CHAPTER 5

Module 5

Consistency, Replication and Fault Tolerance

Syllabus

Introduction to replication and consistency, Data-Centric and Client Centric Consistency Models, Replica Management. Fault Tolerance : Introduction, Process resilience, Reliable client-server and group communication, Recovery.

5.1 Introduction to Replication and Consistency

- The replication of data is carried out in distributed system in order to enhance the reliability and to improve performance. Although reliability and performance can be achieved through replication, it should be ensured to have all the copies of data consistent. If one copy updates then this updates should be propagated to other copies of data also. Otherwise replica will not be same.
- Replication also relates to scalability. If processes access shared data simultaneously the consistency should be ensured. There are many data-centric consistency models. In large scale distributed system, these models are not easy to implement. For this reason, client-centric consistency models are useful, which focus on consistency from the perception of a single client which possibly may be mobile client also.

5.1.1 Reasons for Replication

- Reliability and performance are two main reasons to replicate data. If one replica of the data crashes then other replica can be available for required operation. If single write operation fails in one replica of file out of three replicas then we can consider value returned by other replica. Hence, we can protect our data.
- Replication also improves performance. Scaling of the distributed system can be carried out in numbers or in geographic area. If there is increasing number of processes access the data from same server, then it is better to replicate server and divide the workload. In this way, performance can be improved.
- If we want to scale the system with respect to the size of a geographical area, then replication is also needed. If a copy of data suppose is placed in the proximity of the process which access it, then access time decreases. As a result, the performance as perceived by that process improves.

- Although reliability and performance can be achieved through replication, the multiple copies of the data need to be consistent. If one copy changes then these updates have to be carried out on all copies of data. This price we have to pay against benefits of the replication.

5.1.2 Replication as Scaling Technique

- Replication and caching is mainly used to improve performance. These are also widely applied as scaling techniques. Scalability issues usually come into view in the form of performance problems. If copies of data are placed close to the processes accessing them, then definitely it improves access time and performance as well.
- To have all replicas up to date, updates propagation consumes more network bandwidth. If process P accesses local copy of the data n times per second. Let same copy gets updated m times per second. If n<<m which indicates access-to-update ratio is very low. In this case, many updated versions of data copy will never be accessed by process P. In such case, network communication to update these versions is useless. It requires other approach to update local copy or to not to keep this data copy locally.
- Keeping multiple copies consistent is subject to serious scalability problem. If one process read the data copy then it should always get updated version of data. Hence changes done in one copy should be immediately propagated to all copies (replica) before performing any subsequent operation. Such tight consistency is offered by what is also called synchronous replication.
- Performing update as a single atomic operation on all replicas is difficult as these replicas are widely distributed across large scale network. All replicas should agree on when precisely an update is to be carried out locally. Replicas may need to agree on a global ordering of operations using Lamport timestamps, or this order may be assigned by coordinator. Global synchronization requires lot of communication time, particularly if replicas are spread across a wide-area network. Global synchronization is costly in terms of synchronization.
- The strict assumption about consistency discussed above can be relaxed based on the access and update patterns of the replicated data. It can be agreed that replicas may not be same everywhere. This assumption also depends on purpose for which data are used.

5.2 Data-Centric Consistency Models

- Processes on different machines perform operations on shared data. This shared data is available through use of distributed shared memory, a distributed shared database, or a distributed file system.
- Consider data store which is physically distributed on many machines. Each process has its local copy of the entire data store. The write operation by process changes the data and read operation does not carry out any change in data item. The write operation should be propagated to all other copies of the data store on different machines.
- Read operation by a process on data item should always return the result of last write operation on that data item. A consistency model is basically is a contract between processes and the data store. If processes agree to follow certain rules the data store assures to work correctly. As it is not possible to have global clock, it is difficult to decide exactly which is the last recent write operation. Hence, different model obeys different rules.

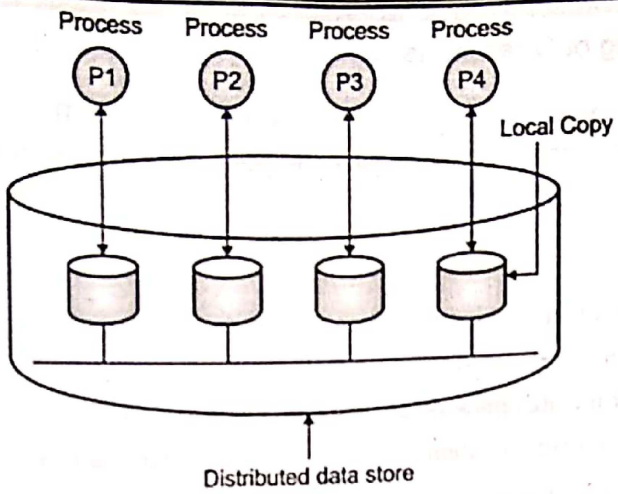

Fig. 5.2.1 : Physically Distributed Data Store

5.2.1 Continuous Consistency

- It is true that, there is no best solution to replicate the data on different machines in network. Efficient solutions can be possible if we consider some relaxation in consistency. The tolerance while considering the relaxation also depends on application. Applications can specify their tolerance of inconsistencies in three ways. These are :
  - o Variation in numerical values between replicas.
  - o Variation in staleness between replicas.
  - o Variation with respect to the ordering of update operations.
- If data have numerical semantics for the applications then they uses numerical deviation or variation. For example, if replicated records contain stock market price data then application may specify that two copies should not deviate more than Rs 10. This would be absolute numeric deviation. On the other hand, a relative numerical deviation could be specified to have deviation between two copies not more than some specific value.
- In both cases of deviations, if a stock goes up without violating the specified numerical deviations, replicas would still be considered to be mutually consistent. A staleness deviation takes in to consideration the last updated time of replica. Some applications can tolerate the data supplied by replica which can be old data, provided it is not too aged. For example, weather reports may remain true for few hours as parameter values do not change suddenly. In such cases, a main server, although it receives current updates, may take decision to propagate updates to the replicas periodically.
- At last, there are some applications that permits ordering of updates that can be different at the different replicas, provided the differences stay bounded. These updates can be viewed as they are applied tentatively to a local copy, until the global agreement from all replicas arrives. As a result, some updates may need to be rolled back and applied in a different order before becoming permanent. Naturally, ordering deviations are not easy to grasp than the other two consistency metrics.

5.2.2 Consistent Ordering of Operations

In parallel and distributed computing, multiple processes share the resources. These processes access these resources simultaneously. The semantics of concurrent access to these resources when resources are replicated led to use of consistency model.

Sequential Consistency

- The time axis is drawn horizontally and interpretation of read and write operation by process on data item is given below. Initially each data item is considered as *NIL*.
  - **$W_i(x)a$:** Process P_i perform write operation on data item x with value a.
  - **$R_i(x)b$:** Process P_i perform read operation on data item x which return value b.
- As shown in following Fig. 5.2.2 (a), process P_1 perform write operation on data item x and modifies its value to a. This write operation is performed by process P_1 on data store which is local to it. We have assumed that data store is replicated on multiple machines. This operation then propagated to other copies of the data store. Process P_2 later reads x from its local copy of the data store and see value a. This is perfectly correct for strictly consistent data store.

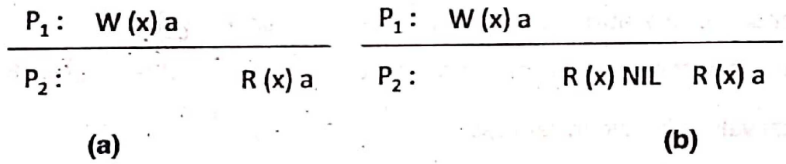

Fig. 5.2.2

- As shown in Fig. 5.2.2 (b), Process P_2 later reads x from its local copy of the data store and see value NIL. After some time, Process P_2 see the value as a. This means it takes some time to propagate the updates from process P_1 to P_2. It is acceptable if we are not considering data store as strictly consistent.
- Sequential consistency model was first defined by Lamport. It is important data-centric consistency model. Following condition data store should satisfy to stay sequentially consistent.
- 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.
- Processes executes concurrently on different machines. Any valid interleaving of read and write operations is tolerable behavior. All these processes see the same interleaving of read and write operations. Time of operation is not considered here.
- Following Fig. 5.2.3 (a) and (b) shows sequentially consistent data store. In Fig. 5.2.3 (a) Process P_1 first performs write on x in its local copy of data store. This sets the value of x as a. Later process P_2 writes b to data item x in its local copy of data store. Both processes P_3 and P_4 first read b and then a. The write operation of process P_2 appears to have occurred before that of process P_1. In Fig. 5.2.3 (b), both processes P_3 and P_4 first read a and then b. Both the data stores shown are sequential consistent.

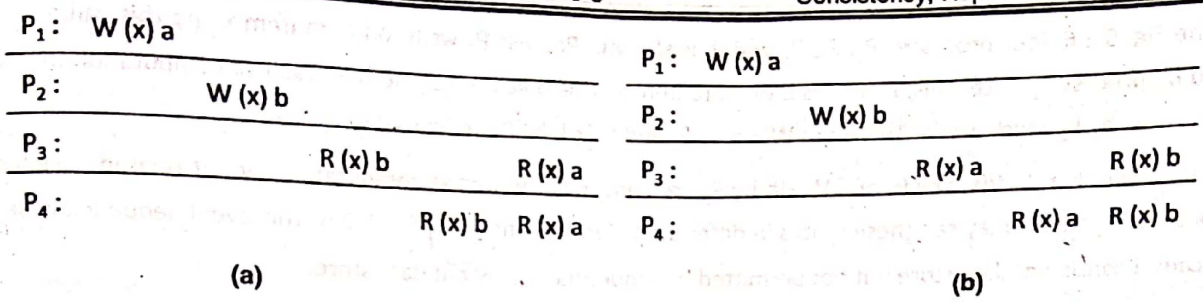

Fig. 5.2.3

- Following Fig. 5.2.4 violets sequentially consistency. Processes P_3 see that first data item been changed to b and later to a. On the other hand, process P_2 will conclude that final value is b.

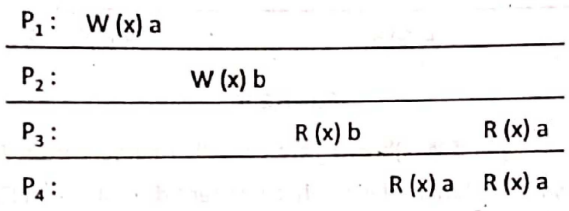

Fig. 5.2.4

- As an example, consider three processes P_1, P_2 and P_3 which are executing concurrently. Three integer type data items a, b, and c are stored in shared sequentially data store. Assignment is considered as write operation and read statement reads two parameters simultaneously. All statement executes indivisibly. Processes are shown below.

Process P_1	Process P_2	Process P_3
a = 1 ;	b = 1 ;	c = 1 ;
read(b, c);	read(a, c);	read(a, b);

Fig. 5.2.5

- Three processes total has 6 statements and hence, 720 (6!) possible execution sequences. If sequence begin with statement a = 1 then total 120 (5!) sequences. The sequences in which statement read(a, c) appears before statement b = 1 and sequences in which statement read(a, b) appears before statement c = 1 are invalid. Hence, only 30 sequences of execution which starts with a = 1 are valid. Similarly 30 starting with b = 1 and other 30 starting with c = 1 sequences are valid. In total, 90 execution sequences are valid out of 720. These can produce different program results which are acceptable under sequentially consistency.

Causal Consistency

- Causal consistency model makes distinction between the events that are causally related and those that are not causally related. If any event b is caused or occurred by previous event y then every process first should see y and then x. Causally consistent data store follows the following condition :
- All processes must see the causally related writes in the same order. Those write which are not causally related (concurrent writes) processes may see in different order on different machines.

- In the Fig. 5.2.6, four processes P_1, P_2, P_3 and P_4 is shown. Process P_1 write on data item x and this value "a" is later read by process P_2. After this, P_2 writes b on data item x. This b value may be the result of computation involving the value read by P_2 which was a. Therefore W(x)a by P_1 and W(x)b by P_2 are causal writes.
- On the other hand, W(x)c by P_1 and W(x)b by P_2 are concurrent writes as computationally not related to each other. Processes P_3 and P_4 may see these writes in different order. It is shown in Fig. 5.2.6. This event sequence is permitted by causal consistent data store but not permitted by sequential consistent data store.

P_1 :	W (x) a			W (x) c	
P_2 :		R (x) a	W (x) b		
P_3 :		R (x) a		R (x) c	R (x) b
P_4 :		R (x) a		R (x) b	R (x) c

Fig. 5.2.6

- In Fig. 5.2.7 (a), W(x)a by process P_1 and W(x)b by P_2 are causally related writes. But, processes P_3 and P_4 see these writes in different order. Hence, it is violation of causally-consistent data store. In Fig. 5.2.7 (b), as W(x)a by process P_1 and W(x)b by P_2 are concurrent writes. This is due to removal of read operation. Hence, figure shows correct sequence of events in a causally consistent data store.

P_1 :	W (x) a			
P_2 :		R (x) a	W (x) b	
P_3 :			R (x) b	R (x) a
P_4 :			R (x) a	R (x) b

(a)

P_1 :	W (x) a		
P_2 :		W (x) b	
P_3 :		R (x) b	R (x) a
P_4 :		R (x) a	R (x) b

(b)

Fig. 5.2.7

FIFO Consistency

- FIFO consistency is the next step in relaxing the consistency by dropping requirement of seeing the causally related writes in same order at all the processes. It is also called the PRAM consistency. It is easy to implement. It says that:
- All the processes must see the writes done by single process in the order in which they were issued. The writes from different processes, processes may see in different order.
- In Fig. 5.2.8, a valid sequence of events for FIFO consistency is shown. Here, W(x)b and W(x)c are the writes from same process P_2. Processes P_3 and P_4 see these writes in the same order i.e. first b and then c.

P_1 :	W (x) a				
P_2 :		R (x) a	W (x) b	W (x) c	
P_3 :				R (x) b	R (x) a	R (x) c
P_4 :				R (x) a	R (x) b	R (x) c

Fig. 5.2.8

Grouping Operations

- Consistency defined at the level of read and write operations does not match its granularity that is offered by applications. The different programs running concurrently use shared data. For this purpose, synchronization mechanisms and transactions are used to control the concurrency between programs. Therefore program level read and write operations are grouped together and bracketed with pair of operations ENTER_CS and EXIT_CS.
- Consider the distributed data store which we have assumed. If process has successfully executed ENTER_CS, it guarantees that its local copy of data store is up to date. Now process can safely execute series of read and write operations inside CS on that store and exit the CS by calling EXIT_CS.
- A series of read and write operations within program are performed on data. This data are protected against simultaneous accesses that would lead to seeing something different than the result of executing the series as a whole. It is needed to have precise semantics about the operations ENTER_CS and EXIT_CS. The synchronization variables are used for this purpose.

Release Consistency

- In release consistency model, acquire operation is used to tell the data store that CS is about to enter and release operation is used to tell that, CS has just been exited. These operations are used to protect the shared data items. The shared data that kept consistent are said to be **protected.** Release consistency guarantees that, when process carry out acquire, the store will make sure that all the local copies of protected data are brought up to date to be consistent with remote copies if must be.
- After release is done, the modified protected data is propagated to other local copies of the store. Carrying out acquire does not guarantee that locally done updates will be propagated immediately to other copies. Carrying out release does not ensure to fetch updates from other copies. Following Fig. 5.2.9 shows valid events for release consistency.

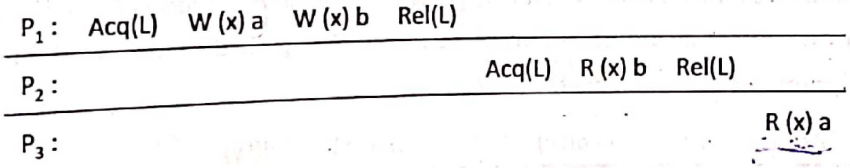

Fig. 5.2.9

- After carrying out acquire, P_1 updates shared data items twice and then carry out release.
- After this, Processes P_2 performs acquire and reads data item x which is b. This value was updated by process P_1 at the time of release. Hence, it is guaranteed that Processes P_2 gets this updated value. On the other hand, process P_3 does not carry out acquire and hence it does not get current value of x. There is no compulsion to get current value of x and returning a is permitted.
- Release consistent distributed data store follows following rules
  - Before carrying out read and write operations on shared data, all previous acquires carried out by process must have completed successfully.
  - Before release is permitted to be carried out, all previous reads and writes carried out by process must have been completed.
  - Accesses to synchronization variables are FIFO consistent.

Entry Consistency

- The shared synchronization variables are used in this model. When process enters the CS, it should acquire the related synchronization variables and when exit the CS, it should release these variables. The current owner of synchronization variable is the process which has last acquired it.
- The owner may enter and exit CSs repetitively without sending any messages on the network. A process not currently having synchronization variable but need to acquire it has to send a message to the current owner asking for ownership and the current values of the data coupled with that synchronization variable. Following rules are needed to follow.
  1. An acquire access of a synchronization variable is not permitted to carry out with respect to a process until all updates to the guarded shared data have been carried out with respect to that process.
  2. Before an exclusive mode access to a synchronization variable by a process is permitted to carry out with respect to that process, no other process may keep the synchronization variable, not even in nonexclusive mode.
  3. After an exclusive mode access to a synchronization variable has been carried out, any other process' next nonexclusive mode access to that synchronization variable may not be carried out until it has carried out with respect to that variable's owner.
- First condition says that at an acquire, all remote updates to the guarded data must be able to be seen. The second condition says that before modification to a shared data item, a process must enter a CS in exclusive mode to confirm that no other process is attempting to update the shared data simultaneously. The third condition says that if a process wants to enter a CS in nonexclusive mode, it must first ensure with the owner of the synchronization variable guarding the CS to obtain the latest copies of the guarded shared data.
- A valid vent sequence for entry consistency is shown Fig. 5.2.10. Lock is associated with each data item. Process P_1 does acquire on x and update it. Later it does acquire on y. Process P_2 does acquire for data item x but not for y. Hence, process P_2 will read value a for data item x but it may get NIL for data item y. Process P_3 does first acquire on y hence it reads value b after y is released by process P_1. The correctly associating the data with synchronization variables is one of the programming problem with this model.

P_1 :	Acq(Lx)	W (x) a	Acq(Ly)	W (y) b	Rel(Lx)	Rel(Ly)		
P_2 :					Acq(Lx)	R (x) a		R (y) NIL
P_3 :						Acq(Ly)	R (y) b	

Fig. 5.2.10

Consistency versus Coherence

- Consistency is concerned with the read and writes operations by the processes that are performed on set of data items. A consistency model explains what can be predictable with respect to that set when numerous processes concurrently work on that data.
- Whereas, coherence models expects result from single data item. In this case, assumption is that a data item is replicated at several places; it is said to be coherent when the different copies stick to the policies as defined by its allied coherence model.

5.3 Client-Centric Consistency Models

- In section 5.2, we have considered simultaneous updates on distributed data store. Some distributed data stores may not get updated simultaneously and if, this situation can be easily resolved. On these data store, most of the operations are only reading the data.
- These data stores obeys eventual consistency model which is considered to be weak consistency model. In client-centric consistency models, many inconsistencies can be hidden in a comparatively cheap manner.

5.3.1 Eventual Consistency

- Practically in many situations, concurrency exists in some limited form. If processes only read data from database and hardly some processes performs updates on it then it is not necessary to propagate updates to all processes immediately.
- In World Wide Web (WWW) web pages either can be updated by authority or by owner of that page. In this case, resolving the write-write conflicts at all not require. Conversely, to get better efficiency, browsers and web proxies are frequently configured to hold accessed page in a local cache and to return that page upon the next request.
- Although the web caches returns old pages, it can be acceptable at some extent to requesting client. Eventual consistency states that, updates will be propagated gradually to update the replica if updates do not take place for long time. Eventual consistent data stores work in correct manner, provided clients always access the same replica. But, problems occur when diverse replicas are accessed over a short period of time.
- Suppose, client is mobile client and performing operations on copy of database. Client disconnects with current database and later connects to other replicated copy. In this case, if updates are not propagated from previous copy to recently connected copy of database then he/she may notice inconsistent behavior. Client-centric consistency models resolve such problem. This is typical example of eventual consistency.
- Particularly, client-centric consistency offer assurance *for a single client* about the consistency of accesses to a data store by that client only. No assurance is provided about simultaneous accesses by different clients.
- Consider data store which is physically distributed on many machines. Each process has its local copy of the entire data store. It is assumed that, network connectivity is unreliable. When process accesses data store, it carry out operation on local or nearest copy available. Updates are then eventually propagated to other copies. Following four models are suggested for client-centric consistency.
  - Monotonic Reads
  - Monotonic Writes
  - Read Your Writes
  - Writes Follow Reads
- Following notations are used to describe above models.
  - $x_i[t]$: It indicate the version of data item x at local copy L_i at time t. This version is result of series of write operations at local copy L_i occurred since initialization.
  - $WS(x_i[t])$: Series of write operations on data item x at local copy L_i at time t.
  - $WS(x_i[t_1];x_j[t_2])$: It denote that operations in $WS(x_i[t_1])$ have also been carried out later at local copy L_j at later time t_2.

5.3.2 Monotonic Reads

- Monotonic-read consistency is guaranteed by data store if 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.***
- This ensures that in later read operation, process will always get (read) latest or same value (which was returned in last read operation) and not older one. Consider the example of distributed email database. In this case, mailboxes of users are distributed and replicated on multiple machines. The received mail at any location is propagated in lazy manner (as per demand) to other copies of mailbox.
- Only that data gets forwarded which is needed to maintain consistency. Suppose user reads the mailbox in Mumbai and later flies to Delhi. The Monotonic-read consistency guarantees that, the messages in mailbox that were read at Mumbai will also be in mailbox at Delhi.
- Fig. 5.3.1(a) shows monotonic consistent data store. L_1 and L_2 are local copies at different machines. Process P carries out read ($R(x_1)$) operation on x at L_1. The value returned to the process P is the result of write operation $WS(x_1)$ carried out at L_1. Later, process P carries out read operation ($R(x_2)$) on x at L_2. Before P performs read operation on L_2, here writes at L_1 is propagated to L_2 and it is shown in diagram by $WS(x_1; x_2)$. Hence, $WS(x_2)$ is the part of $WS(x_1)$. This shows that monotonic-read consistency is guaranteed.

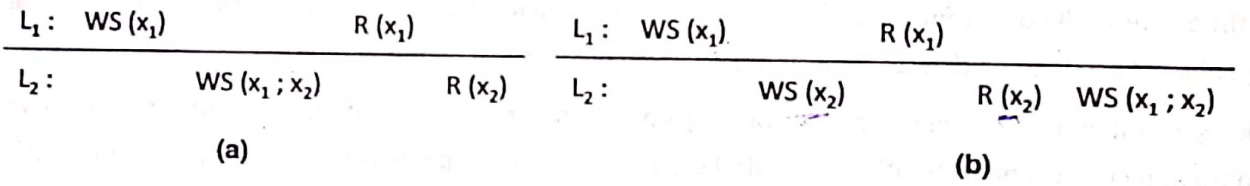

Fig. 5.3.1

- Fig. 5.3.1 (b) shows data store which does not offer monotonic read consistency. After P carries out read ($R(x_1)$) operation on x at L_1, process P carries out read operation ($R(x_2)$) on x at L_2. But only $WS(x_2)$ write operation is performed at L_1. Here, result of writes from L_1 is not propagated to copy L_2. This shows that monotonic-read consistency is not guaranteed in this case.

5.3.3 Monotonic Writes

- In this model, it is important to propagate write operations in the correct order to all copies data store. Monotonic-read consistency is guaranteed by data store if 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.**
- This ensures that, result of current write operation by the process on data item reflects the effect of previous write operation by the same process on same data item. The copy of data item may exist anywhere. But updates should be done on copy that is recently updated.
- Consider the updating the software library. In this case, changes are always carried out in one or more functions. With monotonic write consistency promise is given that, whenever changes will be carried out at other copy of the library, all previous updates will be carried out first.

- Monotonic-write consistency model look like data centric FIFO consistency model. Fig. 5.3.2(a) shows monotonic-writes consistency. Process P carries out write ($W(x_1)$) operation on x at L_1. Later, process P carries out write operation ($W(x_2)$) on x at L_2. Here, it is ensured that, previous write operation at L_1 has already been propagated to local copy L_2. This is indicated by operation $WS(x_1)$ at L_2.
- In Fig. 5.3.2(b), monotonic-writes consistency is not guaranteed as write operation at L_1 has not been propagated to copy L_2.

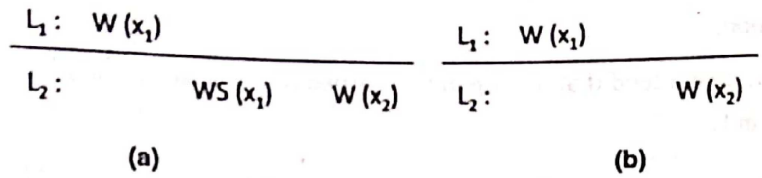

Fig. 5.3.2

5.3.4 Read Your Writes

- Read Your Writes model is closely related to monotonic reads model. Read-your-writes consistency is guaranteed by data store if following condition holds :
- The result 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.
- This ensures that; write operation is always completed before a successive read operation by the same process, regardless of where that read operation occurs.
- Consider the example of changing password of digital library account on web. It takes some time to come in to effect this change. As a result, library may be inaccessible to the user. A reason for delay is use of separate server to manage passwords and it may take some time to subsequently propagate encrypted passwords to the various servers that form the library.
- In Fig. 5.3.3(a), process P carries out write ($W(x_1)$) operation on x at L_1. Later, process P carries out read operation ($R(x_2)$) on x at L_2. Read-your-writes consistency guarantees that, the succeeding read $R(x_2)$ operation at local copy L_2 observes the effects of the write operation $W(x_1)$ carried out at L_1. This is indicated by $WS(x_1; x_2)$, which states that $W(x_1)$ is part of $W(x_2)$.
- Fig. 5.3.3(b) shows that, effect of write operation ($W(x_1)$) on x at L_1 have not been propagated to local copy L_2. This is shown by write operation $WS(x_2)$.

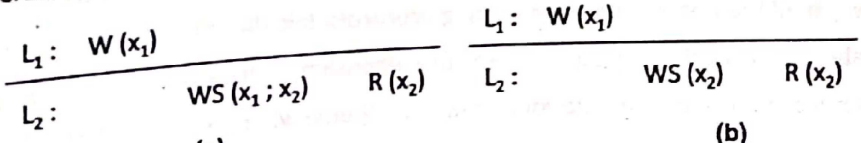

Fig. 5.3.3

5.3.5 Write Follows Reads

- In this consistency model updates of last read operation gets propagated. Write-Follows-Reads consistency is guaranteed by data store if following condition 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.

- It states that, process will always perform successive write operation on up to date copy of data item with value most recently read by that process. Consider the example of posting articles and their reactions in network newsgroup. User first read the article X and then posts its reaction Y. As per write-follows-reads consistency, reaction Y will be written to ant copy of the newsgroup only after article X has been written as well.
- In Fig. 5.3.4 (a), process P carries out read ($R(x_1)$) operation on x at L_1. The value x which just now read at L_1, returns value which just written ($WS(x_1)$) on x at L_1. This value also appears in the write set at L_2 where same process later carries out write operation.
- In Fig. 5.3.4 (b), it is not guaranteed that, the operation carried out at L_2 are carried out on a copy that is consistent with the copy just read at L_1.

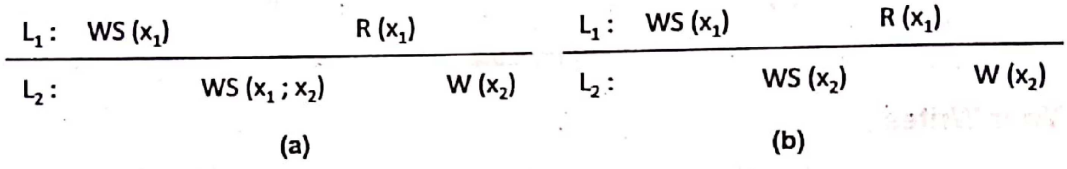

Fig. 5.3.4

5.4 Replica Management

- Replica management is important issue in distributed system. The key issues are to decide where to place replicas, by whom and the time at which placement should be carried out.
- Other issue is selection of approaches to keep the replica consistent. Placement problem also includes placing the servers and placing content. Placing server involves finding the best location and placing content involves which is the best server for placement of content.

5.4.1 Replica-Server Placement

For optimized placement, k best location out of n ($k < n$) needs to be selected. Following are some of the approaches to find best locations.

- Consider distance between clients and locations which is measured in terms of latency or bandwidth. In this solution, one server at a time is selected such that the average distance between that server and its clients is minimal given that previously *k* servers have been already placed (n–k locations are left).
- In second approach, instead of considering the position of clients, take topology of internet formed by autonomous systems (ASs) in which all the nodes runs same routing protocols. Initially take largest AS and place server on router having largest number of network interfaces. Repeat the algorithm with second largest AS and so on. These both algorithms are expensive in terms of computation. It takes $O(n^2)$ time where n are number of locations to be checked.
- In third approach, a region to place the replicas is identified quickly in less time. This identified region comprises collection of nodes accessing the same content, but for which the internode latency is low. The identified region contains more number of nodes with compare to other regions and one of the nodes is selected to play role of replica server.

5.4.2 Content Replication and Placement

Replicas are classified in following three types.

1. Permanent replicas
2. Server-initiated replicas
3. Client-initiated replicas

1. Permanent Replicas

- Initially, distributed data store contains these replicas. These are initial set of replicas that represents the distributed data store. These replicas are less in number in many cases. For example, in distribution of web site, the files that form the site are distributed across few servers in single location. The incoming request for file gets forwarded to one of the server, may be in round robin manner.
- In other type of distribution mirroring is used in which case, a web site is copied to few number of servers called as mirror sites. These servers are geographically spread across internet. The clients request gets forwarded to one of the server. Mirroring approach is used with cluster-based Web sites having limited number of replicas, which are more or less statically configured. As another example, a distributed database can be distributed and replicated to number of servers that collectively form a cluster of servers.

2. Server-Initiated Replicas

- These replicas are created by the owner of the data store in order to enhance the performance. Suppose Web server placed in a city easily handles incoming requests and over a couple of days, sudden burst of requests arrive to this server from unexpected location which is far from server. It may be useful to place the temporary replicas in regions from where requests are coming.
- Web hosting services offers collection servers placed across the internet. These servers stores and offers access to the web files which belongs to third parties. These hosting services can dynamically replicates files to the server where it is needed in order to improve performance. If servers are already placed then content placement is easier than placement of servers.
- The algorithm supports web pages and it is considered that updates are comparatively less than read requests. Two issues are considered by the algorithm. First, it reduces load on server. Second, the needed files can be migrated from current server to the servers that is available in the proximity of clients that issue many requests for those files.
- In this algorithm, each server keeps track on count of number of accesses to the file by clients and the locations of the incoming requests. For a particular client C, any server involved in web hosting service can determine closest server to this client. This information, servers may take from routing database.
- If file F is stored at sever S_2 and sever S_1 is closest to the clients C_1 and C_2. In this scenario, all access requests for file F at server S_2 from C_1 and C_2 are together gets registered at S_2 as a single count of accesses $CS_2(S_1, F)$. A deletion threshold for file F at server S *(del(S, F))* is also maintained. It says that, if count of incoming requests from clients for file F at server S drops below this threshold then file should be removed from server S.
- Removing the file from server in this manner again reduces number of replicas. This can again increase the load on other servers. So, some special actions are carried out in order to survive at least one copy of the same file.

- In the same way, file F is replicated if incoming requests for it exceeds the replication count *rep(S, F)*. If count of requests for file F is in between these two thresholds then file is permitted to migrate on server in proximity of clients requesting it.
- Every server again assesses the placement of file which it stores. In case of number of access requests count if drops below threshold *(del(S, F))* at server S, it deletes F, provided it is not a last copy. Certainly, replication is carried out only if the total number of access requests for *F* at S exceeds the replication threshold *rep(S, F)*.

3. Client-Initiated Replicas

- These replicas are initiated by client and also called as client caches. Caches are always said to be managed by clients. In many cases, the cached data should be inconsistent with data store. Client caches improve the access time to data as accesses are carried out from local cache. This local cache could be maintained on client's machine or in separate machine in same LAN as client. Cache hit occurs if data is accessed from the cache.
- Cache holds the data for short period. This is either to prevent extremely stale data from being used or to make room for other requested data from data store. Caches can be shared between many clients to improve cache hits. The assumption is that, requested data by one client may be useful to another nearby client. This assumption may be correct for some types of data stores.
- A cache either is usually placed on the same machine as its client or on a machine shared by clients on the same LAN. In some other cases, system administrators may place a shared cache between a number of departments, organizations, or for whole country. Other approach is to place servers as caches at some specific locations in wide area network. The client locates nearest server and requests it to keep copies of the data the client was previously accessing from somewhere else.

5.4.3 Content Distribution

Management of the replicas also involves propagation of updated contents to the appropriate servers.

State versus Operations

Following are the different possibilities for propagation of updates.

- Only a notification of an update is propagated.
- Data is transferred from one copy to other copy.
- Update operation is propagated to other copies.

Propagation of notification of update

- In an invalidation protocol, if updates occur in one copy then other copies are informed about these update. This informs to other copies that the data they hold are no longer valid. It may specify that, which part of the data store has been updated, so that only changed part of a copy is actually invalidated.
- In this case, only notification is propagated. Before carrying out the update operation on an invalidated copy, it needs to be updated first, depending on the particular consistency model that is to be supported. The network bandwidth needed to propagate notifications is less. Invalidation protocol works best in situation where many update operations are carried out compared to read operations (read-to-write ratio is small).

- As an example, consider the data store where updates are propagated by transferring modified data to all other copies. Suppose, frequent updates takes place compared to read operation. Consider two consecutive update operations. Since there is no read operation has been performed between two updates, second update will simply overwrite the first update in all copies. In this case sending notification only will be more beneficial.

Propagation of updates by transferring the data

- This approach is more suited when read-to-write ratio is relatively high. In this case, chances are more of modified data will be read is before next update takes place. Hence, in such situation transferring the modified data to all other copies is necessary.
- In order to save the bandwidth, log of changes made can be sent instead of sending the modified data. Communication overhead can also be saved by sending single message for multiple modifications. In this case, these modifications are packed in to single message.

Propagation of update operations

- This approach informs to each copy about which update operation it should carry out. It sends only parameter values which will be required to carry out update operation at target copies. If size of parameters to be sent is relatively small then propagating update operations requires less bandwidth.
- It is also called as active replication. At each replica more processing power is required if operations to be carried out are more complex.

Pull versus Push Protocols

- Push Protocols are server-based protocols in which updates are propagated by servers themselves to the replicas although they do not request for these updates. Push-based approaches usually used between permanent replicas server-initiated replicas, but can also be used to send updates to client caches. Push Protocols are generally used where maintaining high degree of consistency is necessary.
- The largely shared replicas, permanent replica and server-oriented replicas are often shared between many clients. Clients frequently carry out read operation on these replicas. In this case, read-to-update ratio at each replica is comparatively high. Hence, each read operation by clients expected to get updated copies. If client asks for recently updated copy then push-based protocol immediately makes it available.
- On the other hand, pull-based protocols works to get updates as per request issued. In this case, either server or client sends request to another server to obtain any updates done recently there. These protocols are called as client based protocol. For example, In case of web caches whenever requests arrives for the data, they often check with original web server to confirm whether cached data is up to date or not since they were cached.
- If cached data are modified at original web server then it gets transferred to cache and then returned to the requesting client. A pull-based protocol is efficient when update operations are relatively high compared to read operations (read-to-update ration high). This is when there are non shared client caches. In this case there is single client. If caches are shared among many clients then also this protocol is efficient. The main disadvantage of a pull-based approach compared to a push based approach is increase in response time in case of cache miss.

- Consider the situation with single (non-distributed) server and many clients :
  - In this case, push base protocol has to keep track of many client caches. This puts load on server. Server should be stateful server although it is less fault tolerant. In pull-based approach, server does not require to keep track of many client caches.
  - In push-based approach, server sends the messages to clients. These are update or fetch update later messages. In pull-based approach, client sends the messages to server. Client has to poll the server to check if any updates occurred and if then fetch the updates.
  - When a server sends modified data to the client caches, the response time at the client side is zero. When server pushes invalidations, the response time is the same as in the pull-based approach, and is decided by the time involved in accessing the modified data from the server.
- As a balance between push and pull-based approaches, a hybrid form of update propagation based on leases is introduced. A lease is a time interval in which server pushes updates to clients. In other word, it is a promise by the server that it will push updates to the client for a specific time. After expiry of the lease, the client is forced to poll the server for updates and pull in the modified data if required. In other approach, a client requests a new lease to push updates when the earlier lease expires.

Unicasting versus Multicasting

- The use of unicasting or multicasting depends on whether updates needs to be pushed or pulled. If server is a part of data store and push updates to other n number of servers then it sends n separate messages. In this case, with multicasting underlying network takes care of sending n messages.
- Suppose all replicas are placed in LAN then hardware broad casting is available. In that case broadcasting or multicasting is cheaper. In this case, unicasting the update is expensive and less efficient. Multicasting can be combined with push based approach where single server sends message to multicast group of other n servers.

5.5 Fault Tolerance

In distributed system, partial failure may occur. In partial failure, failure of one component of system may affect operation of other component but yet some other components may remain unaffected. The distributed system should be designed to recover from such partial failure automatically without affecting the performance.

5.5.1 Basic Concepts

- Distributed system should tolerate faults. Fault tolerant system is strongly related to dependable system. Dependability covers following requirements for distributed system along with other important requirements.
  - **Availability :** It refers to the probability that system is performing its operation correctly. If system is working at any given instant of time then it is highly available system.
  - **Reliability :** It refers to the working of system continuously without failure or interruption during a relatively long period of time. Although system may be continuously available, but we cannot say that it is reliable.
  - **Safety :** Safety refers to the situation that when a system temporarily fails to function correctly, nothing catastrophic takes place.
  - **Maintainability :** It refers to how failed system can be repaired without any trouble. System should be easily maintainable which will then show high degree of availability.

- Dependable system should also offer high degree of security. The system is said to fail if it is not offering all the services for which it was designed. An error is a part of a system's state that may cause a failure. The cause of an error is called a fault. System should provide its services although faults exist in the system.
- There are three types of faults. These are transient, intermittent, and permanent. Transient faults occur once and then vanish. If the operation is repeated, the fault goes away. An intermittent fault occurs, then disappears of its own accord, then again appears, and so on. A permanent fault continues to live until the faulty part is replaced.

5.5.2 Failure Models

- In distributed system, communication channel, processes, machines can fail during the course of execution. Failure model suggest the ways in which failures in different components of the system may occurs and to provide the understanding of this failure.
- Suppose in distributed system, servers communicate with each other and with their clients. This system is not satisfactorily offering services indicates that servers, communication channels, or possibly both, are not carrying out the work what they are supposed to do.
- In this example, **crash failure** occurs when server halt but it was working correctly until it halts. If server fails to give reply to the incoming requests from other servers or clients then it is considered to be **omission failure**. If server fails to send the message then it is **send omission failure** and if fails to receive the incoming message then it is **receive omission failure**. If server response is not within the specified time interval then it is **timing failure**. If server gives incorrect response then it is **response failure**. **Value failure** occurs when value of the response is wrong. If server deviates from the correct flow of control then it is **state transition failure**. Suppose server produces arbitrary responses at arbitrary times then it is referred as **arbitrary failure**.

Omission Failures

The omission failure refers to the faults that occur due to failure of process or communication channel. Because of this failure, process or communication channel fails to carry out the action or task that it is supposed to do.

Process Omission Failures

- The main omission failure of process is when it crashes and never executes its further action or program step. In this case, process completely stops. When any process does not responds to requesting process repeatedly, the requesting process detects or concludes this crash.
- The detection of crash in above manner relies on use of timeouts. In asynchronous system, timeout can point towards only that process is not responding. The process may have crashed, may be executing slowly or messages have yet not delivered to the system.
- If other process surely detects the crash of process then this process crash is called as fail-stop. This fail-stop behaviour can be formed in synchronous system when processes uses time out to know when other process fail to respond and delivery of messages are guaranteed.

Communication Omission Failures

- Sending process P executes *send* primitive and puts message in its outgoing buffer. Communication channel transport this message to receiving process Q's incoming buffer. Process Q then executes *receive* primitive to take this message from its incoming buffer. It then delivers the message.

- If communication channel is not able to transport message from process P's outgoing buffer to process Q's its incoming buffer then it produces omission failure. The message may be dropped due to non-availability of buffer space at receiving side, no buffer space at intermediate machine or network error detected by checksum calculation with data in message.
- Send-omission failures refer to loss of messages between sending process and its outgoing buffer. Receive- omission failures refer to loss of messages between incoming buffer and the receiving process.

Arbitrary Failures

- In this type of failure process or communication channel behaves arbitrarily. In this failure, the responding process may return wrong values or it may set wrong value in data item. Process may arbitrarily omit intentional processing step or carry out unintentional processing step.
- Arbitrary failure also occurs with respect to communication channels. The examples of these failures are: message content may change, repeated delivery of the same messages or non-existent messages may be delivered. These failures occur rarely and can be recognized by communication software.

Timing Failures

- In synchronous distributed system, limits are set on process execution time, message delivery time and clock drift rate. Hence, timing failures are applicable to this system.
- In timing failure, clock failure affects process as its local clock may drift from perfect time or may exceeds bound on its rate.
- Performance failure affects process if it exceeds the defined bounds on the interval between two steps.
- Performance failure also affects communication channels if transmission of message take longer time than defined bound.

Masking Failures

- Distributed system is collection of many components and components are constructed from collection of other components. Reliable services can be constructed from the components which exhibit failures.
- For example, suppose data is replicated on several servers. In this case, if one server fails then other servers would provide the service. Service mask failure either by hiding it or by transforming it in more acceptable type of failure.

5.5.3 Failure Masking by Redundancy

- The use of redundancy is main technique for masking the faults. Redundancy is categorized as information redundancy, time redundancy, and physical redundancy. With information redundancy extra bits are added for recovery. For example, hamming code added at sender side in transmitted data for recovery from noise.
- Time redundancy is particularly useful in case of transient or intermittent faults. In this, action is performed again and again if needed with no harm. For example, aborted transaction can be redone with no harm.
- With physical redundancy, to tolerate the loss or malfunctioning of some components either extra hardware or software components are added in system.

5.6 Process Resilience

In distributed system, failure of processes may happen. In this case, fault tolerance is achieved by replicating processes into groups.

5.6.1 Design Issues

- If identical processes are organized in group, then message sent to this group gets delivered to all the group members. Every process in group receives this message. If one of the members fails the other can take over this process's job.
- These process groups may be managed dynamically. It allows creating new groups and destroying old groups. During the system operation, a process can join a group or leave it. A process can be a member of multiple groups simultaneously. As a result, mechanisms are required for group management and group membership.
- Groups permit processes to deal with sets of processes as a single abstraction. Hence a process can send a message to a group of servers without knowing to which servers or number of servers and their locations, which may vary from one call to the next.
- Two types of groups exist. In **flat group**, all the processes are at equal level and all decisions are made collectively. The advantage of this organization is that there is no single point of failure. But decision making is complex as all are involved in this process. Voting incur delay and overhead.
- In simple **hierarchical group**, one process acts as coordinator and all the others are workers. In this group, a request from worker or external client is first gets handed over to coordinator and then it decides which worker is appropriate to carry it out, and forwards it there. In this group single point of failure is the problem. If coordinator crashes the whole group will affect.

Group Membership

- A group server is responsible to create and destroy the groups. It also allows the process to join or leave the group. The group server maintains a complete data base of all the groups and their exact membership. In this, single point of failure is again the problem. If group server crashes then group management will affect.
- Another approach is to manage membership in distributed manner. if reliable multicasting is existing, an external process can send a message to all group members declaring its wish to join the group. To leave a group, a member just sends a goodbye message to every other member.
- As soon as process joins the group, it should receive all the messages sent to that group. After leaving group, process should not receive any message from that group and any other group members should not receive messages from it. There should be some protocol to deal with the situation where many machine suppose fails and group can no longer function at all.

5.6.2 Failure Masking and Replication

- Group of identical processes permits us to mask one or more faulty processes in that group. A group can be formed by replicating the processes and organizing then in a group. Hence, single vulnerable process can be replaced by this group to tolerate the fault. Replication can be carried out either with primary-based protocols, or through replicated-write protocols.

- Primary-based replication for fault tolerance usually emerges in the form of a primary-backup protocol. In this case, a process group is organized in a hierarchical manner in which a primary coordinates all write operations. Primary is fixed but its role can be taken over by backups whenever needed. In a situation of crash of primary, the backups run some election algorithm to elect a new primary.
- Replicated-write protocols are used in the form of active replication, in addition to using quorum-based protocols. These solutions consider the flat group. The main advantage is that such groups have no single point of failure, at the cost of distributed coordination. The advantage of this organization is that there is no single point of failure. But distributed coordination incurs delay and overhead.
- Consider the replicated write systems. Amount of replication required is an important issue to consider. A k fault tolerant system survives faults in k components and still meets its specifications. If processes fail slowly, then having k + 1 of them is sufficient to give k fault tolerance. If k of them just stops, then the reply from the other one can be used. In case of Byzantine failures, processes continuing to run when sick and sending out erroneous or random replies, a minimum of 2k +1 processors are required to attain k fault tolerance.

5.6.3 Agreement in Faulty Systems

- Fault tolerance can be achieved by replicating processes in groups. In many cases, it is required that, process groups reaches some agreement. For example; electing a coordinator, deciding whether or not to commit a transaction, dividing up tasks among workers etc. Reaching such agreement is uncomplicated, provided communication and processes are all perfect. Otherwise problem may arise.
- The general aim of distributed agreement algorithms is to have all the correctly working processes reach an agreement on some issue, and to build that consensus within a finite number of steps. Following cases should be considered.
  - Synchronous versus asynchronous systems: A system is synchronous if and only if the processes are known to operate in a lock-step mode.
  - Communication delay is bounded or not: Bounded delay means every message is delivered with a globally and predetermined maximum time.
  - Message delivery is ordered or not: Messages from same sender is delivered in the same order in which they were sent.
  - Message transmission through Unicasting or Multicasting.

Byzantine Agreement Problem

- In this case, we assume that processes are synchronous, messages are unicast while preserving ordering, and communication delay is bounded. Consider n number of processes. In this example, let n = 4 and k = 1 where k is number of faulty processes. The goal is that, each process should build the vector of length n.
- **Step 1 :** Each non-faulty process i send v_i to other process. Reliable multicasting is used to send the v_i. Consider Fig. 5.6.1. In this case, process 1 send 1, process 2 send 2 and process 3 lie to everyone, sending x, y, and z respectively. Process 4 send 4. Process 3 is faulty process and process 1,2 and 4 are non faulty processes.

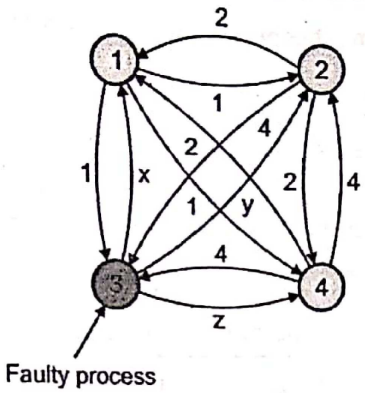

Fig. 5.6.1

- **Step 2 :** The results collected by each process from other processes in the form of vectors is shown below.
  - Process 1: (1, 2, x, 4)
  - Process 2: (1, 2, y, 4)
  - Process 3: (1, 2, 3, 4)
  - Process 4: (1, 2, z, 4)
- **Step 3 :** Now, every process passes its vector to ever other process. The vectors that each process receives from other each process are shown below. Since process 3 is faulty process, it lies and sends new values a to l. these are new 12 values.

Process 1	Process 2	Process 4
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

- **Step 4 :** Now, each process inspects the i^{th} element of each of the newly received vectors. If any value has a majority, that value is put in **Result** vector. If no value has a majority, the corresponding element of the result vector is marked *UNKNOWN*. Processes 1, 2, and 4 all come to agreement on the values for v_1, v_2, and v_4, which is the correct result.

  Result = (1, 2, *UNKNOWN*, 4)
- These processes conclude that about v_3 cannot be decided, but is also irrelevant. The goal of Byzantine agreement is that consensus is reached on the value for the non-faulty processes only.
- If n = 3 and k=1 then only two non-faulty processes and one faulty process. It is shown in Fig. 5.6.2.

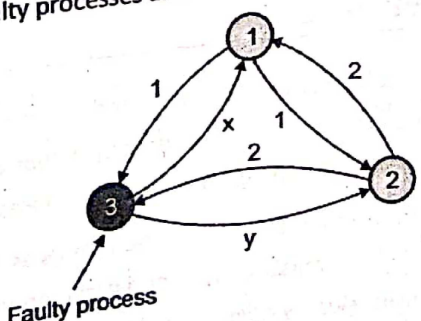

Fig. 5.6.2

- The results collected by each process from other processes in the form of vectors are shown below.
  - Process 1: (1, 2, x)
  - Process 2: (1, 2, y)
  - Process 3: (1, 2, 3)
- Following vectors are received by processes. There is no majority for element 1, element 2 and 3 also. All will be marked as *UNKNOWN* and hence, algorithm is failed.

Process 1	Process 2
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

- In system, if k faulty processes are there then agreement can be achieved if 2k+1 correctly functioning (non-faulty) processes present for a total of 3k+1.

5.6.4 Failure Detection

- For proper masking of failures, it is necessary and requirement to detect them. It is needed to detect faulty member by non-faulty processes. The failure of member detection is main focus here. Two approaches are available for detection of process failure. Either process asks by sending messages to other processes to know whether others are active or not or inactively wait until messages arrive from different processes.
- The latter approach useful only when it can be certain that there is sufficient communication between processes. In fact, actively pinging processes is generally followed. The suggested timeout approach to know whether a process has failed or not, suffers with two major problems. As a first problem, declaring process's failure as it does not return response to ping messages is wrong as this may due to unreliable networks. In other argument, there is no practical implementation to correctly determine failure of process due to timeouts.
- Gossiping between processes can also be used to detect faulty process. In gossiping, process tells other process about its active state. In other solution, neighbors probe each other for their active state instead of relying on single node to take decision. It is also important to distinguish the failure due to process and underlying network.
- This problem can be solved by taking feedback from other nodes or processes. In fact, when observing a timeout on a ping message, a node requests other neighbors to know whether they can reach the presumed failing node. if a node is still active, that information can be forwarded to other interested nodes.

5.7 Reliable Client-Server Communication

- While considering the faulty processes, fault tolerance in distributed system should also consider the failures of the communication channels. Communication failures are also equally important in fault tolerance. Communication channels may also suffer through crash, omission, timing, and arbitrary failures.
- Reliable communication channels are required for exchanging the messages. Practically, while designing the reliable communication channels, main focus is given on masking crash and omission failures. Arbitrary failure may occur in the form of duplicate messages due to retransmission by original sender.

5.7.1 Point-to-Point Communication

- TCP is reliable transport protocol which supports reliable point-to-point communication in distributed system. TCP handles lost of messages is by using acknowledgement and retransmission. Hence, this omission failure is masked by TCP. Client cannot notice such failures.
- If TCP connection is suddenly broken then it is crash failure. In this failure, no more messages can be transmitted through the communication channel. This failure is only masked through establishing new connection by distributed system.

5.7.2 RPC Semantics in the Presence of Failures

- In Remote procedure calls (RPC), communication is established between the client and server. In RPC, a client side process calls a procedure implemented on a remote machine (server). The primary goal of RPC is to hide communication while calling remote procedure so that remote procedure calls looks like just local procedure calls.
- If any errors occur, then masking differences between remote and local procedure calls is not easy. In RPC system, following failure may occur.
  - The client cannot locate the server.
  - The request message from the client to the server is lost.
  - The server crashes after receiving a request.
  - The reply message from the server to the client is lost.
  - The client crashes after sending a request.

The Client Cannot Locate the Server

- In this case, it may happen that all servers are down. In other case, version of client stub is old. Meanwhile server evolves and new interfaces are installed and compiled. At server side new version of stub is present. Hence, binder will be unable to bind with server and failure will be reported.
- As a solution to this problem, an exception can be raised upon errors. Many language supports for writing procedures that are invoked upon errors. Signal handler can also be used for this purpose. Again drawback is that, many languages do not supports exceptions or signals. It is not possible to achieve transparency by using exceptions or signal handlers.

Lost Request Messages

- To deal with this problem, OS or client stub starts timer while sending request. The retransmission of request is carried out if acknowledgement or reply does not arrive before timer expires. Server will not be able to differentiate between original and retransmission if message was truly lost. In this case, no problems arise.
- Suppose so many client request messages are lost. In this case, client will falsely conclude that server is down. Then again situation will be like "cannot locate the server". If request message was not lost then let the server handle it.

Server Crashes

- Following sequence of events takes place when client sends request to server.
  1. A request arrives at server. Server has processed the request and reply is sent to client.

2. A request arrives at server. Server has processed the request. Server crashes before it can send the request.
3. A request arrives at server. Server is crashed before processing the request.

- In case (2) and (3) different solutions are required. In (2), system has to report failure to the client. In (3) retransmission of request is required. Client retransmits when its timer expires.
- In (3), Client wait till server reboots. Client may rebind to new server. **At least one semantics** technique can be used. It means, try until reply arrives from server. It means, RPC has been carried out at least one time, but possibly more.
- In **at-most-once semantics** technique, failure is immediately gets reported to the client. It guarantees that the RPC has been carried out at most one time, but possibly none at all. In other technique, client gets no help and no guarantee. RPC has been carried out from 0 to any number of times. It is easy to implement.

Lost Reply Messages

- In this case client retransmits request message when reply does not arrive before timer expires. It concludes that request is lost and then retransmits it. But, in this case client actually remains unaware about what is happened actually. Idempotent operations can be repeated. These operations produce same result although repeated any number of times. Requesting first 512 bytes of file is idempotent. It will only overwrite at client side.
- Some request messages are non-idempotent. Suppose, request to server is to transfer amount from one account to other account. In this case, each retransmission of the request will be processed by server. Transfer of amount from one account to other account will be carried out many times. The solution to this problem is, try to configure all requests in idempotent way. This is not possible as many requests are inherently non-idempotent.
- In another solution, client assigns sequence number to each request. Server can then keep track on most recent request and it can differentiate between original (first) and retransmitted requests. Now server can reject to process any request a second time. Client may put additional bit in first message header so that server will take it for processing. This is required as retransmitted message requires more care to handle.

Client Crashes

- In this case, client sends request to server and before reply arrives it crashes. The computation of this request is now going on at server. Such computation for which no one is waiting is treated as an **orphan.** This computation at server side (orphan) wastes CPU cycles, locks files, hold resources.
- Suppose client reboots and sends same request. If reply from orphan arrives immediately then there will be confusion state. This orphan needs solution in RPC operation. Four possible solutions are suggested as below.
  1. **Orphan Extermination :** Client maintains log of message on hard disk mentioning what it is about to do. After a reboot, log is verified by client and the orphan is explicitly killed off. This solution needs to write every RPC record on disk. In addition, these orphans may also do RPC, hence, creating grandorphans or further descendents that are difficult or practically not possible to locate.
  2. **Reincarnation :** In this solution, no need to write log on disk. Time is divided in sequentially numbered epochs. Client broadcasts a message to all machines telling the start of a new epoch after its reboot. Receivers of this broadcast message then kills all remote computations going on the behalf of this client. This solution also suffers through network delay and some orphan may stay alive. Fortunately, however, when they report back, their replies will include an out of date epoch number so that they can easily notice it.

3. **Gentle Reincarnation :** When an epoch broadcast arrives, each machine checks about any remote computations running locally, if running, tries its best to locate their owners. Computations are killed in case owners cannot be located.

4. **Expiration :** In this solution, RPC is supposed to finish the job in specified time quantum T. If not finishes in this time, another time quantum is requested which is a nuisance. In contrast, if after a crash the client waits a time T prior to rebooting, all orphans are certain to be gone.

5.8 Reliable Group Communication

5.8.1 Basic Reliable-Multicasting Schemes

- Reliable multicast services promise that messages are delivered to all members in a process group. Most of the transport layers only offer point-to-point connection between two processes. They seldom offer reliable communication to a collection of processes. So each process has to set up reliable point-to-point connection with other process. If number of processes is large then achieving reliability through point-to-point connection is difficult. If number of processes is small then it is simple to achieve reliability through point-to-point connection.
- All the processes in group should receive incoming message for that group. In other word, message sent to particular process group should be delivered to the all members of that group. There are some situations that need to be considered. These are: if process joins the group during communication, whether this newly joined process should receive the message or not and if a sending process crashes during communication.
- It is necessary to differentiate between reliable communication in the presence of faulty processes and reliable communication when processes functions correctly. In case of reliable communication in the presence of faulty processes, multicasting is said to be reliable if it is guaranteed that, all non-faulty processes in group receives message. It is necessary to arrive at an agreement on what the group actually looks like before a message can be delivered, besides various ordering constraints.
- It is simple to handle the case when already agreement exists on who is a member of the group and who is not. Especially, if it is assumed that, processes do not fail, and processes do not join or leave the group during communication then reliable multicasting simply means that every message should be delivered to each current group member.
- Sometimes, it is requirement of receiving the messages by all group members in same order. In simple case, group members can receive messages in any order and it is easy to implement if number of receivers are small.
- Consider underlying communication system offers only unreliable multicasting. In this case, some messages from sender may be lost or may not be delivered to all the receivers (group members). In this case, a sequence number is assigned by sender to each message it multicasts. Suppose messages are received in the order they are sent. Sender maintains buffer to store already sent messages till acknowledgment (ACK) for that message is received. It is easy for a receiver to detect it is missing a message. For missed message by receiver, sender receives negative ACK (NACK). Sender then retransmits the message. In other solution, sender can retransmits message when it has not received all acknowledgments within a certain time. Piggybacking can be used to minimize number of messages.

5.8.2 Scalability in Reliable Multicasting

- Above reliable multicast scheme cannot support large numbers of receivers. Sender has to receive n ACKs from n receivers. If large numbers of receivers are there then the sender may be swamped with such feedback messages called as feedback implosion. Also, the receivers are spread across a wide-area network.
- Instead of sending ACKs by receivers, only NACKs can be sent to sender. Hence, better scaling can be achieved by reducing number of feedback messages. In this case, there will be less chances of feedback implosion. In this case also sender has to keep messages in history buffer to retransmit the message which is not delivered. Sender has to remove messages from history buffer after some time has elapsed in order to prevent it from overflowing. This may cause problem if NACK arrives and same message is already removed from the buffer.

Nonhierarchical Feedback Control

- It is necessary to reduce number of feedback messages while offering the scalable solution to reliable multicasting. A more accepted model used in case of several wide-area applications is **feedback suppression.** This approach underlies the Scalable Reliable Multicasting (SRM) protocol. In SRM, receiver only sends NACK for missed message. For successfully delivered messages, receiver does not report acknowledgements (ACKs) to sender. Only NACKs are returned as feedback. Whenever a receiver notices that it missed a message, it multicasts its feedback to the rest of the group.
- In this way, all the members of the group know that message m is missed by this receiver. If k number of receivers has already missed this message m, then each of k members has to send NACK to sender so that m can be retransmitted. On the other hand, suppose retransmissions are always multicast to the entire group, only a single request for retransmission can be sent to sender.
- For this reason, a receiver R within group who has missed the message m sends the request for retransmission after some random time has elapsed. If, in the meantime, another request for retransmission for m reaches R, R will hold back its own feedback (NACK), knowing that m will be retransmitted soon. In this manner, only a single feedback message (NACK) will reach sender S, which in turn next retransmits m.
- Feedback suppression has been used as the fundamental approach for a number of collaborative Internet applications. Although, this mechanism has shown to scale reasonably well, it also introduces a number of serious problems. It requires accurate scheduling of feedback messages at each receiver so that single request for retransmission will be returned to the sender. It may happen that, many receivers will still return their feedback simultaneously. Setting timers for that reason in a group of processes that is spread across a wide-area network is difficult.
- Other problem in this mechanism is that, a retransmitted message also gets delivered to the group members who have already successfully received this message. Unnecessarily, these processes have to again process this message. A solution to this problem is to let these processes which have missed the message m, join another multicast group to receive message m. This solution requires efficient group management which is practically not possible in wide area network. A better approach is therefore to let receivers that tend to miss the same messages team up and share the same multicast channel for feedback messages and retransmissions.

- The scalability of SRM can be improved, if receivers of message m assist in local recovery. In this solution, a receiver of message m receives a request for retransmission, it can make a decision to multicast m even before the retransmission request reaches the original sender.

Hierarchical Feedback Control

- For large groups of receivers, hierarchical feedback control is better approach.

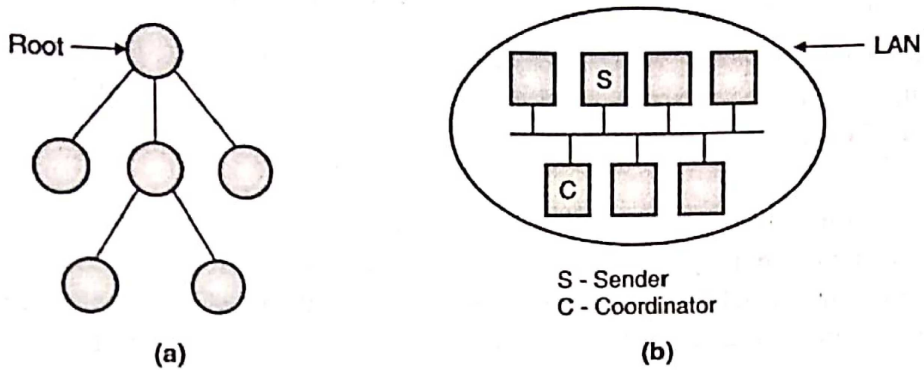

Fig. 5.8.1

- Consider only a single sender multicast messages to a very large group of receivers. The group of receivers is partitioned into a number of subgroups. These subgroups then organized into a tree. The root of the tree is formed by the subgroup in which sender is present. Any reliable multicasting scheme appropriate for small groups can be used within any subgroup. The tree is shown in Fig. 5.8.1 (a). Each node in the tree represents subgroup which has organization as shown in Fig. 5.8.1 (b).
- Each subgroup appoints local coordinator C which handles retransmission requests of the members (receivers) of the same subgroup. The local coordinator also maintains its own history buffer. If the coordinator itself has missed a message m, it requests the coordinator of the parent subgroup to retransmit message m. In a scheme based on ACKs, a local coordinator sends an ACK to its parent if it has received the message. If a coordinator has received ACKs for message m from all members in its subgroup, as well as from its children, it can remove m from its history buffer.
- The construction of the tree is main difficulty in this approach. In many cases, it is necessary to build tree dynamically. One technique is to use the multicast tree in the underlying network, if there is one. In principle, the approach is then to improve each multicast router in the network layer so that it can proceed as a local coordinator in the manner just described. Practically, such adaptations to existing computer networks are difficult to carry out. As a result, application-level multicasting solutions are popular.
- Finally, designing reliable multicast schemes that are scalable to a large number of receivers spread across a wide-area network is a complex problem. No single best solution is available, and each solution introduces new problems.

5.8.3 Atomic Multicast

- Now we consider reliable multicasting in the presence of process failures. In particular, distributed system should guarantee that a message is delivered to either all processes or to none at all. It is often needed in many situations. Also, it is also necessary requirement that that all messages are delivered in the same order to all processes called as the atomic multicast problem.

- Consider a replicated database constructed as an application on top of a distributed system. Suppose distributed system offers reliable multicasting facilities. Hence, it permits construction of process groups to which messages can be reliably sent. The replicated database also forms process group. There is one process for each replica of the database. All these processes belong to one group.
- The update operations are multicast to all replica and operations are carried out locally. A series of update operations is to be carried out on database. If one of the replica crashes during one update operation then this update is lost for this replica. Whereas, same update is correctly carried out at the other replicas.
- The crashed replica can recover to the same state it had before the crash when it recovers. Yet, it has missed a number of updates which are carried out after its crash. Now, it is necessary that it is brought up to date with the other replicas. This requires knowing precisely the missed operations, and in which order these operations are to be carried out
- If atomic multicasting is supported by underlying distributed system, then the update operation that was sent to all replicas just before one of them crashed is either carried out at all nonfaulty replicas or by none at all. The update is carried out if the all nonfaulty replicas have agreed that the crashed replica no longer is member of the group. The crashed replica is now forced to join the group once More after it recovers from crash.
- Update operations will be forwarded to this recovered replica only if it is registered as being a member again. In order to join the group, recovered replica's state needs up to date with the rest of the group members. As a result, atomic multicasting assures that nonfaulty processes preserve a consistent view of the database, and forces settlement when a replica recovers and rejoins the group.

Virtual Synchrony

- We assume that each distributed system has communication layer within which messages are sent and received. On each node, messages are first buffered in communication layer and then delivered to the application which runs in higher layer. **Group view** is view on the set of processes in the group which sender had when message m was multicast. Every process in group should have same view about delivery of message m sent by sender. All should agree that, message should be delivered to each member in group view and to no other member.
- Suppose message m is multicast by sender with **group view** G and while multicast is going on, new process joins the group or leaves the group. Because of this group view changes due to change in group membership. This change in group membership message c (joining or leaving group) is multicast to all the members of the group. Now we have two multicast messages in transit: m and c. In this case, guarantee is needed about either m is delivered to all processes in group view G before each one of them delivered message c, or m is not delivered at all.
- When group membership change is the result of crashing of the sender of m, then delivery of m is permitted to fail. In this case, either all the members of the G should know abort of new member or none. This guarantees that, a message multicast to group view is delivered to each nonfaulty process in G. If sender crashes during multicast then message may be delivered to all remaining processes or ignored by each of them. Reliable multicast with this property is said to be **virtually synchronous.**
- Consider processes P_1, P_2, P_3, and P_4 in group view G. Group view G = (P_1, P_2, P_3, P_4). After some messages have been multicast, P_3 crashes. So group view G = (P_1, P_2, P_4). Before crash, P_3 succeeded in multicasting messages to P_2 and P_4 but not to P_1. Virtual synchrony guarantees that, its messages will not be delivered at all. This indicates that, message had never been sent before crash of P_3. Communication then proceeds between remaining members after removing P_3 from group. After P_3 recovers, it can join the group provided its state has been brought up to date.

- Consider a replicated database constructed as an application on top of a distributed system. Suppose distributed system offers reliable multicasting facilities. Hence, it permits construction of process groups to which messages can be reliably sent. The replicated database also forms process group. There is one process for each replica of the database. All these processes belong to one group.
- The update operations are multicast to all replica and operations are carried out locally. A series of update operations is to be carried out on database. If one of the replica crashes during one update operation then this update is lost for this replica. Whereas, same update is correctly carried out at the other replicas.
- The crashed replica can recover to the same state it had before the crash when it recovers. Yet, it has missed a number of updates which are carried out after its crash. Now, it is necessary that it is brought up to date with the other replicas. This requires knowing precisely the missed operations, and in which order these operations are to be carried out
- If atomic multicasting is supported by underlying distributed system, then the update operation that was sent to all replicas just before one of them crashed is either carried out at all nonfaulty replicas or by none at all. The update is carried out if the all nonfaulty replicas have agreed that the crashed replica no longer is member of the group. The crashed replica is now forced to join the group once More after it recovers from crash.
- Update operations will be forwarded to this recovered replica only if it is registered as being a member again. In order to join the group, recovered replica's state needs up to date with the rest of the group members. As a result, atomic multicasting assures that nonfaulty processes preserve a consistent view of the database, and forces settlement when a replica recovers and rejoins the group.

Virtual Synchrony

- We assume that each distributed system has communication layer within which messages are sent and received. On each node, messages are first buffered in communication layer and then delivered to the application which runs in higher layer. **Group view** is view on the set of processes in the group which sender had when message m was multicast. Every process in group should have same view about delivery of message m sent by sender. All should agree that, message should be delivered to each member in group view and to no other member.
- Suppose message m is multicast by sender with **group view** G and while multicast is going on, new process joins the group or leaves the group. Because of this group view changes due to change in group membership. This change in group membership message c (joining or leaving group) is multicast to all the members of the group. Now we have two multicast messages in transit: m and c. In this case, guarantee is needed about either m is delivered to all processes in group view G before each one of them delivered message c, or m is not delivered at all.
- When group membership change is the result of crashing of the sender of m, then delivery of m is permitted to fail. In this case, either all the members of the G should know abort of new member or none. This guarantees that, a message multicast to group view is delivered to each nonfaulty process in G. If sender crashes during multicast then message may be delivered to all remaining processes or ignored by each of them. Reliable multicast with this property is said to be **virtually synchronous.**
- Consider processes P_1, P_2, P_3, and P_4 in group view G. Group view G = (P_1, P_2, P_3, P_4). After some messages have been multicast, P_3 crashes. So group view G = (P_1, P_2, P_4). Before crash, P_3 succeeded in multicasting messages to P_2 and P_4 but not to P_1. Virtual synchrony guarantees that, its messages will not be delivered at all. This indicates that, message had never been sent before crash of P_3. Communication then proceeds between remaining members after removing P_3 from group. After P_3 recovers, it can join the group provided its state has been brought up to date.

Message Ordering

- The multicasts are classified with following four orderings.
  1. Unordered multicasts
  2. FIFO-ordered multicasts
  3. Causally-ordered multicasts
  4. Totally-ordered multicasts
- An **unordered multicast** which is reliable is a virtually synchronous multicast. In this multicast, guarantees are not assured regarding the order in which received messages are delivered by different processes. Consider the example in which reliable multicasting is offered by a library with **send** and a **receive** primitive. The receive operation blocks the calling process until a message is delivered to it.
- Consider group with three communicating processes P_1, P_2, and P_3. Following are the ordering of events at each process.

Process P_1	Process P_2	Process P_3
sends m_1	receives m_1	receives m_2
sends m_2	receives m_2	receives m_1

- Process P_1 multicasts messages m_1 and m_2 to group members. Assume group view does not change during multicast. In this situation suppose communication layer at P_1 receives first m_1 and then m_2. In contrast, communication layer at P_2 suppose receives first m_2 and then m_1. As there is no ordering constraint, messages may be delivered in the order they were received.
- In the second case of reliable **FIFO-ordered multicasts**, the incoming messages from same process are delivered by the communication layer in the same order as they have been sent. Consider communication between following four processes. In FIFO ordering, as shown below, message m_1 will be always delivered before m_2. In the same way, m_3 will be always delivered before m_4. This rule is followed by all processes in the group. If communication layer at particular process receives m_2 first and then m_1, it should not deliver m_2 till it has received and delivered m_1. On the other hand, messages received from different processes are delivered by communication layer in the order they have received.

Process P_1	Process P_2	Process P_3	Process P_4
sends m_1	receives m_1	receives m_1	sends m_3
sends m_2	receives m_2	receives m_3	sends m_4
	receives m_3	receives m_2	
	receives m_4	receives m_4	

- **In reliable causally ordered multicast**, potentially causally related messages are delivered by communication layer by considering causality between messages. If message m_1 causally precedes message m_2 then at receiver side, communication layer will deliver m_1 first and then m_2. These messages can be from same process or from different processes.

- **Total-order multicast** imposes additional constraint on order of delivery of messages. It says that, message delivery may be unordered, FIFO or causally ordered but should be delivered to all the processes in group in same order. Virtually synchronous reliable multicasting which offers totally ordered delivery messages called as **atomic multicasting.**

5.9 Recovery

5.9.1 Introduction

- It is necessary to recover from the failure of processes. Process should always recover to the correct state. In error recovery, it is necessary to replace erroneous state to error-free state. In **backward recovery,** system is brought from current erroneous state to previous state. For this purpose, a system state is recorded and restored after some interval. When current state of the system is recorded, a **checkpoint** is said to be made.
- In **forward recovery,** an attempt is made to bring system from current erroneous state to new current state from which it can carry on its execution. Forward recovery is possible if type current error occulted is known.
- In distributed system, backward recovery techniques are widely applied as mechanism to recover from failures. It is generally applicable to all systems and processes and can be integrated in middleware as general-purpose service. The disadvantage of this technique is that, it degrades performance as it is costly to restore previous state.
- As backward recovery is general technique applicable to all systems and applications, guarantee cannot be given about occurrence of failure after recovery from the same one. Hence, applications support is needed for recovery which cannot give full-fledged failure transparency. Moreover, rolling back to state such as money is already transferred to other account is practically impossible.
- As **checkpointing** is costlier due to restoring to previous state, many distributed system implements it with **message logging.** In this case, after checkpoint has been taken, process maintains logs of messages prior to sending them off. This is called **sender-based logging.** In other scheme, **receiver-based logging** can be carried out where receiving process first log an incoming message and then deliver it to the application it is currently executing.
- Checkpointing restores processes to previous state. From this state, it may behave differently than before failure had occurred. In this case, messages can be delivered in different order leading to unexpected reactions by receivers. Message logging now helps to deliver messages in the expected order as actual replaying of the events since last taken checkpoint.

5.9.2 Stable Storage

- The information needed to recover to the previous state needed to be stored safely so that it survives process crashes, site failures and also storage media failures. In distributed system, stable storage is important for recovery.
- The storage is of three types: RAM which is volatile, disk which survives CPU failure but can be lost in case of failure of disk heads. Finally, stable storage which is designed to survive from all types of failures except major calamities.
- Stable storage is implemented with pair of ordinary disks. Each block on second drive is an exact copy of the corresponding block on first drive. Update of block first takes place in first drive and once updated, it is verified. Then same block on second drive is done.

- Suppose first drive is updated and then system crashes. Suppose, this crash is occurred before second drive is updated. In this situation, recovery is carried out by comparing both drives block for block. When two corresponding blocks differ then block on first drive is considered as correct one and it is then copied to first drive to another drive. Now both the drive will be identical.
- Due dust particles and general wear and tear, a valid block may give checksum error. In this case, block is recovered by using the corresponding block on other (second) disk. Stable storage is most beneficial for applications that need a high degree of fault tolerance.

5.9.3 Checkpointing

- In backward error recovery, it is required to save system state regularly on stable storage. It also needed to record consistent global state called as distributed snapshot. In distributed snapshot, if one process has recorded the receipt of message then there should be other process who has recorded the sending of the message.
- In backward error recovery schemes, each process saves its state on regular basis to its local stable storage. A consistent global state from these local states is build to recover from process or system failure. A recovery line signifies the most recent consistent collection of checkpoints. The most recent distributed snapshot is recovery line.

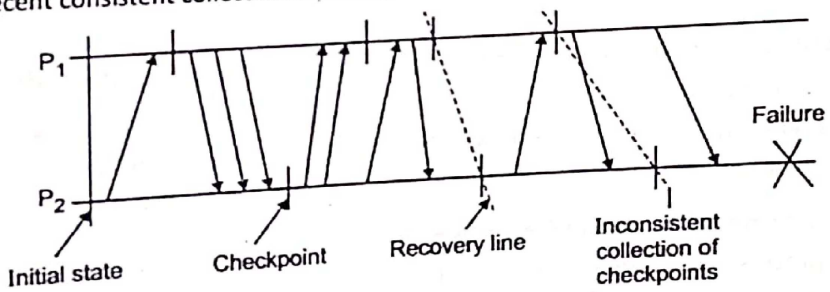

Fig. 5.9.1 : A Recovery Line

Independent Checkpointing

- If each process records its local state from time to time in an uncoordinated way then it is not easy to find recovery line. In order to find out this recovery line, each process is rolled back to its most recently recorded state. If these local states together do not form distributed snapshot then further rolling back is necessary to carry out. This process of cascaded roll back may lead to **domino effect.**

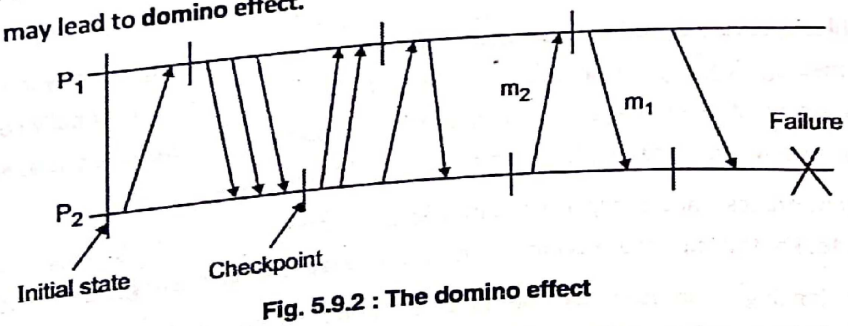

Fig. 5.9.2 : The domino effect

- After crash of process P_2, it is required to restore its state to the most recently recorded checkpoint. As a result, process P_1 will also needs to be rolled back.

- The restored state by P_2 shows receipt of m_1 but sender of the same message is still not recorded in snapshot. Again, it is require P_2 to roll back to previous state. However, the next state to which P_2 is rolled back has not recorded sending of message m_2 although; P_1 has recorded receipt of it. As a result, P_1 again needs to be rolled back.
- This process of rolling back without coordinating with other process is called independent checkpointing The other solution is to globally coordinate checkpointing which requires global synchronization. This may introduce performance problems. Another drawback of independent checkpointing is that, each local storage needs to be cleaned up time to time.

Coordinated Checkpointing

- In coordinated checkpointing, all processes synchronize to write their state to local stable storage in cooperative manner. As a result of which, the saved state is automatically globally consistent. In this way, domino effect is avoided here.
- A two-phase blocking protocol is used to coordinate checkpointing. A coordinator first multicast ***CHECKPOINT_REQUEST*** to all the processes. Receiving process of this message then takes local checkpoint, queues any successive message handed to it by the application it is running, and acknowledges to the coordinator that it is has taken a checkpoint.
- When coordinator receives acknowledgement from all the processes, it multicast ***CHECKPOINT_DONE*** message to permit blocked processes to carry on their work. As incoming messages are not registered as part of checkpoint, this approach guarantees globally consistent state. All the outgoing messages are queued locally till receipt of ***CHECKPOINT_DONE*** message.
- This approach can be improved if checkpoint request is multicast to those processes only that depend on the recovery of the coordinator. A process is dependent on the coordinator if it has received a message that is directly or indirectly causally related to a message that the coordinator had sent since the last checkpoint. This leads to the concept of an incremental snapshot.
- In an incremental snapshot, the coordinator multicasts a checkpoint request only to those processes it had sent a message to since it last took a checkpoint. When a process *P* receives such a request, it forwards the request to all those processes to which *P* itself had sent a message since the last checkpoint, and so on.

5.9.4 Message Logging

- Message logging enables recovery by reducing number of checkpoints. Checkpointing is costly in terms of writing state on stable storage. In message logging, transmission of messages is replayed to achieve globally consistent state. There is no need to write state on stable storage. In this approach, a checkpointed state is taken as starting point and all messages that have been sent since are simply retransmitted and handled accordingly.
- In order to recover from process failure, replaying of messages is carried out in message logging scheme to restore to globally consistent state. For this purpose, it is important to know exactly when messages are to be logged.

The existing message-logging approaches can be easily distinguished on the basis of how they handle orphan processes. An orphan process is a process that survives the crash of another process, but whose state is inconsistent with the crashed process after its recovery.

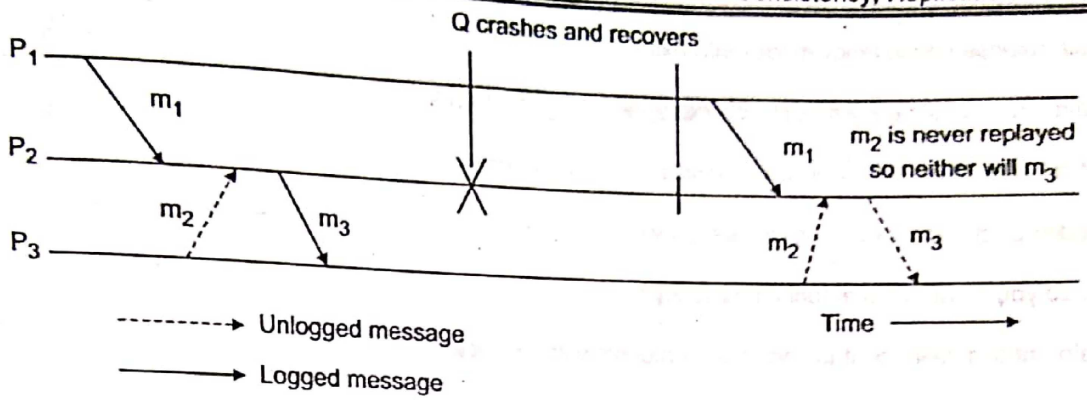

Fig. 5.9.3 : Incorrect replay of messages after recovery, leading to an orphan process

- Consider three processes P_1, P_2 and P_3 as shown in Fig. 5.9.3. Process P_2 receives messages m_1 and m_3 from P_1 and P_3 respectively. Process P_2 then sends messages m_3 to P_3. Message m_1 was logged but m_2 was not logged. After crash process P_2 recovers again. In this case, for recovery of P_2 only logged message m_1 will be replayed. As m_2 was no logged, its transmission will not be replayed and hence, transmission of m_3 may not take place.
- However, the state after the recovery of P_2 is inconsistent with that before its recovery. Especially, P_3 keeps a message m_3 which was sent before the crash, but whose receipt and delivery do not happen when replaying what had taken place before the crash. Such inconsistencies should clearly be avoided

5.9.5 Recovery-Oriented Computing

- The other way of carrying out recovery is basically to start over again. It may be much cheaper to optimize for recovery, then it is aiming for systems that are free from failures for a long time. This approach is called as recovery-oriented computing. One approach is simply reboot the system.
- For clever reboot only a part of the system, it is essential to localize the fault correctly. Just then, rebooting simply involves deleting all instances of the identified components together with the threads operating on them, and (often) to just restart the associated requests. Practically, rebooting as a recovery technique requires few or no dependency between system components.
- In other approach of recovery-oriented computing, checkpointing and recovery techniques are applied, but execution is carried out in a changed environment. The basic idea is that, if programs are allocated more buffer space, changing order of message delivery etc then many failures can be avoided.

Review Questions

Q. 1 What is replication? Write the advantages of replication.

Q. 2 Explain replication as scaling technique.

Q. 3 What is continuous consistency? Explain.

Q. 4 Explain sequential consistency model with example.

Q. 5 Explain causal consistency model with example.

Q. 6 Explain FIFO consistency model with example.

Q. 7 Explain release consistency model with example.

Q. 8 Explain entry consistency model with example.

Q. 9 What is basic difference between consistency and coherence?

Q. 10 What do you mean by eventual consistency?

Q. 11 What do you mean by eventual consistency?

Q. 12 Explain ant two client-centric consistency models with example.

Q. 13 Explain ant two data-centric consistency models with example.

Q. 14 Explain read your writes and write follows reads client centric consistency.

Q. 15 Explain permanent replicas, server-initiated replicas and client-initiated replicas.

Q. 16 Explain different techniques of propagation of updates.

Q. 17 Explain different failure models.

Q. 18 Explain Byzantine Agreement Problem with example.

Q. 19 Explain different failures that can occur in RPC and their solutions.

Q. 20 Explain different approaches for scalability in reliable multicasting.

Q. 21 What is mean by atomic multicast? Explain.

Q. 22 What is mean by atomic multicast? Explain.

Q. 23 What is virtually synchronous reliable multicast? Explain.

Q. 24 Explain different types of ordering of the messages in group communication.

Q. 25 What is stable storage? How it plays role in recovery of distributed system?

Q. 26 What is checkpointing?

Q. 27 Explain independent and coordinated checkpointing approaches.

Q. 28 What is message logging? What are the advantages of message logging?

Q. 29 Explain recovery-oriented computing.

□□□