UNIT V P2P & DISTRIBUTED SHARED MEMORY

Peer-to-peer computing and overlay graphs: Introduction – Data indexing and overlays – Chord – Content addressable networks – Tapestry. Distributed shared memory: Abstraction and advantages – Memory consistency models –Shared memory Mutual Exclusion

5.1 Peer-to-peer Computing and Overlay Graphs

Characteristics

- Peer-to-peer (P2P) network systems use an application-level organization of the network overlay for flexibly sharing resources (e.g., files and multimedia documents) stored across network-wide computers.
- All nodes are equal; communication directly between peers (no client-server) Allow location of arbitrary objects; no DNS servers required
- Large combined storage, CPU power, other resources, without scalability costs
- Dynamic insertion and deletion of nodes, as well as of resources, at low cost

Features	Performance
self-organizing	large combined storage, CPU power, and resources
distributed control	fast search for machines and data objects
role symmetry for nodes	Scalable
anonymity	efficient management of churn
naming mechanism	selection of geographically close servers
security, authentication, trust	redundancy in storage and paths

Table:Desirable characteristics and performance features of P2P systems.

Napster

- One of the earliest popular P2P systems, Napster [25], used a server-mediated central index architecture organized around clusters of servers that store direct indices of the files in the system.
- Central server maintains a table with the following information of each registered client: (i) the client's address (IP) and port, and offered bandwidth, and (ii) information about the files that the client can allow to share.
  1. A client connects to a meta-server that assigns a lightly-loaded server.
  2. The client connects to the assigned server and forwards its query and identity.
  3. The server responds to the client with information about the users connected to it and the files they are sharing.
  4. 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 P2P connection between the client and the selected user is provided by the server to the client.

Users are generally anonymous to each other. 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

- A core mechanism in P2P networks is searching for data, and this mechanism depends on how (i) the data, and (ii) the network, are organized. Search algorithms for P2P networks tend to be data-centric, as opposed to the host-centric algorithms for traditional networks.
- P2P search uses the *P2P overlay*, which is a logical graph among the peers that is used for the object search and object storage and management algorithms. Note that above the P2P over-lay is the application layer overlay, where communication between peers is point-to-pont (representing a logical all-to-all connectivity) once a connection is established.
- The P2P overlay can be *structured* (e.g., hypercubes, meshes, butterfly networks, de Bruijn graphs) or *unstructured*

Structured and Unstructured Overlays

- Search for data and placement of data depends on P2P overlay (which can be thought of as being below the application level overlay)
- Search is data-centric, not host-centric Structured P2P overlays:
  -) E.g., hypercube, mesh, de Bruijngraphs
  -) rigid organizational principles for object storage and object search
- Unstructured P2P overlays:
  -) Loose guidelines for object search and storage
  -) Search mechanisms are ad-hoc, variants of flooding and random walk
- Object storage and search strategies are intricately linked to the overlay structure as well as to the data organization mechanisms.

5.2 Data indexing

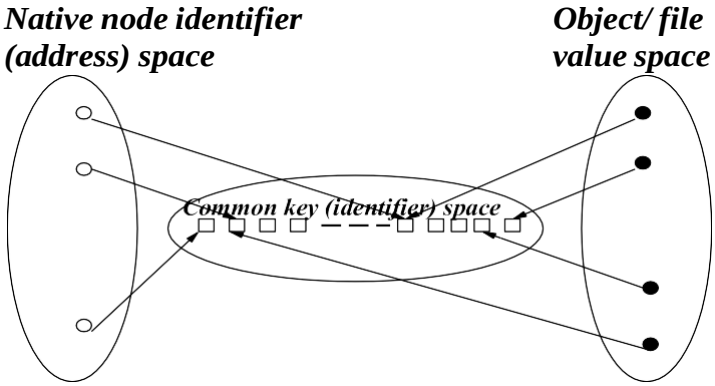
The data in a P2P network is identified by using indexing. Data indexing allows the physical data independence from the applications. Indexing mechanisms can be classified as being centralized, local, or distributed

- **Centralized indexing**, e.g., versions of Napster, DNS
- **Distributed indexing**. Indexes to data scattered across peers. Access data through mechanisms such as Distributed Hash Tables (DHT). These differ in hash mapping, search algorithms, diameter for lookup, fault tolerance, churn resilience.
- **Local indexing**. Each peer indexes only the local objects. Remote objects need to be searched for. Typical DHT uses flat key space. Used commonly in unstructured overlays (E.g., Gnutella) along with flooding search or random walk search.

An alternate way to classify indexing mechanisms is as being a *semantic index mechanism* or a *semantic-free* index mechanism.

- **Semantic indexing** - human readable, e.g., filename, keyword, database key. Supports keyword searches, range searches, approximate searches.
- **Semantic-free indexing**. Not human readable. Corresponds to index obtained by use of hash function.

Simple Distributed Hash Table scheme



Mappings from node address space and object space in a simple DHT.

- Highly deterministic placement of files/data allows fast lookup. But file insertions/deletions under churn incurs some cost.
- Attribute search, range search, keyword search etc. not possible.

5.2.1 Distributed indexing

Structured overlays

- The P2P network topology has a definite structure, and the placement of files or data in this network is highly deterministic as per some algorithmic mapping. (The placement of files can sometimes be "loose," as in some earlier P2P systems like Freenet, where "hints" are used.)
- The objective of such a deterministic mapping is to allow a very fast and deterministic lookup to satisfy queries for the data. These systems are termed as lookup systems and typically use a hash table interface for the mapping.

Unstructured overlays

-
- The P2P network topology does not have any particular controlled structure, nor is there any control over where files/data is placed. Each peer typically indexes only its local data objects, hence, local indexing is used.
- Node joins and departures are easy – the local overlay is simply adjusted. File placement is not governed by the topology. Search for a file may entail high message overhead and high delays. However, complex queries are supported because the search criteria can be arbitrary.
- Although the P2P network topology does not have any controlled structure, some topologies naturally emerge.
    - Power law random graph (PLRG) This is a random graph where the node degrees follow the power law. Here, if the nodes are ranked in terms of their degree, then the ith node has c/i neighbors, where c is a constant.
    - Normal random graph This is a normal random graph where the nodes typically have a uniform degree.

Structured vs. unstructured overlays

Structured Overlays:

- structure $\Longrightarrow$ placement of files is highly deterministic, file insertions and deletions have some overhead
- Fast lookup
- Hash mapping based on a single characteristic (e.g., file name)
- Range queries, keyword queries, attribute queries difficult to support

Unstructured Overlays:

- No structure for overlay $\Longrightarrow$ no structure for data/file placement
- Node join/departures are easy; local overlay simply adjusted
- Only local indexing used
- File search entails high message overhead and high delays
- Complex, keyword, range, attribute queries supported
- Some overlay topologies naturally emerge:
  - Power Law Random Graph (PLRG) where node degrees follow a power law. Here, if the nodes are ranked in terms of degree, then the i^{th} node has c/i^{α} neighbors, where c is a constant.
  - simple random graph: nodes typically have a uniform degree

Unstructured Overlays: Properties

- Semantic indexing possible =⇒ keyword, range, attribute-based queries Easily accommodate high churn
- Efficient when data is replicated in network Good if user satisfied with ”best-effort” search
- Network is not so large as to cause high delays in search

Gnutella features

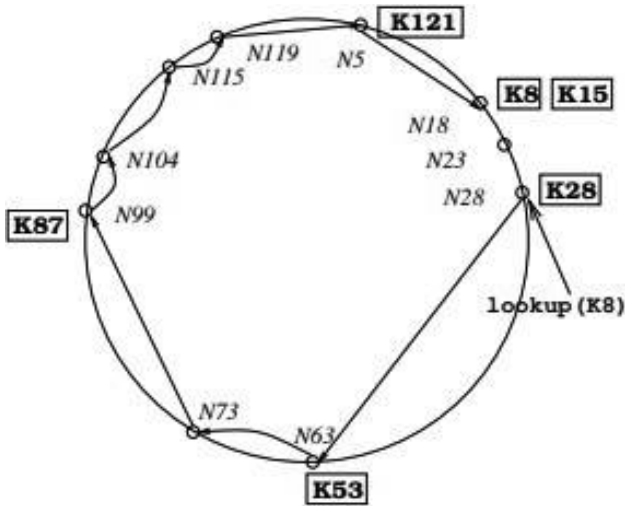
- A joiner connects to some standard nodes from Gnutella directory
- *Ping* used to discover other hosts; allows new host to announce itself
- *Pong* in response to *Ping* ; *Pong* contains IP, port #, max data size for download
- *Query* msgs used for flooding search; contains required parameters
- *QueryHit* are responses. If data is found, this message contains the IP, port #, file size, download rate, etc. Path used is reverse path of *Query*

5.3 Chord

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

- When a node joins or leaves the network of *n* nodes, only 1/*n* keys have to moved.
- The Chord key space is flat, thus giving applications flexibility in map-ping their files/data to keys. Chord supports a single operation, lookup x , which maps a given key x to a network node. Specifically, Chord stores a file/object/value at the node to which the file/object/value’s key maps.
- Two steps involved.

-) Map the object value to its key
-) Map the key to the node in the nativeaddressspaceusing *lookup*
- Common address space is a *m*-bit identifier (2^m addresses), and this space is arranged on a logical ring *mod*(2^m).
- A key *k* gets assigned to the first node such that the node identifier equals or is greater than the key identifier *k* in the logical space address.



Chord: SimpleLookup

- A simple key lookup algorithm that requires each node to store only 1 entry in its routing table works as follows.
- 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.

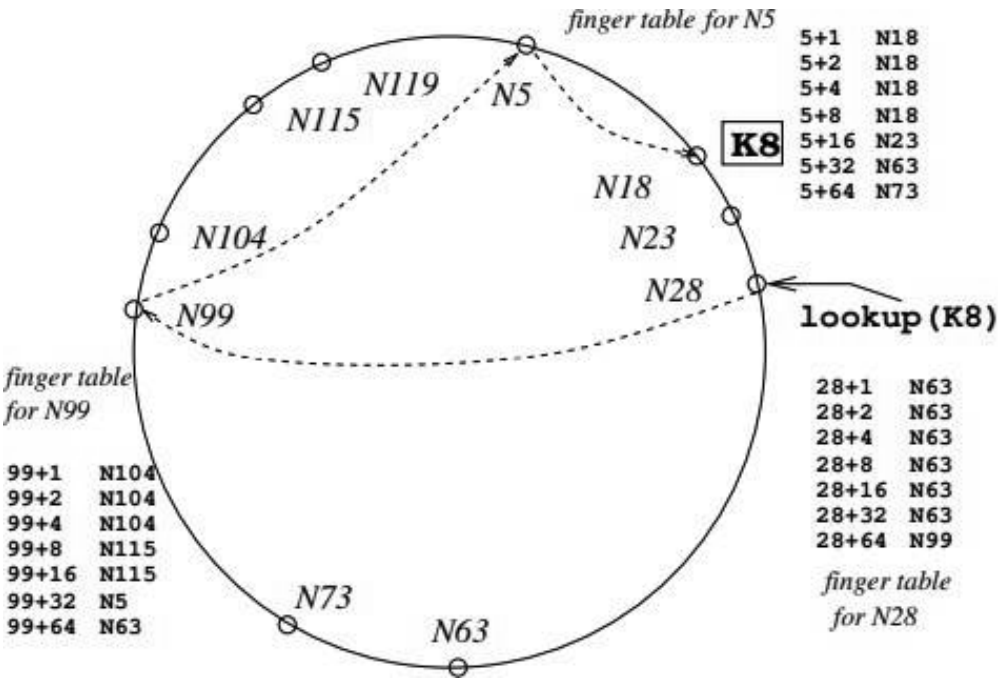
Chord: Scalable Lookup

- Each node i maintains a routing table, called the *finger table*, with $O(\log n)$ entries, such that the xth entry ($1 \leq x \leq m$) is the node identifier of the node $succ(i + 2^{x-1})$.
- This is denoted by $i.finger[x] = succ(i + 2^{x-1})$. This is the first node whose key is greater than the key of node i by at least $2^{x-1} \bmod 2^m$.
- Complexity: $O(\log n)$ message hops at the cost of $O(\log n)$ space in the local routing tables
- Due to the *log* structure of the finger table, there is more info about nodes closer by than about nodes further away.
- Consider a query on key *key* at node i,
  - if *key* lies between i and its successor, the *key* would reside at the successor and its 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.

Chord

```
(variables)
integer: successor ⟵ initial value;
integer: predecessor ⟵ initial value;
array of integer finger[1 . . . log n];

(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]).
```

Scalable Lookup - Example



Chord: Managing Churn

The code to manage dynamic node joins, departures, and failures is given in Algorithm

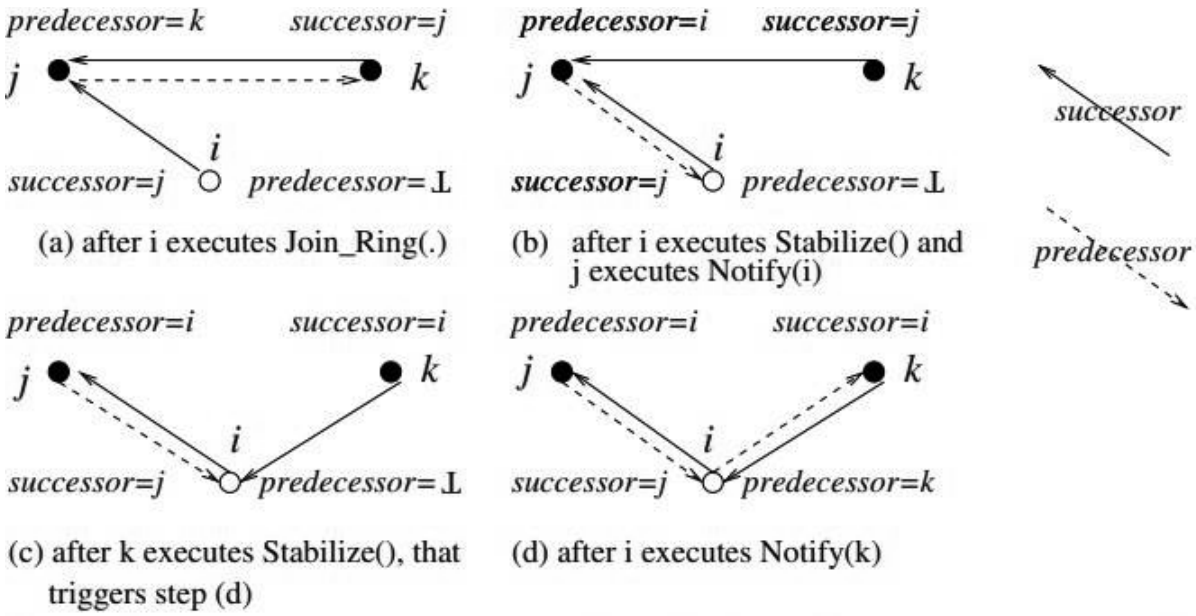
Node joins

- To create a new ring, a node i executes Create_New_Ring which creates a ring with 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.

Algorithm Managing churn in Chord. Code shown is for node

```
(variables)
integer: successor ⟵ initial value;
integer: predecessor ⟵ initial value;
array of integer finger[1 . . . log m];
integer: next_finger ⟵ 1;


(1) i.Create_New_Ring():
(1a) predecessor ⟵ ⊥;
(1b) successor ⟵ i.

(2) i.Join_Ring(j), where j is any node on the ring to be joined:
(2a) predecessor ⟵ ⊥;
(2b) successor ⟵ j.Locate_Successor(i).

(3) i.Stabilize():        // executed periodically to verify and inform successor
(3a) x ⟵ successor.predecessor;
(3b) if x ∈ (i, successor) then
(3c)      successor ⟵ x;
(3d) successor.Notify(i).

(4) i.Notify(j):          // j believes it is predecessor of i
(4a) if predecessor = ⊥ or j ∈ (predecessor, i)) then
(4b)      transfer keys in the range (predecessor, j] to j;
(4c)      predecessor ⟵ j.

(5) i.Fix_Fingers():      // executed periodically to update the finger table
(5a) next_finger ⟵ next_finger + 1;
(5b) if next_finger > m then
(5c)      next_finger ⟵ 1;
(5d) finger[next_finger] ⟵ Locate_Successor(i + 2^(next_finger − 1)).

(6) i.Check_Predecessor():      // executed periodically to verify whether predecessor still exists
(6a) if predecessor has failed then
(6b)      predecessor ⟵ ⊥.
```

Figure illustrates the main steps of the joining process. A recent joiner node i that has executed Join_Ring · gets integrated into the ring by the following sequence:



Node i integrates into the ring, where $j > i > k$, as per the steps shown.

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 $\in$ i successor , then i updates its successor to k. In either case, i notifies 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.

- How are node departures handled? or node failures?
- For a Chord network with n nodes, each node is responsible for at most $(1 + s)$ K/n keys, with "high probability", where K is the total number of keys. Using consistent hashing, s can be shown to be bounded by $O(\log n)$.
- 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 $\log (n) \leq m$. The average lookup time is $1/2 \log (n)$.

5.4 Content Addressable Network (CAN)

- An indexing mechanism that maps objects to locations in CAN
- object-location in P2P networks, large-scale storage management, wide-area name resolution services that decouple name resolution and the naming scheme
- Efficient, scalable addition of and location of objects using location-independent names or keys.
- 3 basic operations: insertion, search, deletion of (*key, value*) pairs
- *d*-dimensional logical Cartesian space organized as a *d*-torus logical topology, i.e.. *d* -dimensional mesh withwraparound.
- Space partitioned dynamically among nodes, i.e., node *i* has space *r* (*i*). For object v , its key r (v) is mapped to a point ˙p in the space. (v, key (v)) tuple stored at node which is the present owner containing the point ˙p.
- Analogously to retrieve object *v*.

3 components of CAN

) Set up CAN virtual coordinate space, partition among nodes

❯ Routing in virtual coordinate space to locate the node that is assigned the region corresponding to *p*˙

) Maintain the CAN in spite of node departures and failures

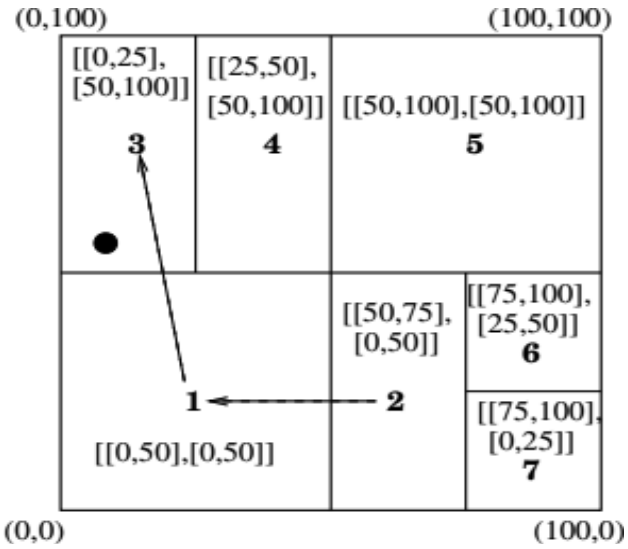
CAN Initialization

- Each CAN has a unique DNS name that maps to the IP address of a few bootstrap nodes. Bootstrap node: tracks a partial list of the nodes that it believes are currently in the CAN.
- A joiner node queries a bootstrap node via a DNS lookup. Bootstrap node replies with the IP addresses of some randomly chosen nodes that it believes are in the CAN.
- 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 CAN routing algorithm.
- 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. This also helps to methodically merge regions, if necessary. The (k, v) tuples for which the key k now maps to the zone to be transferred to the joiner, are also transferred to the joiner.
- The joiner learns the IP addresses of its neighbours from old owner (˙p). The neighbors are old owner (˙p) and a subset of the neighbours of old owner (˙p). old owner (˙p) also updates its set of neighbours. The new joiner as well as old owner (˙p) inform their neighbours of the changes to the space allocation, In fact, each node has to send an immediate update of its assigned region, followed by periodic HEARTBEAT refresh messages, to all its neighbours.

When a node joins a CAN, only the neighbouring nodes in the coordinate space are required to participate. The overhead is thus of the order of the number of neighbours, which is $O(d)$ and independent of n.

CAN routing

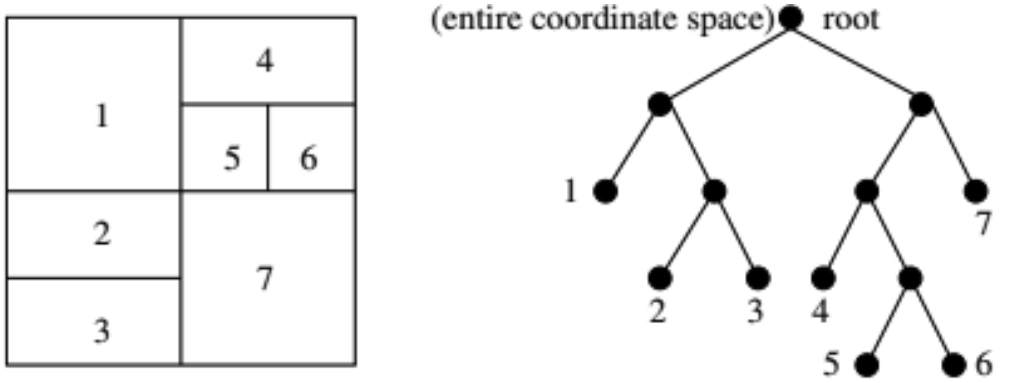
- CAN routing uses the straight-line path from the source to the destination in the logical Euclidean space.
- This routing is realized as follows. Each node maintains a routing table that tracks its neighbor nodes in the log-ical coordinate space. In d-dimensional space, nodes x and y are neigh-bors if the coordinate ranges of their regions overlap in d − 1 dimensions, and abut in one dimension.



- The routing table at each node tracks the IP address and the virtual coor-dinate region of each neighbor. To locate value v, its key k v is mapped to a point p- whose coordinates are used in 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

CAN Maintainence

- Voluntary departure: Hand over region and (*key*, *value*) tuples to a neighbor. Neighbor choice: formation of a convex region after merger of regions
- Otherwise, neighbor with smallest volume. However, regions are not merged and neighbor handles both regions until background reassignment protocol is run.
- Node failure detected when periodic HEARTBEAT message not received by neighbors. They then run a TAKEOVER protocol to decide which neighbor will own dead node's region. This protocol favors region with smallest volume.
- Despite TAKEOVER protocol, the (*key*, *value*) tuples remain lost until background region reassignment protocol is run.
- Background reassignment protocol: for 1-1 load balancing, restore 1-1 node to region assignment, and prevent fragmentation.



CAN Optimizations

Improve per-hop latency, path length, fault tolerance, availability, and load balancing. These techniques typically demonstrate a trade-off.

- **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 or coordinate spaces.** The same node will store different (k, v) tuples belonging to the region assigned to it in each reality, and will also have a different neighbour set. The data contents (k, v) get replicated, leading to higher availability. Furthermore, the multiple copies of each (k, v) tuple offer a choice.
- Routing fault tolerance also improves.
- Use delay metric instead of Cartesian metric for routing
- Overloading coordinate regions by having multiple nodes assigned to each region. Path length and latency can reduce, fault tolerance improves, per-hop latency decreases.
- Use multiple hash functions. Equivalent to using multiple realities. Topologically sensitive overlay. This can greatly reduce per-hop latency.

CAN Complexity: $O(d)$ for a joiner. $O(d/4 \log(n))$ for routing.
Node departure $O(d^2)$.

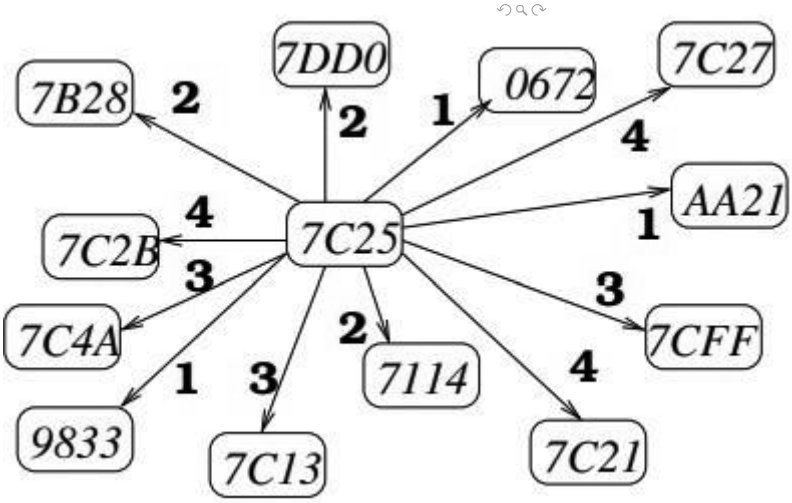
5.5 Tapestry

- The Tapestry P2P overlay network provides efficient scalable location-independent routing to locate objects distributed across the Tapestry nodes
- Nodes and objects are assigned IDs from common space via a distributed hashing.
- Hashed node ids are termed VIDs or v_{id} . Hashed object identifiers are termed GUIDs or O_G .
- ID space typically has m = 160 bits, and is expressed in hexadecimal.
- If a node v exists such that $v_{id} = O_G$ exists, then that v become the root. If such a v does not exist, then another unique node sharing the largest common prefix with O_G is chosen to be **the *surrogate root.***
- The object O_G is stored at the root, or the root has a direct pointer to the object.

- To access object *O*, reach the root (real or surrogate) using ***prefix routing*** Prefix routing to select the next hop is done by increasing the prefix match of the next hop's VID with the destination O_GR . Thus, a message destined for O_GR = 62*C* 35 could be routed along nodes with VIDs 6****, then 62***, then 62C**, then 62C3*, and then to 62C35

Tapestry - Routing Table

- Let $M = 2^m$. The routing table at node v_{id} 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)*.
- Each entry denotes some "neighbour" node VIDs with a $(i - 1)$-digit prefix match with v_{id} – thus, the entry's w_{id} matches v_{id} in the $(i - 1)$-digit prefix. Further, in level *i*, for each digit *j* in the chosen base (e.g., $0, 1, \ldots E, F$ when $b = 16$), there is an entry for which the i^{th} digit position is *j*.
- For each forward pointer, there is a backward pointer.

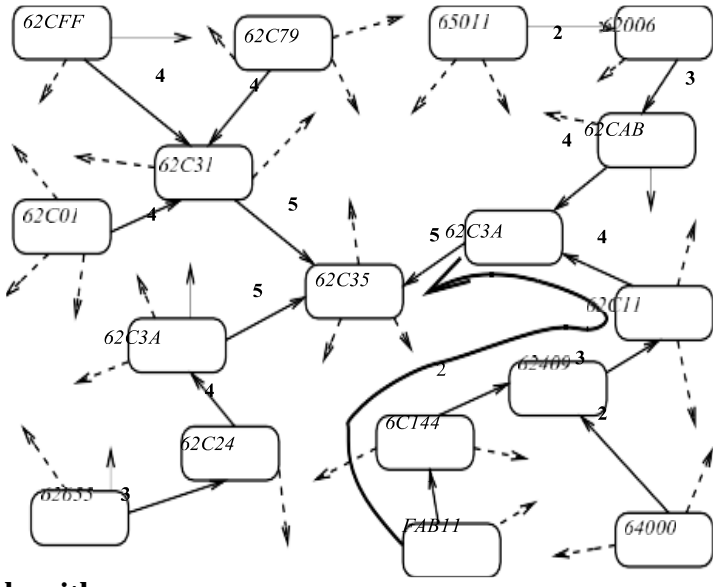


Some example links at node with identifier "7C25". Three links each of levels 1 through 4 are labeled.

Tapestry: Routing

- The j^{th} 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 j^{th} 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.

An example of routing from FAB11 to 62C35. The numbers on the arrows show the level of the routing table



Tapestry: RoutingAlgorithm

- Surrogate routing leads to a unique root.
- For each v_{id} , the routing algorithm identifies a unique spanning tree rooted at v_{id} .

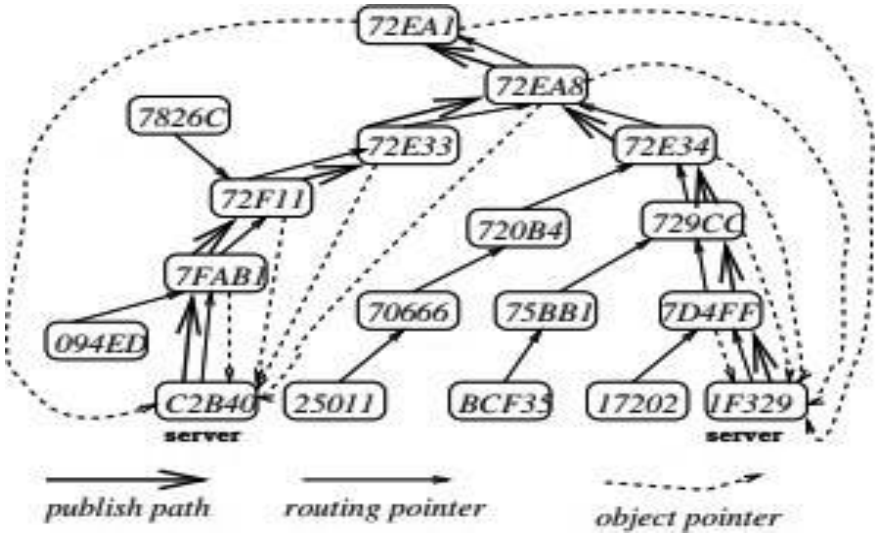
```
(variables)
array of array of integer Table[1 ... log_b 2^m, 1 ... b];          // routing table

(1) NEXT_HOP(i, O_G = d_1 ∘ d_2 ... ∘ d_{log_b M}) 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 ith digit of destination
(1b)      d_i ⟵ (d_i + 1) mod b;
(1c) if Table[i, d_i] = v then     // node v also acts as next hop (special case)
(1d)      return NEXT_HOP(i + 1, O_G)   // locally examine next digit of destination
(1e) else return(Table[i, d_i]).     // node Table[i, d_i] is next hop
```

Tapestry: Object Publication and Object Search

- The unique spanning tree used to route to v_{id} 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 pointer to the object
- This is the directory info and is maintained in *soft-state*.
- 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 towards the root O_{GR} which is guaranteedto 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



An example showing publishing of object with identifier 72EA1 at two replicas 1F329 and C2B40. A query for the object from 094ED will find the object pointer at 7FAB1. A query from 7826C will find the object pointer at 72F11. A query from BCF35 will find the object pointer at 729CC.

Tapestry: Node Insertions

- For any node *Y* on the path between a publisher of object *O* and the root
- G_{OR} , node *Y* should have a pointer to *O*.
- Nodes which 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 efficient routing.

The main steps in node insertion are as follows:

1. Node X uses some gateway node into the Tapestry network to route a message to itself. This leads to its "surrogate," i.e., the root node with identifier closest to that of itself (which is

X_{id}). The surrogate Z identifies the length of the longest common prefix that Z_{id} shares with X_{id}.

2. Node Z initiates a MULTICAST-CONVERGECAST on behalf of X by essentially creating a logical spanning tree as follows. Acting as a root,

   Z contacts all the j nodes, for all j ∈ 0 1 b − 1 (tree level 1). These are the nodes with prefix followed by digit j. Each such (level 1) node Z1 contacts all the prefix Z1 + 1 j nodes, for all j ∈ 0 1 b − 1 (tree level 2). This continues up to level $\log_b 2^m$ − and completes the MULTICAST. The nodes at this level are the leaves

Tapestry: Node Deletions and Failures

Node deletion

- Node *A* informs the nodes to which it has (routing) backpointers. It also provides them with replacement entries for each level from its routing table. This is to prevent holes in their routing tables. (The notified neighbours can periodically run the nearest neighbour algorithm to fine-tune their tables.)
- The servers to which *A* has object pointers are also notified. The notified servers send object republish messages.
- During the above steps, node *A* routes messages to objects rooted at itself to their new roots. On completion of the above steps, node *A* informs thenodes reachable via its backpointers and forward pointers that it is leaving, and then leaves.

Node failures: Repair the object location pointers, routing tables and mesh, using the redundancy in the Tapestry routing network. Refer to the book for the algorithms

Complexity

- A search for an object expected to take ($log_b 2^m$) hops. However, the routing tables are optimized to identify nearest neighbour hops (as per the space metric). Thus, the latency for each hop is expected to be small, compared to that for CAN and Chord protocols.
- The size of the rou·ti·ng table at each node is $c\ b\ log_b 2^m$, where *c* is the constant that limits the size of the neighbour set that is maintained for fault-tolerance.

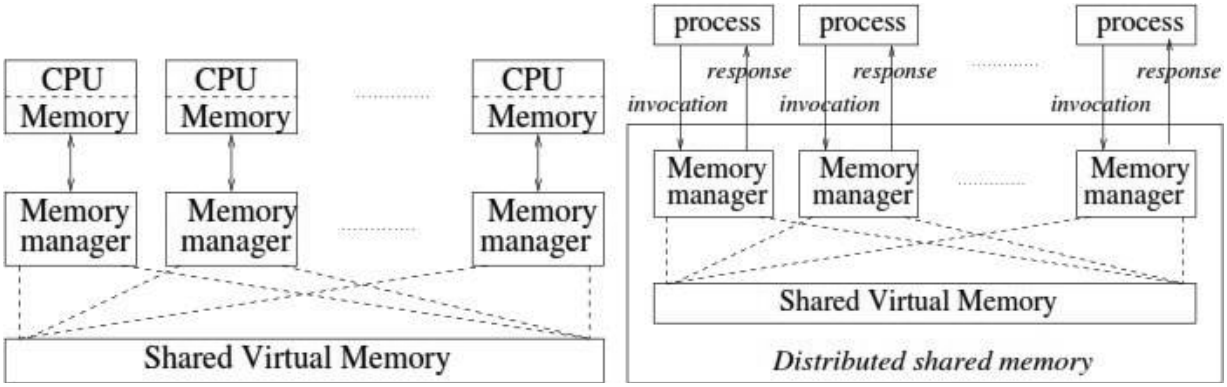
The larger the Tapestry network, the more efficient is the performance. Hence, better if different applications share the same overlay.

5.6 Distributed Shared Memory

Distributed Shared Memory Abstractions

Distributed shared memory (DSM) is an abstraction provided to the programmer of a distributed system. It gives the impression of a single monolithic memory, as in traditional von Neumann architecture. Programmers access the data across the network using only *read* and *write* primitives, as they would in a uniprocessor system. 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.

- communicate with Read/Write ops in shared virtual space No Send and Receive primitives to be used by application
  -) Under covers, Send and Receive used by DSM manager
- *Locking is too restrictive; need concurrent access*
- With replica management, problem of consistency arises!



Advantages/Disadvantages of DSM

Advantages:

Shields programmer from Send/Receive primitives
Single address space; simplifies passing-by-reference and passing complex data structures
Exploit locality-of-reference when a block is moved
DSM uses simpler software interfaces, and cheaper off-the-shelf hardware. Hence cheaper than dedicated multiprocessor systems
No memory access bottleneck, as no single bus Large **virtual memory space**

- DSM programs portable as they use common DSM programming interface Disadvantages:
- Programmers need to understand consistency models, to write correct programs
- DSM implementations use async message-passing, and hence cannot be more efficient than msg-passing implementations
- By yielding control to DSM manager software, programmers cannot use their own msg-passing solutions.

Issues in Implementing DSM Software

- Semantics for concurrent access must be clearly specified Semantics – replication? partial? full? read-only? write-only? Locations for replication (for optimization)
- If not full replication, determine location of nearest data for access Reduce delays, # msgs to implement the semantics of concurrent access
- Data is replicated or cached Remote access by HW or SW
- Caching/replication controlled by HW or SW
- DSM controlled by memory management SW, OS, language run-time system

Comparison of Early DSM Systems

Type of DSM	Examples	Management	Caching	Remote access
single-bus multiprocessor	Firefly, Sequent	by MMU	hardware control	by hardware
switched multiprocessor	Alewife, Dash	by MMU	hardware control	by hardware
NUMA system	Butterfly, CM*	by OS	software control	by hardware
Page-based DSM	Ivy, Mirage	by OS	software control	by software
Shared variable DSM	Midway, Munin	by language runtime system	software control	by software
Shared object DSM	Linda, Orca	by language runtime system	software control	by software

Memory consistency models

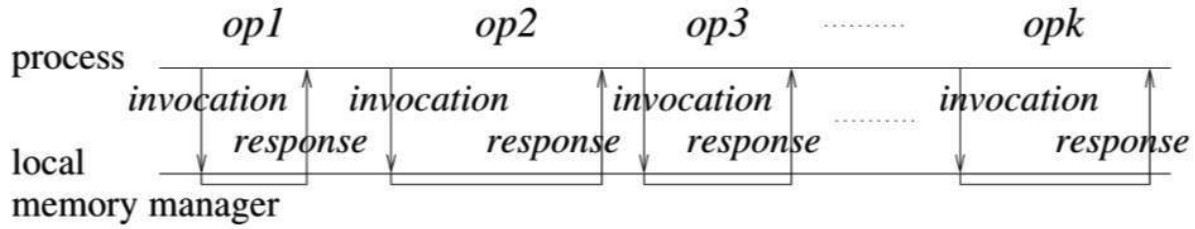
The *memory consistency model* defines the set of allowable memory access orderings.

Memory Coherence

Memory coherence is the ability of the system to execute memory operations correctly.

- s_i *memory operations by* P_i
- $(s_1 + s_2 + \ldots s_n)!/(s_1!s_2! \ldots s_n!)$ possible interleavings
- *Memory coherence model defines which interleavings are permitted Traditionally, Read returns the value written by the most recent Write "Most recent" Write is ambiguous with replicas and concurrent accesses*

DSM consistency model is a *contract* between DSM system and application programmer



Sequential invocations and responses in a DSM system, without any pipelining

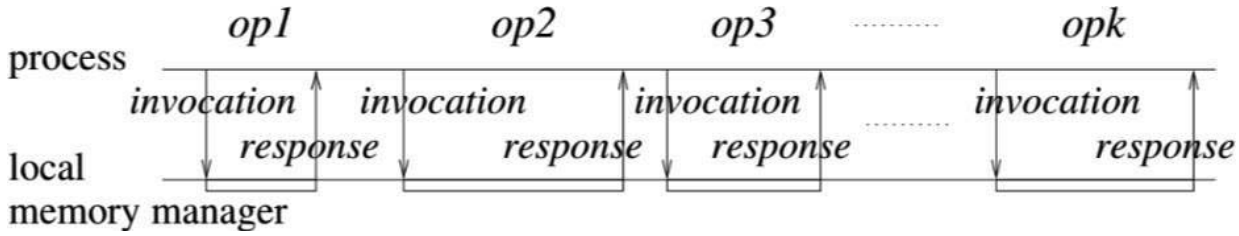
Strict Consistency/Linearizability/Atomic Consistency

Strict consistency

The strictest model, corresponding to the notion of correctness on the tradi-tional Von Neumann architecture or the uniprocessor machine, requires that any *Read* to a location (variable) should return the value written by the most recent *Write* to that location (variable).

Two salient features of such a system are the following: (i) a common global time axis is implicitly available in a uniprocessor system; (ii) each write is immediately visible to all processes.

1. A Read should return the most recent value written, per a global time axis. For operations that overlap per the global time axis, the following must hold.
2 All operations appear to be atomic and sequentially executed.
3 All processors see the same order of events, equivalent to the global time ordering of non-overlapping events.



Sequential invocations and responses to each Read or Write operation.

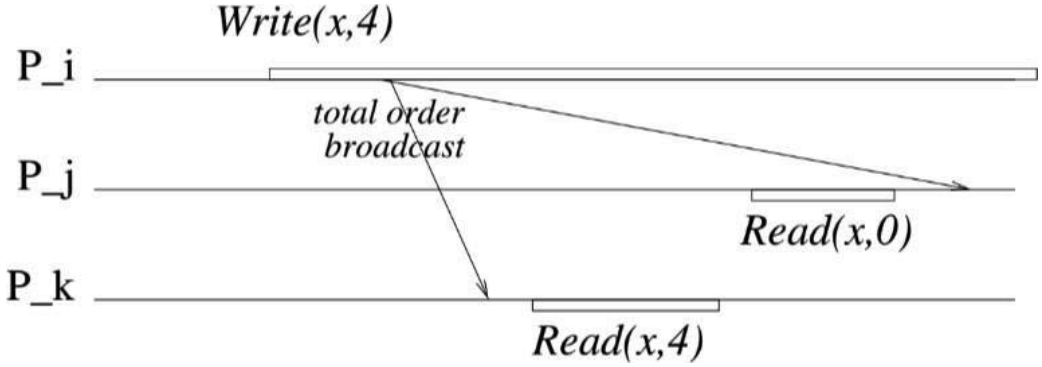
Strict Consistency / Linearizability: Examples
Linearlzability: Implementation

- Simulating global time axis is expensive.
- Assume full replication, and total order broadcast support.

```
(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.
```

Linearizability: Implementation

When a Read in simulated at other processes, there is a no-op. Why do Reads participate in total order broadcasts?
Reads need to be serialized w.r.t. other Reads and all Write operations. See counter-example where Reads do not participate in total order broadcast.



Sequential Consistency

Linearizability or strict/atomic consistency is difficult to implement because the absence of a global time reference in a distributed system necessitates that the time reference has to be simulated. This is very expensive. Programmers can deal with weaker models. The first weaker model, that of *sequential con-sistency* (SC) was proposed by Lamport and uses logical time reference instead of the global time reference.

- The result of any execution is the same as if all operations of the processors were executed in *some* sequential order.
- The operations of each individual processor appear in this sequence in the local program order.

Any interleaving of the operations from the different processors is possible. But all processors must see *the same* interleaving. Even if two operations from different processors (on the same or different variables) do not overlap in a global time scale, they may appear in reverse order in the *common* sequential order seen by all. See examples used for linearizability

Only Writes participate in total order BCs. Reads do not because:

- all consecutive operations by the same processor are ordered in that same order (no pipelining), and
- *Read* operations by different processors are independent of each other; to be ordered only with respect to the *Write* operations.

Direct simplification of the LIN algorithm. Reads executed atomically. Not so for Writes. Suitable for Read-intensive programs.

Sequential Consistency using Local Read Algorithm

(shared var)
int: x;

(1) When the Memory Manager at P_i 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 P_i receives a **total_order_broadcast**$_j$(*Write*, x, *val*) from networ
(2a) **write** *val* to local replica of x;
(2b) **if** $i = j$ **then return** ack to application.

Sequential Consistency using Local Write Algorithm

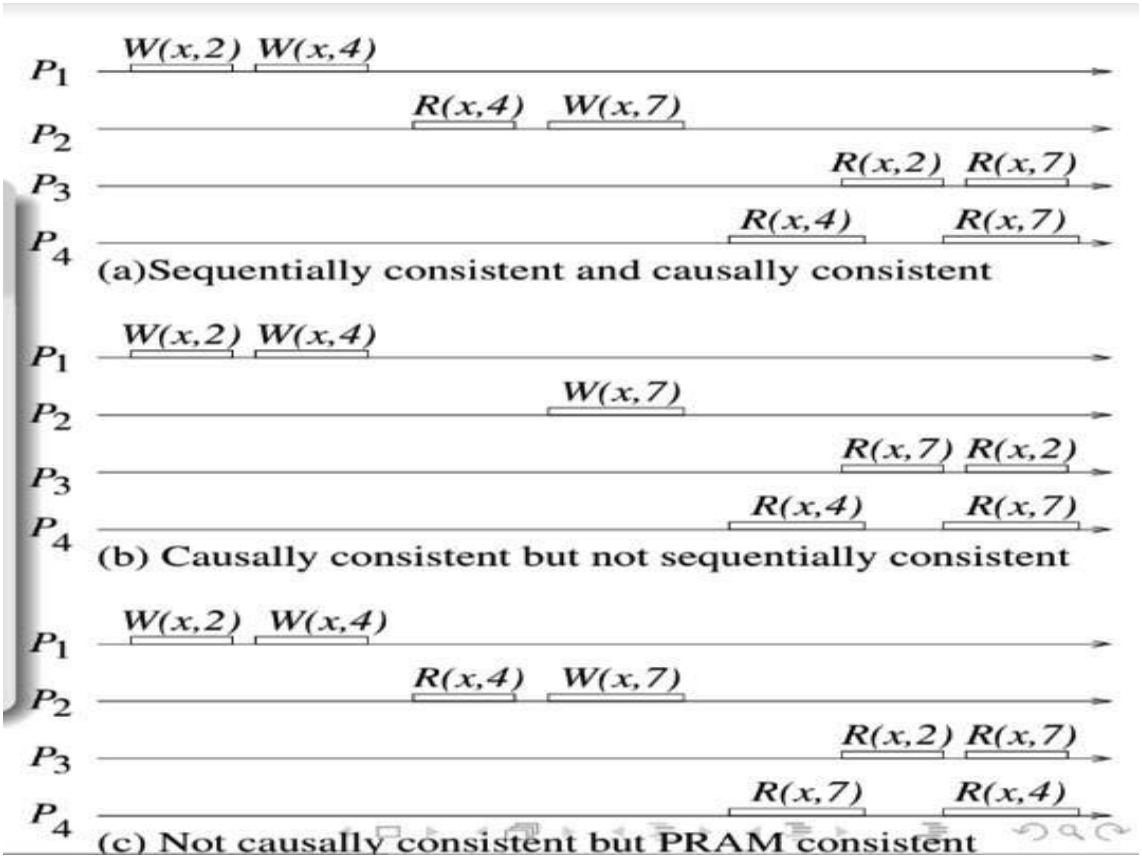
```
(shared var)
int: x;

(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) counter ⟵ counter + 1;
(2b) total_order_broadcast_i the Write(x, val);
(2c) return ack to the application.

(3) When the Memory Manager at P_i receives a total_order_broadcast_j(Write, x, val) from networ
(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.
```

Causal Consistency

In SC, all Write ops should be seen in common order.
For *causal consistency*, only causally *related Writes should be seen in common* [P1] order.

Causal relation for shared memory systems

- At a processor, local order of events is the causal order
- A Write causally precedes Read issued by another processor if the Read returns the value written by the Write.
- The transitive closure of the above two orders is the causal order



Pipelined RAM or Processor Consistency

PRAM memory

- Only Write ops issued by the same processor are seen by others in the order they were issued, but Writes from different processors may be seen by other processors in different orders.

- PRAM can be implemented by FIFO broadcast? PRAM memory can exhibit counter-intuitive behavior, see below.

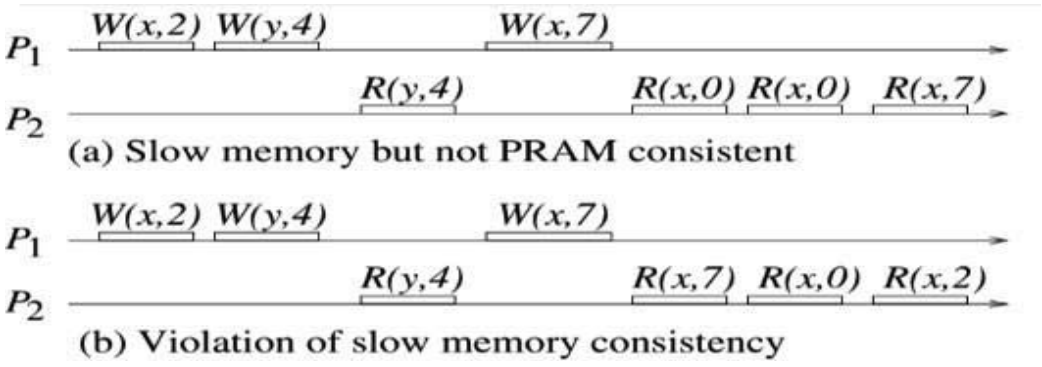
(shared variables)
int: x, y;

Process 1	Process 2
...	...
(1a) $x \longleftarrow 4$;	(2a) $y \longleftarrow 6$;
(1b) **if** $y = 0$ **then kill**(P_2).	(2b) **if** $x = 0$ **then kill**(P_1).

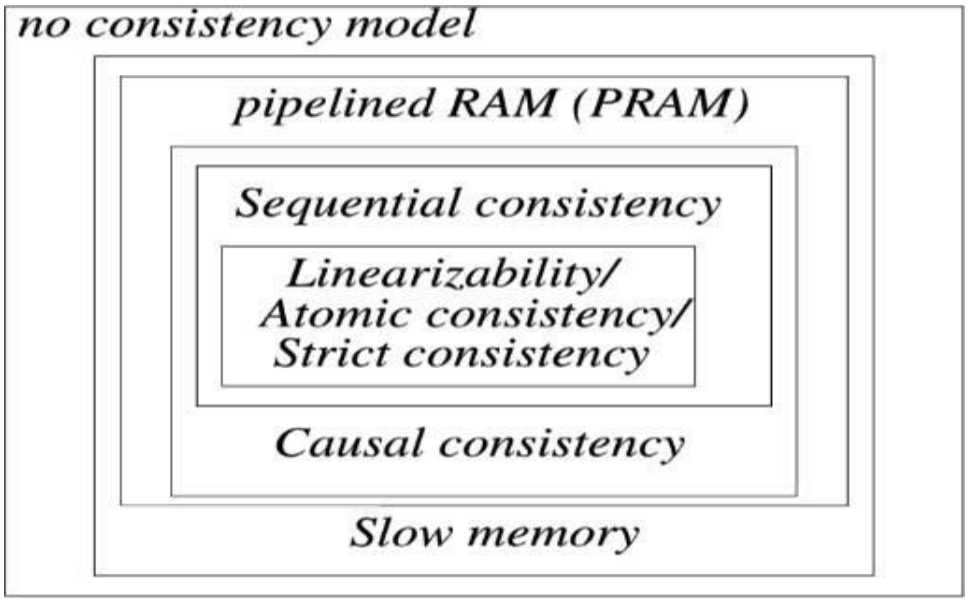
Slow Memory

The next weaker consistency model is that of *slow memory*]. This model represents a location-relative weakening of the PRAM model. In this model, only all *Write* operations issued by the

same processor and to the same memory location must be observed in the same order by all the processors.



Hierarchy of Consistency Models



Synchronization-based Consistency Models: Weak Consistency
Consistency conditions apply only to special synchronization" instructions, e.g.,

barrier synchronization
Non-sync statements may be executed in any order by various processors.
E.g.,weak consistency, release consistency, entry consistency

Weak consistency:
All Writes are propagated to other processes, and all Writes done elsewhere are brought locally, at a sync instruction.

- Accesses to sync variables are sequentially consistent
- Access to sync variable is not permitted unless all Writes elsewhere have completed
- No data access is allowed until all previous synchronization variable accesses have

been performed

Drawback: cannot tell whether beginning access to shared variables (enter CS), or finished access to shared variables (exit CS).

Synchronization based Consistency Models:

Release Consistency and Entry Consistency

Two types of synchronization Variables: *Acquire* and *Release*

Release Consistency

Acquire indicates CS is to be entered. Hence all *Writes* from other processors should be locally reflected at this instruction

Release indicates access to CS is being completed. Hence, all Updates made locally should be propagated to the replicas at other processors.

Acquire and *Release* can be defined on a subset of the variables.

If no CS semantics are used, then Acquire and Release act as barrier synchronization variables.

Lazy release consistency: propagate updates on-demand, not the PRAM way.

Entry Consistency

Each ordinary shared variable is associated with a synchronization variable (e.g., lock, barrier)

For Acquire /Release on a synchronization variable, access to only those ordinary variables guarded by the synchronization variables is performed.

5.8 Shared Memory Mutual Exclusion: Bakery Algorithm

Lamport's bakery algorithm

- Lamport proposed the classical *bakery algorithm* for n-process mutual exclusion in shared memory systems [18]. The algorithm is so called because it mimics the actions that customers follow in a bakery store. A process wanting to enter the critical section picks a token number that is one greater 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 processes may have the same token number. In this case, a unique *lexicographic order* is defined on the tuple token pid , and this dictates the order in which processes enter the critical section. The algorithm for process i is given in Algorithm.
- The algorithm can be shown to satisfy the three requirements of the critical section problem: (i) mutual exclusion, (ii) bounded waiting, and (iii) progress.

```
(shared vars)
array of boolean: choosing[1 . . . n];
array of integer: timestamp[1 . . . n];

repeat
(1) P_i executes the following for the entry section:
(1a) choosing[i] ⟵ 1;
(1b) timestamp[i] ⟵ max_{k∈[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 critical section (CS) after the entry section
(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;
```

Mutual exclusion

) Role of line (1e)? Wait for others' timestamp choice to stabilize ...

❩ Role of line (1f)? Wait for higher priority (lex. lower timestamp) process to enter CS

Bounded waiting: P_i can be overtaken by other processes at most once (each)

Progress: lexicographic order is a total order; process with lowest timestamp in lines (1d)-(1g) enters CS

Space complexity: lower bound of *n* registers Time complexity: (*n*) time for Bakery algorithm

Lamport's WRWR mechanism and fast mutual exclusion

Lamport's fast mutex algorithm takes $O(1)$ time in the absence of contention. However it compromises on bounded waiting. Uses $W(x)$ - $R(y)$ - $W(y)$- $R(x)$ sequence necessary and sufficient to check for contention, and safely enter CS

Lamport's Fast Mutual Exclusion Algorithm

```
(shared variables among the processes)
integer: x, y;                                   // shared register initialized
array of boolean b[1 . . . n];                   // flags to indicate interest in critical section

repeat
(1) Pi (1 ≤ i ≤ n) executes entry section:
(1a)    b[i] ⟵ true;
(1b)    x ⟵ 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;
(1j)            for j = 1 to N do
(1k)                    await ¬b[j];
(1l)            if y ≠ i then
(1m)                    await y = 0;
(1n)                    goto (1a);
(2) Pi (1 ≤ i ≤ n) executes critical section:
(3) Pi (1 ≤ i ≤ n) executes exit section:
(3a)    y ⟵ 0;
(3b)    b[i] ⟵ false;
forever.
```

Shared Memory: Fast Mutual Exclusion Algorithm

Need for a boolean vector of size n: For Pi, there needs to be a trace of its identity and that it had written to the mutex variables. Other processes need to know who (and when) leaves the CS. Hence need for a boolean array b[1..n].

Process P_i	Process P_j	Process P_k	variables
	$W_j(x)$		$\langle x = j, y = 0 \rangle$
$W_i(x)$			$\langle \mathbf{x = i}, y = 0 \rangle$
$R_i(y)$			$\langle x = i, y = 0 \rangle$
	$R_j(y)$		$\langle x = i, y = 0 \rangle$
$W_i(y)$			$\langle x = i, \mathbf{y = i} \rangle$
	$W_j(y)$		$\langle x = i, y = j \rangle$
$R_i(x)$			$\langle x = i, y = j \rangle$
		$W_k(x)$	$\langle x = k, y = j \rangle$
	$R_j(x)$		$\langle x = k, y = j \rangle$

Examine all possible race conditions in algorithm code to analyze the algorithm.

Hardware Support for Mutual Exclusion

Hardware support can allow for special instructions that perform two or more operations atomically.
Test&Set and Swap are each executed atomically!!

Definitions of synchronization operations Test&Set and Swap.

Test&Set and *Swap* are each executed atomically!!

```
(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) returns value:
(1a) old ⟵ Reg;
(1b) Reg ⟵ 1;
(1c) return(old).

(2) Swap(Reg, new) returns value:
(2a) old ⟵ Reg;
(2b) Reg ⟵ new;
(2c) return(old).
```

Mutual Exclusion using Swap

```
(shared variables)
register: Reg ⟵ false;                                  // shared register initialized
(local variables)
integer: blocked ⟵ 0;                                   // variable to be checked before entering CS

repeat
(1) Pi executes the following for the entry section:
(1a) blocked ⟵ true;
(1b) repeat
(1c)      Swap(Reg, blocked);
(1d) until blocked = false;
(2) Pi executes the critical section (CS) after the entry section
(3) Pi executes the following exit section after the CS:
(3a) Reg ⟵ false;
(4) Pi executes the remainder section after the exit section
until false;
```

Mutual Exclusion using *Test&Set*, with Bounded Waiting

```
(shared variables)
register: Reg ⟵ false;                                   // shared register initialized
array of boolean: waiting[1 . . . n];
(local variables)
integer: blocked ⟵ initial value;                        // value to be checked before entering CS

repeat
(1) Pi executes the following for the entry section:
(1a) waiting[i] ⟵ true;
(1b) blocked ⟵ true;
(1c) while waiting[i] and blocked do
(1d)      blocked ⟵ Test&Set(Reg);
(1e) waiting[i] ⟵ false;
(2) Pi executes the critical section (CS) after the entry section
(3) Pi executes the following exit section after the CS:
(3a) next ⟵ (i + 1)mod n;
(3b) while next ≠ i and waiting[next] = false do
(3c)      next ⟵ (next + 1)mod n;
(3d) if next = i then
(3e)      Reg ⟵ false;
(3f) else waiting[next] ⟵ false;
(4) Pi executes the remainder section after the exit section
until false;
```

Code shown is for process Pi, $1 \leq i \leq n$.

UNIT-4 QUESTIONS

What is rollback? and explain the several types of messages for rollback. (13)
Examine briefly about global states with examples. (13)
Describe the issues involved in a failure recovery with the help of a distributed computation. (13)
Elaborate the various checkpoint-based rollback-recovery techniques.(13)
Describe the pessimistic logging , optimistic logging and casual logging.(13)
What are min-process check pointing algorithms? Explain it detail.(7) Examine Deterministic and non-deterministic events. (6)
Summarize the koo–toueg coordinated check pointing algorithm.(7) Explain the rollback recovery algorithm. (6)
. **Demonstrate** in detail about the juang–venkatesan algorithm for asynchronous check pointing and recovery.(13)

. **Discuss** in detail about some assumptions underlying the study of agreement algorithms. (13)
. What is byzantine agreement problem? **Explain** the two popular flavours of the byzantine agreement problem.
. **Develop** an overview of the results and lower bounds on solving the consensus problem under different assumptions.
. **Explain** agreement in (message-passing) synchronous systems with failures.(13)
. Give byzantine agreement tree algorithm and illustrate with an example. (13)
. Analyze on phase-king algorithm for consensus.(13)
. **Design** a system model of distributed system consisting of four processes and explain the interactions with the outside world.(15)
. **Explain** with examples of consistent and inconsistent states of a distributed system.(15)
. Consider the following simple check pointing algorithm. A process takes a local checkpoint right after sending a message. **Create** that the last checkpoint at all processes will always be . consistent. What are the trade-offs with this method?(15)
. Give and **analyse** a rigorous proof of the impossibility of a min- process, non blocking check pointing algorithm.(15)

UNIT 5

Explain the structured overlays and unstructured overlays in distributed indexing. (13)
i) **What** is meant by napster legacy? Explain.(7) Give a brief account on Indexing mechanisms. (6
Examine the chord protocol with simple key lookup algorithm.(13)
Illustrate in detail about A scalable object location algorithm in chord.(13)
Discuss on managing churn in chord.(13)
Describe briefly about the following: i) Content-Addressable Network (CAN) initialization (6) ii) CAN routing (7).
Point out tapestry P2P overlay network and its routing with an example. (13)
Discuss the CAN maintenance and CAN optimizations. (13)
State about the consistency models: entry consistency, weak consistency, and release consistency.(13)
Summarize in detail how node insertion and node deletion are applied in tapestry. (13)
i) Illustrate the advantages and disadvantages of DSM.(6) **ii)** Point out the main issues in designing a DSM system (7)

Examine how to implement linearizability (LIN) using total order broadcasts.(13)
Analyse how to implement Sequential consistency in a distributed system.(13)
Describe lamport's bakery algorithm lamport's WRWR mechanism and fast mutual exclusion. (13)
User 'A' in delhi wishes to send a file for printing to user 'B' in florida, whose system is connected to a printer; while user 'C' from tokyo wants to save a video file in the hard disk of user 'D' in london. **Analyze** and discuss the required peer-to-peer network architecture.(15)
Evaluate a formal proof to justify the correctness of algorithm that implements sequential consistency using local read operations.(15)
Develop a detailed implementation of causal consistency, and provide a correctness argument for your implementation.(15)