CHORD

- Chord is a protocol proposed by Stoica that associates mapping of key/ value pairs in distributed environment using a hash table.
- A **distributed hash table** (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table: (key, value) pairs are storedin a DHT, and any participating node can retrieve the value associated with a givenkey.
- Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption.
- This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

Properties of DHT:

- **Autonomy and decentralization:** the nodes collectively form the system without any central coordination.
- **Fault tolerance:** the system should be reliable even with nodes continuously joining, leaving, and failing.
- **Scalability:** the system should function efficiently even with thousands or millions of nodes.

CHORD

> ***Chord uses a flat key space to associate the mapping between network nodes and data objects/files/values.***

- Chord is a protocol and algorithm for a peer-to-peer distributed hash table.
- A distributed hash table stores key-value pairs by assigning keys to different computers (known as “nodes”); a node will store the values for all the keys for which it is responsible.
- Chord specifies how keys are assigned to nodes, and how a node can discover the

- value for a given key by first locating the node responsible for that key.
- The node address as well as the data object/file/value is mapped to a logical identifier in the common key space using a consistent hash function.
- Chord supports a single operation, lookup(x), that maps a given key x to a network node.
- Chord stores a file/object/value at the node to which the file/object/value's key maps.
- The steps involved in lookup are:
    1. Map the object/file/value to its key in the common address space.
    2. Map the key to the node in its native address space using lookup. The design oflookup is the main challenge.

Basic Querying

- The Chord protocol is used to query a key from a client i.e. to find successor(k).
- The basic approach is to pass the query to a node's successor, if it cannot find the key locally.

- This will lead to a O(N) query time where N is the number of machines in the ring.
- To avoid the linear search above, Chord implements a faster search method by requiring each node to keep a **finger table** containing up to m entries, where m is the number of bits in the hash key.
- In Chord, a node's IP address is hashed to an m-bit identifier that serves as thenode identifier in the common key (identifier) space.
- The file/data key is hashed to an m-bit identifier that serves as the key identifier.
- The value of m is sufficiently large so that the probability of collisions during the hash is negligible.
- The Chord overlay uses a logical ring of size 2^m.
- The identifier space is ordered on the logical ring modulo 2^m.

- A key k gets assigned to the first node such that its node identifier equals or follows the key identifier of k in the common identifier space.

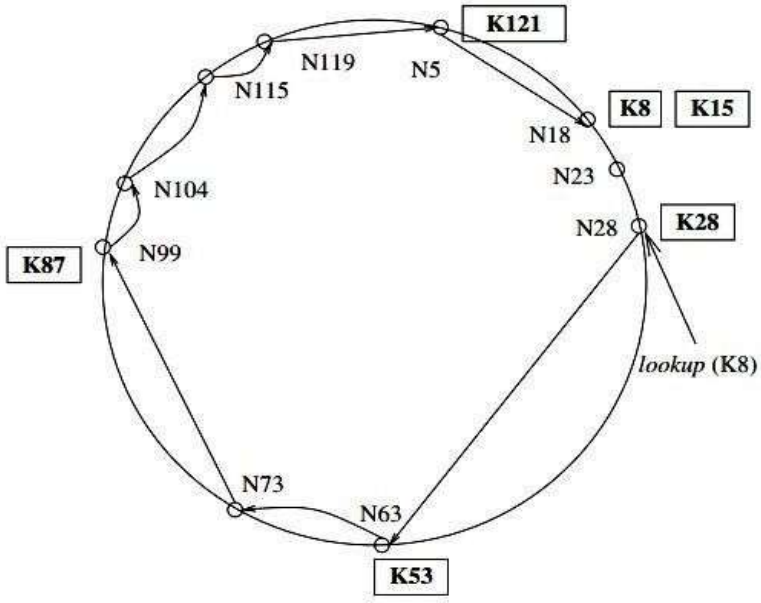

Fig : Chord ring with m=7

- The node is the successor of k denoted as succ(k).
- N denoted the nodes and K denoted the keys stored by the nodes.

Succ(8)=18

Succ(15)=18

Succ(28)=28

Succ(53)=63

Succ(87)=99

Succ(121)=5

Simple lookup

- Each node tracks its successor on the ring, in the variable successor; a query for key x is forwarded to the successors of nodes until it reaches the first node such that that node's identifier y is greater than the key x, modulo 2^m.
- The result, which includes the IP address of the node with key y, is returned to the querying node along the reverse of the path that was followed by the query.
- This mechanism requires O(1) local space but O(n) hops, where n is the number of nodes in the P2P network.

(variables)

Integer: successor ← initial value;

(1) i .Locate_Successor(key), where key≠i:

(1a) if key ∈ (i, successor] then

(1b) return(successor)

(1c) else return (successor.Locate_Successor(key)).

Fig : Simple object lookup algorithm

Scalable Lookup

- A scalable look up algorithm that uses O(log n) message hops at the cost of O(m) space in the local routing tables, uses the following idea.
- Each node i maintains a routing table, called the finger table, with at most O(log n) distinct entries, such that the x^{th} entry ($1 \leq x \leq m$) is the node identifier of the node succ($i + 2^{x-1}$).
- This is the first node whose key is greater than the key of node i by at least $2x-1$ mod 2^m.
- The size of the finger table is bounded by m entries.
- Due to the logarithmic structure, the finger table has more information about nodes closer ahead of it in the Chord overlay, than about nodes further away.

(variables)

Integer: successor ← initial value;

Integer: predecessor ← initial value;

Integer finger[1…..m];

(1) i .Locate_Successor(key), where key≠i:

(1a) if key ∈ (i, successor] then

(1b) return(successor)

(1c) else

(1d) j ← Closest_Preceding_Node(key);

(1e) return(j.Locate_Successor(key)).

(2) i.Closest_Preceding_Node(key), where key≠i:

(2a) for count = m down to 1 do

(2b) if finger[count] ∈ (i, key) then

(2c) break( );

(2d) return (finger[count]).

Fig: A scalable object location algorithm

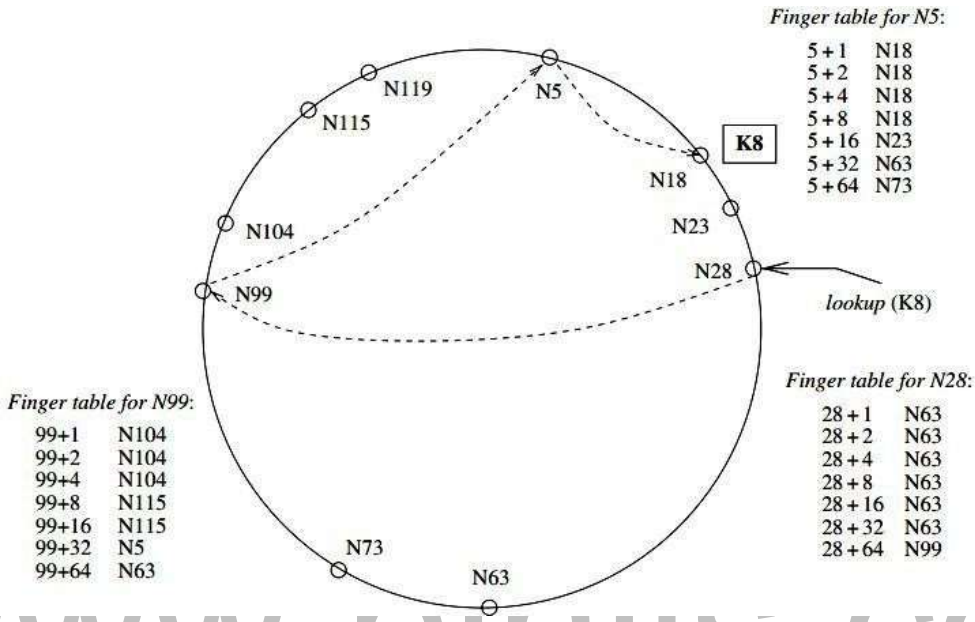

Fig : Query lookup using the logarithmically-structured finger tables

- For a query on key key at node i, if key lies between i and its successor, the key would reside at the successor and the successor's address is returned.
- If key lies beyond the successor, then node i searches through the m entries in its finger table to identify the node j such that j most immediately precedes key, among all the entries in the finger table.
- As j is the closest known node that precedes key, j is most likely to have the most information on locating key, i.e., locating the immediate successor node to which key has been mapped.

Managing Churn

The behavior of the protocol during node joins, failures and departures is discussed here.

i) Node Join

- To create a new ring, a node i executes Create_New_Ring which creates a ringwith the singleton node.
- To join a ring that contains some node j, node i invokes Join_Ring(j).
- Node j locates i's successor on the logical ring and informs i of its successor

- Before i can participate in the P2P exchanges, several actions need to happen: i's successor needs to update its predecessor entry to i, i's predecessor needs to revise its successor field to i, i needs to identify its predecessor, the finger table at i needs to be built, and the finger tables of all nodes need to be updated to account for i's presence.
- This is achieved by procedures Stabilize( ), Fix_Fingers(), and Check_Predecessor() that are periodically invoked by each node.
- A recent joiner node i that has executed Join_Ring() gets integrated into the ring isshown in Fig

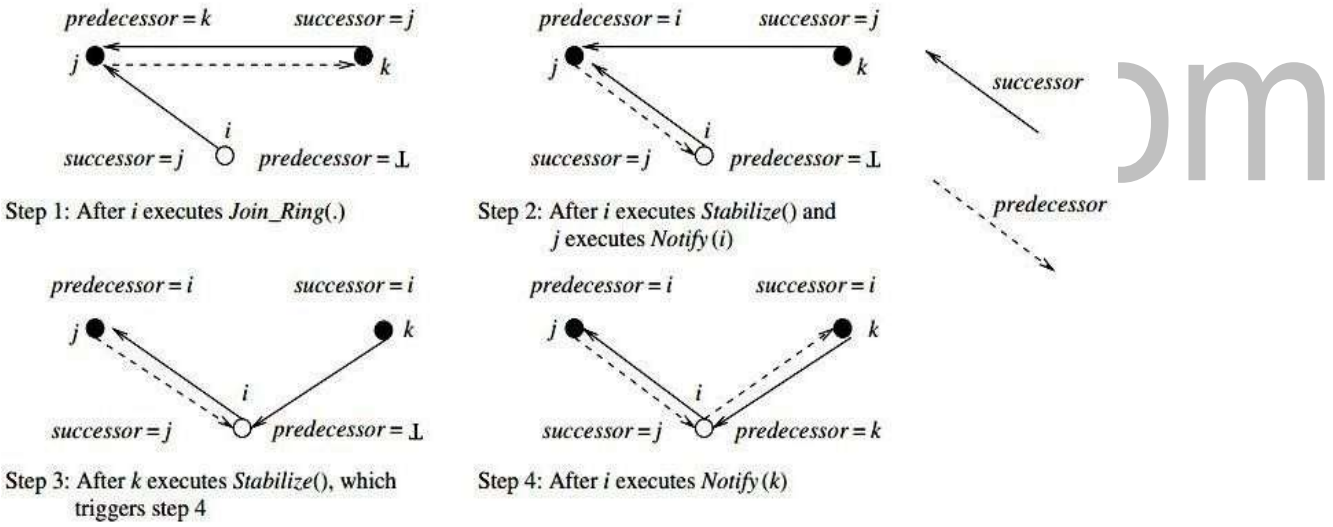

Fig : Steps in the integration of node i in the ring, where j > i > k

The following are the sequence:

1. The configuration after a recent joiner node i has executed Join_Ring().
2. Node i executes Stabilize(), which allows its successor j to adjust j's variable

predecessor to i. Specifically, when node i invokes Stabilize(), it identifies the successor's predecessor k. If k ∈ (i, successor), then I updates its successor to k.

In either case, i notify its successor of itself via successor. Notify(i), so the successor has a chance to adjust its predecessor variable to i.

3. The earlier predecessor k of j (i.e., the predecessor in Step 1) executes Stabilize() and adjusts its successor pointer from j to i.

4. Node i executes Fix_Fingers() to build its finger table, and other nodes also execute the procedure to update their finger tables if necessary.

- Once all the successor variables and finger tables have stabilized, a call by anynode to Locate_Successor() will reflect the new joiner i.
- Until then, a call to Locate_ Successor() may result in the Locate_Successor() call performing a conservative scan.
- The loop in Closest_Preceding_Node that scans the finger table will result in a search traversal using smaller hops rather than truly logarithmic hops, resultingin some
- It can be shown that for any set of concurrent join operations, at some point after the last join operation completes, all the pointers and finger tables will becorrect. This ensures the correctness.

- But the transit period can have the following outcomes:

1. The finger tables used in a search are up to date and the correct successor of the key is sought in O(log n) hops.
2. The finger tables are not up to date but the successor pointers are correct. The sought key will be located but may take more steps as the full advantage of a logarithmic search space pruning cannot be used.
3. If the successor pointers are incorrect, or the key transfer to the new joiners in procedure Notify has not completed, the search may fail. This is during a transient duration, and the source has the choice of reissuing the query.

ii) Node failures and departures

- When a node j fails abruptly, its successor i on the ring will discover the failure when the successor i executes Check_Predecessor() periodically.
- Process i gets a chance to update its predecessor field when another node k causes i
  to execute Notify(k). But that can happen only if k's successor variable is i.
- This requires the predecessor of the failed node to recognize that its successor hasfailed, and get a new functioning successor.
- In fact, the successor pointers are required for object search; the predecessor variables are required only to accommodate new joiners.
- A solution such as introducing a Check_Successor() procedure analogous to Check_Predecessor procedure will not solve the problem because it does not helpto identify the functional successor.
- The Chord protocol proposes that, rather than maintain a single successor, each node maintains a list of successors, which are the node's first successors.

- If the first successor does not respond, the node can try the next successor from thelist, and so on. Only the simultaneous failure of all the successors can then causethe protocol to fail.
- The provision for a successor list at each node provides a natural mechanism for the application to manage replicated objects.
- The replicas get placed at the node corresponding to the object key, as well as at the nodes in the successor list of that node.
- As Chord is able to update its successor list as the successor list changes, Chord can also interface with the application to let it track the locations of the replicas.
- A voluntary departure from the ring can be treated as a failure. However, a failed node causes all the data (keys) stored at that node to be lost until

corrective actionis taken.

- When a node departs voluntarily, it should first transfer all the keys it is responsible for to its successor.
- The departing node should also inform its successor and predecessor.
- This will enable the successor to update its predecessor to the predecessor of the departing node.
- The predecessor will also be able to update its successor list by deleting the departing node and adding the last successor of the departing node's successor list to its own successor list.

Complexity

- For a Chord network with n nodes, each node is responsible for at most (1 +) /n keys.
- The search for a successor in Locate_Successor in a Chord network with n nodes requires time complexity O(log n) with high probability.
- The size of the finger table is log(n) ≤ m
- The average lookup time is 1/2 log(n)

CONTENT ADDRESSABLE NETWORKS (CAN)

> ***A content-addressable network (CAN) is scalable indexing mechanism that maps objects to their locations in the network.***

- The real motivation behind CAN is the existing networks are not scalable.
- CAN support basic hash table operations on key-value pairs (K,V): insert, search,delete
- CAN is composed of individual nodes and each node stores a chunk (zone) of the hash table
- A hash table is formed as a subset of the (K,V) pairs in the table.
- Each node stores state information about neighbor zones.
- The requests (insert, lookup, or delete) for a key are routed by intermediate nodesusing a greedy routing algorithm.
- It do not need any centralized control (completely distributed).
- The small per-node state is independent of the number of nodes in the system (scalable) and also the nodes can route around failures (fault-tolerant).

Properties of CAN

i) Distributed

ii) fault-tolerant

iii) scalable

iv) independent of the naming structure

v) implementable at the application layer

vi) self-organizing and self-healing.

> ***CAN is a logical d-dimensional Cartesian coordinate space organized as a d-torus logical topology, i.e., a virtual overlay d-dimensional mesh with wrap-around.***

- A d-torus logical topology is a virtual overlay d-dimensional mesh with wrap-around.
- The entire space is partitioned dynamically among all the nodes present, so that each node i is assigned a disjoint region r(i) of the space.
- As nodes arrive, depart, or fail, the set of participating nodes, as well as the assignment of regions to nodes
- For any object v, its key k(v) is mapped using a deterministic hash function to apoint p in the Cartesian coordinate space.

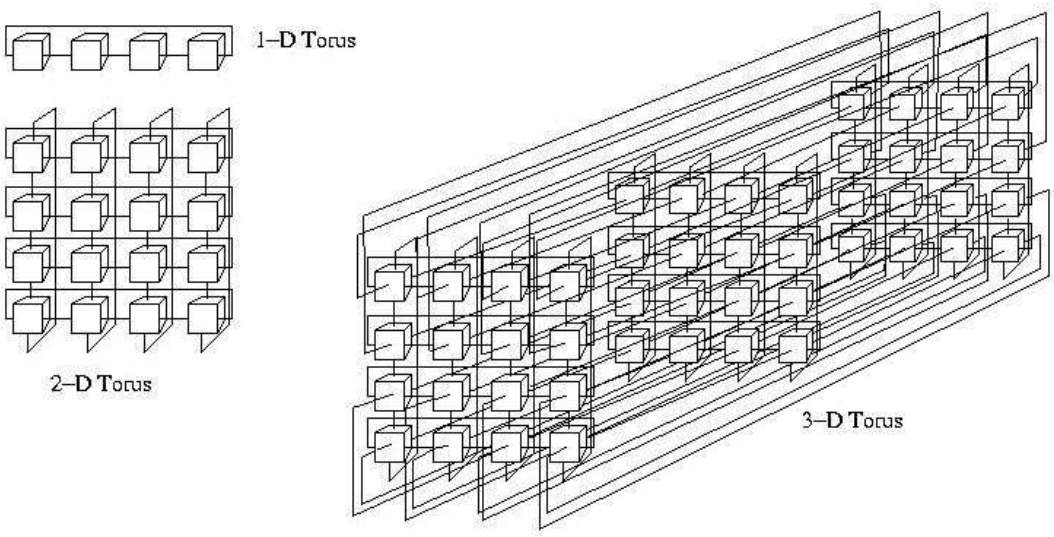

Fig : d-Torus topology

- The (k, v) pair is stored at the node that is presently assigned the region that contains the point p. This means the (k, v) pair is stored at node i if presently the point p corresponding to (k, v) lies in region (r, i).
- To retrieve object v, the same hash function is used to map its key k to the same point p.
- The node that is presently assigned the region that contains p- is accessed to retrieve v.
- The three core components of a CAN design are the following:

1. Setting up the CAN virtual coordinate space, and partitioning it among the nodes as they join the CAN.

2. Routing in the virtual coordinate space to locate the node that is assigned the region containing p.
3. Maintaining the CAN due to node departures and failures.

Initialization of CAN

The following are the steps in CAN initialization:

1. Each CAN is assumed to have a unique DNS name that maps to the IP address of one or a few bootstrap nodes of that CAN.

> ***A bootstrap node is responsible for tracking a partial list of the nodes that itbelieves are currently participating in the CAN.***

2. To join a CAN, the joiner node queries a bootstrap node via a DNS lookup, and the bootstrap node replies with the IP addresses of some randomly chosen nodes that it believes are participating in the CAN.
3. The joiner chooses a random point p in the coordinate space. The joiner sends a request to one of the nodes in the CAN, of which it learnt in step 2, asking to be assigned a region containing p. The recipient of the request routes the request to the owner old_owner(p) of the region containing p, using the CAN routing algorithm.
4. The old_owner(p) node splits its region in half and assigns one half to the joiner. The region splitting is done using an a priori ordering of all the dimensions, so asto decide which dimension to split along. This also helps to methodically merge regions, if necessary. The (k, v) tuples for which the key k now maps to the zone tobe transferred to the joiner, are also transferred to the joiner.
5. The joiner learns the IP addresses of its neighbors from old_owner(p). The neighbors are old_owner(p) and a subset of the neighbors of old_owner(p). The old_owner(p) also updates its set of neighbors. The new joiner as well as old_owner(p) inform their neighbors of the changes to the space allocation, so that they have correct information about their neighborhood and can route correctly. Each node has to send an immediate update of its assigned region, followed by periodic Heartbeat refresh messages, to all

its neighbors.

- When a node joins a CAN, only the neighboring nodes in the coordinate space are required to participate in the joining process.
- The overhead is the order of the number of neighbors, which is O(d) and independent of n, the number of nodes in the CAN.

CAN Routing

- CAN routing uses the straight-line path from the source to the destination in the logical Euclidean space.
- Each node maintains a routing table that tracks its neighbor nodes in the logical coordinate space.
- In d-dimensional space, nodes x and y are neighbors if the coordinate ranges of their regions overlap in d − 1 dimensions, in one dimension.
- All the regions are convex.
- Let the region x be $[[x^1_{min}, x^1_{max}], \ldots [x^a_{min}, x^a_{max}]]$ and the region y be $[[y^1_{min}, y^1_{max}], \ldots [y^d_{min}, y^d_{max}]]$.
- X and y are neighbors if there is some dimension j such that $x^j_{max} = y^i_{min}$ and for all dimensions, $[x^i_{min}, x^i_{max}]]$ and $[y^i_{min}, y^i_{max}]]$ overlap.

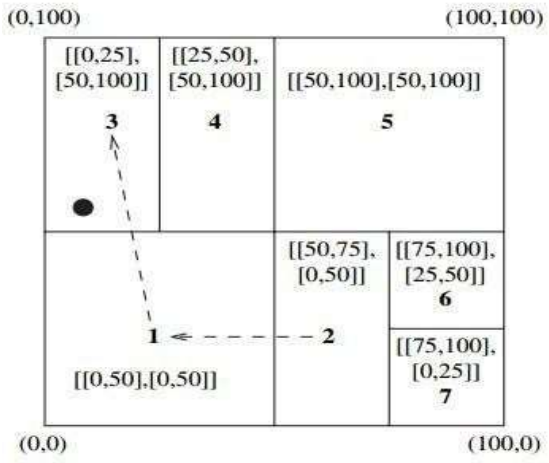

Fig : Two-dimensional CAN space

- The routing table at each node tracks the IP address and the virtual coordinate region of each neighbor.
- To locate value v, its key (k, v) is mapped to a point p- whose coordinates are usedin the message header.
- Knowing the neighbors' region coordinates, each node follows simple greedy routing by forwarding the message to that neighbor having coordinates that are closest to the destination's coordinates.
- To implement greedy routing to a destination node x, the present node routes a message to that neighbor among the neighbors k ∈ Neighbors:
- Assuming equal-sized zones in d-dimensional space, the average number ofneighbors for a node is O(d).
- The average path length is $(d/4)\, n^{1/d}$.
- The implication on scaling is that each node has about the same number of neighbors and needs to maintain about the same amount of state information, irrespective of the total number of nodes participating in the CAN.

- The CAN structure is superior to that of Chord.
- Unlike in Chord, there are typically many paths for any given source-destination pair.
- This greatly helps for fault-tolerance.
- Average path length in CAN scales as $O(n^{1/d})$ as opposed to log n for Chord.

Maintenance in CAN

- When a node voluntarily departs from CAN, it hands over its region and the associated database of (key, value) tuples to one of its neighbors.
- If the node's region can be merged with that of one of its neighbors to form a valid convex region, then such a neighbor is chosen.
- Otherwise the node's region is handed over to the neighbor whose region has the smallest volume or load – the regions are not merged and the neighbor handles both

zones temporarily until a periodic background region reassignment process runs to integrate the regions and prevent further fragmentation.

- AN requires each node to periodically send a HEARTBEAT update message to each neighbor, giving its assigned region coordinates, the list of its neighbors, and their assigned region coordinates.
- When a node dies, the neighbors suspect its death and initiate a TAKEOVER protocol to decide who will take over the crashed node's region.
- Despite this TAKEOVER protocol, the (key, value) tuples in the crashed node's database remain lost until the primary sources of those tuples refresh the tuples.
- Requiring the primary sources to periodically issue such refreshes also serves the dual purpose of updating stale or dirty objects in the CAN.

TAKEOVER protocol

- When a node suspects that a neighbor has died, it starts a timer in proportion to its region's volume.

- On timeout, it sends a TAKEOVER message, with its region volume piggybacked on the message, to all the neighbors of the suspected failed node.
- When a TAKEOVER message is received, a node cancels its bid to take over the failed node's region if the received TAKEOVER message contains a smaller region volume than that of the recipient's region.
- This protocol thus helps in load balancing by choosing the neighbor whose region volume is the smallest, to take over the failed node's region. As all nodes initiate the TAKEOVER protocol, the node taking over also discovers its neighbors andvice versa.
- In the case of multiple concurrent node failures in one vicinity of the Cartesian space, a more complex protocol using an expanding ring search for the TAKEOVER messages can be used.

- A graceful departure as well as a failure can result in a neighbor holding more than one region if its region cannot be merged with that of the departed or failed node.

- To prevent the resulting fragmentation and restore the 1 → 1node to region assignment, there is a background reassignment algorithm that is run periodically.
- Conceptually, consider a binary tree whose root represents the entire space. An internal node represents a region that existed earlier but is now split into regions represented by its children nodes.
- A leaf represents a currently existing region, and overloading the semantics and the notation, also the node that represents that region.
- When a leaf node x fails or departs, there are two cases:
    1. If its sibling node y is also a leaf, then the regions of x and y are merged and assigned to y. The region corresponding to the parent of x and y becomes a leafand it is assigned to nodey.
    2. If the sibling node y is not a leaf, run a depth-first search in the sub tree rootedat y until a pair of sibling leaves (say, z1 and z2) is found. Merge the regions ofz1 and z2, making their parent z a leaf node, assign the merged region to nodez2, and the region of x is assigned to node z1.

- A distributed version of the above depth-first centralized tree traversal can be performed by the neighbors of a departed node.
- The distributed traversal leverages the fact that when a region is split, it is done in accordance to a particular ordering on the dimensions.
- Node i performs its part of the depth first traversal as follows:
    1. Identify the highest ordered dimension dim_a that has the shortest coordinate range [$i^{dima}_{min,}$ i^{dima}_{max}]. Node i's region was last halved along dimension dim_a.
    2. Identify neighbor j such that j is assigned the region that was split off from i'sregion in the last partition along dimension dim_a. Node j's region i's region along dimension dim_a.
    3. If j's region volume equals i's region volume, the two nodes are siblings

and the regions can be combined. This is the terminating case of the depth first treesearch for siblings. Node j is assigned the combined region, and node i takes over the region of the departed node x. This take over by node i is done by returning the recursive search request to the originator node, and communicating i's identity on the replies.

4. Otherwise, j's region volume must be smaller than i's region volume. Node i forwards a recursive depth-first search request to j.

CAN Optimizations

The following are the design techniques to improve the performance of factors:

- **Multiple dimensions:** As the path length is $O(d \cdot n^{1/d})$, increasing the number of dimensions decreases the path length and increases routing fault tolerance at the expense of larger state space per node.
- **Multiple realities:** A coordinate space is termed as a reality. The use of multiple independent realities assigns to each node a different region in each different reality. This implies that in each reality, the same node will store different (k, v) tuples belonging to the region assigned to it in that reality, and will also have a different neighbor set. The data contents (k, v) get replicated in each reality, leading to higher data availability. The multiple copies of each (k, v) tuple, one in each reality, offer a choice – the closest copy can be accessed. Routing fault tolerance improves because each reality offers a set of different paths to the same(k, v) tuple. All these contribute to more storage.
- **Delay latency:** The delay latency on each of the candidate logical links can also beused in making the routing decision.
- **Overloading coordinate regions:** Each region can be shared by multiple nodes, up to some upper limit. This reduces path length and path latency. The fault tolerance improves because a region becomes empty only if all the nodes assignedto it depart or fail concurrently. The per-hop latency decreases because a node can select the closest node from the neighboring region to forward a message towards the destination. This demands many of the aspects of the basic CAN protocol need to be reengineered to

accommodate overloading of coordinate regions.

- **Multiple hash functions:** The use of multiple hash functions maps each key to different points in the coordinate space. This replicates each (k, v) pair for each hash function used. The effect is similar to that of using multiple realities.

- **Topologically sensitive overlay:** The CAN overlay has no correlation to the physical proximity or to the IP addresses of domains. Logical neighbors in the overlay may be geographically far apart, and logically distant nodes may be physical neighbors. By constructing an overlay that accounts for physical proximity in determining logical neighbors, the average query latency can be significantly reduced.

CAN Complexity

- The time overhead for a new joiner is O(d) for updating the new neighbors in the CAN, and O(d/4 log(n)) for routing to the appropriate location in the coordinate space.

- The time overhead and the overhead in terms of the number of messages for a node departure is $O(d^2)$, because the TAKEOVER protocol uses a message exchange between each pair of neighbors of the departed node.

DATA INDEXING AND OVERLAYS

Data stored in distributed systems are located through indexing mechanisms. There are three types of indexing:

1. **Centralized:** This indexing entails the use of one or a few central servers to store references or indexes to the data on many peers. The DNS lookup as well as the lookup by some early P2P networks such as Napster used a central directory lookup.

2. **Local:** This indexes to the objects at various peers being scattered across other peers throughout the P2P network. To access the indexes, a structure is used in the P2P overlay to access the indexes. Distributed indexing is the most challenging of

   the indexing schemes, and many novel mechanisms have been proposed, most notably the distributed hash table (DHT). Various DHT schemes differ in the hash mapping, search algorithms, diameter for lookup, search diameter, fault-tolerance,and resilience to churn.

3. **Distributed**: This requires each peer to index only the local data objects and remote objects need to be searched for. This form of indexing is typically used inunstructured overlays in conjunction with flooding search or random walk search.

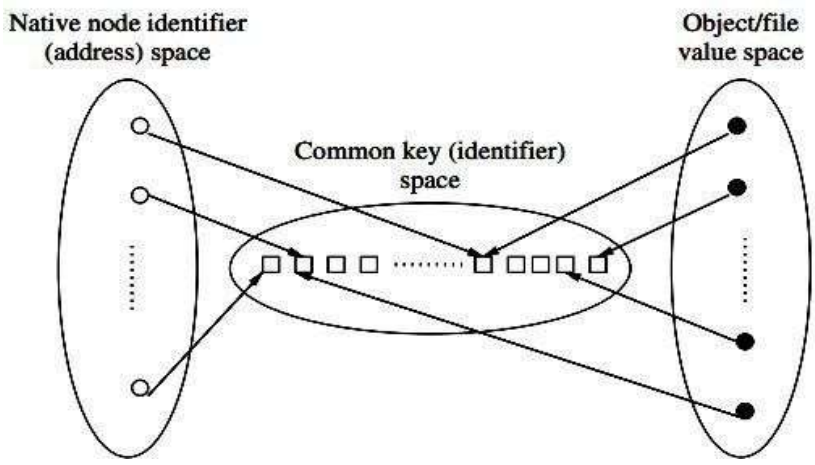

Fig: Mapping between address space and object space

Another criterion for classifying indexing mechanism are based on their semantic structure: semantic indexing and semantic free indexing.

Semantic indexing: A semantic index is human readable. A semantic index

mechanism supports keyword searches, range searches, and approximate searches.

- **Semantic free indexing:** This is not human readable and corresponds to the index obtained by a hash mechanism. These searches are not supported by semantic free index mechanisms.

Distributed IndexingStructured Overlay

> ***The P2P network topology has a definite structure, and the placement of files or data in this network is highly deterministic according to an algorithmic mapping.***

- Deterministic mapping allows a very fast and deterministic lookup to satisfy queries for the data. These lookup systems use a hash table for the mapping.
- The hash function maps keys to values, along with the regular structure of the overlay. This facilitates fast search for the location of the file.
- An implicit characteristic of such a deterministic mapping of a file to a location isthat the mapping can be based on a single characteristic of the file.
- The main drawback in this mapping is that arbitrary queries cannot be handled directly.
- Another notable limitation is the overhead occurred due to insertions and deletions of files in distributed environment.

Unstructured overlays

- This P2P network topology does not have any particular controlled structure.
- It do not have any control over where files or data is located.
- Each peer typically indexes only its local data objects, hence, it uses **localindexing.**
- Node joins and departures are easy since, the local overlay is simply adjusted.
- File placement is independent of the topology.
- But searching a file may incur high message overhead and high delays.
- The major advantage is that unstructured overlays support complex queriesbecause

the search criteria can be arbitrary.

- The lack of fixed topology paves way for the formation of new topology.
- **Some of the topologies are:**

Power law random graph (PLRG): This is a random graph where the node degrees followthe power law

Normal random graph: This is a normal random graph where the nodes typically have auniform degree.

Differences between structured and unstructured overlay networks

Structured overlay	**Unstructured overlay**
The networks are constructed over a predetermined topology.	There is no specific topology.
The connections are also predetermined.	Random and dynamic connections can be established.
The insertion and deletion of nodes imposeshigh overhead over the network performance.	They offer better resilience to network dynamics. Insertion and deletions of nodesis simpler.
This offers faster response time, better performance and lower diameter.	This has comparatively worse performance, node reachability, response time and no guarantee for the diameter.
They are more scalable.	They lack scalability.
They do not support arbitrary searches.	They support arbitrary searches.
Vulnerable to attacks.	Resilience to attacks.
EG: Chord	EG: Gnutella

DISTRIBUTED SHARED MEMORY

Abstraction and its advantages

Distributed Shared Memory is a resource management component of a distributed operating system that implements the shared memory model in distributed systems, which have no physically shared memory. The shared memory model provides a virtual address space that is shared among all computers in a distributed system.

- It is an abstraction provided to the programmer of a distributed system.
- It gives the impression of a single memory. Programmers access the data across the network using only read and write primitives.
- Programmers do not have to deal with send and receive communication primitives and the ensuing complexity of dealing explicitly with synchronization and consistency in the message passing model.
- A part of each computer's memory is earmarked for shared space, and the remainder is private memory.

- To provide programmers with the illusion of a single shared address space, a memory mapping management layer is required to manage the shared virtual memory space.

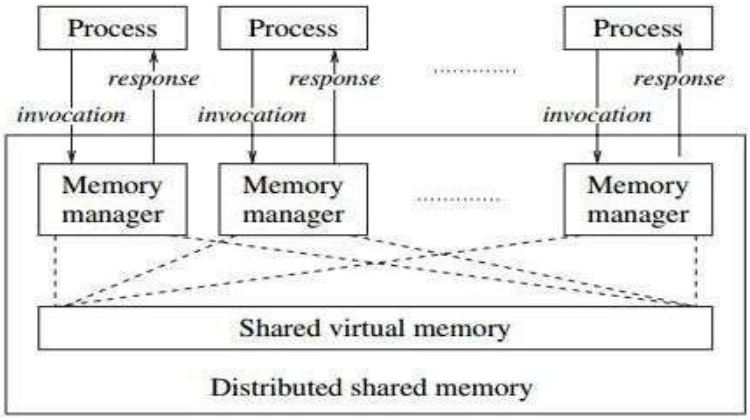

Fig : Abstract view of Distributed Shared MemoryAdvantages of DSM

- Communication across the network is achieved by the read/write abstraction that simplifies the task of programmers.

- A single address space is provided, thereby providing the possibility of avoiding data movement across multiple address spaces, and simplifying passing-by- reference and passing complex data structures containing pointers.
- If a block of data needs to be moved, the system can exploit locality of reference to reduce the communication overhead.
- DSM is economical than using dedicated multiprocessor systems, because it uses simpler software interfaces and off-the-shelf hardware.
- There is no bottleneck presented by a single memory access bus.
- DSM effectively provides a large (virtual) main memory.
- DSM provides portability of programs written using DSM. This portability arises due to a common DSM programming interface, which is independent of the operating system and other low-level system characteristics
- When multiple processors wish to access the same data object, a decision about how tohandle concurrent accesses needs to be made. If concurrent access is permitted by different processors to different replicas, the problem of replica consistency needs to be addressed.

Challenges in implementing replica coherency in DSM systems

1. Programmers are aware of the availability of replica consistency models and from coding their distributed applications according to the semantics of these models.
2. As DSM is implemented as asynchronous message passing, it faces the overhead of asynchronous synchronization.
3. Since the control is given to memory management, the programmers lose the ability to use their own message-passing solutions for accessing shared objects.

Issues in designing a DSM system:

- Determining the semantics to allow for concurrent access to shared objects.

- Determining the best way to implement the semantics of concurrent access to shared data either to use read or write replication.
- Selecting the locations for replication to optimize efficiency from the system's viewpoint.
- Determining the location of remote data that the application needs to access, if full replication is not used.
- Reducing communication delays and the number of messages that are involved under the covers while implementing the semantics of concurrent access to shareddata.

MEMORY CONSISTENCY MODELS

> ***A memory consistency model is a set of rules which specify when a written value by one thread can be read by another thread.***

- These rules are essential to write a correct program**.**
- **Memory coherence** is the ability of the system to execute memory operations correctly.
- The problem of ensuring memory coherence is identifying which of theinterleaving are correct, which of course requires a clear definition of correctness.
- The memory consistency model defines the set of allowable memory access orderings.
- In DSM system, the programmers write their programs keeping in mind the allowable interleaving permitted by that specific memory consistency model.
- A program written for one model may not work correctly on a DSM system that enforces a different model.

- The model can thus be viewed as a contract between the DSM system and the programmer using that system.
- The memory consistency model affects:
  i) System implementation: hardware, OS, languages, compilers
  ii) Programming correctness
  iii) Performance

Strict consistency, atomic consistency, linearizability

- According to Von Neumann architecture/ uniprocessor machine, any Read operation to a location should return the value or variable written by the most recent Write to that location or a variable.
- The system built over the above principle is called strict or atomic consistency model.

- The features of the atomic consistency model area:

i) Common global time axis is implicitly available in a uniprocessor system

ii) The write operation is immediately visible to all processes

Atomic Consistency Model:

i) Any Read to a location is required to return the value written by the most recent Write to that location in accordance with global time reference. For non overlapping operations, with respect to the global time reference, the specification is clear. For overlapping operations the following further specifications are necessary.

ii) All operations appear to be executed atomically and sequentially.

iii) All processors see the same ordering of events, which is equivalent to the global-time occurrence of non-overlapping events.

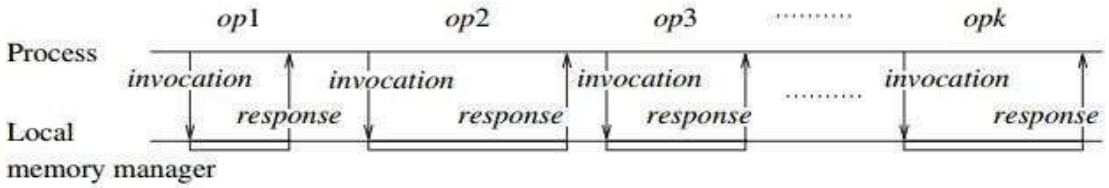

Fig : Invocations and responses in sequential system

The invocation and the response to each invocation can be viewed as being atomic events. An execution sequence in global time is viewed as a sequence Seq of such invocationsand responses. The Seq must satisfy the following conditions:

- **Liveness:** Each invocation must have a corresponding response.
- **Correctness:** The projection of Seq on any processor i, denoted Seqi, must be a sequence of alternating invocations and responses if pipelining is disallowed.

A linearizable execution needs to generate an equivalent global order on the eventsthat is a permutation of Seq, satisfying the semantics of linearizability.

Linearizable property:

A sequence Seq of invocations and responses is linearizable (LIN) if there is a permutation Seq' of adjacent pairs of corresponding (invoc, resp) events satisfying:

1. ***For every variable v, the projection of Seq' on v, denoted Seqv' , is such that every Read (adjacent (invoc, resp) event pair) returns the most recent Write (adjacent, (nvoc, resp) event pair) that immediately preceded it.***
2. ***If the response op1(resp) of operation op1 occurred before the invocation op2(invoc) of operation op2 in Seq, then op1 (adjacent (invoc, resp)event pair) occurs before op2 (adjacent (invoc, resp) event pair) in Seq.***

- Linearizability is a guarantee about single operations on single objects.
- It provides a real- guarantee on the behavior of a set of single operations on a single object.

Linearizability requires that each operation appears to occur atomically at some point between its invocation and completion. This point is called the linearization point.

Implementation of Linearizability

- Implementing linearizability is expensive because a global time scale needs to besimulated.
- As all processors need to agree on a common order, the implementation needs touse total order.
- For simplicity, the algorithm described here assumes full replication of each dataitem at all the processors.
- This demands the total ordering to be combined with a broadcast.
- The memory manager software is placed between the application above it and thetotal order broadcast layer below it.

(shared var)

int: x:

(1) When the memory manager receives a Read or Write from application:

(1a) **total_order_broadcast** the Read or Write request to all processors;(1b) **await** own request that was broadcast;

(1c) **perform** pending response to the application as follows

(1d) **case** Read: return value from local replica;

(1e) **case** Write: write to local replica and return ack to application.

(2) When the memory manager receives a **total_order_broadcast**(Write, x, val) from network;

(2a) **write** val to local replica of x.

(3) When the memory manager receives a **total_order_broadcast**(Read, x) from network;

(3a) **no operation**,

Fig : Implementing Linearizability

The algorithm in Fig. ensures total order broadcast such that all processors follow the same order:

1. For two non-overlapping operations at different processors, the response to the former operation precedes the invocation of the latter in global time.
2. For two overlapping operations, the total order ensures a common view by allprocessors.

Sequential Consistency

Sequential consistency requires that a shared memory multiprocessor appear to be a multiprogramming uniprocessor system to any program running on it.

Sequential consistency requires that:

1. All instructions are executed in order.
2. Every write operation becomes instantaneously visible throughout the system.

The main motivation behind sequential consistency is that the atomic consistency is very difficult to implement since the it is very difficult for a system to synchronize to global clock. Sequential consistency is specified as follows:

- The result of any execution is the same as if all operations of the processorswere executed in some sequential order.
- The operations of each individual processor appear in this sequence in the localprogram order.

Sequential Consistency:

A sequence Seq of invocation and response events is sequentially consistent if there is a permutation Seq' of adjacent pairs of corresponding (invoc, resp) events satisfying:

1. ***For every variable v, the projection of Seq' on v, denoted Seqv' , is such that every Read (adjacent, (invoc,resp) event pair) returns the most recent Write (adjacent, (invoc, resp) event pair) that immediately preceded it.***
2. ***If the response op1(resp) of operation op1 at process Pi occurred before the invocation op2(invoc) of operation op2 by process Pi in Seq, then op1 (adjacent (invoc, resp) event pair) occurs before op2 (adjacent (invoc,resp) event pair) in Seq.***

Implementation of Sequential Consistency

All processors are required to see the same global order, but global time ordering neednot be preserved across processes. So it is sufficient to use total order broadcasts for the Write operations only. In the simplified algorithm, no total order broadcast is required for Read operations, because:

1. all consecutive operations by the same processor are ordered in the same orderbecause pipelining is not used.
2. Read operations by different processors are independent of each other and need tobe ordered only with respect to the Write operations in the execution.

(shared var)

int: x:

(1) When the memory manager receives a Read or Write from application:

(1a) **case** Read: **return** value from local replica;

(1b) **case** Write(x, val): **total_order_broadcast**$_i$(Write(x, val)) to all processors including itself.

(2) When the memory manager at Pi receives a **total_order_broadcasts**$_j$(write, x, val) from network;

(2a) **Write** val to local replica of x;

(2b) **if** i=j **then return** acknowledgement to application.

Fig : Sequential Consistency using Local Read algorithm

Local-read algorithm

- A Read operation completes atomically, whereas a Write operation does not.
- Between the invocation of a Write by Pi (line 1b) and its c knowledgement (lines 2a, 2b), there may be multiple Write operations initiated by other processors that take effect at Pi (line 2a).
- Thus, a Write issued locally has its completion locally delayed. Such an algorithmis acceptable for Read intensive programs.

Local-write algorithm

- This does not delay acknowledgement of Writes.
- For Write intensive programs, it is desirable that a locally issued Write gets acknowledged immediately even though the total order broadcast for the Write, and the actual update for the Write may not go into effect by updating the variableat the same time.
- The algorithm achieves this at the cost of delaying a Read operation by a processor until all previously issued local Write operations by that same processor have locally gone into effect.
- The variable counter is used to track the number of Write operations that have beenlocally initiated but not completed at any time.
- A Read operation completes only if there are no prior locally initiated Write operations that have not written to their variables.
- Else, a Read operation is delayed until after all previously initiated Write operations have written to their local variables, which happens after the total orderbroadcasts associated with the Write have delivered the broadcast message locally.

(shared var)int:x;

(1) When the memory manager at P_i receives a Read(x) from application:

(1a) if counter = 0 then

(1b) return x

(1c) else keep the Read pending

(2) When the memory manager at P_i receives a Write(x, val) from application:

(2a) count ← counter + 1;

(2b) total_order_broadcast$_i$ Write(x, val)

(2c) return acknowledgement to the application.

(3) When the memory manager at P_i receives a total_order_broadcast$_j$ Write(x, val) from network:

(3a) write val to local replica of x;

(3b) if i=j then

(3c) counter ← counter – 1;

(3d) if (counter = 0 and any Reads are pending) then

(3e) perform pending responses for the Reads to the application.

Fig : Sequential Consistency using local write algorithm

Casual Consistency

- The causal consistency model represents a weakening of sequential consistency inthat it makes a distinction between events that are potentially causally related andthose that are not.

According to casual consistency model, only that Writes that are causally related must be seen in that same order by all processors, whereas concurrent Writes may be seen by different processors in different orders.

The causality relation is defined as follows:

- **Local order:** At a processor, the serial order of the events defines the local causalorder.
- **Inter-process order:** A Write operation causally precedes a Read operation issued by another processor if the Read returns a value written by the Write.

- **Transitive closure:** The transitive closure of the above two relations defines the (global) causal order.

Pipelined RAM (PRAM) or Processor Consistency

- In causal consistency, the concurrent writes be seen in a different order on different machines, although causally-related ones must be seen in the same order by all machines.
- PRAM consistency or Pipelined RAM states that Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.
- This is a weaker form of consistency requires only that Write operations issued by the same processor are seen by all other processors in the same order that they were issued, but Write operations issued by different processors may be seen in different orders by different processors.
- The local causality relation, as defined by the local order of Write operations, needs to be seen by other processors. Hence, this form of consistency is termed processor consistency.
- An equivalent name for this consistency model is **pipelined RAM (PRAM**), to capture the behavior that all operations issued by any processor appear to the otherprocessors in a FIFO pipelined sequence.

Slow Memory

- The use of weakly consistent memories or slow memory in distributed shared memory systems to combat unacceptable network delay and to allow such systemsto scale is proposed.
- Slow memory is presented as a memory that allows the effects of writes to propagate slowly through the system, eliminating the need for costly consistency maintenance protocols that limit concurrency.
- Slow memory processes a valuable locality property and supports a reduction

from traditional atomic memory. Thus slow memory is as expressive as atomic memory.

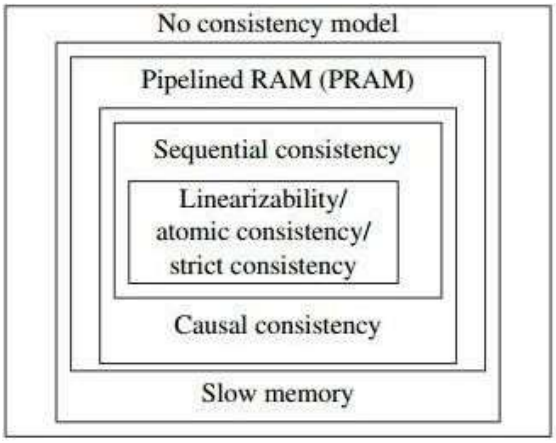

Fig : Hierarchy of memory consistency models

Models based on synchronization instructions

Synchronization instructions are like run-time library. The synchronization statements across the various processors must satisfy the consistency conditions; other program statements between synchronization statements may be executed by the different processors without any conditions.

i) Weak Consistency

The protocol is said to support weak consistency if:

- All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
- All other accesses may be seen in different order on different processes (or nodes,processors).
- The set of both read and write operations in between different synchronization operations is the same in each process.

Drawbacks:

When a synchronization variable is accessed, the memory does not know

whether this is being done because the process is finished writing the shared variables or about to begin reading them.

ii) Release Consistency

The drawbacks of weak consistency are overcome by:

1. Ensuring that all locally initiated Writes have been completed, i.e., propagated to all other processes.
2. Ensuring that all Writes from other machines have been locally reflected

To differentiate the entering and leaving of CS, release consistency providesacquire and release operations.

Acquire:

- Acquire accesses are used to tell the memory system that a critical region is about to be entered.
- The actions for case 2 need to be performed to ensure that local replicas of variables are made consistent with remote ones.

Release:

- This accesses say that a critical region has just been exited.
- Hence, the actions for case 1 need to be performed to ensure that remote replicas of variables are made consistent with the local ones that have been updated.

The Acquire and Release operations can be defined to apply to a subset of the variables. The accesses themselves can be implemented either as ordinary operationson special variables or as special operations. If the semantics of a CS is not associated with the Acquire and Release operations, then the operations effectively provide for barrier synchronization.

The barrier synchronization states that until all processes complete the previous phase, none can enter the next phase.

This is implemented through protected variables which follows the given rules:

- All previously initiated Acquire operations must complete successfully before a process can access a protected shared variable.
- All accesses to a protected shared variable must complete before a Releaseoperation can be performed.
- The Acquire and Release operations effectively follow the PRAM consistency model.

The lazy release consistency model is relaxation of the release consistency model in which rather than propagating the updated values throughout the system as soon as a process leaves a critical region, the updated values are propagated to the rest of the system only on demand, i.e., only when they are needed.

iii) Entry Consistency

- Entry consistency requires the programmer to use Acquire and Release at the startand end of each CS, respectively.
- Entry consistency requires each ordinary shared variable to be associated with some synchronization variable such as a lock orbarrier.
- When an Acquire is performed on a synchronization variable, only access to those ordinary shared variables that are guarded by that synchronization variable is regulated.

P2P & DISTRIBUTED SHARED MEMORY

PEER TO PEER COMPUTING AND OVERLAY GRAPHS

Peer to peer (P2P) systems refers to the applications that take advantage of resources like storage, time cycles, content, manpower available at the end systems of the internet. In other words, the peer to peer computing architecture contains nodes that are equal participantsin data sharing. All the tasks are equally divided between all the nodes. The nodes interact with each other as required as share resources. This deals with application layer organization of network overlay for flexibility of sharing resources.

The prominent feature of P2P networks is their ability to provide a large combined storage, CPU power, and other resources while imposing a low cost for scalability, andfor entry into and exit from the network.

The ongoing entry and exit of various nodes, as well as dynamic insertion and deletionof objects in P2P network is called **churn.** The impact of churn should be as transparent as possible. There are two types of P2P systems: structured and unstructured.

Differences between structured and unstructured P2P

Unstructured P2P	**Structured P2P**
The construction of overlay network is highly flexible.	The construction of overlay network has lowlevel of flexibility.
The resources are indexed locally.	The resources are distributed remotely andindexed using hash tables.
The messages can be broadcast or randomwalk.	The messages are unicast.
The network puts best effort content location.	The network guarantees the content location.
High overhead.	Comparatively low overhead.
This supports high failure rates.	Supports moderate failure rates.
This is suitable for small scale and highlydynamic applications.	This is suitable for large scale and relativelystable applications.

Overlay network is constructed over another network. For example, connecting internet over telephone lines is an overlay network. The topology of the overlay network is independent from the underlying network.

Characteristics of Peer to Peer Computing

- Peer to peer networks are usually formed by groups computers. These computers all store their data using individual security but also share data with all the other nodes.
- The nodes in peer to peer networks both use resources and provide resources. So, ifthe nodes increase, then the resource sharing capacity of the peer to peer networkincreases.
- The nodes in peer to peer networks act as both clients and servers. Hence, it is difficult to provide adequate security for the nodes. This can lead to denial of service attacks.
- Most modern operating systems such as Windows and Mac OS contain software to implement peer to peer networks.

- Efficient usage ofresources.
- Self -organizing nature: because of scalable storage, CPU power and other resources.
- Distributed control: fast and efficient searching for data.
- Symmetric: highly scalable
- Anonymity: efficient management of churns
- Naming mechanism: selection of geographically close server.
- Security, authentication, trust: Redundancy in storage

Advantages of Peer to Peer Computing

- Each computer in the peer to peer network manages itself. The network is quite easy to set up and maintain.
- The server handles all the requests of the clients. This provision is not required in

peer to peer computing and the cost of the server is saved.

- It is easy to scale the peer to peer network and add more nodes. This only increases the data sharing capacity of the system.
- None of the nodes in the peer to peer network are dependent on the others for their functioning. Hence the network is fault tolerant.
- Easy deployment and organization.

Disadvantages of Peer to Peer Computing

- It is difficult to back-up the data as it is stored in different computer systems and there is no central server.
- It is difficult to provide overall security in the peer to peer network as each systemis independent and contains its own data.

Napster P2P system

- The developers of the original Napster launched the service as a peer-to-peer (P2P)file sharing network.
- The software application was easy to use with a free account, and it was specifically designed for sharing digital music files (in the MP3 format) across a Web-connected network.

Napster used a server-mediated, central index architecture organized aroundclusters of servers that store direct indices of the files in the system.

- The central server maintains a table with the:
    i) the client's address (IP) and port, and offered bandwidth
    ii) information about the files that the client can allow to share
- The basic steps of operation to search for content and to determine a node fromwhich to download the content are the following:
    - A client connects to a meta-server that assigns a lightly loaded server fromone of

the close-by clusters of servers to process the client's query.

- The client connects to the assigned server and forwards its query along withits own identity.
- The server responds to the client with information about the usersconnected to it and the files they are sharing.
- On receiving the response from the server, the client chooses one of the users from whom to download a desired file. The address to enable the P2Pconnection between the client and the selected user is provided by the server to the client.
- The directory serves to provide the mapping from a particular host thatcontains the required content, to the IP address needed to download from it.

Application Layer Overlays

- The fundamental mechanism in P2P networks is data searching.
- This depends on the organization of data and networks.
- The search algorithms for P2P networks tend to be data-centric.
- P2P search uses the P2P overlay, a logical graph among the peers that is used for the object search and object storage and management algorithms.
- Overlays can be thought as a network built over another network.
- Above the P2P overlay is the application layer overlay, where communication between peers is point-to-point once a connection is established.
- The P2P overlay can be structured or unstructured, i.e., no particular graph structure is used.
- Structured overlays use some rigid organizational principles based on the properties of the P2P overlay graph structure, for the object storage algorithms andthe object search algorithms.
- Unstructured overlays use very loose guidelines for object storage. As there

is nodefinite structure to the overlay graph, the search mechanisms are more ad-hoc, and use some forms of flooding or random walk strategies.

- Thus, object storage and search strategies are intricately linked to the overlay structure as well as to the data organization mechanism
- On receiving the response from the server, the client chooses one of the users from whom to download a desired file. The address to enable the P2Pconnection between the client and the selected user is provided by the server to the client.
- The directory serves to provide the mapping from a particular host that contains the required content, to the IP address needed to download from it.

Application Layer Overlays

- The fundamental mechanism in P2P networks is data searching.
- This depends on the organization of data and networks.
- The search algorithms for P2P networks tend to be data-centric.

- P2P search uses the P2P overlay, a logical graph among the peers that is used for the object search and object storage and management algorithms.
- Overlays can be thought as a network built over another network.
- Above the P2P overlay is the application layer overlay, where communication between peers is point-to-point once a connection is established.
- The P2P overlay can be structured or unstructured, i.e., no particular graphstructure is used.
- Structured overlays use some rigid organizational principles based on the properties of the P2P overlay graph structure, for the object storage algorithms andthe object search algorithms.
- Unstructured overlays use very loose guidelines for object storage. As there is no definite structure to the overlay graph, the search mechanisms

are more ad-hoc, and use some forms of flooding or random walk strategies.

- Thus, object storage and search strategies are intricately linked to the overlay structure as well as to the data organization mechanisms.

SHARED MEMORY MUTUAL EXCLUSION

Shared memory model is implemented in operating systems through semaphoresmonitors and atomically executable special purpose hardware.

Lamport's bakery algorithm

- Lamport proposed the classical bakery algorithm for n-process mutual exclusion inshared memory systems.
- This algorithm satisfies the requirements of the critical section problem namelymutual exclusion, bounded waiting, and progress.
- All process threads must take a number and wait their turn to use a sharedcomputing resource or to enter their critical section.
- The number can be any of the global variables, and processes with the lowestnumber will be processed first.
- If there is a tie or similar number shared by both processes, it is managed throughtheir process ID.
- If a process terminates before its turn, it has to start over again in the processqueue.
- A process wanting to enter the critical section picks a token number that is onegreater than the elements in the array choosing [1…n].
- Processes enter the critical section in the increasing order of the token numbers.
- In case of concurrent accesses to choosing by multiple processes, the processesmay have the same token number.
- Then, a unique lexicographic order is defined on the tuple (token, pid) and thisdictates the order in which processes enter the critical section.

(shared vars)

boolean: choosing[1…n];

integer: timestamp[1….n];

repeat

(1) P_i executes the following for the entry section:

(1a) choosing[i] ← 1;

(1b) timestamp[i] ←$\max_{k\in[1...n]}$(timestamp[k]) + 1;

(1c) choosing[i] ← 0;

(1d) for count = 1 to n do

(1e) while choosing[count] do no-op;

(1f) while timestamp[count] ≠0 and (timestamp[count], count)

<(timestamp[i], i) do

(1g) no-op.

(2) P_i executes the critial section (CS) after the entrysection

(3) P_i executes the following exit section after the CS:

(3a) timestamp[i] ← 0

(4) P_i executes the remainder section after the exit section until false;

until false;

Fig : Lamport's Bakery algorithm for shared memory exclusionMutual exclusion

- In the entry section, a process chooses a timestamp for itself, and resets it to 0when it leaves the exit section.
- These steps are non-atomic in the algorithm. Thus multiple processes could bechoosing timestamps in overlapping durations.
- When process i reaches line 1d, it has to check the status of each other process j, todeal with the effects of any race conditions in selecting timestamps.
- In lines 1d–1f, process i serially checks the status of each other process j.
- If j is selecting a timestamp for itself, j's selection interval may have overlappedwith that of i, leading to an unknown order of timestamp values.
- Process i needs to make sure that any other process j(j < i) that had begun to execute line 1b concurrently with itself and may still be executing line 1b does notassign itself the same timestamp.

- If this is not done mutual exclusion could be violated as i would enter the CS, andsubsequently, j, having a lower process identifier and hence a lexicographically lower time stamp, would also enter the CS.
- The i waits for j's timestamp to stabilize, i.e., choosing [j] to be set to false.
- Once j's timestamp is stabilized, i moves from line 1e to line 1f.
- Either j is not requesting or j is requesting. Line 1f determines the relative prioritybetween i and j.
- The process with a lexicographically lower timestamp has higher priority and enters the CS; the other process has to wait(line 1g).
- Thus mutual exclusion is satisfied by the algorithm.

Bounded Waiting

- Bounded waiting is satisfied because each other process j can overtake process i atmost once after i has completed choosing its timestamp.
- The second time j chooses a timestamp, the value will necessarily be larger than i's timestamp if i has not yet entered its CS.

Progress

- Progress is guaranteed because the lexicographic order is a total order and the process with the lowest timestamp at any time in the loop is guaranteed to enter theCS.

Improvements in Lamport's Bakery Algorithm

i) Space complexity

- A lower bound of n registers, specifically, the timestamp array, has been shown forthe shared memory critical section problem.

ii) Time complexity

- When the level of contention is low, the overhead of the entry section does not scale.
- This issue is addressed his concern is addressed by fast mutual exclusion with O(1).
- The limitation of this approach is that it does not guarantee bounded delay.

Lamport's WRWR mechanism and fast mutual exclusion

- This algorithm illustrates an important technique – the (W − R − W – R) sequencethat is a necessary and sufficient sequence of operations to check for contention and to ensure safety in the entry section, by employing just two registers.
- The basic sequence of operations for W(x)–R(y)–W(y)–R(x):

1. The first operation needs to be a Write to x. If it were a Read, then all contending processes could find the value of the variable even outside the entrysection.
2. The second operation cannot be a Write to another variable, for that could equally be combined with the first Write to a larger variable. The second operation should not be a Read of x because it follows Write of x and if there isno interleaved operation from another process, the Read does not provide anynew information. So the second operation must be a Read of another variable,say y.
3. The sequence must also contain Read(x) and Write(y) because there is no pointin reading a variable that is not written to, or writing a variable that is never read.
4. The last operation in the minimal sequence of the entry section must be a Read,as it will help determine whether the process can enter CS. So the last

operation should be Read(x), and the second-last operation should be the Write(y).

(shared variable among the processes)

integer: x, y; // shared register initialized

boolean b[1….n]; //flags to indicate interest in critical section

repeat

(1) $P_i(1 \leq i \leq n)$ executes entry section:

(1a) b[i] ← true; x

(1b) ← i;

(1c) if y ≠0 then

(1d) b[i] ← false;

(1e) await y=0;

(1f) goto(1a);

(1g) y ← i;

(1h) if x ≠i then

(1i) b[i] ← false; for

(1j) j = 1 to n do

(1k) await y = 0;

(1l) if y ≠i then

(1m) await y = 0;

(1n) goto(1a);

(2) $P_i(1 \leq i \leq n)$ executes entry section:

(3) $P_i(1 \leq i \leq n)$ executes exit section:

(3a) y ← 0;

(3b) b[i] ← false

Forever.

Fig : Lamport's fast mutual exclusion algorithm

Hardware Support for Mutual Exclusion

- Hardware support can allow for special instructions that perform two ormore operations atomically.
- Two such instructions, Test &Set and Swap are defined and implemented.
- The atomic execution of two actions, a Read and a Write operation can simplify amutual exclusion algorithm.

(shared variables among the processes accessing each of the different object types)

register: Reg ← initial value; // shared register initialized

(local variables)

integer: old ← initial value; // value to be returned

(1) Test & Set(Reg) return value:

(1a) old ← Reg;

(1b) Reg ← 1;

(1c) return(old).

(2) Swap(Reg, new) return value:

(2a) old ← Reg;

(2b) Reg ← new;

(2c) return(old).

Fig : Definitions for Test&Set, Swap operations

(shared variables)

register: Reg ← false; // shared register initialized

(local variables)

integer: blocked ← 0 // variable to be checked before entering CS

repeat

(1) P_i executes the following for the entry section:

(1a) blocked ← true;

(1b) repeat

(1c) blocked ← Swap(reg, blocked);

(1d) until blocked = false;

(2) P_i executes the critical section (CS) after the entry section

(3) P_i executes the following exit section after the CS:

(3a) Reg ← false;

(4) P_i executes the remainder section after the exit section

until false;

Fig : Code for Swap operation

(shared variable)

register: Reg ← false; // shared register initialized

boolean: waiting[1…n];

(local variables)

integer: blocked ← initial value // value to be checked before // entering CS

repeat

(1) P_i executes the following for the entry section:

(1a) waiting[i] ← true;

(1b) blocked ← true;

(1c) repeat waiting[i] and blocked do

(1d) blocked ← Test&Set(Reg);

(1e) waiting[i] ← false;

(2) P_i executes the critical section (CS) after the entry section

(3) P_i executes the following exit section after the CS:

(3a) next ← (i + 1) mod n;

(3b) while next ≠1 and waiting [next] = false do

(3c) next ← (next + 1) mod n;

(3d) if next = i then

(3e) Reg ← false;

(3f) else waiting[j] ← false;

(4) P_i executes the remainder section after the exit section

until false;

Fig : Code for Test & Set operation

TAPESTRY

> ***Tapestry is a peer-to-peer overlay network which provides a distributed hash table, routing, and multicasting infrastructure for distributed applications. The Tapestry peer- to-peer system offers efficient, scalable, self-repairing, location-aware routing to nearby resources.***

- Tapestry is a decentralized distributed system.
- It is an overlay network that implements simple key-based routing.
- It is a prototype of a decentralized, scalable, fault-tolerant, adaptive location and routing infrastructure
- Each node serves as both an object store and a router that applications can contactto obtain objects.
- In a Tapestry network, objects are published at nodes, and once an object has been successfully published, it is possible for any other node in the network to find the location at which that object is published.

- The difference between Chord and Tapestry is that in Tapestry the application chooses where to store data, rather than allowing the system to choose a node to store the object at.
- The application only publishes a reference to the object.
- The Tapestry P2P overlay network provides efficient scalable location independent routing to locate objects distributed across the Tapestry nodes.
- The hashed node identifiers are termed **VIDs** (Virtual ID) and the hashed object identifiers are termed as **GUIDs** (Globally Unique ID).

Routing and Overlays

> ***Routing and overlay are the terms coined for looking objects and nodes in any distributed system.***

- It is a middleware that takes the form of a layer which processes the route requestsfrom

the clients to the host that holds the objects.

- The objects can be placed and relocated without the information from the clients.

Functionalities of routing overlays:

- A client requests an object with GUID to the routing overlay, which routes therequest to a node at which the object replica resides.
- A node that wishes to make the object available to peer-to-peer service computes the GUID for the object and announces it to the routing overlay that ensures that the object is reachable by all other clients.
- When client demands object removal, then the routing overlays must make them unavailable.
- Nodes may join or leave the service.

Routing overlays in Tapestry

- Tapestry implements Distributed Hash Table (DHT) and routes the messages to thenodes based on GUID associated with resources through prefix routing.

- **Publish (GUID**) primitive is issued by the nodes to make the network aware of its possession of resource.
- Replicated resources also use the same publish primitive with same GUID. This results in multiple routing entries for the same GUID.
- This offers an advantage that the replica of objects is close to the frequent users toavoid latency, network load, improve tolerance and host failures.

Roots and Surrogate roots

- Tapestry uses a common identifier space specified using m bit values and presently Tapestry recommends m = 160.
- Each identifier O_G in this common overlay space is mapped to a set of unique nodes that exists in the network, termed as the identifier's root set denoted $O_{GR.}$
- If there exists a node v such that $v_{id} = O_{GR,}$ then v is the root of identifier O_G.

- If such a node does not exist, then a globally known deterministic rule is used to identify another unique node sharing the largest common prefix with O_G, that acts as the surrogate root.
- To access object O, the goal is to reach the root O_{GR}.
- Routing to O_{GR} is done using distributed routing tables that are constructed using prefix routing information.

Prefix Routing

> ***Prefix routing at any node to select the next hop is done by increasing the prefix match of the next hop's VID with the destination O_{GR}.***

- Let $M = 2^m$. The routing table at node vid contains $b \cdot \log_b M$ entries, organized in$\log_b M$ levels $i = 1,\ldots, \log_b M$.
- Each entry is of the form <w_{id}, IP address>.
- The following is the property of entry (b) at level i:

  Each entry denotes some neighbor node VIDs with an (i – 1) digit prefix match with v id . Further, in level i, for each digit j in the chosen base, there is an entry for which the ith digit position is j. The jth entry (counting from 0) in level i has value j for digit position i. Let an i digit prefix of vid be denoted as prefix (vid, i). Then the jth entry (counting from 0) in level i begins with an i-digit prefix prefix (vid,i– 1). j.

Routing Table

- The nodes in the router table at v_{id} are the neighbors in the overlay, and these are exactly the nodes with which v_{id} communicates.
- For each forward pointer from node v to v', there is a backward pointer from v' to v.
- There is a choice of which entry to add in the router table. The jth entry in level i can be the VID of any node whose i-digit prefix is determined; the (m – i) digit suffix can vary.

- The flexibility is useful to select a node that is close, as defined by some metric space.
- This choice also allows a more fault-tolerant strategy for routing.
- Multiple VIDs can be stored in the routing table.
- The jth entry in level i may not exist because no node meets the criterion. This is a hole in the routing table.
- Surrogate routing can be used to route around holes. If the jth entry in level i should be chosen but is missing, route to the next non-empty entry in level i, using wraparound if needed.
- All the levels from 1 to $\log_b 2^m$ need to be considered in routing, thus requiring $\log_b 2^m$ hops.

(variables)

Integer Table[1…$\log_b 2^m$, 1….b]; //routing table

(1) NEXT_HOP(i, OG = d_1 o d_2 … o d_{logbm}) executed at node v_{id} to route to O_G:

// i is (1 + Length of longest common prefix), also level of the table

(1a) while Table[i, d_i] = ⊥do // d_j is the ith digit of destination

(1b) $d_i \leftarrow (d_i+1) \bmod b$;

(1c) if Table[i, d_i] = v then // node v also acts as next hop // (special case)

(1d) return (NEXT_HOP(i+1, OG) // locally examine next digit of //destination

(1e) else return (Table[i, d_i]). // node Table[i, d_i] is next hop

Fig : NEXT_HOP(i, O_G)

Object Publication and object searching

- The unique spanning tree used to route to vid is used to publish and locate an object whose unique root identifier O_{GR} is v_{id}.
- A server S that stores object O having GUID O_G and root O_{GR} periodically publishes the object by routing a publish message from S towards O_{GR}.

- At each hop and including the root node O_{GR}, the publish message creates a pointerto the object.
- Each node between O and O_{GR} must maintain a pointer to O despite churn.
- If a node lies on the path from two or more servers storing replicas, that node will store a pointer to each replica, sorted in terms of a distance metric.
- This is the directory information for objects, and is maintained as a soft-state, i.e., it requires periodic updates from the server, to deal with changes and to provide fault-tolerance.
- To search for an object O with GUID O_G, a client sends a query destined for the root O_{GR}
- Along the $\log_b 2^m$ hops, if a node finds a pointer to the object residing on server S,the node redirects the query directly to S. Otherwise, it forwards the query towardsthe root O_{GR} which is guaranteed to have the pointer for the location mapping.
- A query gets redirected directly to the object as soon as the query path overlaps the publish path towards the same root.
- Each hop towards the root reduces the choice of the selection of its next node by a factor of b; hence, the more likely by a factor of b that a query path and a publishpath will meet.
- As the next hop is chosen based on the network distance metric whenever there is a choice, it is observed that the closer the client is to the server in terms of the distance metric, the more likely that their paths to the object root will meet sooner, and the faster the query will be redirected to the object.

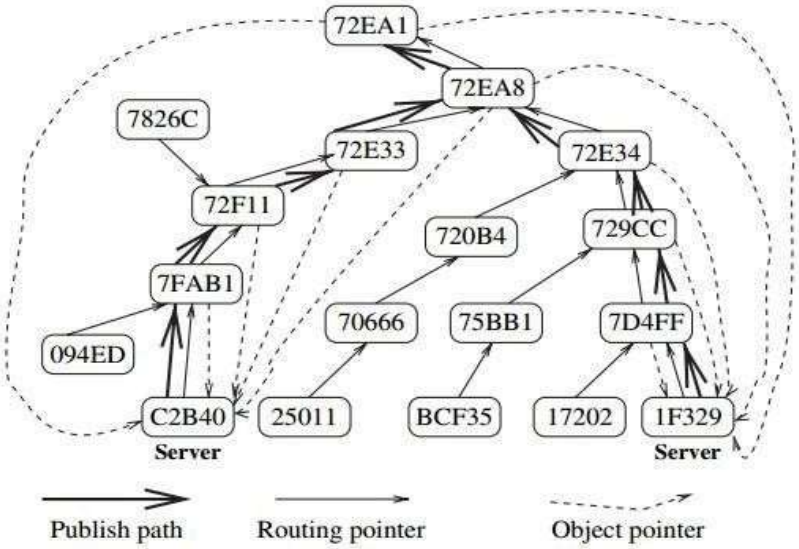

Fig : Publishing of object with identifier 72EA1 at two replicas 1F329 and C2B40Node Insertion

- When nodes join the network, the result should be the same as though the networkand the routing tables had been initialized with the nodes as part of the network.
- The procedure for the insertion of node X should maintain the following property of Tapestry: For any node Y on the path between a publisher of object O and the root O_{GR}, node Y should have a pointer to O.

Properties for node insertion:

- Nodes that have a hole in their routing table should be notified if the insertion of node X can fill that hole.
- If X becomes the new root of existing objects, references to those objects should now lead to X.
- The routing table for node X must be constructed.
- The nodes near X should include X in their routing tables to perform more efficientrouting.

Steps in insertion

- Node X uses some gateway node into the Tapestry network to route a message toitself. This leads to its surrogate, i.e., the root node with identifier closest to that of itself (which is Xid). The surrogate Z identifies the length α of the longest common prefix that Zid shares with Xid.
- Node Z initiates a MULTICAST-CONVERGECAST on behalf of X by creating a

logical spanning tree as follows. Acting as a root, Z contacts all the (α, j) nodes, for all j $\in$ {0, 1, …, b – 1}.

- These are the nodes with prefix αfollowed by digit j. Each such (level 1) node Z1 contacts all the prefix ((Z1, $|\alpha|$ + 1),j) nodes, for all j $\in$ {0, 1,…,b – 1}. This continues up to level $\log_b 2^m$ and completes the MULTICAST.
- The nodes at this level are the leaves of the tree, and initiate the CONVERGECAST, which also helps to detect the termination of this phase.
- The insertion protocols are fairly complex and deal with concurrent insertions.

Node Deletion

When a node A leaves the Tapestry overlay:

1. Node A informs the nodes to which it has back pointers. It also provides them with replacement entries for each level from its routing table. This is to prevent holes in their routing tables.
2. The servers to which A has object pointers are also notified. The notified servers send object republish messages.
3. During the above steps, node A routes messages to objects rooted at itself to theirnew roots. On completion of the above steps, node A informs the nodes reachable via its back pointers and forward pointers that it is leaving, and then leaves.

- Node failures are handled by using the redundancy that is built in to the routing tables and object location pointers.
- A node X detects a failure of another node A by using soft-state beacons or when a node sends a message but does not get a response.
- Node X updates its routing table entry for A with a suitable substitute node, running the nearest neighbor algorithm if necessary.
- If A's failure leaves a hole in the routing table of X, then X contacts the suggorate of A in an effort to identify a node to fill the hole.

- To repair the routing mesh, the object location pointers also have to be adjusted.
- Objects rooted at the failed node may be inaccessible until the object is republished.
- The protocols for doing so essentially have to:
    i) maintain path availability
    ii) optionally collect garbage/dangling pointers that would otherwise persist until the next soft-state refresh and timeout

Complexity

- A search for an object is expected to take $\log_b 2^m$ hops. The routing tables are optimized to identify nearest neighbor hops.
- The size of the routing table at each node is $c \cdot b \cdot \log_b 2^m$ where c is the constant that limits the size of the neighbor set that is maintained for fault-tolerance.