Principles of Distributed Database Systems

TS. Phan Thị Hà

Outline

- Introduction
- Distributed and Parallel Database Design
- Distributed Data Control
- Distributed Query Processing
- Distributed Transaction Processing
- Data Replication
- Database Integration – Multidatabase Systems
- Parallel Database Systems
- Peer-to-Peer Data Management
- Big Data Processing
- NoSQL, NewSQL and Polystores
- Web Data Management

Outline

- Peer-to-Peer Data Management
  - P2P infrastructure
  - Schema mapping
  - Querying over P2P systems
  - Replication
  - Blockchain

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
  - Why? How?

Why High-level P2P Data Sharing?

- Professional community example
  - Medical doctors in a hospital may want to share (some of) their patient data for an epidemiological study
  - They have their own, independent patient descriptions
  - They want to ask queries such as “age and weight of male patients diagnosed with disease X …” over their own descriptions
  - They don’t want to create a database and set a centralized server

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

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

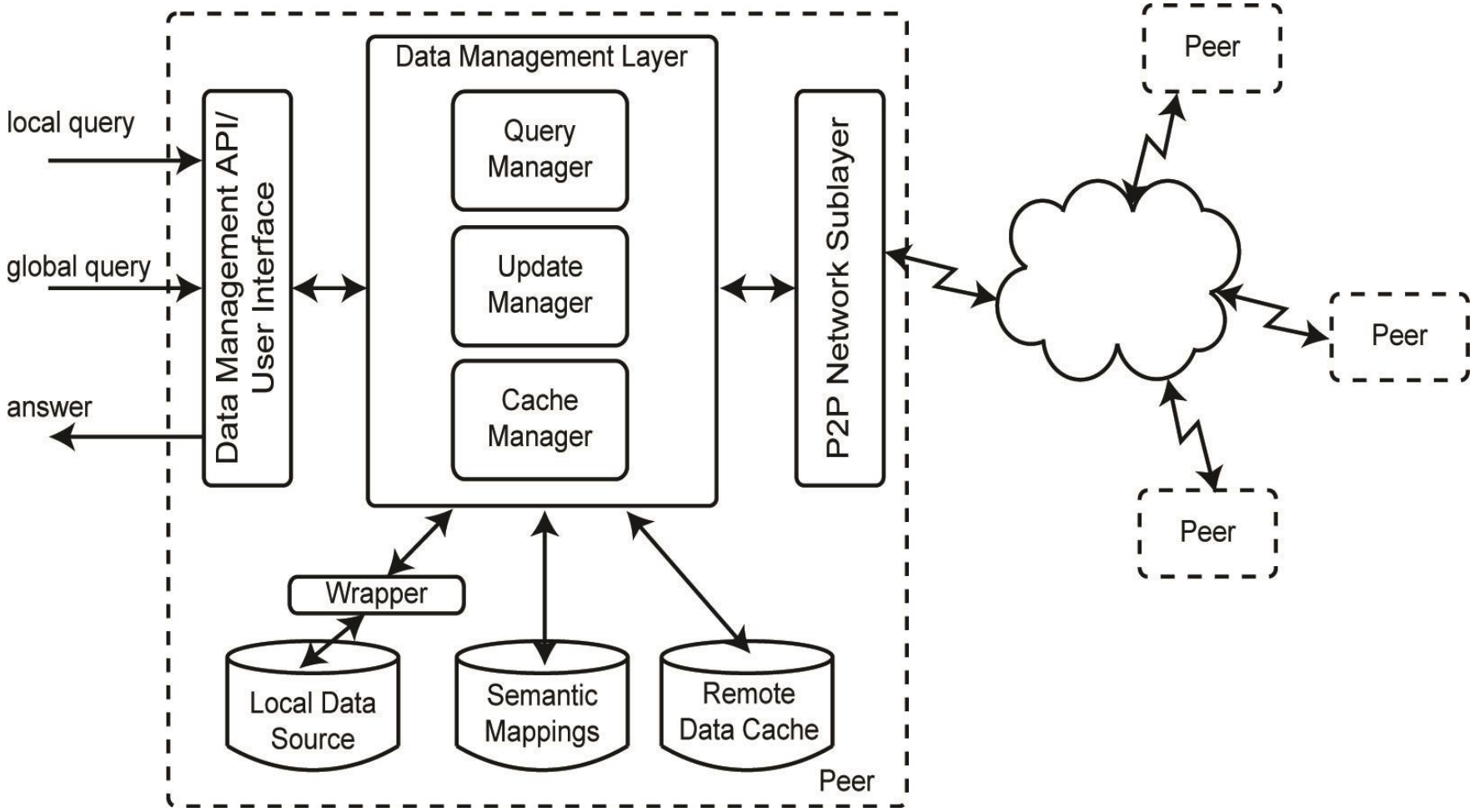
Requirements for P2P Data Management (cont'd)

- 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

Outline

- Peer-to-Peer Data Management
  - P2P infrastructure
  - Schema mapping
  - Querying over P2P systems
  - Replication
  - Blockchain

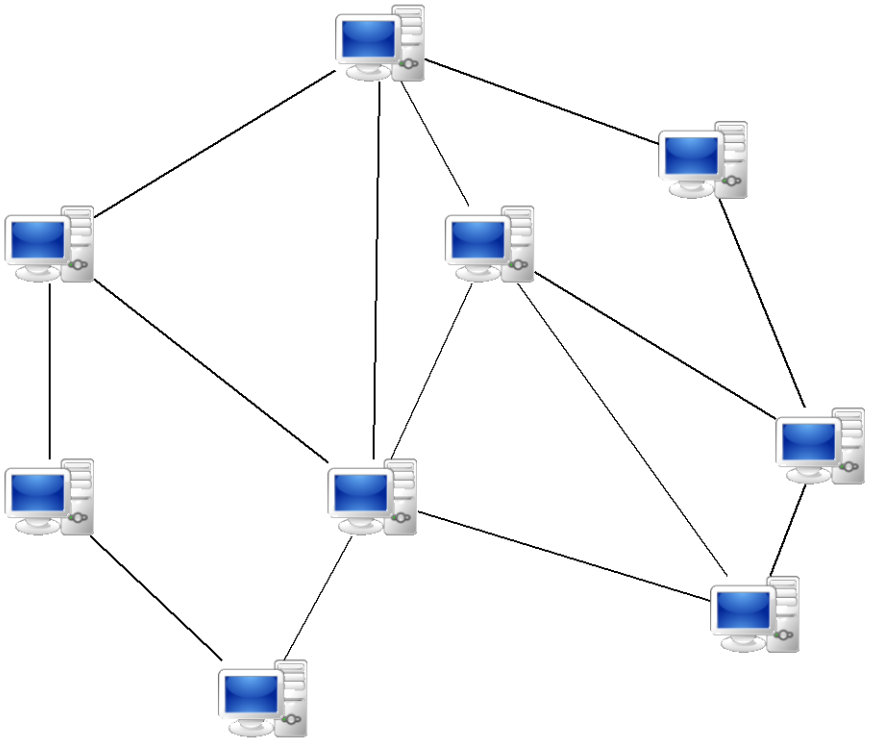
Peer Reference Architecture



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

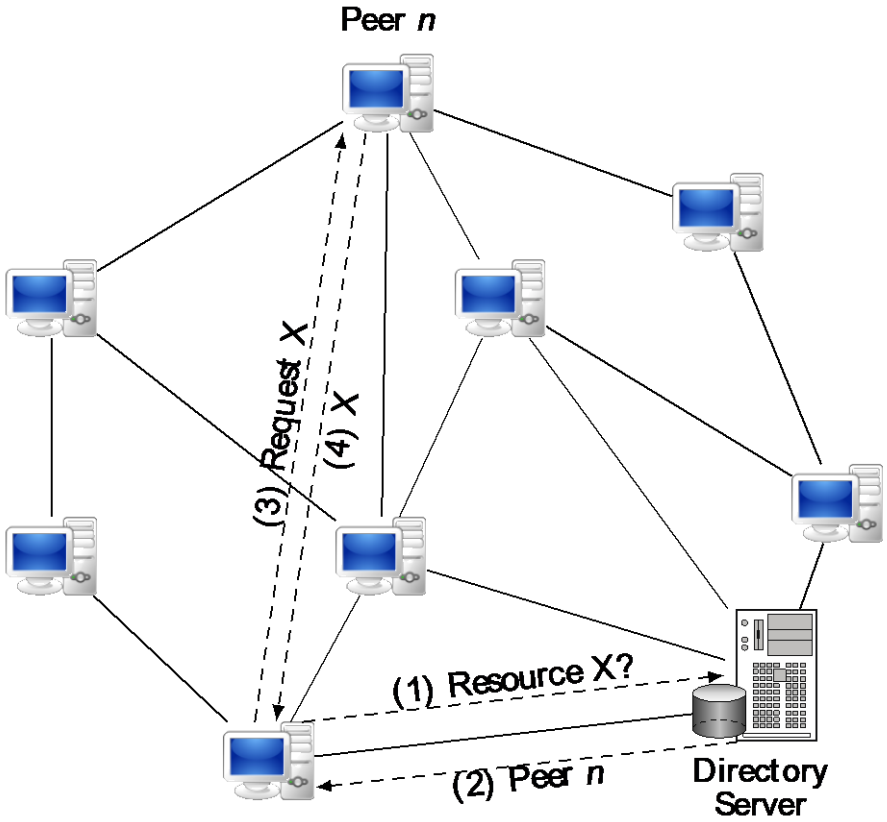
P2P Unstructured Network



- 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

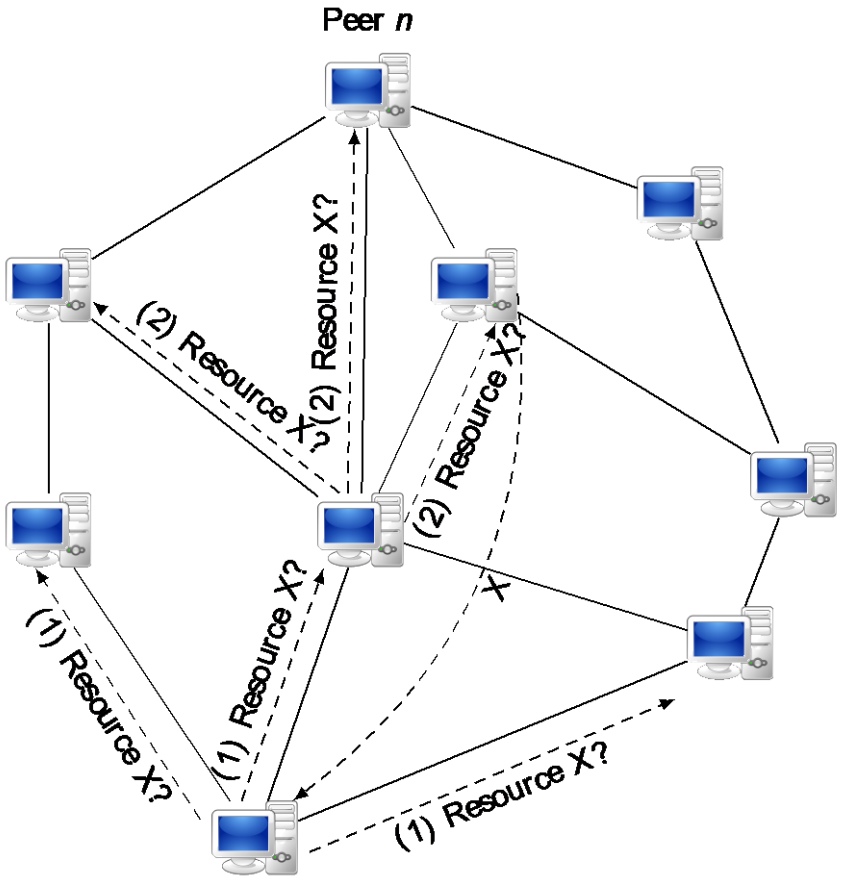
Search over Centralized Index

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

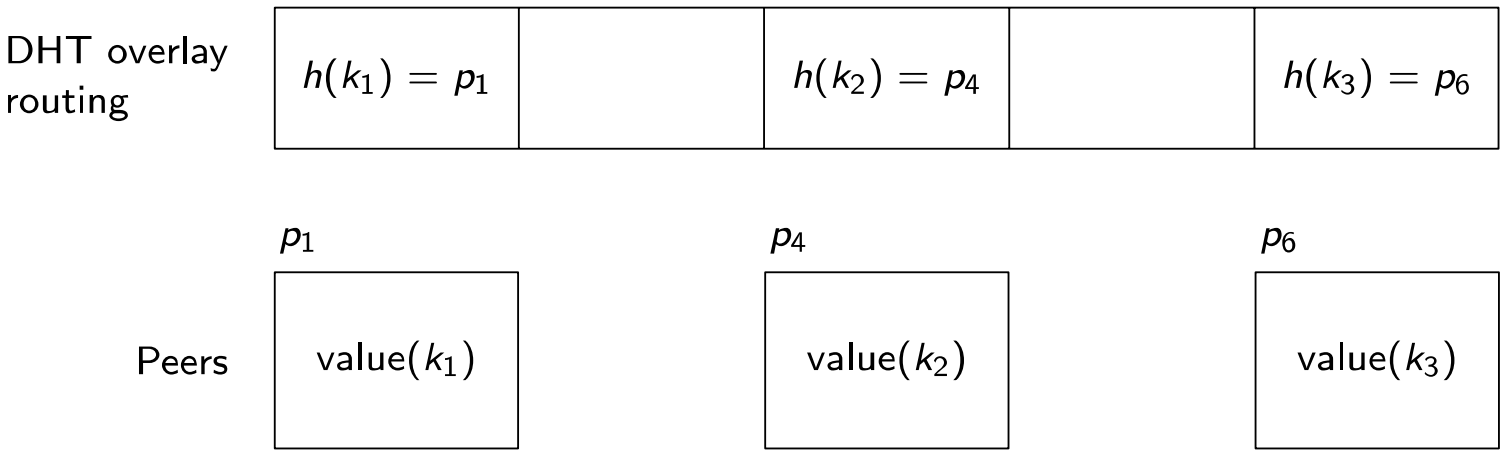


Search over Distributed Index

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



Structured P2P Network



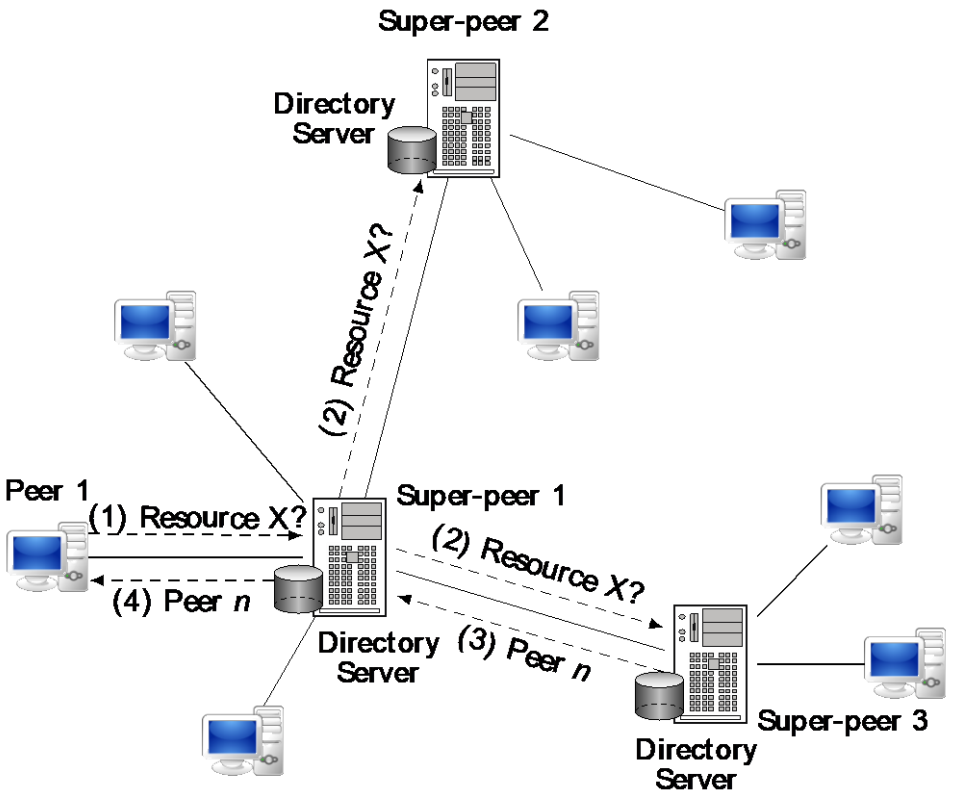
- Simple API with put(key, data) and get(key)
  - The key (an object id) is hashed to generate a peer id, which stores the corresponding data
- Efficient exact-match search
- *O*(*log n*) for put(key, data), get(key)
- Limited autonomy since a peer is responsible for a range of keys

Super-peer Network

- Not all peers are equal
- Super-peers can perform complex functions (meta-data management, indexing, access control, etc.)
  - Efficiency and QoS
  - Restricted autonomy
  - SP = single point of failure $\Rightarrow$ 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	Structured	Superpeer
Autonomy	Low	Low	Moderate
Query expressiveness	High	Low	High
Efficiency	Low	High	High
QoS	Low	High	High
Fault-tolerance	High	High	Low
Security	Low	Low	High

Outline

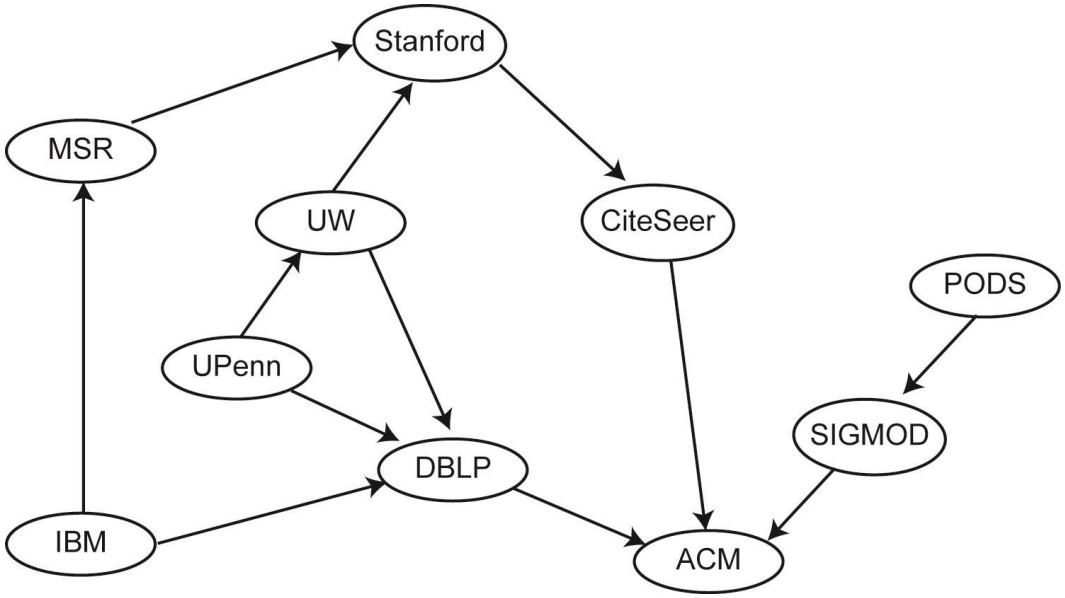
- Peer-to-Peer Data Management
  - P2P infrastructure
  - Schema mapping
  - Querying over P2P systems
  - Replication
  - Blockchain

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

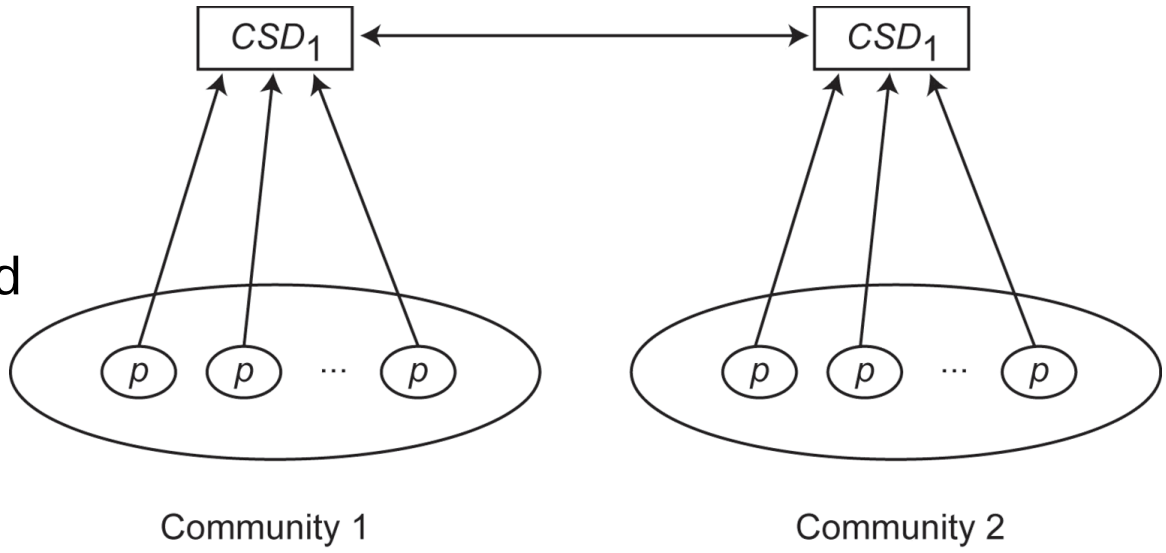


Mapping Based on Machine Learning

- 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



Outline

- Peer-to-Peer Data Management
  - P2P infrastructure
  - Schema mapping
  - Querying over P2P systems
  - Replication
  - Blockchain

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 ≥ *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 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
  - 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, $T = 75$

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

Top-k Query Processing in DHTs

- Given
  - A DHT network
  - *T* : a set of tuples which are stored in the DHT
  - *Q* : a top-k query
  - *sf* : a scoring function
- Goal
  - Retrieve efficiently the *k* tuples stored in the DHT whose scores are the highest according to *Q* and *sf*
  - While avoiding centralized data storage
- DHTop algorithm
  - TA style

Data Storage Mechanism

- Two complementary methods for storing tuples in the DHT
  - **Tuple storage**: each tuple of a relation is stored in the DHT using its tuple's identifier (e.g. primary key)
    - Allows to retrieve tuples using their identifier
  - **Attribute storage**: the values of some attributes of a tuple are stored individually in the DHT
    - Acts as secondary indices
    - Good support for exact match queries

Tuple Storage Method

- Let
  - R ($\underline{a_1}$, a_2, …, a_m) be a relation
  - t ⟨v_1, v_2, …, v_m⟩ be a tuple of R
  - v_1 be the primary key of tuple t
  - h : a hash function that hashes its inputs into a DHT key
- To store tuple t in the DHT, a peer does as follows:
  - $ts\_key = h(R, v_1)$;
  - put(ts_key,⟨v_1, v_2, …, v_m⟩);

Attribute-value Storage

- Domain partitioning
  - Let D_a be the domain of an attribute *a*
  - We partition D_a into *n* nonempty sub-domains $d_1, d_2, ..., d_n$
  - Example:
    - Attribute “weight”, with domain [0..200] in kilograms, can be partitioned into *n*=40 sub-domains [0..5), [5..10), …, [190..195), [195..200]
- Exploitation by peers
  - All peers know the sub-domains of each attribute
  - Thus, each peer can locally execute the following function:
    - *sd*(*a, v*) : returns the sub-domain of attribute *a* to which value *v* belongs

Attribute-Value Storage (cont'd)

- Let
  - $R(a_1, a_2, \ldots, a_m)$ be a relation
  - $t \langle v_1, v_2, \ldots, v_m \rangle$ be a tuple of R
  - ts_key_t : the key used for storing tuple t in the DHT, i.e. $h(R, v_1)$;
- For storing the value v_2 of attribute a_2 in the DHT :
  - let d_i be the sub-domain to which v_2 belongs, i.e. $d_i = sd(a_2, v_2)$;

    $avs\_key \leftarrow h(R, a_2, d_i)$;

    put(avs_key, $\langle v_2 , ts\_key_t \rangle$);
- Two important features:
  1. After retrieving an attribute value, we are able to retrieve its entire tuple
  2. The values that are "relatively close" are stored at the same peer

DHTop

- Two phases, executed by the peer where *Q* is originated
- Phase 1: Prepare ordered lists of attributes' sub-domains
  - For each scoring attribute *a* :
    - Step 1: create a list of *a*'s sub-domains, denoted by L_a
    - Step 2: remove useless sub-domains by considering *Q*'s conditions
      - E.g. condition *a* = *c* , remove all sub-domains except sub-domain of *c*
    - Step 3: *s*ort L_a according to the positive impact of its sub-domains on the scoring function

DHTop (cont'd)

- Phase 2: Continuously retrieve attribute values and their tuples until finding the *k* top tuples
  - For each scoring attribute *a* do in parallel

    i=1

    Repeat

    1. Ask the peer *p* that maintains attribute values for $L_a[i]$ (i.e. *i*-th sub-domain in the list) to return its attribute values in order of their positive impact on the scoring function
    2. After receiving each attribute value *v*
       - Retrieve the entire tuple
       - Check End-Condition: are there at least *k* tuples (among the retrieved tuples) whose scores are greater than the threshold?
       - If end-condition holds, then exit
    3. If all values maintained by *p* are received then
       - $i = i + 1$

    Until (end-condition holds) or (*i* ≥ number of *a*'s sub-domains)

DHTop Threshold

- Let a_1, a_2, ..., a_m be the scoring attributes, and v_1, v_2, ..., v_m be the last values retrieved for them
- Let f be the scoring function
- The threshold is $\delta = f(v_1, v_2, \ldots, v_m)$

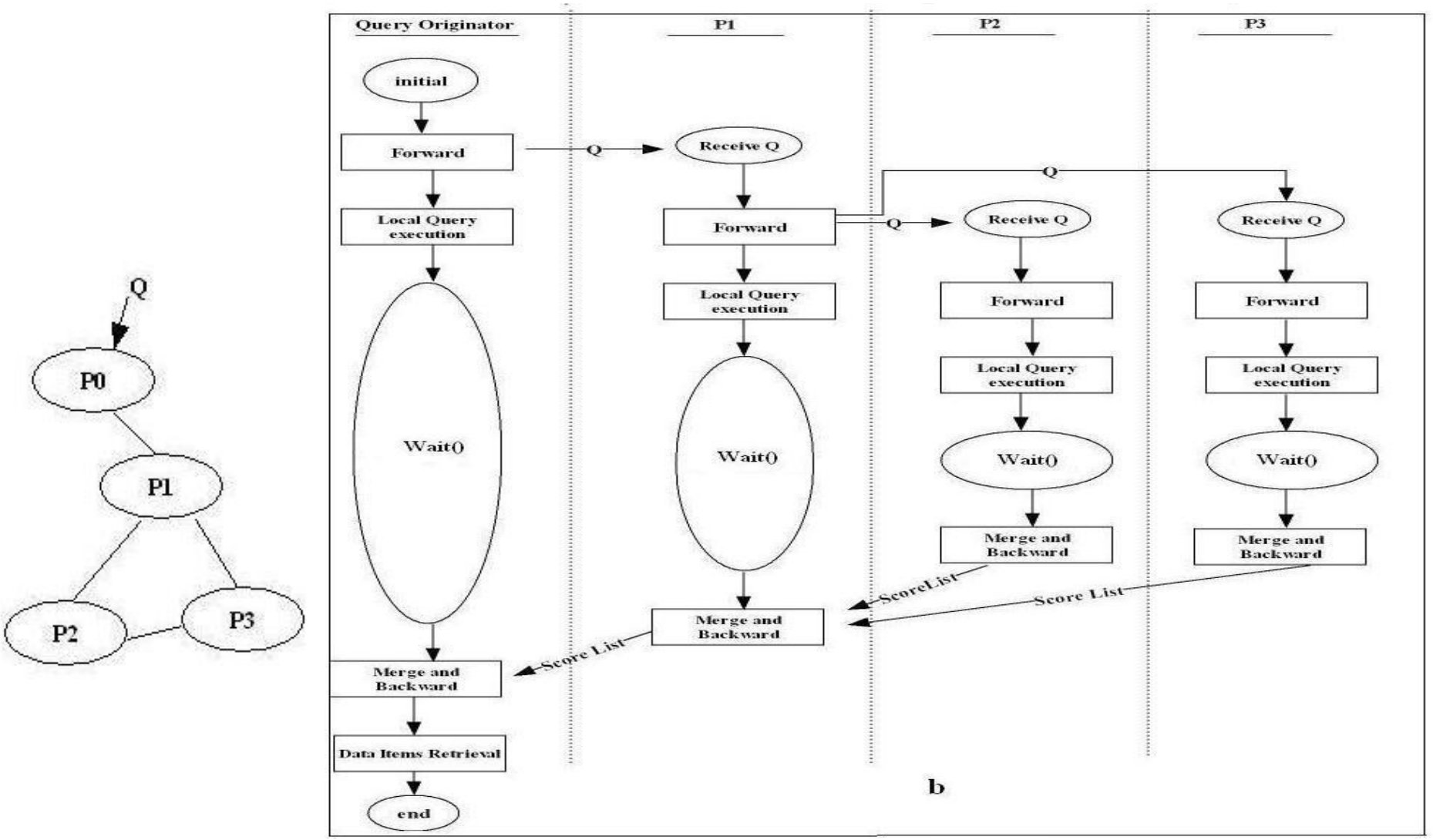
Top-k Query Processing in Unstructured Systems

- Fully Decentralized (FD) algorithm
  - Fully decentralized query execution
  - No centralized information
  - Considers dynamicity of peers
  - Simple, efficient and fault-tolerant
    - Many optimizations possible

FD Algorithm Overview

1. Query Forward
   - If TTL > 0, send *Q* to your neighbors
2. Local query execution
   - Extract the *k* local top scores
3. Wait
   - Wait to receive from the neighbors their *k* top scores
   - The wait time is based on the received TTL, network parameters and peer's local processing parameters
4. Merge-and-Backward
   - Extract *k* top scores from the local and received scores
   - Send the *k* top scores to your parent
5. Data retrieval (at the query originator)
   - Retrieve the overall top-k data using the final top scores

State Transition Diagram



FD Optimizations

- What if a peer receives a score-list after its wait time has expired?
  - Urgent Score-list: a mechanism to backward a score-list to another peer still in the wait phase
- What if the parent of a peer *p* leave the system?
  - If *p* has a neighbor that is not *p*'s child, the score-list can be sent to it as an urgent score-list
  - If *p* has not such a neighbor, it can send its merged score-list directly to the query originator
- Return results ASAP
  - Without waiting for the entire result set

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

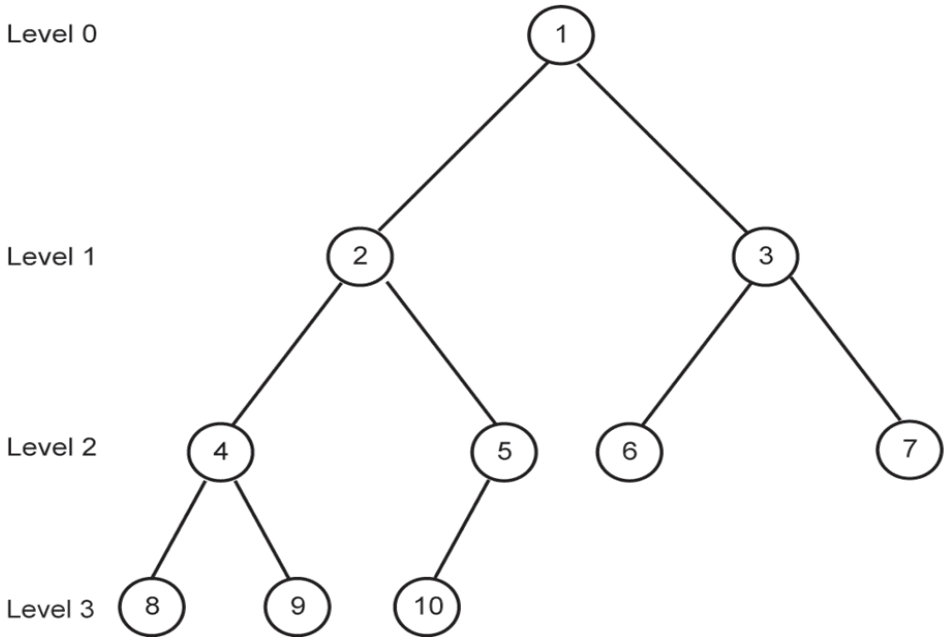
- Range query
  - WHERE clause of the form “attribute A in range [*a*; *b*]”
- Difficult to support in structured P2P systems, in particular, DHTs
  - Hashing tends to destroy the ordering of data that is useful in finding ranges quickly
- Two main approaches for supporting range queries in structured 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

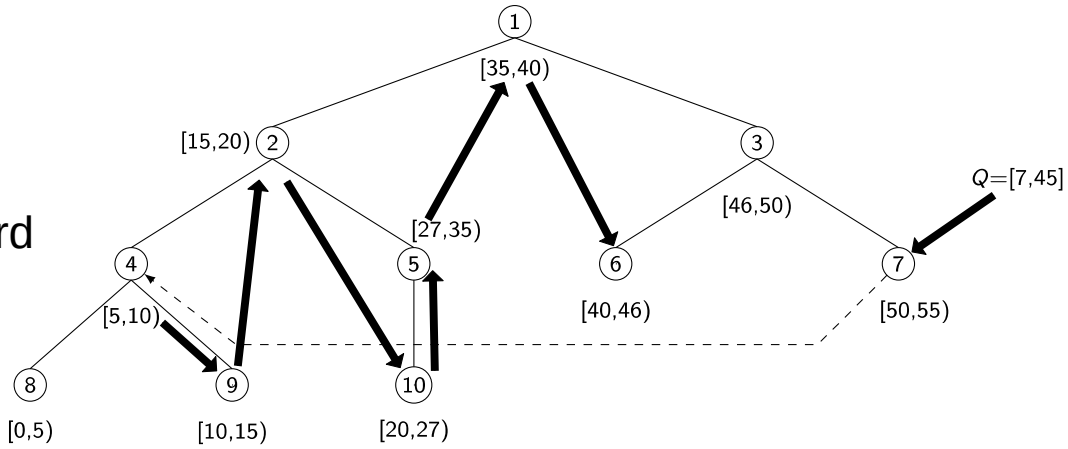
   If (Range(RighAdjacent(p)) $\cap$ [*a*,*b*]) not empty

   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



Outline

- Peer-to-Peer Data Management
  - P2P infrastructure
  - Schema mapping
  - Querying over P2P systems
  - Replication
  - Blockchain

Replica Consistency

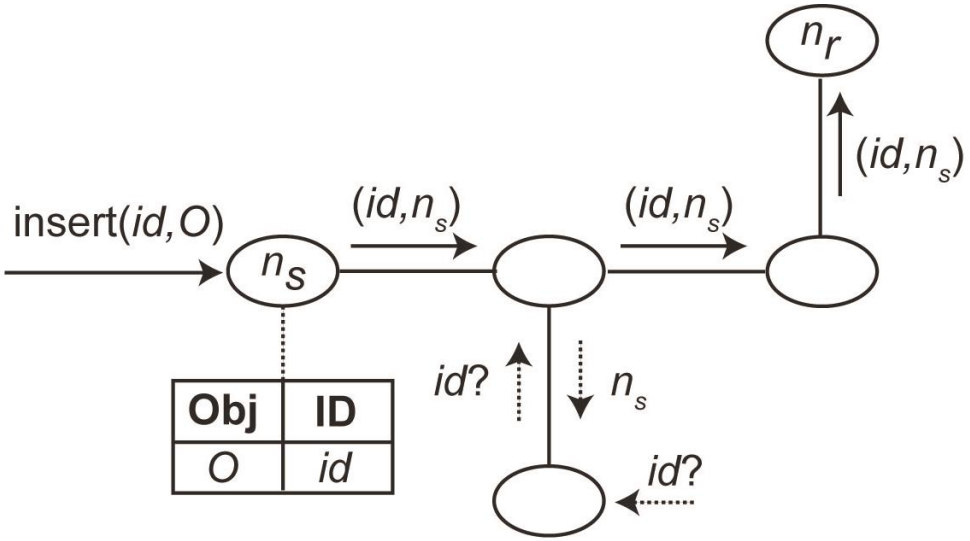
- To increase data availability and access performance, P2P systems replicate data, however, with very different levels of replica consistency
  - The earlier, simple P2P systems such as Gnutella and Kazaa deal only with static data (e.g., music files) and replication is “passive” as it occurs naturally 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

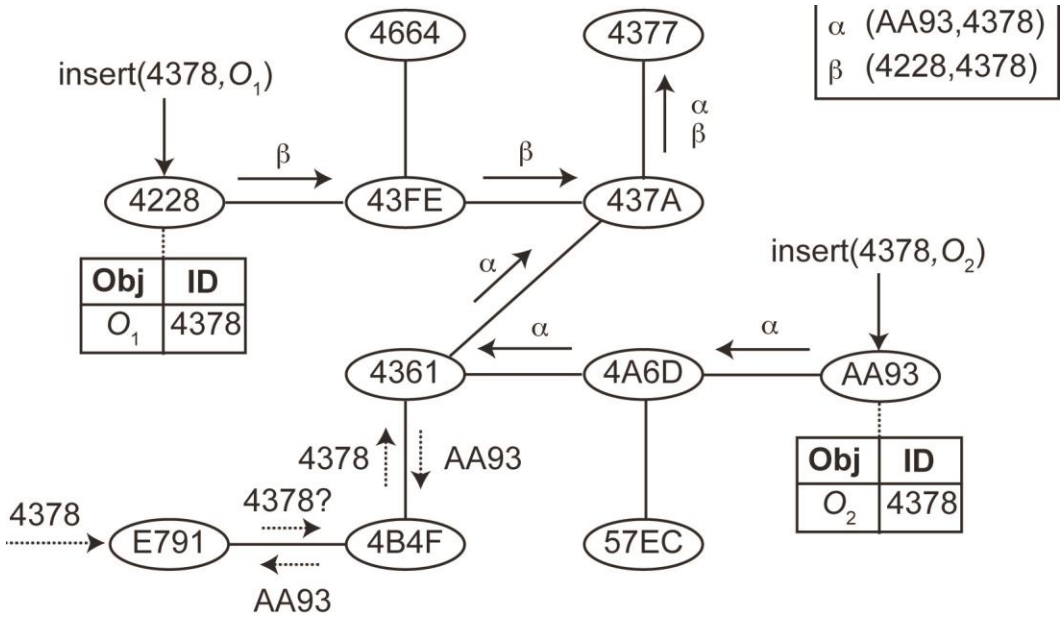
Object Publishing in Tapestry

- 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

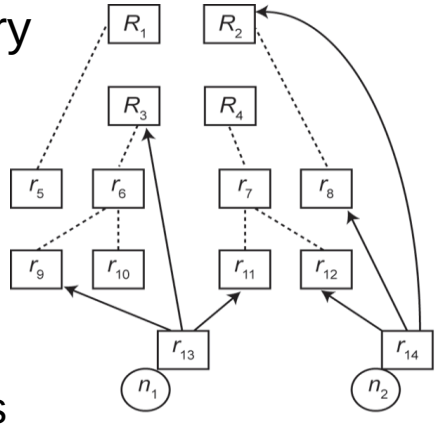


OceanStore

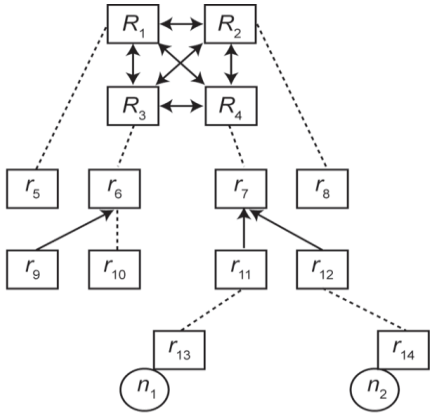
- OceanStore is a data management system designed to provide continuous access to persistent information
- Relies on Tapestry and assumes untrusted powerful servers connected by high-speed links
- To improve performance, data are allowed to be cached anywhere, anytime
- Allows concurrent updates on replicated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

Reconciliation in OceanStore

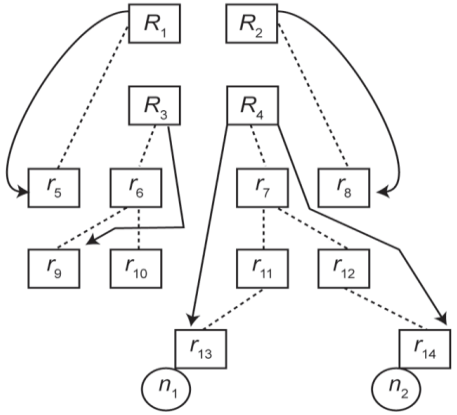
- R_i and r_i denote, respectively, a primary and a secondary copy of object R
- Nodes n_1 and n_2 are concurrently updating R as follows
  - Nodes that hold primary copies of R, called the master group of R, are responsible for ordering updates
  - (a) n_1 and n_2 perform tentative updates on their 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



(a)



(b)



(c)

Outline

- Peer-to-Peer Data Management
  - P2P infrastructure
  - Schema mapping
  - Querying over P2P systems
  - Replication
  - Blockchain

History: Bitcoin

- Bitcoin: A Peer-to-Peer Electronic Cash System
  - Satoshi Nakamoto (pseudo), Oct. 31, 2008 (Halloween)
  - Cryptocurrency and payment system
  - Blockchain is the infrastructure
- Since then
  - Many blockchains: Etherum in 2013, Ripple in 2014, etc.
  - Increasing use for high-risk investment
    - Initial Coin Offerings
  - But also in fraudulent or illegal activities !
    - Scam, purchase on the dark web, money laundering, tax evasion, …
  - Warnings from market authorities and beginning of regulation

Blockchain Definition

- A distributed ledger
  - Shared by all participants
    - Replicated
  - Decentralized
  - Append-only
    - No update, no delete
  - Distributed transaction validation
    - Consensus
  - Unfalsiable, verifiable

Blockchain Promises

- Increased trust in value exchange
  - Trust the data, not the participants
- No single point of failure
  - Increased security
- Efficient, consistent transactions between participants
  - Faster and cheaper than relying on a long chain of intermediaries, with incompatible systems and rules

Public versus Private Blockchain

- Public blockchain
  - Open P2P network
    - Participants can join and leave without notification
  - Anonymous, untrusted participants
  - Large-scale distributed ledger
- Private blockchain
  - Closed permissioned network
  - Identified, trusted participants
  - Regulated control
  - Small to medium-scale distributed ledger

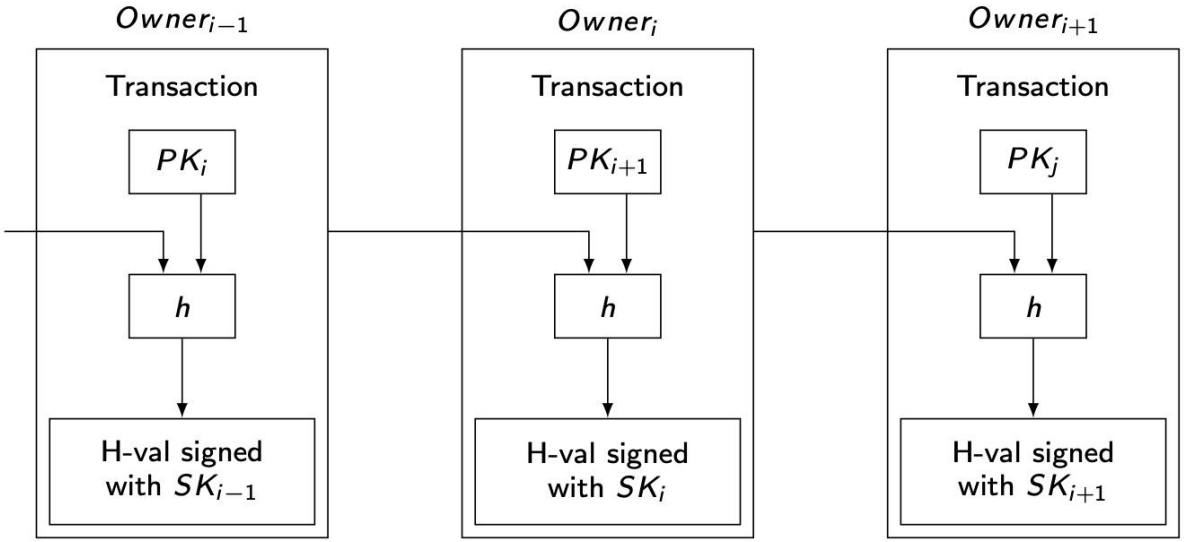
Blockchain Concepts

- Blockchain
  - An *immutable* distributed database, i.e. a log of blocks, which are linked and replicated on *full nodes*
- A block
  - Digital container for transactions, contracts, property titles, etc.
  - Transactions are secured using public key encryption
- The code of each new block is built on that of the preceding block
  - Guarantees that it cannot be changed or tampered
- The blockchain is viewed by all participants
  - Enables validating the entries in the blocks
  - Privacy: users are pseudonymized

Blockchain Protocol (Nakamoto 2008)

0. Initialization (of a *full node*)
   - Synchronization with the network to obtain the blockchain (a few hundred GB)
1. Two users agree on a transaction
   - Information exchange: wallet addresses, public keys, …
2. Grouping with other transactions in a block and validation of the block (and of the transactions)
   - Consensus using "mining"
3. Addition of the validated block in the blockchain and replication in the P2P network
4. Transaction confirmation

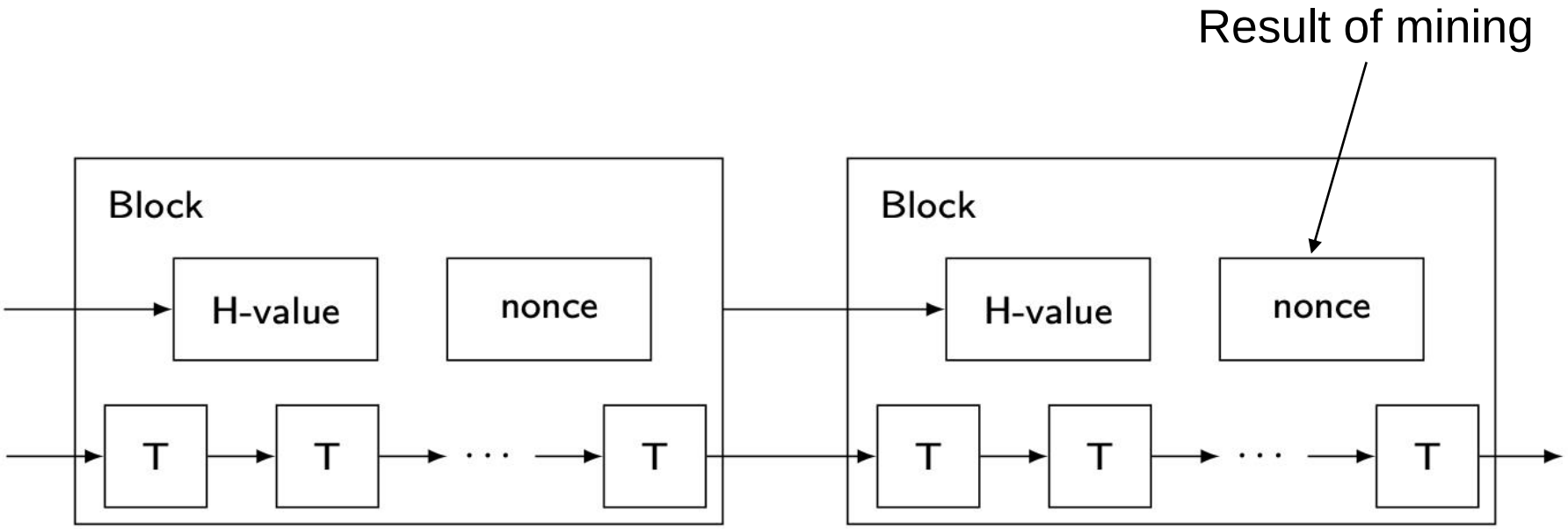
Transaction



- The coin owner signs the transaction by
  1. Creating a hash value of
     - The previous transaction
     - And the public key (PK) of the next owner
  2. Signing it with its secret key (SK)

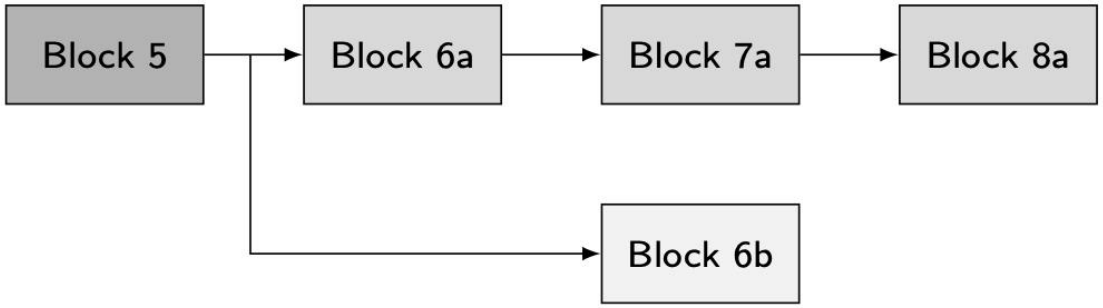
Block Management

- Transactions are placed into blocks, validated (by checking inputs/outputs, etc.) and linked by their addresses
  - Size of a bitcoin block = 1 Megabyte



Validation by the Network

- Each block is validated by network nodes, the *miners*, by a consensus protocol (see next)
- Problem: accidental fork
  - As different blocks are validated in parallel, one node can see several candidate chains at any time
  - Solution: longest chain rule



Intentional Fork

- Main reasons
  - To add new features to the blockchain (protocol changes) => new software
  - To reverse the effects of hacking or catastrophic bugs
- Soft versus hard fork
  - Soft fork: backward compatible
    - The old software recognizes blocks created with new rules as valid
    - Makes it easy for attackers
  - Hard fork
    - The old software recognizes blocks created with new rules as invalid
    - Example: the battle between (new) Ethereum and Ethereum Classic
      - In 2016, after an attack against the Decentralized Autonomous Organization (DAO), a complex smart contract for venture capital, the blockchain forked but without momentum
      - Battle is more philosophical and ethical than technical

Consensus Protocol: *mining*

- Why not Paxos?
  - Remember: participants are unknown
- To validate a block, miner nodes compete (as in a lottery) to produce a *nonce* (number used once)
  - One of the first competing solutions is selected, e.g. the one that includes the largest number of transactions
  - The winner miner is paid, e.g. 12.5 bitcoins today (originally 50)
  - This increases the money supply
- Mining is designed to be difficult
  - The more mining power the network has, the harder it is to compute the nonce
  - This allows controlling the injection of new blocks ("inflation") in the system, on avg. 1 block every 10mn
  - Advantages powerful nodes

Mining Difficulty : Proof of Work (PoW)

- PoW
  - A piece of data that is difficult to calculate but easy to verify
  - First proposed to prevent DoS attacks
- Hashcash PoW
  - Computed by each miner to produce the nonce
- Goal: produce a value *v* such that $h(v)<T$ where
  - *h* is a hash function (SHA-256)
  - *T* is a target value which is shared by all nodes and reflects the size of the network
  - *v* is a 256-bit number starting with *n* zero bits
    - Low probability of success : $1/2^n$

The 51% Attack

- Also called Goldfinger attack
  - Enables the attacker to invalidate valid transactions and double spend funds
- How
  - By holding more than 50% of the total computing power for mining
    - Miners coalition
  - It then becomes possible to modify a received chain (e.g. by removing a transaction) and produce a longer chain that will be selected by the majority
- Solution: monitoring by the community
  - In January 2014, Ghash.io reached 42%, then dropped to 9% after the Bitcoin community alert

Transaction Confirmation

- A provisionally validated transaction in a candidate block ensures that it has been verified and is viable
- Each new block accepted in the chain after the validation of the transaction is considered as a confirmation
  - A transaction is considered mature after 6 confirmations (1 hour on average)
  - New bitcoins (mining products) are only valid after 120 confirmations, to avoid the 51% attack

Public Blockchain Limitations

- Complexity and low scalability
  - Difficult evolution of operating rules
  - Increasing chain size
  - Low number of transactions per second (TPS)
    - 5-7 TPS for Bitcoin versus 25K TPS for VISA
  - Unpredictable duration of transactions, from minutes to days
- Cost
  - High energy consumption
  - Favors concentration of miners
- Users are pseudonymized, not anonymized
  - Making a transaction with a user reveals all its other transactions
- Lack of control and regulation
  - Hard for states to watch and tax transactions

Blockchain 2.0

- Evolution of Paradigm
- Beyond Bitcoin and other cryptocurrencies
  - Recording and exchange of assets without powerful intermediaries
  - Example: smart contracts
- Positioning in the internet
  - TCP/IP: the communication protocol
  - Blockchain: the value-exchange protocol?

Blockchain 2.0 Technology

- Programmable blockchain, e.g. Etherum
  - Allows application developpers to build APIs on the Blockchain protocol
    - APIs to allocate digital resources (bandwidth, storage, etc.) to the connected devices, e.g. FileCoin
    - Micropayment APIs tailored to the type of transaction (e.g. tipping a blog versus tipping a car share driver)
- Private blockchain
  - Efficient transaction validation since participants are trusted
    - No need to produce a PoW
  - Efficient management, e.g. in the cloud

Smart Contracts

- “Code is law”, Lawrence Lessig, Harvard Law School
- Smart contract (Nick Szabo, 1993)
  - Self-executing contract, with code that embeds the terms and conditions of a contract
  - Early application: digital rights management schemes
- Deployment in the blockchain 2.0 (e.g. Etherum)
  - Participants can be unknown to each other
  - Contracts can be with many third parties, e.g. IoT devices, at low cost
- Challenges
  - Bug-free code, which requires code certification
  - Compliance with mandatory regulation, which requires collaboration between programmers and lawyers

Hyperledger Project (Linux Foundation)

- Started in 2015 (IBM, Intel, Cisco, …)
- Open source blockchains and related tools
- Major frameworks
  - Hyperledger Fabric (IBM, digital Asset): a permissioned blockchain infrastructure
    - Smart contracts, configurable consensus (PBFT, …) and membership services
  - Sawtooth (Intel): a new consensus "Proof of Elapsed Time" that builds on trusted execution environments
  - Hyperledger Iroha (Soramitsu): based on Hyperledger Fabric, with a focus on mobile applications

Blockchain 2.0 Apps

- Critical characteristics of the applications
  - Asset and value are exchanged (transactions)
  - Multiple participants, unknown to each other
  - Trust is critical
- Top use cases
  - Financial services, micropayments
  - Digital rights using smart contracts
  - Digital identity
  - Supply chain management
  - Internet of Things (IoT)
- POCs in many industries
  - Publishing, retail, music, healthcare, rental, real estate, government, energy, agriculture, etc.

Blockchain Research Issues

- Scalability of the public blockchain
  - Alternatives to PoW : proof of stake, proof of hold, proof of use, proof of stake/time, …
  - New generation blockchains, e.g. Bitcoin-NG [Usenix 2016]
- Smart contracts
  - Code certification and verification
- Blockchain interoperability
  - Blockchain Interoperability Alliance (BIA), to promote cross-blockchain transactions
- Blockchain and big data
  - Blockchain-generated data analysis, e.g. fraud prevention based on real-time transactions
  - Blockchain-based DBMS, e.g. BigchainDB

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