

# P2P systems and P2P indexing

## Web data management and distribution

Serge Abiteboul   Philippe Rigaux  
Marie-Christine Rousset   Pierre Senellart

*<http://gemo.futurs.inria.fr/wdmd>*

February 2, 2010

# Outline

- 1 P2P Systems
- 2 Unstructured P2P Indexing
- 3 Structured: Distributed Hash Table
- 4 Issues in P2P indexing

# Peer-to-Peer (P2P) Systems

P2P system = a (very) large number of systems called **peers** that cooperate to achieve a particular task

All the peers have equal rights

- The main difference with a server/clients architecture
- Each peer at the same time, *server for* and *client of* other peers

What is a peer?

- An autonomous system with computing power and private storage
- A powerful computer, a personal computer, a PDA, a communicating object, a tripleplay box, etc.
- Any device or system with access to the network and the appropriate **P2P software**

# Applications

It is a very recent technology

It already has some applications

- P2P/Grid computing
  - ▶ Seti@home: Search for Extra-Terrestrial Intelligence
- File sharing
  - ▶ For music or video, eMule, Kazaa, Napster, Nutella, etc.
- Communications: ad hoc networks
- Data transfer: e.g, BitTorrent
  - ▶ BitTorrent is about 35% of traffic over the Internet according to some sources
- Large scale persistent storage: e.g., OceanStore
- Many others: Web caching, event notification, naming systems, backup system/archiving

# Main driving force in a P2P system

## Exploiting existing (often free) resources

- There are huge amounts of available CPU, memory, disk, network resources available on the Web
- Possibly also human resources, e.g., in social networks

## Pushing the limits of technology

- The Web brings a scale that is stressing the possibilities of clients/server architecture - e.g. Search engines and Web indexing
- Solutions: expensive super computers or huge farms of PCs
- Alternative: P2P exploiting the power of a different architecture with massive parallelism

## Particularities of P2P Systems

Scalability based on massive distribution  $\Rightarrow$  massive parallel computation

High communication overhead

- Possibly a bottleneck compared to a cluster of machine on a very high-speed local network

All participants (peers) have the same functionality

Highly volatile (peers join and leave) but resilient to node failure

No costly infrastructure

Not adapted to very rapidly changing data and high quality of service, e.g., not adapted to transactional tasks

## Particularities of P2P data management

Query processing: localization of data relevant to a query

Data consistency in presence of replication/caching

Access control with data residing in autonomous peers

Fault-tolerance based on redundancy: Copies available in many peers

# P2P Indexing

We will look at a particular class of P2P systems

P2P systems for indexing data

Task: implement a particular abstract data type, **an index**

## Abstract data type: Index

Underlying structure  $\Delta$  : key  $\rightarrow$  list of items

The **key domain** is very large: e.g., the set of strings or the set of URLs

An **item** often involves some associated data: e.g.

- An item is the ID of a video file
- The associated data is the file

Interface

- **get**(k): return  $\Delta(k)$  - or part of it
- **append**(k,c): add c to the list  $\Delta(k)$
- **delete**(k,c'): remove c' from  $\Delta(k)$
- sometimes: **fetch**(c): get the data associated to the item

# Examples

## Web indexing

- key is a word (keyword search)
- item is the URL of a file containing this keyword

## Backup file system

- key is the logical ID of a file
- an item is the ID of a copy of this file

These are typical of indexes implemented in P2P systems

# Variants of P2P indexing systems

## 1 2 functionalities

- ▶ *store the index*: keyword → list of items (e.g., list of IDs)
- ▶ In this context, a list of item is called a **posting list** (the items have been published/posted in the network)
- ▶ *store the data* associated to each item (the files corresponding to the IDs)

## 2 Pure P2P system

- ▶ the index is distributed
- ▶ the data is distributed

## 3 Hybrid system

- ▶ the index is on a server or a cluster of machines
- ▶ the data are distributed

## 4 Superpeer system

- ▶ the index is distributed between all superpeers
  - ▶ the data are distributed between all peers
- Sometimes, a superpeer is in charge of a set of peers

# Examples

Gnutella, Freenet: pure P2P

Napster and eDonkey: hybrid with a central cluster

Kazaa: superpeers

## Two main classes of implementation

### Unstructured P2P index

Each peer chooses the peers it knows and the index information it has

### Structured P2P index

The system chooses for the peers

# Outline

- 1 P2P Systems
- 2 Unstructured P2P Indexing**
- 3 Structured: Distributed Hash Table
- 4 Issues in P2P indexing

# Unstructured P2P index

Each peer  $p$  stores a portion of  $\Delta(k)$  for each  $k$ , denoted  $\Delta_p(k)$

- Typically, the items the peer owns
- E.g., the file name of a video file it stores
- $\Delta(k) = \cup_p \Delta_p(k)$

## Overlay network

- This is a network that is built on top of the Internet.
- There are links from a peer  $p$  to the peers it knows, called its “friends”,  $\text{friends}(p)$

# Implementation of get

## Implementation of get(k): **flooding**

- get(k) at peer p: first look locally for  $\Delta_p(k)$
- ask (in parallel) each  $p'$  in friends(p) for get(k)
  - ▶ They return  $\Delta_{p'}(k)$
  - ▶ They ask their friends, and so on
- Control of the computation
  - ▶ Do not propagate the query to all edges of the overlay network
  - ▶ Only propagate on a spanning tree of the graph rooted at  $p$ 
    - ★ Avoid loops
    - ★ How do you find the spanning tree?
  - ▶ Flood only to a certain depth called **TimeToLive**
  - ▶ Do not open too many channels in parallel

## Implementation of updates

A peer  $p$  publishes some new item  $v$  for  $\Delta(k)$

- Typically, add it to  $\Delta_p(k)$
- Alternatives
  - ▶ publish it in a “super” peer that has more resources
  - ▶ publish it in a peer that is specialized in  $k$

A peer  $p$  wants to delete some item  $v$  for  $\Delta(k)$

- Typically, just remove it from  $\Delta_p(k)$  - may still remain in the network
- Possible: flood the deletion to the entire network

# Possible improvements of unstructured P2P systems

## Topic-based organization

- Store additional knowledge to better route queries: send query to peers most likely to have an answer

## Store neighborhood knowledge

- Bloom filter: some traces of the matches at a certain distance.
- If the keyword does not match, no need to forward.
- If it does, there may still be no answer there (false positive of the Bloom filter)

## Caching

- Replicate popular information from other peers

# Outline

- 1 P2P Systems
- 2 Unstructured P2P Indexing
- 3 Structured: Distributed Hash Table**
- 4 Issues in P2P indexing

## Hash table in centralized systems

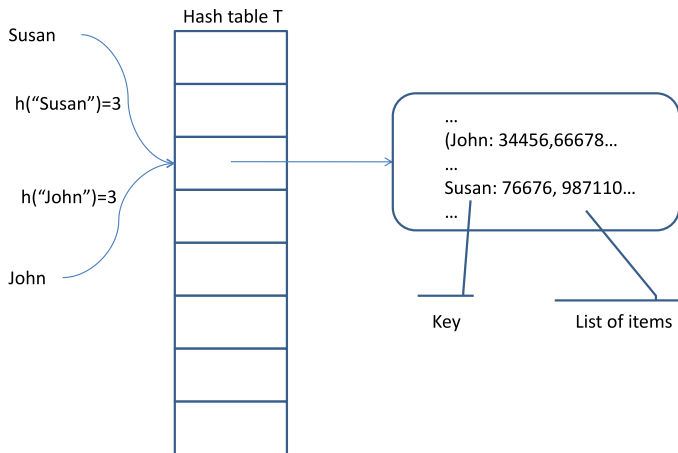
- Implementation of an index on a single machine
- Table  $T[0..N]$  of pointers to  $N$  blocks
- **Hash function**  $h : \text{key domain} \rightarrow [0..N]$ 
  - ▶ Distribute (statistically) uniformly the keys between the  $N$  values
- Each  $T[i]$  is a pointer to a block  $B_i$
- The block  $B_i$  contains

$$\{ (k ; \Delta(k)) \mid h(k)=i \}$$

### Operations

- ▶  $\text{get}(k)$ : read block  $B_{h(k)}$ ; find  $(k ; \Delta(k))$  in it
- ▶ similarly for append and delete
- When the block is full:
  - ▶ Use a list of blocks instead
  - ▶ Use dynamic hashing - variable table size

# Hash Table



# Distributed Hash Table

Each peer has an identifier  $pId$  that is unique (e.g., URI)

Distribute uniformly the keys into blocks using a hash function  $h$

Assign uniformly each block  $B_i$  to a peer

Abstract interface

- $get$ ,  $append$ ,  $delete$  as in index
- $locate(i)$ : find the peer in charge of  $B_i$
- $join(pId)$ : peer  $pId$  joins the network
  - ▶ Some blocks are transferred to the new partner
- $leave(pId)$ : peer  $pId$  leaves the network
  - ▶ Its blocks are distributed to other peers

## Some aspects common to all DHTs

Each peer maintains some routing information about the network to be able to locate  $B_j$ .

The goal is to minimize the number of messages needed on average (or in the worst case) to locate a block

There is a tradeoff between accuracy/performance, and maintenance cost:

- a highly precise routing table helps to quickly find a peer in the DHT, but requires heavy maintenance as peers join/leave the DHT;
- a routing table that stores sparse information on the DHT topology is easier to maintain, but incur more hops to locate an item.

Highly unstable networks tend to prefer lightweight routing tables (e.g., Chord); networks managed and controlled by institutions favor efficient routing through large tables (e.g., Dynamo, the Amazon system).

# Replication

Replication is used:

- For performance: several accesses to the same key in parallel
- For robustness: if a peer fails, replicas of the indexing information it contains exist
- How: instead of considering only key  $k$ , consider  $(k,1)$ ,  $(k,2)$ ,  $(k,3)$  as keys

## Implementation: Chord Ring

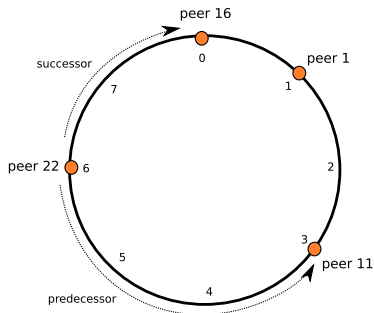
The hash function  $h'$  takes the URL of a peer and maps it to the address space  $[0..2^m - 1]$ . Hashing is based on arithmetic modulo  $2^m$ .

$$h' : pld \rightarrow pld \bmod 2^m$$

$h'$  distributes the peers around a *ring*. We assume no two peers have the same  $h'(pld)$  ( $m$  large enough).

The Chord Ring with  $m = 3, 2^m = 8$ . Each peer with id  $n$  is located at node  $n \bmod 8$  on the ring.

Ex: peer 22 is assigned to node 6.



## Distribute the keys to the peers

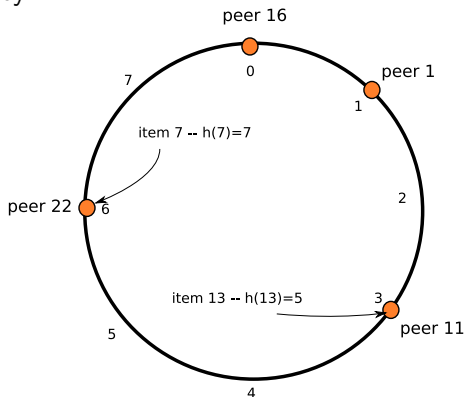
Keys are hashed with a function  $h$  with range  $[0..2^m - 1]$

Rule: a key  $k$  is assigned to the unique peer  $p$  such that

- $h'(p) \leq h(k)$
- **and** there is no  $p'$  such that  $h'(p') < h'(p) \leq h(k)$

We say that  $p$  is *responsible* for key  $k$ .

Assignment of item to peer:  
 item 13 is hashed to  $h(13) = 5$   
 and assigned to the peer  $p$  with  
 the largest  $h'(p) \leq 5$ .



## Routing tables

For each peer  $p$ ,  $friends_p$  contains (at most)  $\log 2^m = m$  peer addresses.  
 For each  $i$  in  $[1..m]$ , the  $i^{th}$  friend  $p_i$  is such that

- $h'(p_i) \leq h'(p) + 2^{i-1}$
- there is no  $p'$  such that  $h'(p_i) < h'(p') \leq h'(p) + 2^{i-1}$

In other words:  $p_i$  is the peer responsible for key  $h'(p) + 2^{i-1}$ .

Examples (Let  $m = 10$ ,  $2^m = 1024$ ; consider peer  $p$  with  $h'(p) = 10$ .)

- The first friend  $p_1$  is the peer resp. for  $10 + 2^0 = 11$
- The second friend  $p_2$  is the peer resp. for  $10 + 2^1 = 12$
- The third friend  $p_3$  is the peer resp. for  $10 + 2^2 = 14$
- friend  $p_7$  is the peer resp. for  $10 + 2^6 = 74$
- The last friend  $p_{10}$  is the peer resp. for  $(10 + 512) = 522$

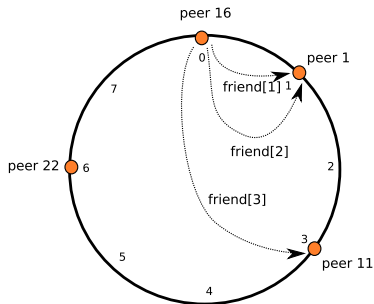
Exercise: express the gap between  $friends[i]$  and  $friends[i+1]$

## Understanding routing tables

Important properties:

- 1 a peer maintains a small routing table, e.g., 16 friends for each peer, in a ring with  $2^{16} = 65,536$  nodes;
- 2 each peer knows better the peers close on the ring than the peers far away;
- 3 a peer  $p$  cannot (in general) find directly the peer  $p'$  responsible for a key  $k$ ; *but  $p$  can find a friend which holds a more accurate information about  $k$ .*

The figure shows the friends of peer  $p_{16}$ , located at node 0 (note the collisions).  $p_{16}$  does *not* know  $p_{22}$ .



Exercise: what are the friends of  $p_{11}$ , located at node 3?



## Joining Chord

In a P2P network, nodes can join and leave at any time. When a peer  $p$  wants to join, it uses a *contact peer*  $p'$  which carries out three tasks:

- $p$  must initialize its own routing table  
 $\Rightarrow p'$  uses its routing table to locate  $p$ 's friends. Cost:  $O(\log^2 N)$ , where  $N$  is the current number of nodes.
- the routing table of the existing nodes must be updated to reflect the addition of  $p$ ;  
 $\Rightarrow$  more tricky: see details on next slide.
- finally  $p$  takes from its predecessor all the items  $k$  such that  $h'(p) \leq h(k)$ .

For highly dynamic networks, Chord relies on a stabilization protocol (not presented).

## Details: updating the routing tables of existing nodes

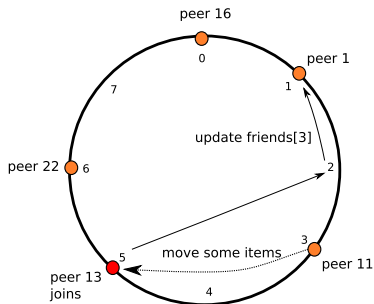
Peer  $p$  joins the network. Existing routing tables must be updated

Fact:  $p$  becomes the  $i^{\text{th}}$  friend of a peer  $p'$  iff the following conditions hold:

- 1  $h'(p) - 2^{i-1} \leq h'(p') < h'(p) - 2^{i-2}$
- 2 the current  $i^{\text{th}}$  friend of  $p'$  is before  $p$  on the ring.

Example: Peer 13 joins. It takes the slot 5. Then

- 1  $p_{13}$  computes its own routing table (explain how, assuming its contact node is  $p_{22}$ ).
- 2  $p_{13}$  is the third friend of a peer at slot 2 ( $5 - 2^{3-2} - 1$ ) or 1 ( $5 - 2^{3-1}$ ) (check that it's true).
- 3  $p_{13}$  moves part of the data stored on peer 11.



## Leaving Chord

A peer  $p$  may leave “cold” (you know you are leaving  $\Rightarrow$  take appropriate measures) or leave “hot” (failure). In both cases

- local index at  $p$  must be transmitted to the predecessor (OK if  $p$  exits gracefully, but what if  $p$  fails?)
- existing routing tables must be updated
- meanwhile, the other peers must be ready to fail when they attempt to contact a friend

Two extensions allow to cope with failures:

- **Predecessor list:** each peer maintains the list of its  $r$  immediate predecessor ( $r$  depends on the network churn).  
 $\Rightarrow$  if a friend of  $p$  does not answer, the query is routed to a predecessor of  $p$  which chooses another route.
- **Replication:** items are replicated on the  $r$  predecessors.

## Range queries

Possible but costly - Several proposals

For instance:

- Consider a particular attribute A (say with positive integer values)
- Split its domain into  $[0..a_1[; [a_1..a_2[; [a_2..a_3[; \dots [a_n..∞[$
- This information is kept in a catalog
- All the entries corresponding to one of the ranges are kept in a block, e.g., using an index entry “range:A:[0:a<sub>1</sub>]”, etc.
- To find all the entries with  $\alpha < A < \beta$ 
  - ▶ query the catalog to find all the ranges that intersect  $]α..β[$
  - ▶ one query is used for each intersecting block

# Improvement for DHT

Cache along the path to the peer that stores it

- Data: Popular files found quickly because they are replicated
- Overlay network: Cache pointers to peers accessed frequently to speed up locate

# Outline

- 1 P2P Systems
- 2 Unstructured P2P Indexing
- 3 Structured: Distributed Hash Table
- 4 Issues in P2P indexing

## Performance issues

- Many systems don't consider **locality**
  - ▶ Assume implicitly that all **communications costs** are the same
  - ▶ Not true over the Internet; e.g. across ocean queries
- Many systems assume **equal resources** between peers - not true
- **Hotspot**: Too many requests on the same peer
  - ▶ Very popular key - too many calls to  $get(k)$
  - ▶ Very popular data (e.g., music file) - many fetches
- Hotspot: A key with too many items
  - ▶ The processing of a single  $get(k)$  is expensive if the list of items for  $k$  is very large

## Other issues

Support flash crowds: bursts in usage of the system

Support high churn: peers not staying in the system for long

Support peer failures

Security

Access control

Free riders

Consequence: Difficult to guarantee quality of data and performance

## Security issues and trust

What can go wrong?

- fetch: Bad guy returns bad data
  - ▶ fix: signature of the data in the index
  - ▶ possible to verify that the data is not corrupted
- locate: Bad guy supplies incorrect routing information
- get: bad guy returns incorrect indexing information
  - ▶ e.g., filters out part of the items
- get: bad guy floods the system with dummy queries
  - ▶ denial of service attack
- append: Bad guy floods network with data
- delete: Bad guy deletes correct information

Need to evaluate **trust** and have the means to outcast bad guys

## No access control and privacy

Well-adapted to “free” communities

Anonymous sharing of information between peers

Not exposing the identity of the publisher

Not exposing the identity of the reader

No access control

Typically difficult or impossible to delete like on the Web

P2P = Power 2 People

# Access control

Use of secure HTTP: HTTPS

- Need for authentication

Read access: based on encryption

- Publish data encrypted with public key
- Only peers knowing the private key can decrypt it

Write access: more complicated

## An issue in P2P: Free riders

Why do people participate in P2P systems? e.g.,

- Why do they publish music/video in eMule?
- Why are they willing to store backups for others?
- Why are they editors in Wikipedia?

Selfish attitude: Someone who takes and does not give, **free riders**

- 66% share no files - 73% share 10 files or less
- Top 25% peers accounted for 98% of query hits
- Top 1% peers accounted for 47% of query hits

Incentive

- Make the system work (if everybody is a free rider...)
- Get popularity or influence
- Be part of a community
- Altruism
- Get advantages such as better/more access to resources

# Conclusion

The internet has already changed society

With P2P, we can expect further changes

Many technical issues, notably

- Performance
- Security
- Access control

Social dimension: social networks

The end for today

Merci