Outline

- Introduction & architectural issues
- Data distribution
- Distributed query processing
- Distributed query optimization
- Distributed transactions & concurrency control
- Distributed reliability
- Data replication
- Parallel database systems
- Database integration & querying
- Peer-to-Peer data management
  - P2P Infrastructure
  - Schema mapping
  - Querying
  - Replication
- Stream data management
- MapReduce-based distributed data management

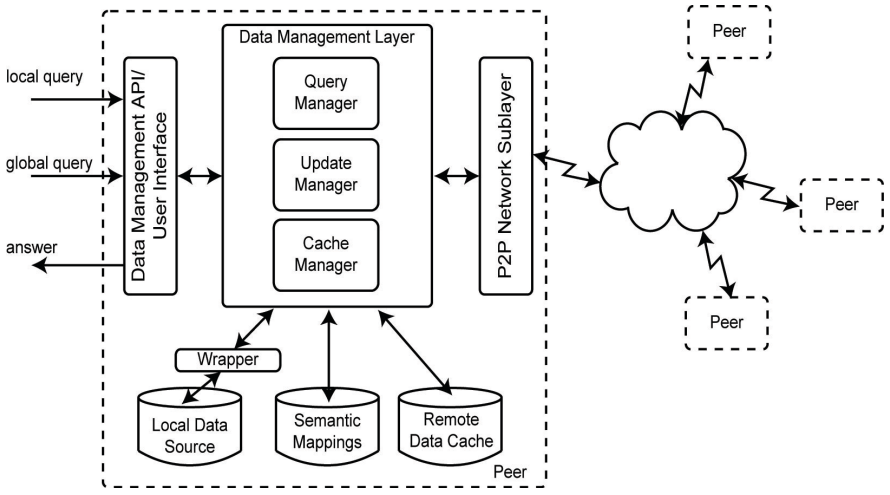
Motivations

- P2P systems
  - Each peer can have same functionality
  - Decentralized control, large scale
  - Low-level, simple services
    - File sharing, computation sharing, com. sharing
- Traditional distributed DBMSs
  - High-level data management services
    - queries, transactions, consistency, security, etc.
  - Centralized control, limited scale
- P2P + distributed database

Problem Definition

- P2P system
  - No centralized control, very large scale
  - Very dynamic: peers can join and leave the network at any time
  - Peers can be autonomous and unreliable
- Techniques designed for distributed data management need be extended
  - Too static, need to be decentralized, dynamic and self-adaptive

Peer Reference Architecture



Potential Benefits of P2P Systems

- Scale up to very large numbers of peers
- Dynamic self-organization
- Load balancing
- Parallel processing
- High availability through massive replication

P2P vs Traditional Distributed DBMS

	P2P	Distributed DBMS
Joining the network	Upon peer's initiative	Controled by DBA
Queries	No schema, key-word based	Global schema, static optimization
Query answers	Partial	Complete
Content location	Using neighbors or DHT	Using directory

Requirements for P2P Data Management

- Autonomy of peers
  - Peers should be able to join/leave at any time, control their data with respect to other (trusted) peers
- Query expressiveness
  - Key-lookup, key-word search, SQL-like
- Efficiency
  - Efficient use of bandwidth, computing power, storage

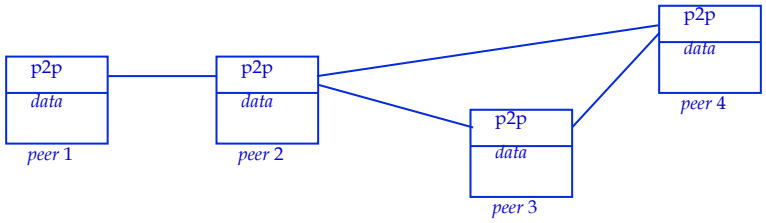
Requirements for P2P Data Management (cont'd)

- Quality of service (QoS)
  - User-perceived efficiency: completeness of results, response time, data consistency, …
- Fault-tolerance
  - Efficiency and QoS despite failures
- Security
  - Data access control in the context of very open systems

P2P Network Topologies

- Pure P2P systems
  - Unstructured systems
    - e.g. Napster, Gnutella, Freenet, Kazaa, BitTorrent
  - Structured systems (DHT)
    - e.g. LH* (the earliest form of DHT), CAN, CHORD, Tapestry, Freepastry, Pgrid, Baton
- Super-peer (hybrid) systems
  - e.g. Edutela, JXTA
- Two issues
  - Indexing data
  - Searching data

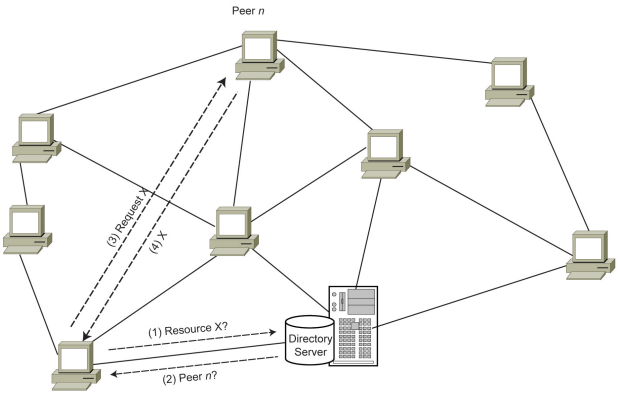
P2P Unstructured Network



- High autonomy (peer only needs to know neighbor to login)
- Searching by
  - flooding the network: general, may be inefficient
  - Gossiping between selected peers: robust, efficient
- High-fault tolerance with replication

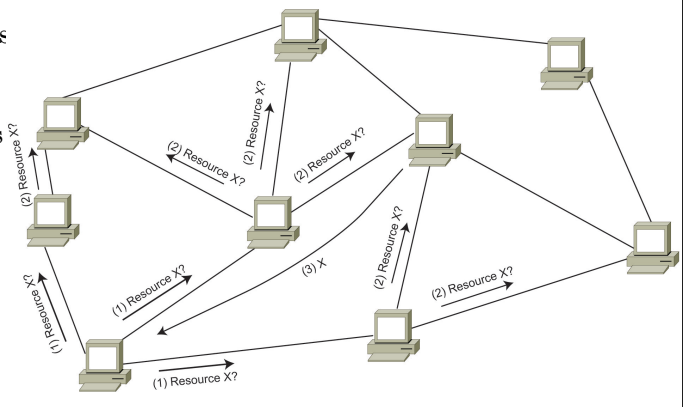
Search over Centralized Index

1. A peer asks the central index manager for resource
2. The response identifies the peer with the resource
3. The peer is asked for the resource
4. It is transferred

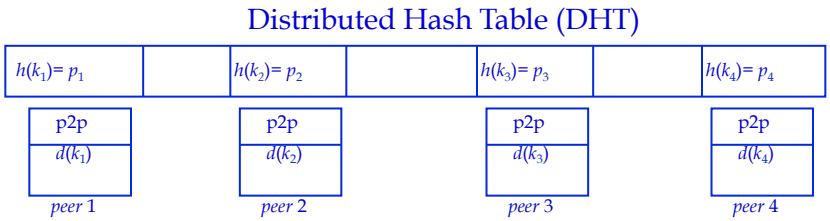


Search over Distributed Index

1. A peer sends the request for resource to all its neighbors
2. Each neighbor propagates to its neighbors if it doesn't have the resource
3. The peer who has the resource responds by sending the resource

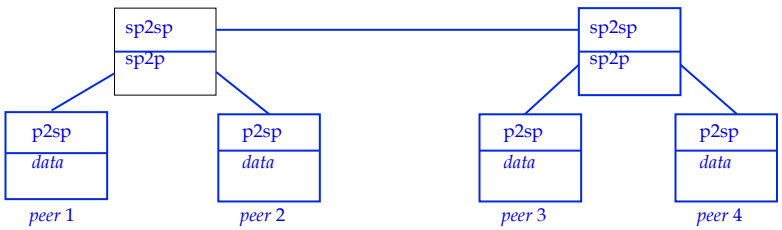


P2P Structured Network



- Simple API with `put(key, data)` and `get(key)`
  - The key (an object id) is hashed to generate a peer id, which stores the corresponding data
- Efficient exact-match search
  - *O*(*log n*) for put(key, data), get(key)
- Limited autonomy since a peer is responsible for a range of keys

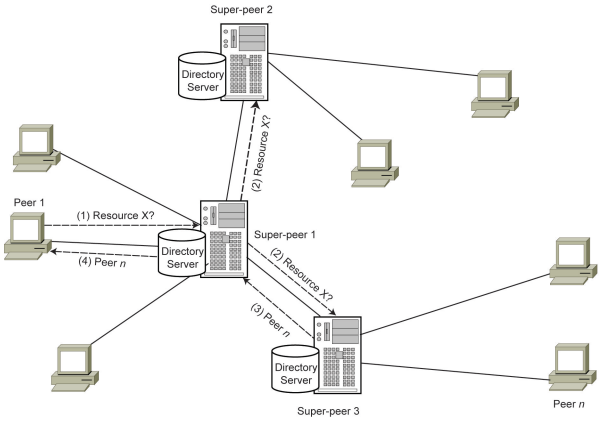
Super-peer Network



- Super-peers can perform complex functions (meta-data management, indexing, access control, etc.)
  - Efficiency and QoS
  - Restricted autonomy
  - SP = single point of failure ⇒ use several super-peers

Search over a Super-peer System

1. A peer sends the request for resource to all its super-peer
2. The super-peer sends the request to other super-peers if necessary
3. The super-peer one of whose peers has the resource responds by indicating that peer
4. The super-peer notifies the original peer



P2P Systems Comparison

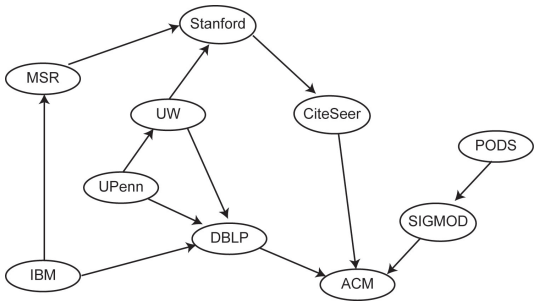
Requirements	Unstructured	DHT	Super-peer
Autonomy	high	low	avg
Query exp.	high	low	high
Efficiency	low	high	high
QoS	low	high	high
Fault-tolerance	high	high	low
Security	low	low	high

P2P Schema Mapping

- Problem: support decentralized schema mapping so that a query expressed on one peer's schema can be reformulated to a query on another peer's schema
- Main approaches
  - Pairwise schema mapping
  - Mapping based on machine learning
  - Common agreement mapping

Pairwise Schema Mapping

- Each user defines the mapping between the local schema and the schema of any other peer that contains data that are of interest
- Relying on the transitivity of the defined mappings, the system tries to extract mappings between schemas that have no defined mapping

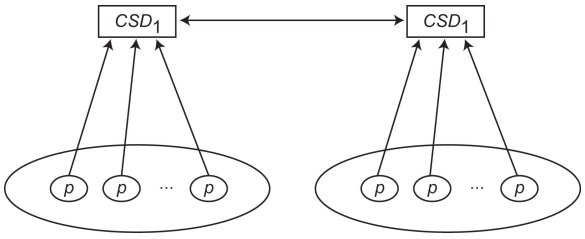


Mapping Based on Machine Learning

- This approach is generally used when the shared data are defined based on ontologies and taxonomies as proposed for the semantic web
- It uses machine learning techniques to automatically extract the mappings between the shared schemas
- The extracted mappings are stored over the network, in order to be used for processing future queries

Common Agreement Mapping

- Some (cooperating) peers must agree on a Common Schema Description (CSD)
- Given a CSD, a peer schema can be specified using views
  - Similar to the LAV approach
- When a peer decides to share data, it needs to map its local schema to the CSD



Querying over P2P Systems

- P2P networks provide basic query routing
  - Sufficient for simple, exact-match queries, e.g. with a DHT
- Supporting more complex queries, particularly in DHTs, is difficult
- Main types of complex queries
  - Top-k queries
  - Join queries
  - Range queries

Top-k Query

- Returns only *k* of the most relevant answers, ordered by relevance
  - Like for a search engine
- Scoring function (sf) determines the relevance (*score*) of answers to the query
- Example

```
SELECT    *
FROM      Patient    P
WHERE     (P.disease = "diabetes") AND
          (P.height < 170) AND (P.weight > 70)
ORDER BY  sf(height, weight)
STOP AFTER 10
```

  - Example of sf: *weight* − (*height -100*)

General Model for Top-k Queries

- Suppose we have:
  - n data items
    - items can be document, tuples, etc.
  - m lists of n data items such that
    - Each data item has
      - a local score in each list
      - an overall score computed based on its local scores in all lists using a given scoring function
    - Each list
      - contains all n data items (or item ids)
      - is sorted in decreasing order of the local scores
- The objective is:
  - *Given a scoring function, find the k data items whose overall scores are the highest*

Execution Cost of Top-k Queries

- Two modes of access to a sorted list
  - Sorted (sequential) access
    - Starts with the first data item, then accesses each next item
  - Random access
    - Looks up a given data item in the list by its identifier (e.g. TID)
- Given a top-k algorithm A and a database D (*i.e. set of sorted lists*), the cost of executing A over D is:
  - $Cost(A, D)$ = (#sorted-access * sorted-access-cost) + (#random-access * random-access-cost)

Basic Top-k Algorithms

- Fagin's Algorithm (FA)
  - General model of top-k queries using sorted lists
  - Simple algorithm
    - Do sorted access in parallel to the lists until *at least k data items have been seen in all lists*
- Threshold Algorithm (TA)
  - Proposed independently by several groups
  - Efficient algorithm over sorted lists
  - The basis for many TA-style distributed algorithms
    - Mainly for DHTs
    - Algorithms for unstructured or super-peer simpler

TA

- Similar to FA in doing sorted access to the lists
  - But different stopping condition
- Unlike FA, no need to wait until the lists give k items
  - Once an item has been seen from a sorted access, get all its scores through random access
- *But how do we know that the scores of seen items are higher than those of unseen items?*
  - Use a *threshold* (T) to predict maximum possible score of unseen items
    - based on the last scores seen in the lists under sorted access
  - Then stop when there are at least k data items whose overall score $\geq T$

TA Example

- Assume sf( ) = $s_1 + s_2 + s_3$, $k = 3$, Y: {top seen items with overall scores}
- At position 1
  - Look up the local scores of items d_1, d_2 and d_3 in other lists using random access and compute their overall scores (which are 65, 63 and 70, respectively)
  - $Y = \{(d_1,70)\ (d_2,65)\ (d_3,63)\}$, $T = 30 + 28 + 30 = 88$
- Then
  - At position 2, $Y = \{(d_3,70)\ (d_4,70)\ (d_5,65)\}$, $T = 84$
  - At position 3, $Y = \{(d_3,71)\ (d_5,70)\ (d_8,70)\}$, $T = 80$
  - At position 4, Y = same, $T = 75$
  - At position 5, Y = same, $T = 72$
  - At position 6, Y = same, $T = 63$
    - which is less than the overall score of the three data items in Y. Thus, TA stops
- Note that the contents of Y at position 6 is exactly the same as at position 3

Position	List 1		List 2		List 3	
	Data Item	Local score s_1	Data Item	Local score s_2	Data Item	Local score s_3
1	d_1	30	d_2	28	d_3	30
2	d_4	28	d_6	27	d_5	29
3	d_9	27	d_7	25	d_8	28
4	d_3	26	d_5	24	d_4	25
5	d_7	25	d_9	23	d_2	24
6	d_8	23	d_1	21	d_6	19
7	d_5	17	d_8	20	d_{13}	15
8	d_6	14	d_3	14	d_1	14
9	d_2	11	d_4	13	d_9	12
10	d_{11}	10	d_{14}	12	d_7	11
...	...	...	...	...	...	...

Improvement over TA: BPA

- Best Position Algorithm
- Main idea: *keep track of the positions (and scores) of the items seen under sorted or random access*
  - Enables BPA to stop as soon as possible
    - In the previous example, BPA stops at position 3
- Best position = the greatest seen position in a list such that any position before it is also seen
  - Thus, we are sure that all positions between 1 and *best position* have been seen
- Stopping condition
  - Based on *best positions overall score*, i.e. the overall score computed based on the best positions in all lists

Join Query Processing in DHTs

- A DHT relies on hashing to store and locate data
  - Basis for parallel hash join algorithms
- Basic solution in the context of the PIER P2P system
  - Let us call it PIERjoin
  - Assume that the joined relations and the result relations have a *home* which are the peers that store horizontal fragments of the relation
    - Recall def. of home from Chapter 8
  - Make use of the put method for distributing tuples onto a set of peers based on their join attribute so that tuples with the same join attribute values are stored at the same peers
  - Then apply the probe/join phase

PIERjoin Algorithm

1. Multicast phase
   - The query originator peer multicasts Q to all peers that store tuples of the join relations R and S, i.e., their homes.
2. Hash phase
   - Each peer that receives Q scans its local relation, searching for the tuples that satisfy the select predicate (if any)
   - Then, it sends the selected tuples to the home of the result relation, using put operations
   - The DHT key used in the put operation uses the home of the result relation and the join attribute
3. Probe/join phase
   - Each peer in the home of the result relation, upon receiving a new tuple, inserts it in the corresponding hash table, probes the opposite hash table to find matching tuples and constructs the result joined tuples

Range Query Processing

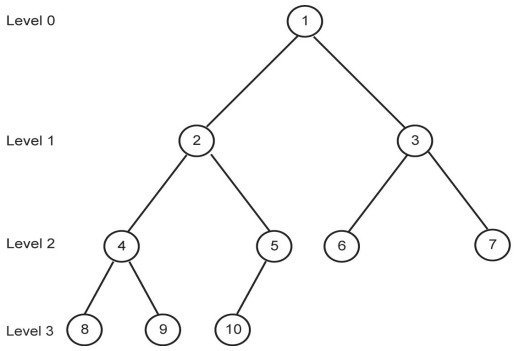
- Range query
  - WHERE clause of the form "attribute A in range [*a*; *b*]"
- Difficult to support in structured P2P systems, in particular, DHTs
  - Hashing tends to destroy the ordering of data that is useful in finding ranges quickly
- Two main approaches for supporting range queries in structured P2P systems
  - Extend a DHT with proximity or order-preserving properties
    - Problem: data skew that can result in peers with unbalanced ranges, which hurts load balancing
  - Maintain the key ordering with a tree-based structure
    - Better at maintaining balanced ranges of keys

BATON

- BATON (BAlanced Tree Overlay Network)
- Organizes peers as a balanced binary tree
  - Each node of the tree is maintained by a peer
  - The position of a node is determined by a (level, number) tuple, with level starting from 0 at the root, number starting from 1 at the root and sequentially assigned using in-order traversal
  - Each tree node stores links to its parent, children, adjacent nodes and selected neighbor nodes that are nodes at the same level
  - Two routing tables: a left routing table and a right routing table store links to the selected neighbor nodes

BATON Structure-tree Index

- Each node (or peer) is assigned a range of values
  - Maintained at the routing table of each link
  - Required to be to the right of the range managed by its left subtree and less than the range managed by its right subtree



Node 6: level 2, number=3
parent=3, leftchild=null, rightchild=null
leftadjacent=1, rightadjacent=3

Left routing table

	Node	Left Child	Right Child	Lower Bound	Upper Bound
0	5	10	null	LB5	UB5
1	4	8	9	LB4	UB4

Right routing table

	Node	Left Child	Right Child	Lower Bound	Upper Bound
0	7	null	null	LB7	UB7

Range Query Processing in BATON

Input: Q, a range query in the form $[a,b]$
Output: T: result relation

1. Search for the peer storing the lower bound of the range

   At query originator node do
   find peer p that holds value a
   send Q to p

2. A peer p that receives Q (from query originator or its left adjacent

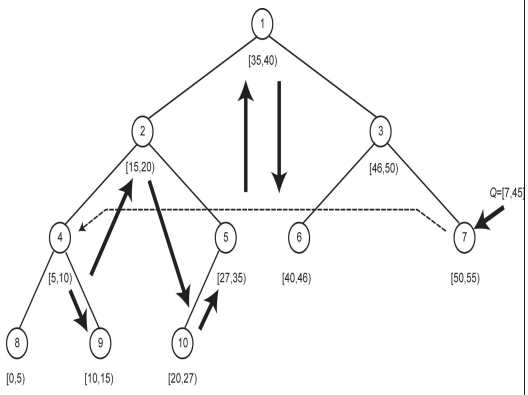
peer) searches for local tuples and sends Q to its right adjacent node

At each peer p that receives Q
Tp = Range(p) $\cap$ $[a,b]$
send Tp to query originator
If (Range(RighAdjacent(p)) $\cap$ $[a,b]$) not empty
send Q to right adjacent peer of p

- With X nodes covering the range, Q is answered *in* $O(\log n + X)$ steps

Example of Range Query Execution

- Consider Q with range [7; 45] issued at node 7
- First, execute an exact match query looking for a node containing the lower bound of the range (see dashed line)
- Since the lower bound is in node 4's range, check locally for tuples belonging to the range and forward Q to its adjacent right node (node 9)
- Node 9 checks for local tuples belonging to the range and forwards Q to node 2
- Nodes 10, 5, 1 and 6 receive Q, check for local tuples and contact their respective right adjacent node until the node containing the upper bound of the range is reached



Replica Consistency

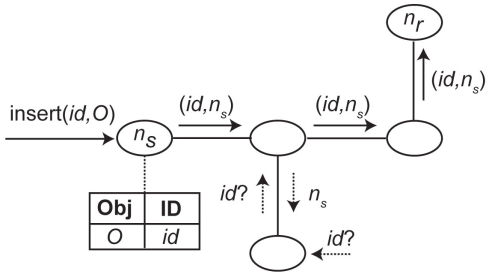
- To increase data availability and access performance, P2P systems replicate data, however, with very different levels of replica consistency
  - The earlier, simple P2P systems such as Gnutella and Kazaa deal only with static data (e.g., music files) and replication is "passive" as it occurs naturally as peers request and copy files from one another (basically, caching data)
- In more advanced P2P systems where replicas can be updated, there is a need for proper replica management techniques
- Replica consistency in DHTs
  - Basic support - Tapestry
  - Replica reconciliation - OceanStore

Tapestry

- Decentralized object location and routing on top of a structured overlay
- Routes messages to logical end-points (i.e., not associated with physical location), such as nodes or object replicas.
  - This enables message delivery to mobile or replicated endpoints in the presence of network instability
- Location and routing
  - Let O be an object identified by $id(O)$, the insertion of O involves two nodes: the server node (noted n_s) that holds O and the root node (noted n_r) that holds a mapping in the form $(id(O); n_s)$ indicating that the object identified by $id(O)$ is stored at node n_s
  - The root node is dynamically determined by a globally consistent deterministic algorithm

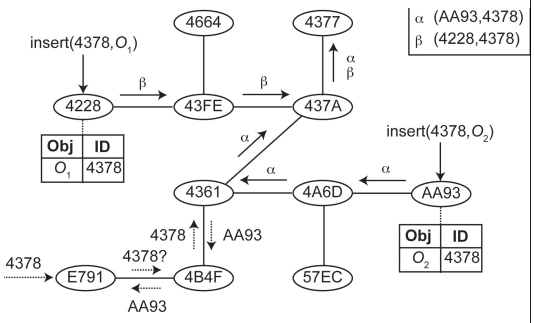
Object Publishing in Tapestry

- When O is inserted into n_s, n_s publishes $id(O)$ at its root node by routing a message from n_s to n_r containing the mapping $(id(O); n_s)$
- This mapping is stored at all nodes along the message path
- During a location query (e.g., "$id(O)$?", the message that looks for $id(O)$ is initially routed towards n_r, but it may be stopped before reaching it once a node containing the mapping $(id(O); n_s)$ is found
- For routing a message to $id(O)$'s root, each node forwards this message to its neighbor whose logical identifier is the most similar to $id(O)$



Replica Management in Tapestry

- Each node represents a peer and contains the peer's logical identifier in hexadecimal format
- Two replicas O_1 and O_2 of object O (e.g., a book file) are inserted into distinct peers (O_1 at peer 4228 and O_2 at peer AA93). The identifier of O_1 is equal to that of O_2 (i.e., 4378)
- When O_1 is inserted into its server node (peer 4228), the mapping (4378; 4228) is routed from peer 4228 to peer 4377 (the root node for O_1's identifier)
- As the message approaches the root node, the object and the node identifiers become increasingly similar
- In addition, the mapping (4378; 4228) is stored at all peers along the message path

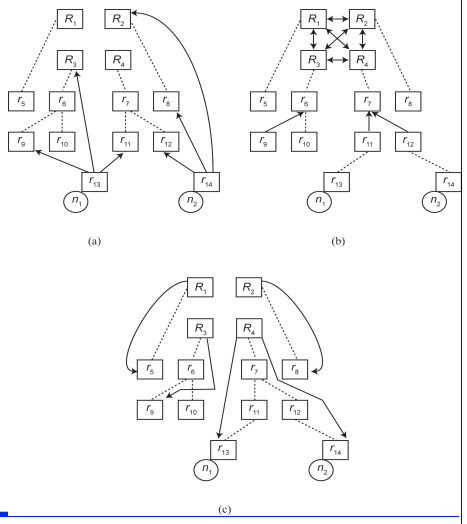


OceanStore

- OceanStore is a data management system designed to provide continuous access to persistent information
- Relies on Tapestry and assumes untrusted powerful servers connected by high-speed links
- To improve performance, data are allowed to be cached anywhere, anytime
- Allows concurrent updates on replicated objects; it relies on reconciliation to assure data consistency

Reconciliation in OceanStore

- R_i and r_i denote, respectively, a primary and a secondary copy of object R
- Nodes n_1 and n_2 are concurrently updating R as follows
  - Nodes that hold primary copies of R, called the master group of R, are responsible for ordering updates
  - (a) n_1 and n_2 perform tentative updates on their local secondary replicas and send these updates to the master group of R as well as to other random secondary replicas
  - (b) The tentative updates are ordered by the master group based on timestamps assigned by n_1 and n_2, and epidemically propagated among secondary replicas
  - (c) Once the master group obtains an agreement, the result of updates is multicast to secondary replicas



Conclusion

- Advanced P2P applications will need high-level data management services
- Various P2P networks will improve
  - Network-independence crucial to exploit and combine them
- Many technical issues
  - Decentralized schema management, complex query processing, transaction support and replication, and data privacy
- Important to characterize applications that can most benefit from P2P with respect to other distributed architectures