# Architectures for Big Data

## Peer-to-peer (P2P) data management

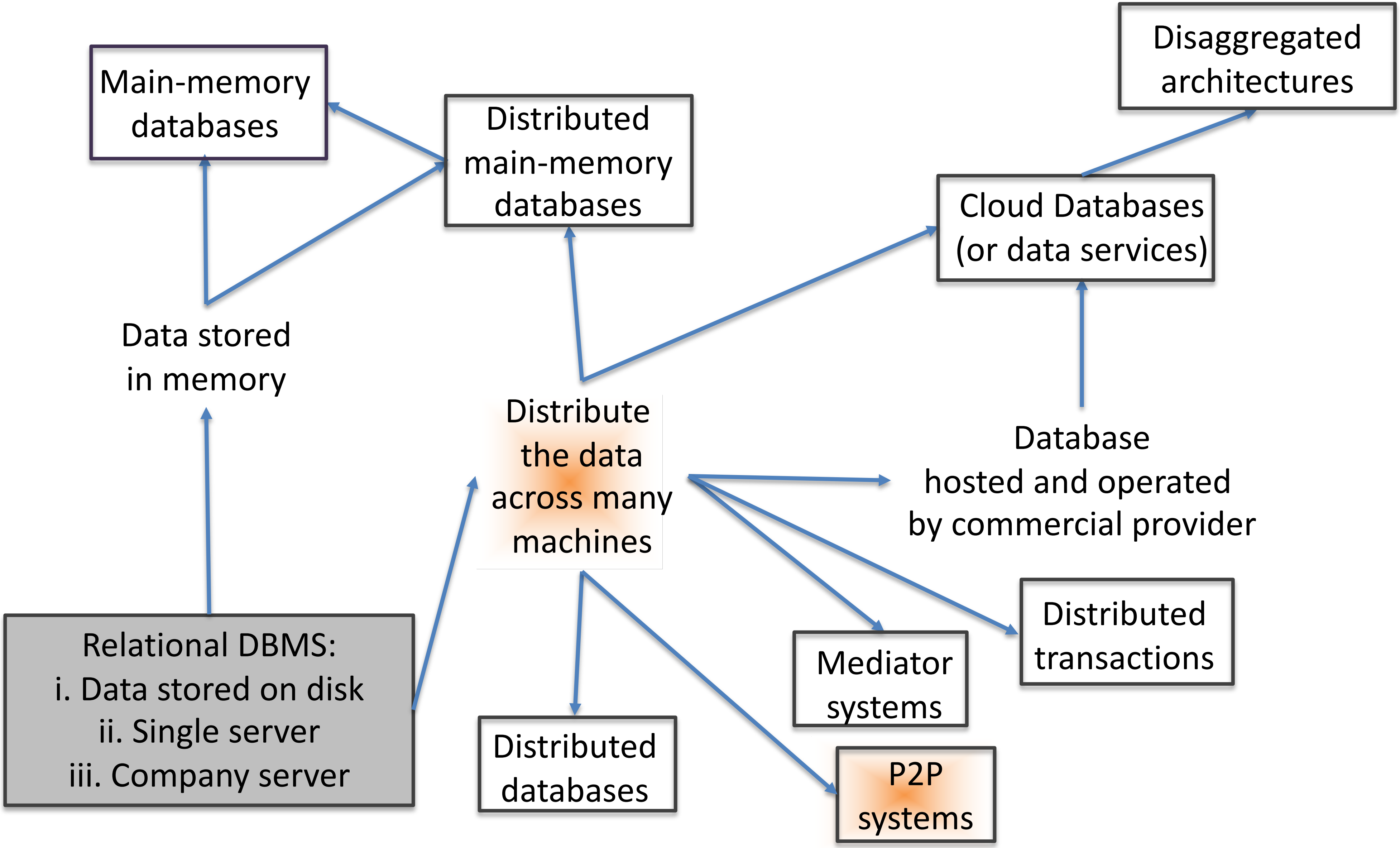
**Ioana Manolescu**

Inria Saclay & Ecole Polytechnique

ioana.manolescu@inria.fr

http://pages.saclay.inria.fr/ioana.manolescu/

# From databases to Big Data



Main-memory databases

Distributed main-memory databases

Disaggregated architectures

Cloud Databases (or data services)

Data stored in memory

Distribute the data across many machines

Database hosted and operated by commercial provider

Relational DBMS:
i. Data stored on disk
ii. Single server
iii. Company server

Distributed databases

P2P systems

Mediator systems

Distributed transactions

# Peer-to-peer architectures

- Idea: easy, **large-scale** sharing of data with **no central point of control**
- All the peers play identical roles
- Peers may join the peer network or leave it at any time

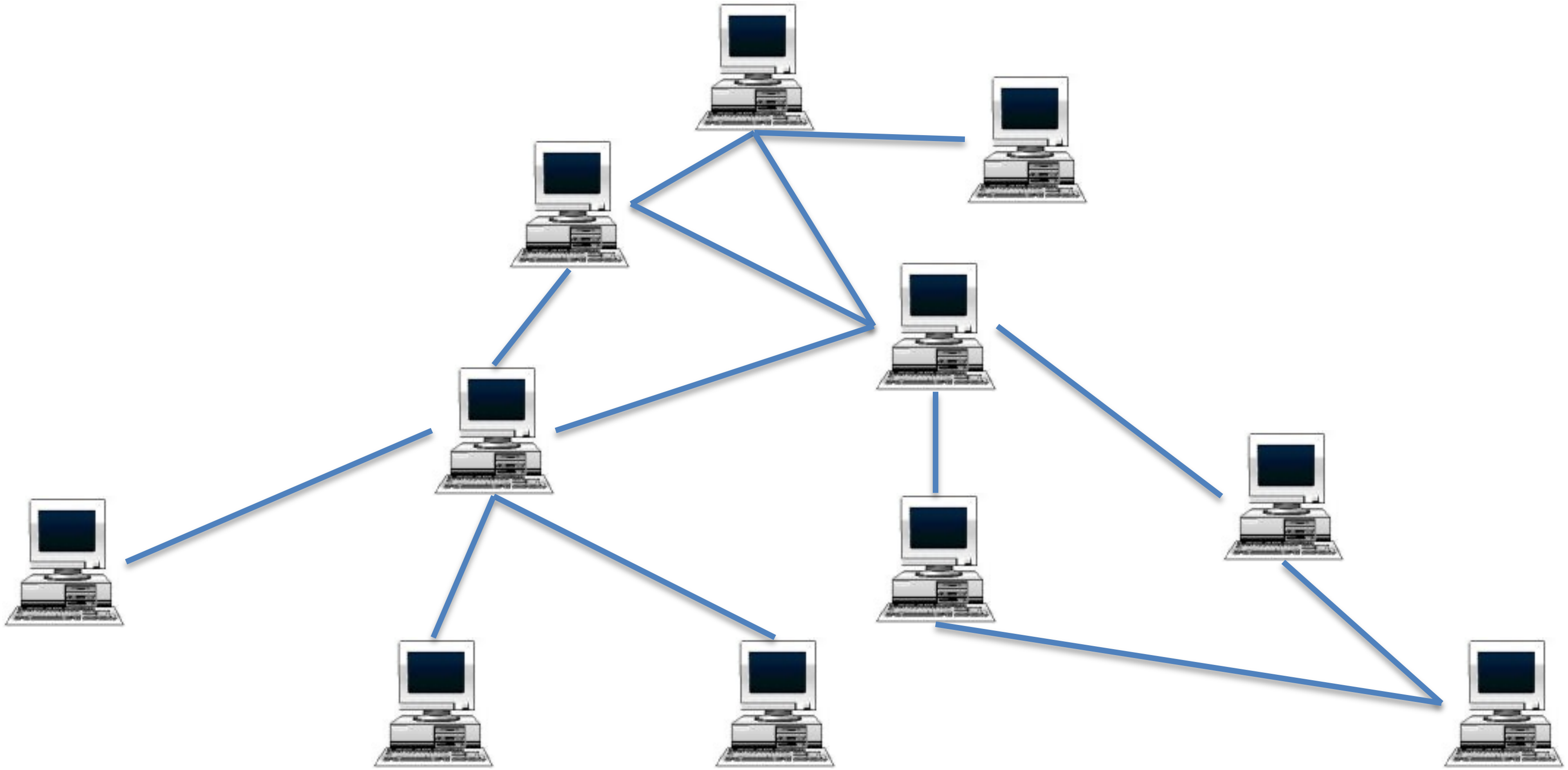
- **Advantages**:
  - Distribute work; preserve peer independence
- **Disadvantages**:
  - Lack of control over peers which may leave or fail → need for mechanisms to cope with peers joining or leaving (*churn*)
  - Schema unknown in advance; need for data discovery

# Peer-to-peer architectures

- **Large-scale** sharing of data with **no central point of control**
- Two main families of P2P architectures:
  - Unstructured P2P networks
    - Each peer is free to connect to other peers;
    - Variant: super-peer networks
  - Structured P2P networks
    - Each peer is connected to a set of other peers determined by the system
- Also: hybrid P2P architectures
  - A "central" subset of the network is structured, the rest is unstructured

# Unstructured P2P networks

A peer joins the network by connecting to another peer (« getting introduced »)

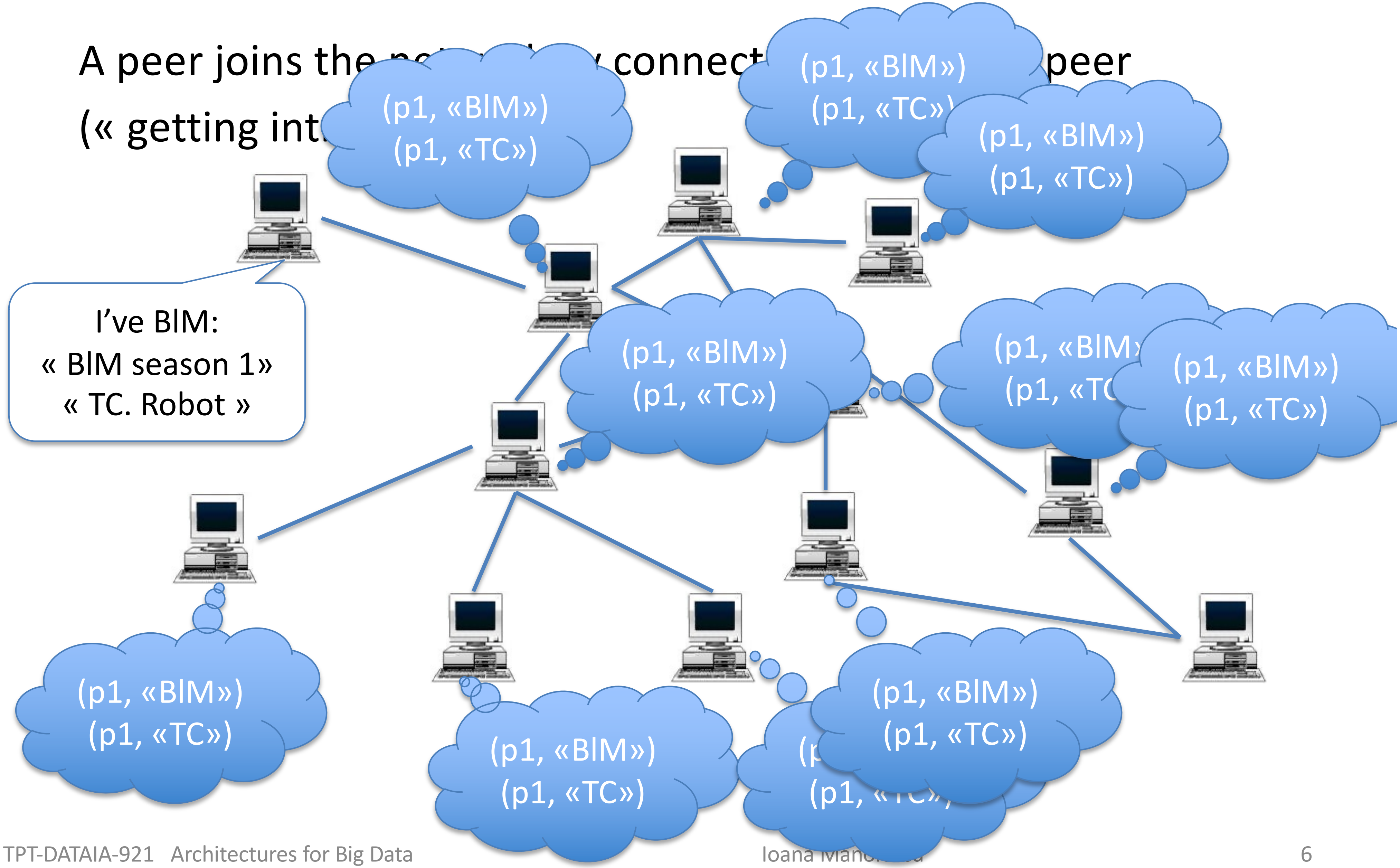


Each peer may advertise data that it publishes→peers « know their neighbors » up to some level

# Unstructured P2P networks

A peer joins the ne y connect peer

(« getting int



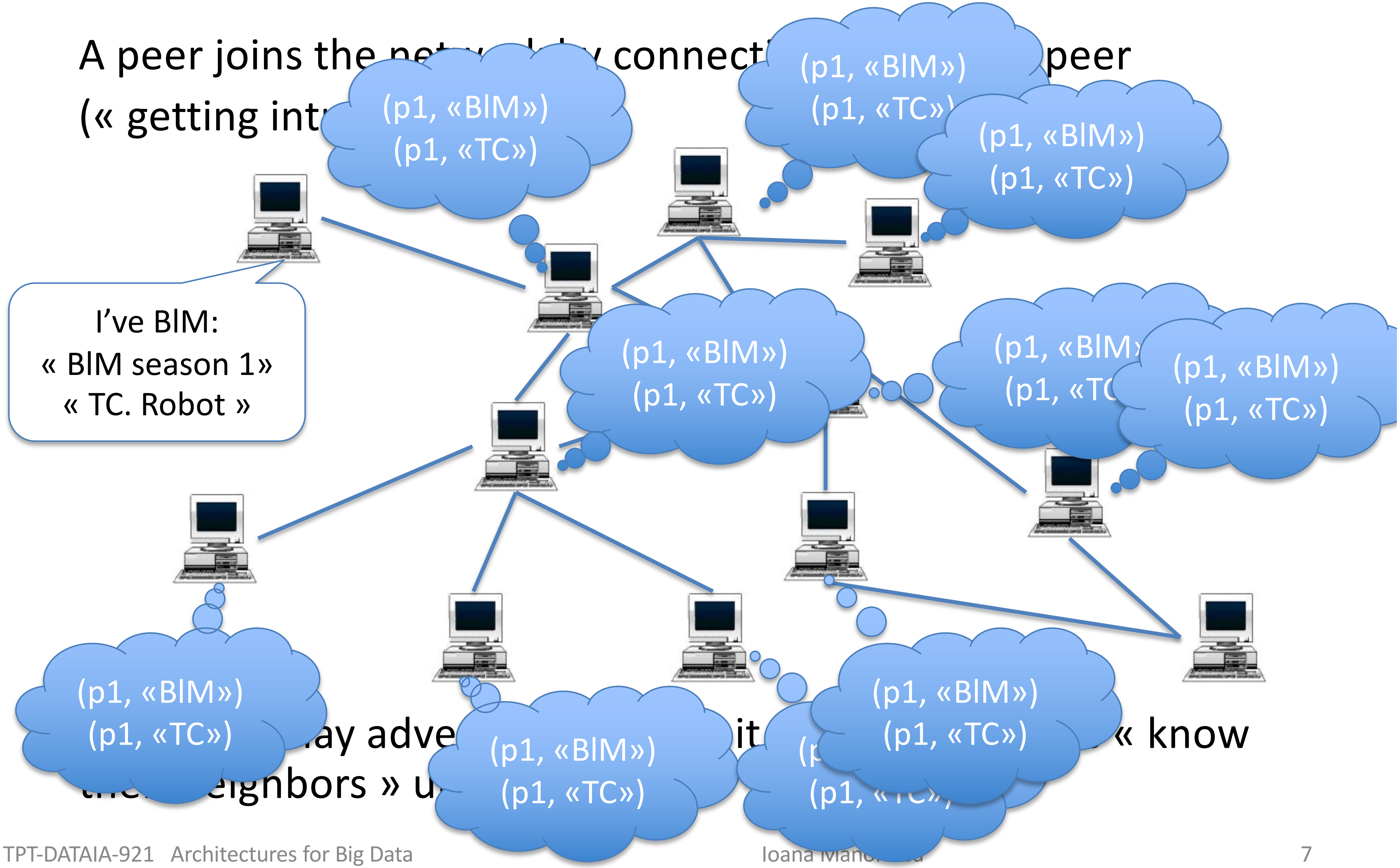
I’ve BlM:

« BlM season 1»

« TC. Robot »

# Unstructured P2P networks

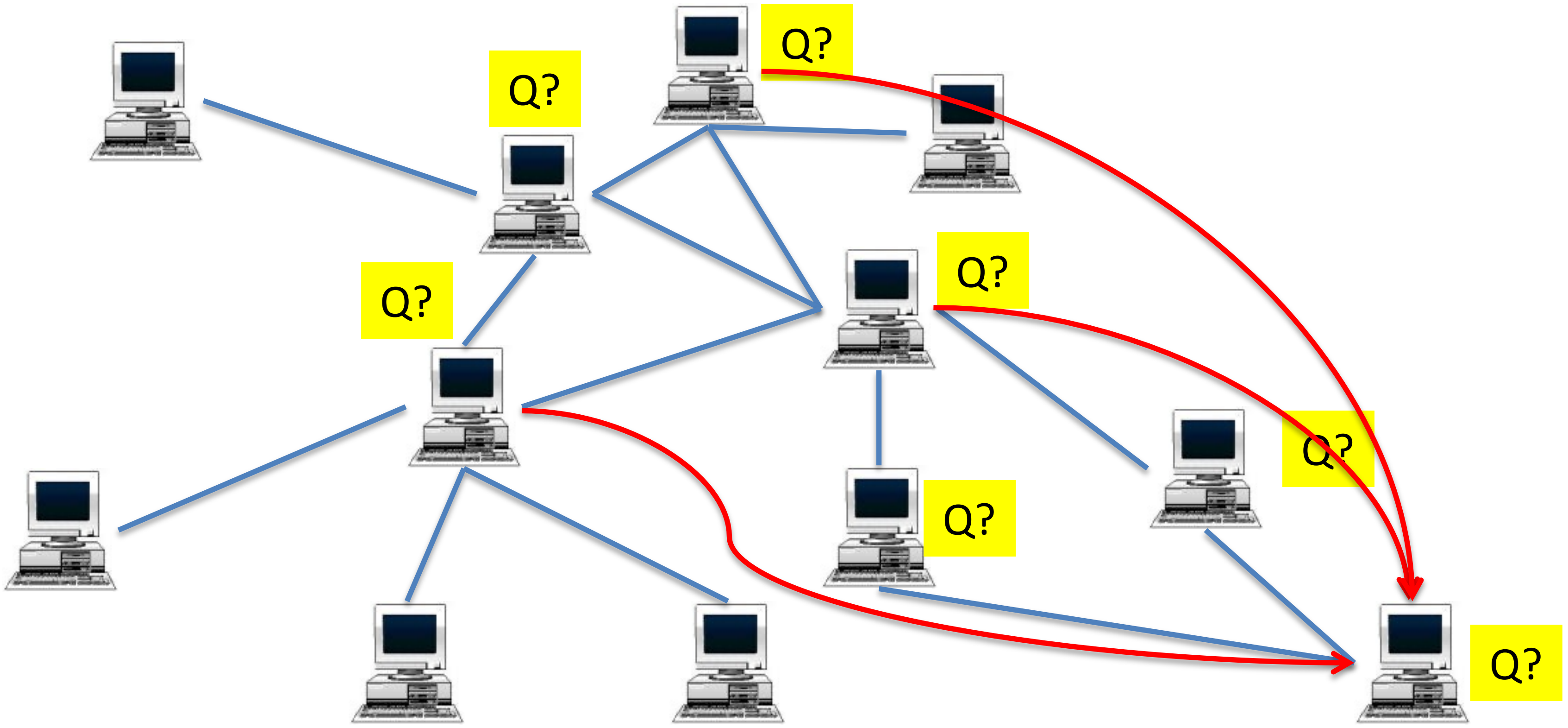
A peer joins the net … connecti … peer (« getting int …



… ay adve … it … « know the … eighbors » u …

# Unstructured P2P networks

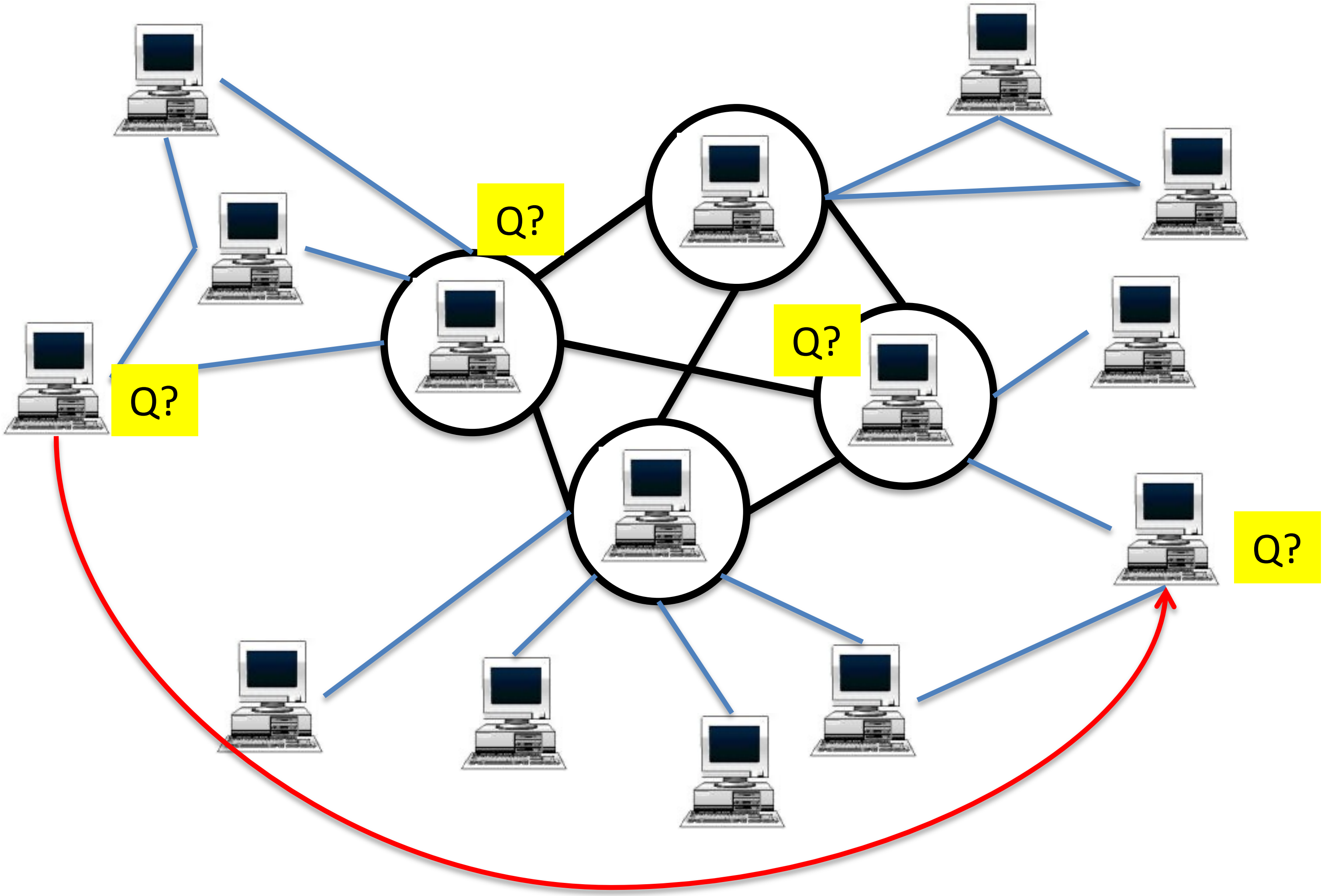
Queries are evaluated by propagating them from the query peer to its neighbors and so on recursively (flooding)



To avoid saturating the network, queries have TTL (time-to-live)

This may lead to missing answers → a. replication; b. superpeers

# Hybrid P2P network



- Small subset of superpeers all connected to each other
- Specialized by data domain, e.g. [Aa—Bw], [Ca—Dw], … or by address space
- Each peer is connected at least to a superpeer, which routes the peer's queries

# Structured P2P network

- Peers form a **logical address space** 0… $2^k$-1
  - Some logical addresses (positions) may be empty
  - The peer address is obtained with the help of a **hash function**
  - e.g., H(peer IP address)=n, 0 <= n <= $2^k$-1
  - This also leads to the name: **distributed hash table, DHT**

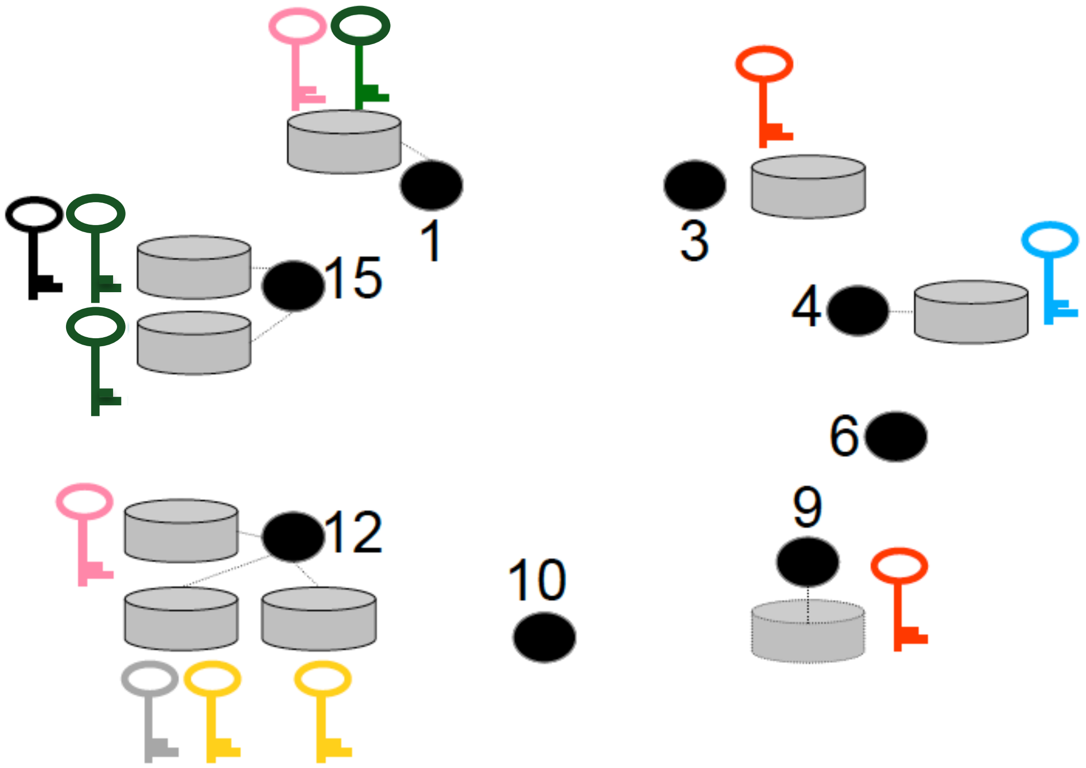
- The global data catalog is created and distributed across the peers, using the same hash function (see next)

# Catalog construction (indexing) in structured P2P networks

- The **catalog** is built as a set of key-value pairs
  - *Key*: expected to occur in search queries, e.g. «BlackMirror », «TheCrown»
  - **Value**: the address of content in the network matching the key, e.g. « peer5/Users/a/movies/BlackMirror »

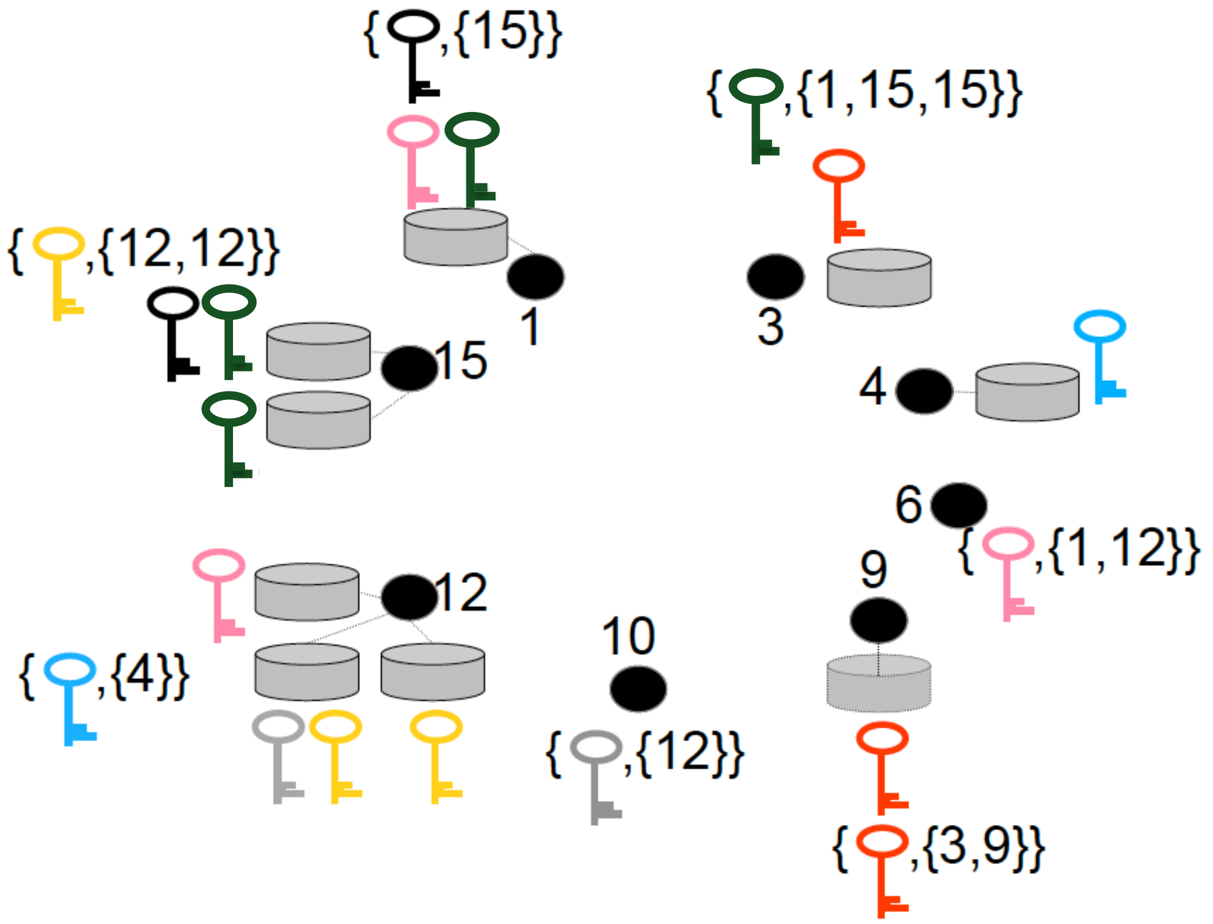
- A **hash function** is used to map every *key* into the address space; this distributes (*key*, value) pairs
  - H(*key*)=n → the (*key*, value) pair is sent to peer n
  - If n is empty, the next peer in logical order is chosen

# Catalog construction (indexing) in structured P2P networks



| *key* | *hash* |
|---|---|
| 🔑 | 6 |
| 🔑 | 2 |
| 🔑 | 7 |
| 🔑 | 1 |
| 🔑 | 12 |
| 🔑 | 8 |
| 🔑 | 14 |

# Catalog construction (indexing) in structured P2P networks



| *key* | *hash* |
|---|---|
| 🔑 | 6 |
| 🔑 | 2 |
| 🔑 | 7 |
| 🔑 | 1 |
| 🔑 | 12 |
| 🔑 | 8 |
| 🔑 | 14 |

# Searching in structured P2P networks

Locate all items characterized by 🔑 ?
Hash(🔑)=6
Peer 6 knows all the locations

Locate all items characterized by 🔑?
Hash(🔑)=14
Peer 15 knows all the locations

How do we *physically* find peers 6 and 15?

# Connections between peers in structured P2P networks

A peer's connections are dictated by the network organization and the logical address of each peer in the space 0… $2^k$-1

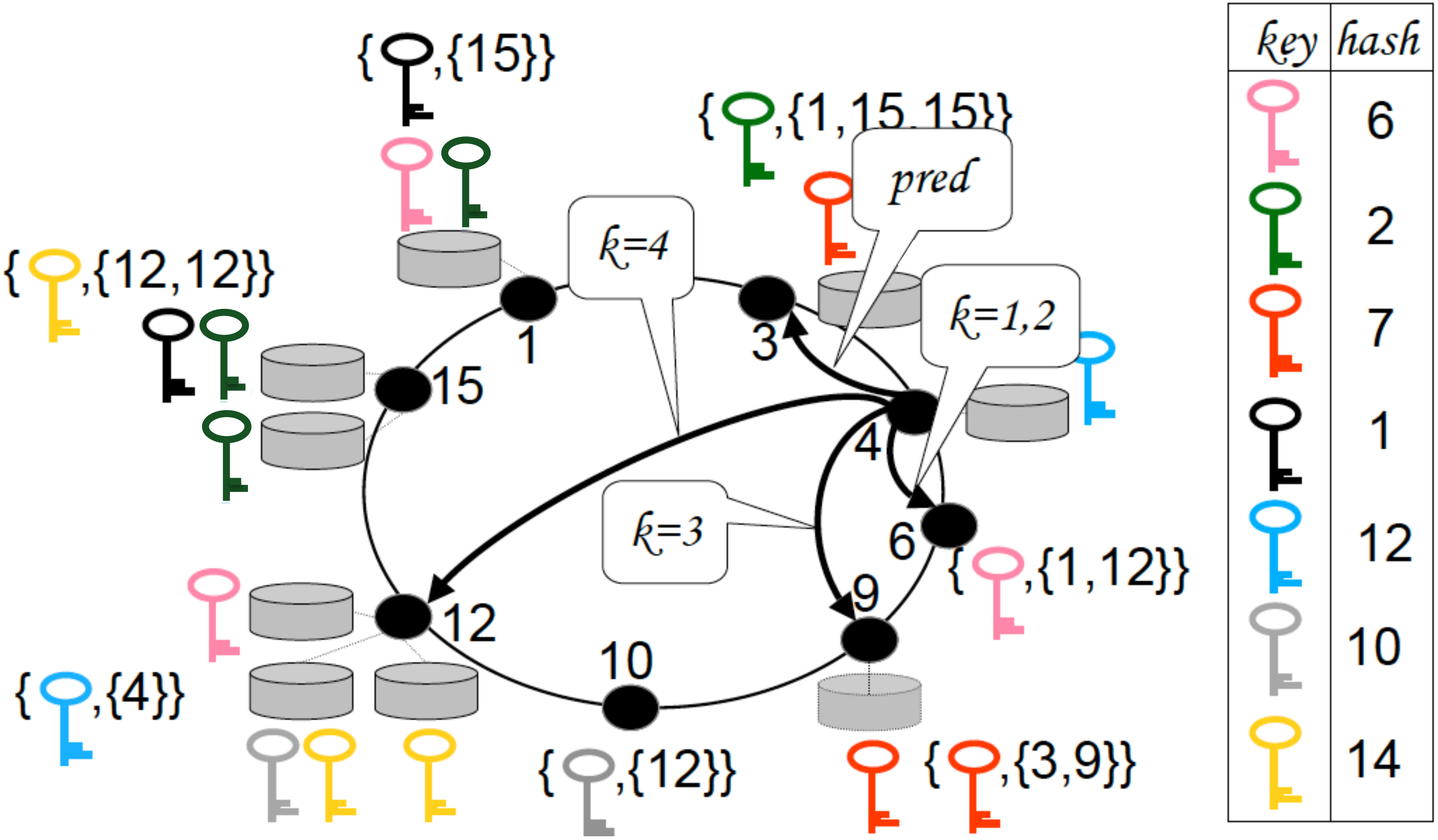
**Example: Chord (MIT, most popular)**

Each peer n is connected to

- *n+1, n+2, …,* $n+2^{k-1}$, or to the first peer *following* that position in the address space;
- The *predecessor of n*

The connections are called *fingers*

# Connections between peers in Chord



# Searching in structured P2P networks

Locate all items characterized by 🔑?

- Hash(🔑)=6
- Peer 6 knows all the locations

How does peer 1 find peer 6?

- 6-1=5; 2 <= $\log_2(5)$ <= 3, thus 6 is between 1 + $2^2$ and 1 + $2^3$
- Redirect the question to the 2nd finger of 1.

How does peer 10 find peer 3?

- (3 $-_{\text{modulo } 16}$ 10)=9; 3 <= $\log_2(9)$ <= 4, thus 3 is after 10 + $2^3$
- Redirect the question to the 3rd finger of 10.

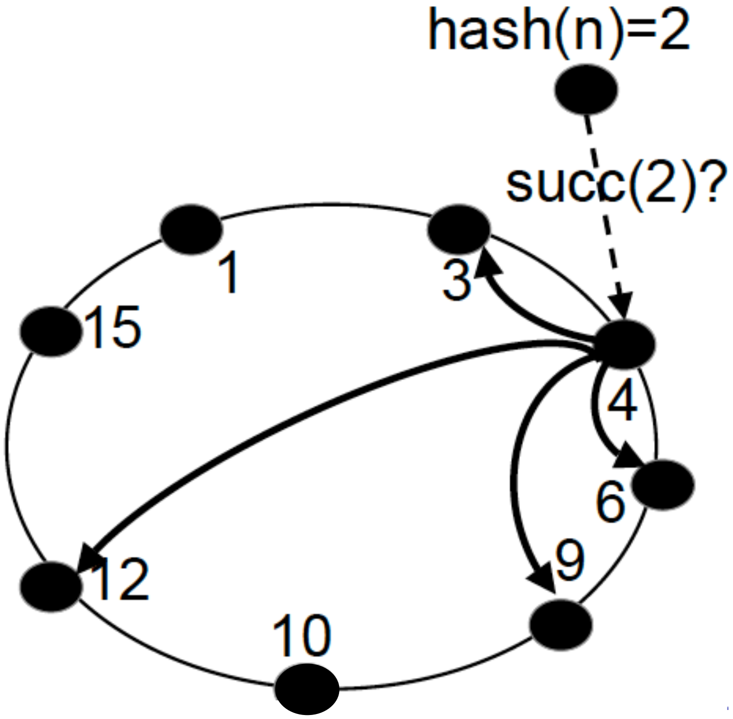
Quick traversals of the ring ($\log_2(N)$ hops)

# Peers joining in Chord

To join, a peer n must know (any) peer n' already in the network

Procedure **n.join(n')**:

```
    s = n'.findSuccessor(n);
    buildFingers(s);
    successor=s;
```

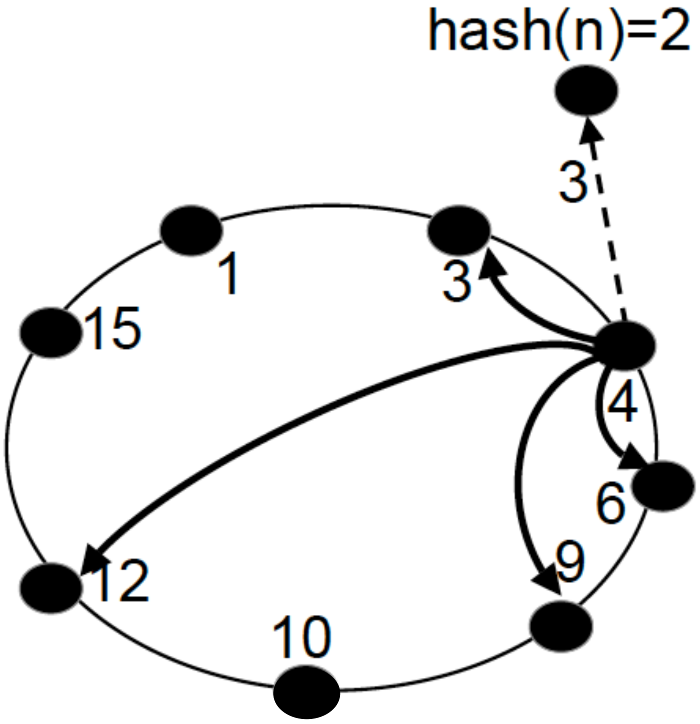


# Peers joining in Chord

To join, a peer n must know (any) peer n' already in the network

Procedure **n.join(n')**:

```
    s = n'.findSuccessor(n);
    buildFingers(s);
    successor=s;
```



# Peers joining in Chord

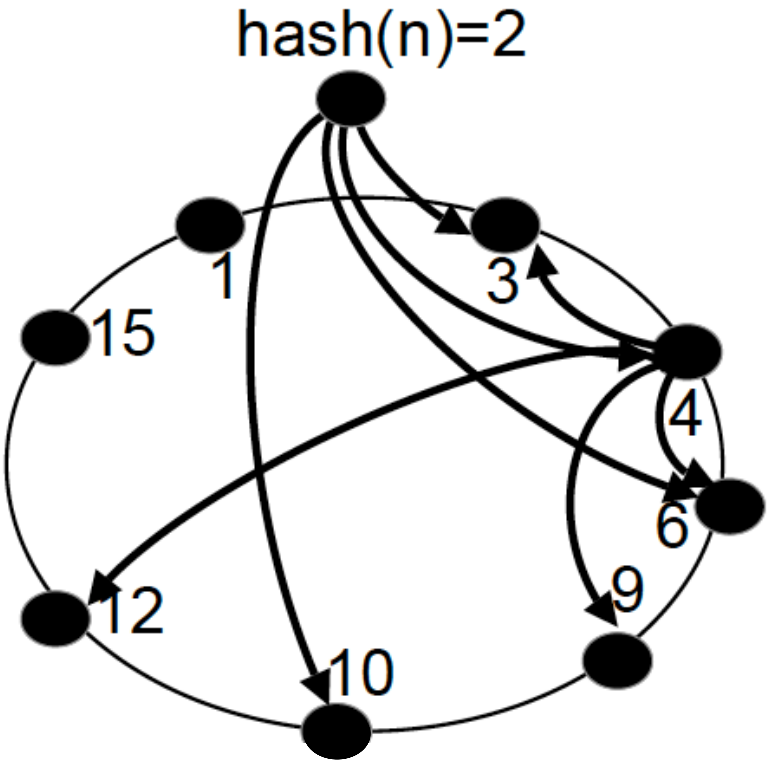
To join, a peer n must know (any) peer n' already in the network

Procedure **n.join(n')**:

```
    s = n'.findSuccessor(n);
    buildFingers(s);
    successor=s;
```

If 3 had some key-value pairs for the key 2, 3 gives them over to 2

The network is not *stabilized* yet…

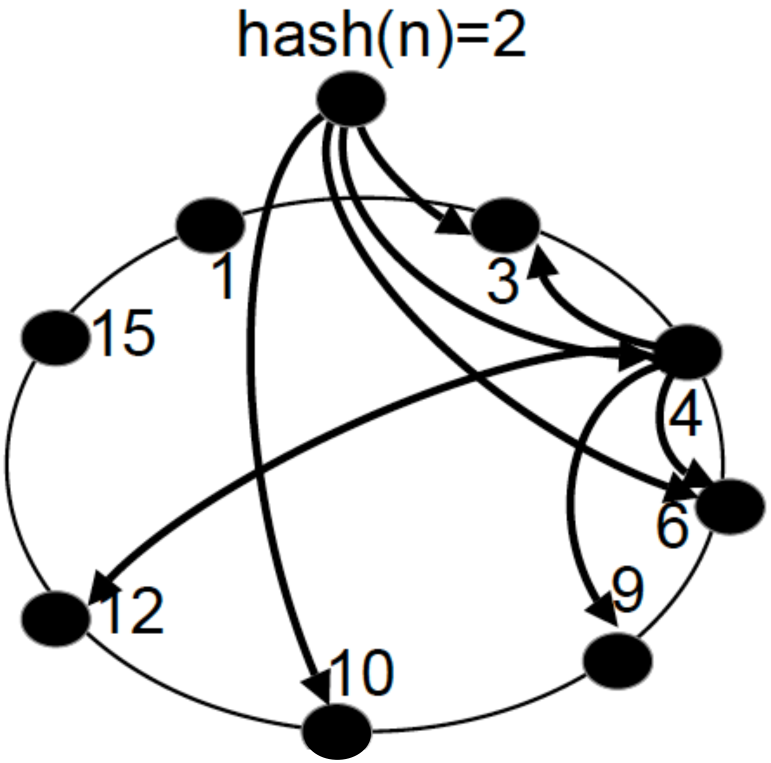


# Network stabilization in Chord

Each peer periodically runs stabilize()

```
n.stabilize():
    x = n.succ().pred()
    if (n < x < succ) then succ = x;
    succ.notify(n)

n.notify(p):
    if (pred < p < n)
    then pred = p
```

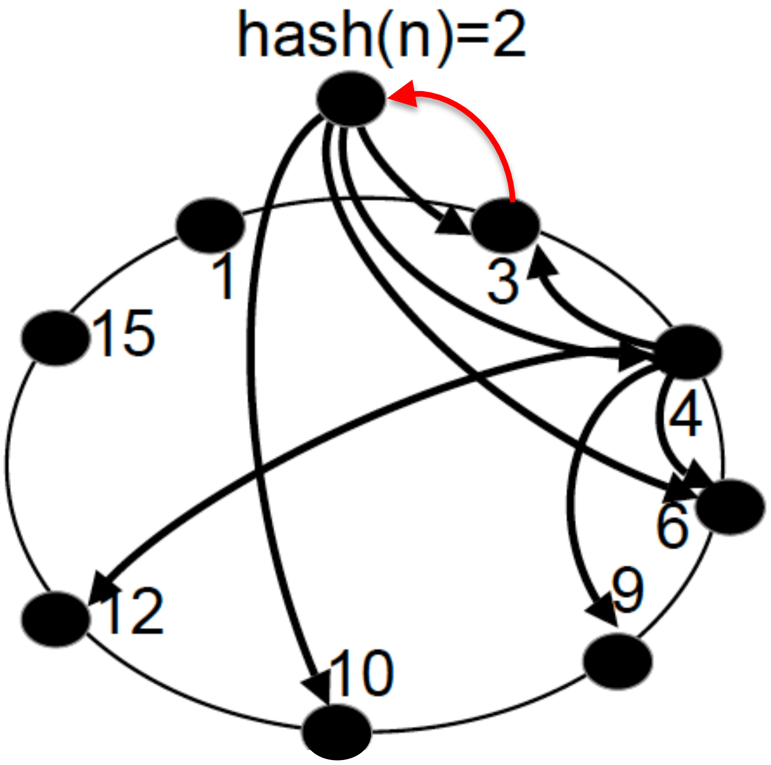


# Network stabilization in Chord

First stabilize() of 2: 3 learns its new predecessor

```
n.stabilize():
    x = n.succ().pred()
    if (n < x < succ) then succ = x;
    succ.notify(n)

n.notify(p):
    if (pred < p < n)
    then pred = p
```

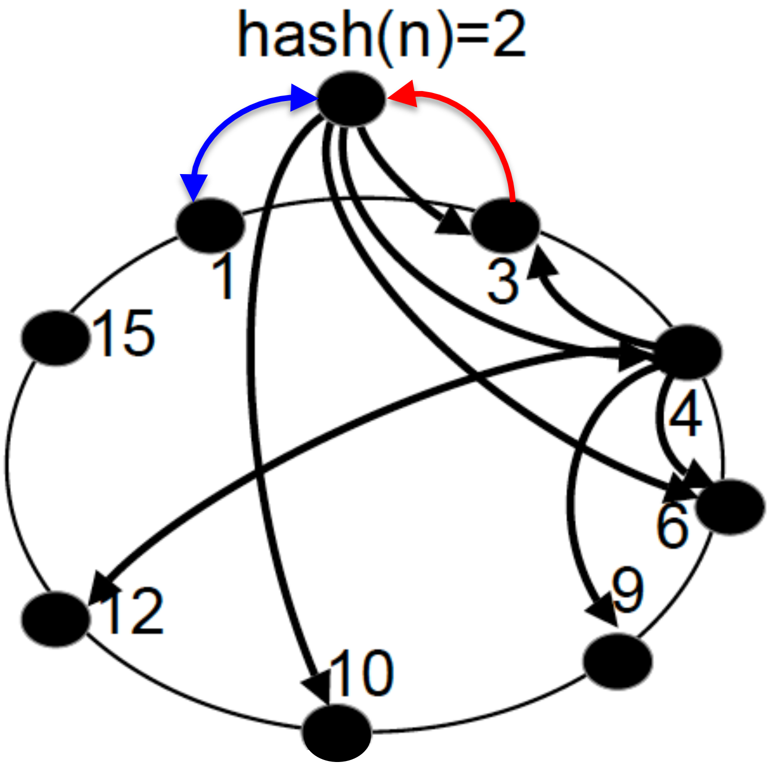


# Network stabilization in Chord

First stabilize() of 1: 1 and 2 connect

```
**n.stabilize**():
    x = n.succ().pred()
    if (n < x < succ) then succ = x;
    succ.notify(n)

**n.notify**(p):
    if (pred < p < n)
    then pred = p
```



# Peer leaving the network

- The peer leaves (with some advance notice, « in good order »)

- Network adaptation to peer leave:
  - (key, value) pairs: those of the leaving peer are moved to its successor
  - Routing:  P notifies successor and predecessor, which reconnect "over P"

# Peer failure

- Without warning
- In the absence of replication, the (key, value) pairs held on P are lost
  - Peers may also re-publish periodically

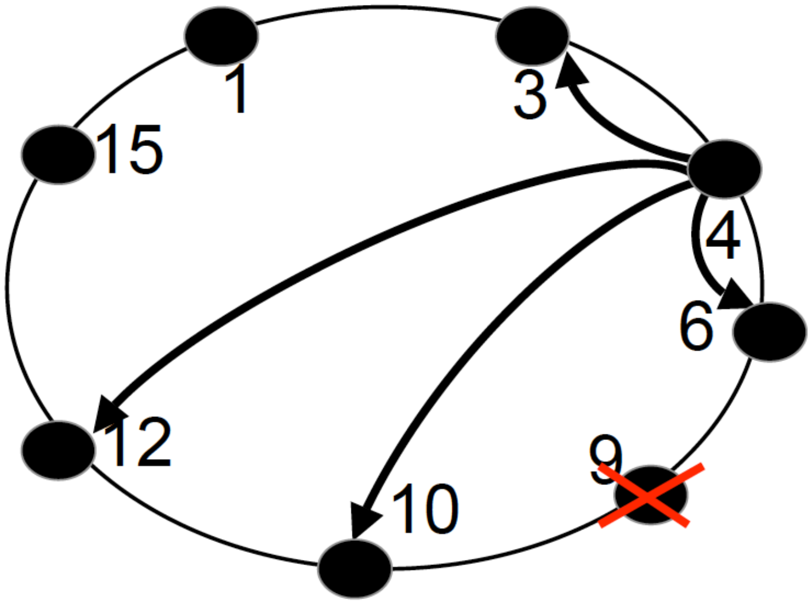
**Example** Running stab(), 6 notices 9 is down

6 replaces 9 with its next finger 10 →

all nodes have correct successors,
but fingers are wrong

Routing still works, even if a
little slowed down

Fingers must be recomputed



# Peer failure

Chord uses successors to adjust to any change

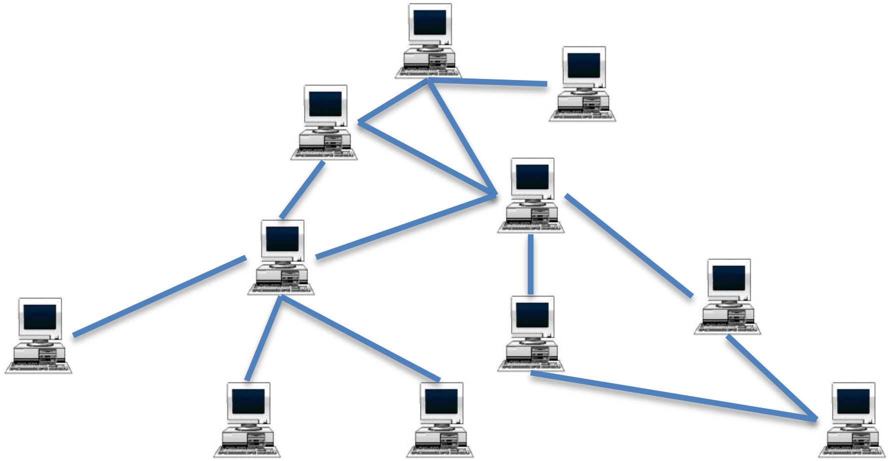
- Adjustment may « slowly propagate » along the ring, since it is relatively rare

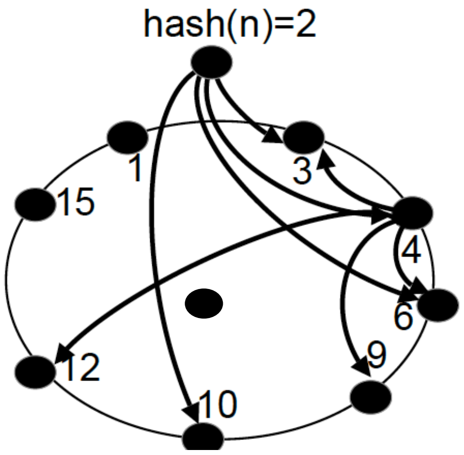
To prevent erroneous routing due to successor failure, each peer maintains a list of its *r* direct successors (*2 $log_2N$*)

- When the first one fails, the next one is used...
- All *r* successors must fail simultaneously in order to disrupt search

# Gossip in P2P architectures

- Constant, « background » communication between peers
- Structured or unstructured networks
- Disseminates information about peer network, peer data



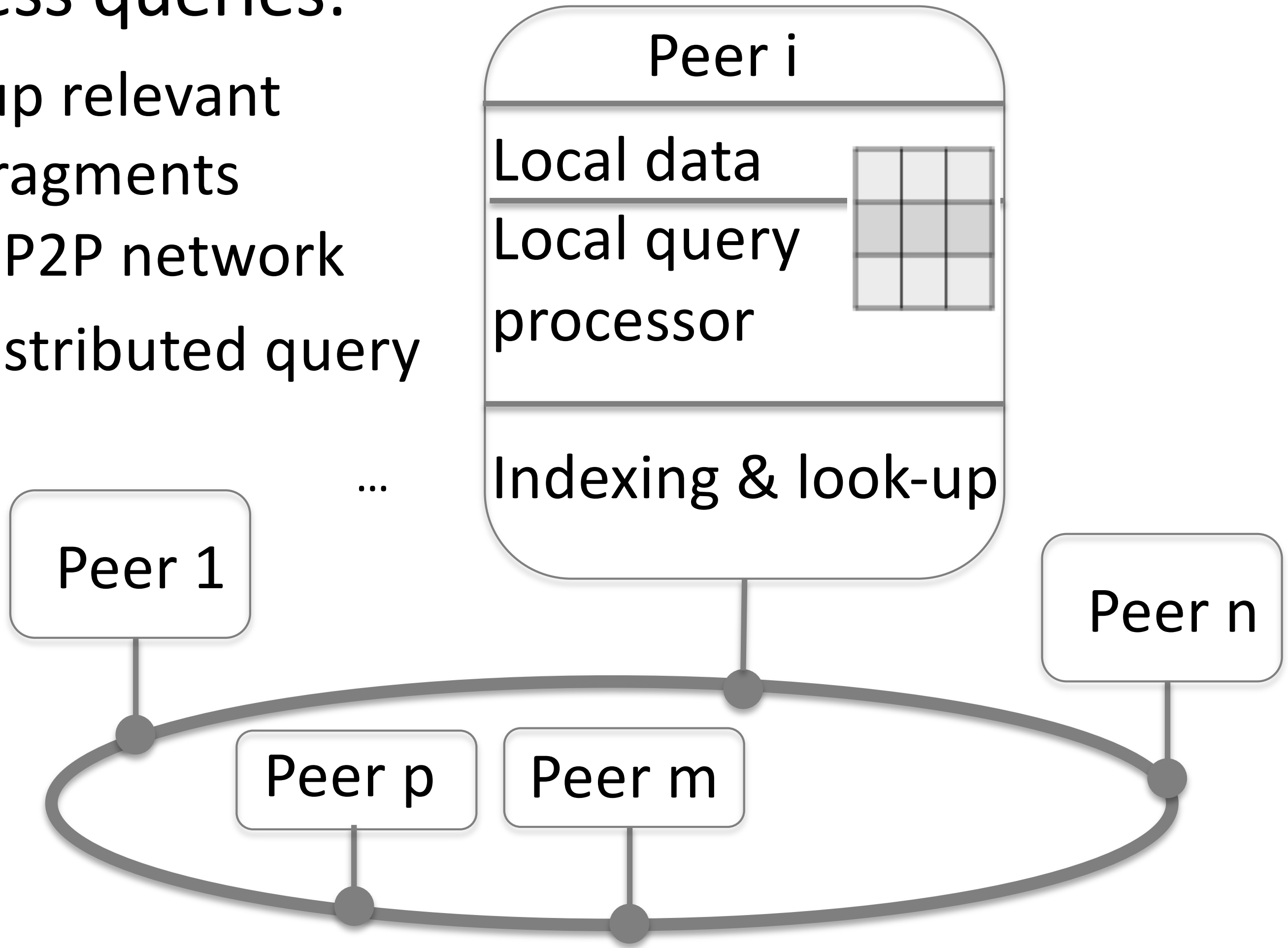


# Peer-to-peer networks: wrap-up

- **Data model**:
  - Catalog and search at a simple key level
- **Query language**: keys
- **Heterogeneity**: not the main issue
- **Control**:
  - peers are autonomous in storing and publishing
  - query processing through symetric algorithm (except for superpeers)

# Peer-to-peer data management

- Extract key-value pairs from the data & index them
- To process queries:
  - Look up relevant data fragments in the P2P network
  - Run distributed query plan



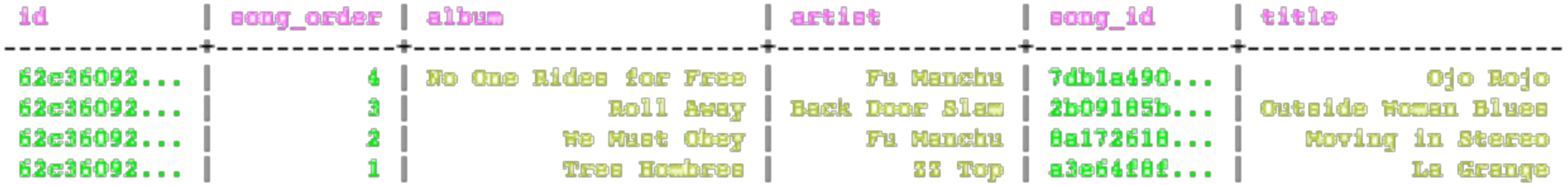
# Example: storing relational data in P2P data management platform

- Each peer stores a horizontal slice of a table
- Catalog **at the granularity of the table**:
  - Keys: table names, e.g. Singer, Song
  - Value: fragment description, e.g., peer1:postgres:sch1/Singer&u=u1&p=p1,
  - Query:
    ```
    select Singer.birthday
    from Singer, Song
    where Song.title= « Come Away » and
          Singer.sID=Song.singer
    ```
  - What can happen?
- Try other granularities

# Modern P2P data management system: Cassandra

- Partitioned row store, fully symetric structured P2P architecture
- Some nesting; indexes. Queries: select, project.

Table **songs**:

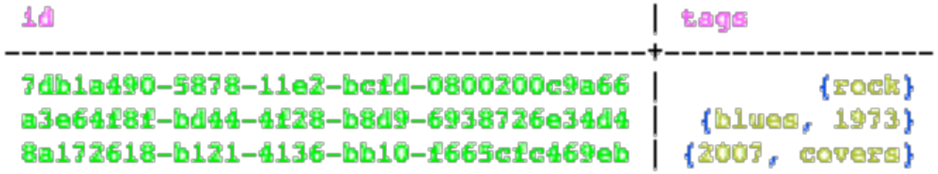


```
 id          | song_order | album                 | artist          | song_id     | title
-------------+------------+-----------------------+-----------------+-------------+---------------------
 62c36092... |          4 | No One Rides for Free |       Fu Manchu | 7db1a490... |            Ojo Rojo
 62c36092... |          3 |             Roll Away | Back Door Slam  | 2b09185b... | Outside Woman Blues
 62c36092... |          2 |         We Must Obey  |       Fu Manchu | 8a172618... |     Moving in Stereo
 62c36092... |          1 |          Tres Hombres |          ZZ Top | a3e64f8f... |           La Grange
```

ALTER TABLE songs *ADD tags set<text>*;

UPDATE songs SET tags = tags + {'2007'} WHERE id = 8a172618…;
UPDATE songs SET tags = tags + {'covers'} WHERE id = 8a172618…;
UPDATE songs SET tags = tags + {'1973'} WHERE id = a3e64f8f-…;

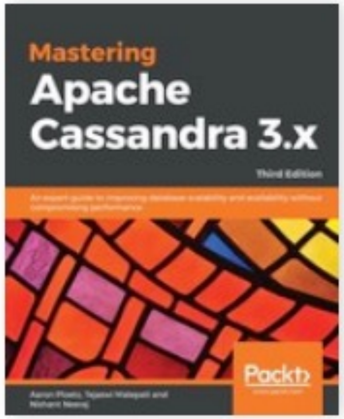
SELECT id, tags from songs;



```
 id                                   | tags
--------------------------------------+-----------------
 7db1a490-5878-11e2-bcfd-0800200c9a66 |          {rock}
 a3e64f8f-bd44-4f28-b8d9-6938726e34d4 |  {blues, 1973}
 8a172618-b121-4136-bb10-f665cfc469eb | {2007, covers}
```

# Modern P2P data management system: Cassandra

Cassandra



Large Cassandra deployments:

- Apple: over 75,000 nodes storing over 10 PB of data
- Netflix: 2,500 nodes, 420 TB, over 1 trillion requests per day

CAP trade-off: timeout for deciding when a node is dead

« *During gossip exchanges, every node maintains a* ***sliding window of inter-arrival times of gossip messages from other nodes in the cluster****. Configuring the* *phi_convict_threshold* *property adjusts the sensitivity of the failure detector. Lower values increase the likelihood that an unresponsive node will be marked as down, while higher values decrease the likelihood that transient failures causing node failure.*
*Use the default value for most situations, but* ***increase it to 10 or 12 for Amazon EC2*** *(due to frequently encountered network congestion) to help prevent false failures.*
*Values* ***higher than 12 and lower than 5*** *are not recommended.* »