Peer To Peer Systems

- Distributed application where nodes are:
  - Autonomous
  - Very loosely coupled
  - Equal in role or functionality
  - Share & exchange resources with each other

Related Terms

- File Sharing
  - Ex: Napster, Gnutella, Kaaza, E-Donkey, BitTorrent, FreeNet, LimeWire, Morpheus
- Grid Computing
- Autonomic Computing

Search in a P2P System

Query: Who has X?

Request resource

Receive resource

Resources: R1,1, R1,2, ...
Resources: R2,1, R2,2, ...
Resources: R3,1, R3,2, ...

Answers:

Values:

Resources: R1,1, R1,2, ...
Resources: R2,1, R2,2, ...
Resources: R3,1, R3,2, ...
**Distributed Lookup**

- Have \(<k, v>\) pairs
- Given \(k\), find matching values

\[
\begin{array}{c|c}
 k & v \\
1 & a \\
1 & b \\
4 & a \\
7 & c \\
3 & a \\
1 & a \\
4 & d \\
\end{array}
\]

\[\text{lookup}(4) = \{a, d\}\]

**Data Distributed Over Nodes**

- \(N\) nodes
- Each holds some \(<k, v>\) data

**Notation**

- \(X.\text{func}(\text{params})\) means RPC of procedure \(\text{func}(\text{params})\) at node \(X\)
- \(X.A\) means send message to \(X\) to get value of \(A\)
- If \(X\) omitted, we refer to local procedure or data structures

\[
\begin{align*}
B & := X.A \\
A & := X.f(B)
\end{align*}
\]

**Distributed Hashing**

- Chord
- Replicated HT

**Hashing Values and Nodes**

- \(H(v)\) is \(m\)-bit number (\(v\) is value)
- \(H(X)\) is \(m\)-bit number (\(X\) is node id)
- Hash function is “good”

\[
\begin{align*}
0 & \leq H(k) < 2^m \\
0 & \leq H(X) < 2^m \\
0 & \leq H(Y) < 2^m
\end{align*}
\]

**Chord Circle**

Chord Paper:
**Rule**

- Consider nodes X, Y such that Y follows X clockwise
- Node Y is responsible for keys k such that H(k) in \( \{ H(X), H(Y) \} \)

**Succ, pred links**

Use hashed values... e.g., N54 is node whose id hashes to 54.

**Search using succ links**

Search using succ links

**Code**

- X.find_succ(k)
  - if k in \( \{ X, \text{succ} \} \)
  - return succ
  - else
  - return succ.find_succ(k);

**Notation: Use of Hashed Values**

- X.find_succ(k)
  - if k in \( \{ X, \text{succ} \} \)
  - return succ
  - else
  - return succ.find_succ(k);

- X.find_succ(k)
  - if H(k) in \( \{ H(X), H(\text{succ}) \} \)
  - return succ
  - else
  - return succ.find_succ(k);

**Code**

- X.find_succ(k)
  - if k in \( \{ X, \text{succ} \} \)
  - return succ
  - else
  - return succ.find_succ(k);

Note: What happens if k stored by X?
- Case 1: k < X
- Case 2: k = X
Looking Up Data

- \( X.DHT\text{lookup}(k) \)
  - \( Y := \text{find\_succ}(k) \)
  - \( S := Y.\text{lookup}(k) \)
  - return \( S \)
- \( Y.\text{lookup}(k) \)
  returns local values associated with \( k \)

Another Version

- \( X.DHT\text{lookup}(k) \)
  - \( X.\text{find\_succ}(k, X) \)
  - \( \text{wait}(\text{ans}(k, S)) \): return \( S \)
- \( X.\text{find\_succ}(k, Y) \)
  - if \( k \) in \((X, \text{succ})\)
    - \( Y.\text{ans}(k, \text{succ.\text{lookup}(k)}) \)
  - else
    - \( \text{succ.\text{find\_succ}(k, Y)} \)

Inserting Data

- \( X.DHT\text{insert}(k, v) \)
  - \( Y := \text{find\_succ}(k) \)
  - \( Y.\text{insert}(k, v) \)
- \( Y.\text{insert}(k, v) \)
  inserts \([k,v]\) in local storage

Another Version

- \( X.DHT\text{insert}(k, v) \)
  - if \( k \) in \((X, \text{succ})\)
    - \( \text{succ.\text{insert}(k,v)} \)
  - else
    - \( \text{succ.\text{DHT\text{insert}(k, v)}} \)

Finger Table

\[ m=6 \]

Node responsible for key \( 8+32 = 40 \)
Example

```
findsucc(K54)
```

**Code**

- `X.find_succ(id)`
  
  ```
  if id in (X, succ) return succ 
  else 
    Y := closest_preceed(id); 
    return Y.find_succ(id); 
  ```

- `X.closest_preceed(id)`
  
  ```
  for i := m downto 1 
  if finger[i] in (X, id) return finger[i]; 
  return X; 
  ```

Another Example

```
findsucc(K7)
```
Example

\[ \text{find\_succ}(K7) \]

return N8

Yet Another Example

\[ \text{find\_succ}(K8) \]

there seems to be a bug in code of Slide 26 (which is the same as Figure 5 of Chord paper).

Yet Another Example

Adding Nodes to Circle

need to
1. update links
2. move data

for now, assume nodes never die

New Node X Joins

- Node Y is known to be in ring
- \( X.j \)(Y)
  
  pred := nil;
  succ := Y.find\_succ(X);

Periodic Stabilization

- \( X.stabilize() \)
  
  Y := succ.pred;
  if Y in (X, succ) succ := Y;
  succ.notify(X);

- \( X.notify(Z) \)
  
  if pred = nil OR Z in (pred, X)
  pred := Z;
**Periodic Finger Check**

- $X$.fix_fingers()
  
  ```
  next := next + 1;
  if next > m
    next := 1;
  finger[next] := find_succ(X+2^{next-1})
  ```

**Join Example**

**Join Example**  

after join:

**Join Example**  

after $N_x$.stabilize:  

$Y = N_p$

**Join Example**  

after $N_p$.stabilize:  

$Y = N_x$

**Moving Data: When?**

keys in $(N_p, N_x)$

keys in $(N_p, N_s)$
Move Data After Ns.notify(Nx)

- Send all keys in range (Np, Nx] when Ns.prev is updated...

Periodic Stabilization

- X.notify(Z)
  - if pred = nil OR Z in (pred, X)
    - Z.give(data in range (pred, Z])
    - pred := Z
    - X.remove(data in range (pred, Z])

Question: What data do we give when pred=nil?

Note: We are glossing over concurrency issues, e.g., what happens to lookups while we are moving data?

Lookup May be at Wrong Node!

- resp. for (Np, Nx]
- resp. for (Nx, Ns]
- lookup for k in (Np, Nx]
  - directed to Ns!

Looking Up Data (Revised)

- X.DHTlookup(k)
  - ok := false
  - while not ok do
    - Y := find_succ(k)
    - [ok, S] := Y.lookup(k)
  - return S

Why Does This Work?

- pred, succ links eventually correct
- data ends up at correct node
- key k is at node X where k in (X.prev, X]
- finger pointers speed up searches but do not cause problems

Inserting Data (Revised)

- X.DHTinsert(k, v)
  - ok := false
  - while not ok do
    - Y := find_succ(k)
    - ok := Y.insert(k, v)

- Y.insert(k, v)
  - if k in (pred, Y]
    - insert [k, v] in local storage
    - return true
  - else return false
Results for N Node System

• With high probability, the number of nodes that must be contacted to find a successor is $O(\log N)$
• Although finger table contains room for m entries, only $O(\log N)$ need to be stored
• Experimental results show average lookup time is $(\log N)/2$

Node Failures

data at Nx:

$N_p$ $v_7$
$N_x$ $v_8$
$N_s$ $v_9$

$N_x$ dies!!
- links screwed up
- data lost

To Fix Links

• $X$.check_pred()
  if (pred has failed)
    pred := nil;
• Also, keep links to r successors in ring

Failure Example

Initially...

backup succ link (r=2)

$N_x$ dies...

Failure Example

After $N_s$.check_pred

Failure Example

After $N_p$ discovers $N_x$ down...
Failure Example
After stabilization...

Protecting Against Data Loss
• One idea: robust node (see notes on replicated data)

Replicated Hash Table

Looking Up Data
• X.DHTlookup(k)
  id := H(k);
  lock(table);
  if X = X.node(id)
    temp := local values for k;
    unlock(table);  return temp
  else
    Y := X.node(id);
    return Y.DHTlookup(k)

Concurrency Control
• X.DHTlookup(k)
  id := H(k);
  lock(table);
  if X = X.node(id)
    temp := local values for k;
    unlock(table);  return temp
  else
    unlock(table);
    Y := X.node(id);
    return Y.DHTlookup(k)

Shorthand
• X.DHTlookup(k)
  id := H(k);
  [[ if X = X.node(id)
    return local values for k
  else ]]
    Y := X.node(id);
    return Y.DHTlookup(k)
Node Joins

- Node N0
  - hash node: 0 1 2 3
  - data for keys that hash to 0,1
  - data for keys that hash to 0
  - data for keys that hash to 1

- Node N2
  - hash node: 0 1 2 3
  - data for keys that hash to 0,1
  - data for keys that hash to 2,3

Node N0 overloaded, asks N1 for help...

First, set up N1...

Copy data to N1...

Change control...

Which do we update first, N0 or N1??

Drop data at N0...

Update Other Nodes

How do other nodes get updated?
- Eagerly by N0?
- Lazily by future lookups?
What about inserts?

node N0
hash node:
0: N0
1: N0
2: N0
3: N0
data for keys that hash to 0,1

node N1
hash node:
0: N1
1: N1
2: N1
3: N1
data for keys that hash to 2,3

node N2
hash node:
0: N2
1: N2
2: N2
3: N2
data for keys that hash to 0,1

copy

insert arrives at this point...
• apply at N0 and then copy?
• re-direct to N1?
Example

N0.join(N1) is executed...

Example: Step 1

Bucket 0 is reserved for N1...
Note: What happens to lookups & inserts at this point in time? (Inserts can bounce back and forth from N0 to N1... Problem?)

Example: Step 2

N1 starts accepting inserts and data from N0...
Note: What happens to lookups & inserts at this point in time? (Lookups will miss recently inserted data...)

Example: Step 3

Copy complete; Node N1 activated (part 1)
Note: What happens to lookups & inserts at this point in time?

Example: Step 4

Copy complete; Node N1 activated (part 2)
Note: What happens to lookups & inserts at this point in time?

Example: Step 5

Node N0 records new master (part 1)
Note: What happens to lookups & inserts at this point in time?
Example: Step 6
Node N0 records new master (part 2)
Note: What happens to lookups & inserts at this point in time?

Node N0 removes data for bucket 0...
Note: What happens to lookups & inserts at this point in time?

Can we prove scheme works?
- Assume only joins, no failures
- Remember: OK to miss recent inserts
  - if not, need 2PC to handoff control...
Exercise

- Make HT extensible
- Start with m entries in HT, then dynamically double size
- Make sure we don’t get confused when HTs of different sizes exist

Chord vs Replicated HT

- Which code is simpler?
  - note that Chord code does not show data migration!
- Lookups O(log N) vs O(1)
- Impact of caching
- Routing table: Log N vs N
- Anonymity?
- Bootstrapping

Neighborhood Search (Gnutella)

- Each node stores its own data searches nearby nodes

Storing Data

- X.DTinsert(k, v)
  insert (k, v) locally at X

Lookup

- X.DTlookup(k)
  TTL := desired value
  return( X.find(k, TTL, X) )
- X.find(k, TTL, Z)
  TTL := TTL – 1
  S := local data pairs with key k
  if TTL > 0 then
    for all Y in X.neighbors (Y neq Z) do
      S := S union Y.find(k, TTL, X)
  return( S )

Example

N1.DTlookup(13), TTL = 4
**Example**

N1.DTlookup(13), TTL = 4

answer so far = \{c, f, x\}

---

**Optimization**

- Queries have unique identifier
- Nodes keep cache of recent queries (query id plus TTL)

---

**Joins**

- \( X \) \( \cdot \) join

\[
\text{neighbors} := \{} \quad \text{cand} := \text{nodes a “friend” recommends} \\
\text{Z} := \text{bootstrap server we hear about} \\
\text{cand} := \text{cand union Z.getNodes for Y in cand do} \\
\text{ok} := \text{Y.wantMe(X)} \\
\text{if ok then} \\
\text{neighbors} := \text{neighbors} \cup \{Y\} \\
\text{if |neighbors| > limit, return}
\]
Joins (continued)

- Y.wantMe(X)
  if I want X as neighbor then
  neighbors := neighbors ∪ \{X\}
  return (true)
  return (false)

Bootstrap Server

known nodes:
S = X1, X2, x3, ...
server
add to S
X5
X1
X7
X2
if no response,
remove from S
sometimes called “pong server”

Problems with Neighborhood Search

- Unnecessary messages
- High load and traffic
  - Example: nodes have M new neighbors, number of messages is M^TTL
- Low capacity nodes are a bottleneck
- Do not find all answers

Why is Neighborhood Search Good?

- Can pose complex queries
- Simple robust algorithm
- Works well if data is highly replicated

Super-Nodes

- Regular nodes index their content at super-nodes
- Super-nodes run neighborhood search

Motivation for Super-Nodes

- Take advantage of powerful nodes
- Searching larger index better than searching many smaller ones
Napster (Original One)

- Single Super-Node

Napster (Original One)

- Actually, had several disconnected SNs

Performance Evaluation


Aggregate Bandwidth

![Aggregate Bandwidth Graph]

Individual Bandwidth Load

![Individual Bandwidth Load Graph]
**Individual Compute Load**

![Individual Compute Load Diagram](image)

**SP Redundancy**

![SP Redundancy Diagram](image)

**Content Addressable Network (CAN)**

![CAN Diagram](image)

**What is a P2P System?**

- Multiple sites (at edge)
- Distributed resources
- Sites are autonomous (different owners)
- Sites are both clients and servers
- Sites have equal functionality

**Comparison**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Gnutella</th>
<th>CAN</th>
<th>Others?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressiveness</td>
<td>★★★★★</td>
<td>★★</td>
<td></td>
</tr>
<tr>
<td>Comprehensiveness</td>
<td>★★★</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Autonomy</td>
<td>★★★★★</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td>★</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robustness</td>
<td>★★★</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Topology</td>
<td>pwr law</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Placement</td>
<td>arbitrary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message Routing</td>
<td>flooding</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison

<table>
<thead>
<tr>
<th></th>
<th>Gnutella</th>
<th>CAN</th>
<th>Others?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressiveness</td>
<td>★★★★</td>
<td>★</td>
<td></td>
</tr>
<tr>
<td>Comprehensiveness</td>
<td>★★★☆</td>
<td>★★</td>
<td></td>
</tr>
<tr>
<td>Autonomy</td>
<td>★★★★☆</td>
<td>★★</td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td>★</td>
<td>★★★☆</td>
<td></td>
</tr>
<tr>
<td>Robustness</td>
<td>★★★★☆</td>
<td>★★</td>
<td></td>
</tr>
<tr>
<td>Topology</td>
<td>pwr law</td>
<td>grid</td>
<td></td>
</tr>
<tr>
<td>Data Placement</td>
<td>arbitrary</td>
<td>hashing</td>
<td></td>
</tr>
<tr>
<td>Message Routing</td>
<td>flooding</td>
<td>directed</td>
<td></td>
</tr>
</tbody>
</table>

Open Problems

Performance
- Efficiency
- Load-balancing

Correctness
- Authentic Services
- Prevention of DoS
- Incentives
- Participation

Open Problems: “Bad Guys”
- Availability (e.g., coping with DOS attacks)
- Authenticity
- Anonymity
- Access Control (e.g., IP protection, payments,...)

Authenticity

- title: origin of species
- author: charles darwin
- date: 1859
- body: In an island far, far away ...

More than Fetching One File

- T=origin Y=1800 A=darwin B=?
- T=origin Y=1859 A=darwin
- T=origin Y=1859 A=darwin
More than Fetching One File

T=origin
Y=1800
A=darwin
B=abcd

T=origin
Y=1859
A=darwin

T=origin
Y=1859
A=darwin

T=origin
Y=1859
A=darwin

Solutions

• Authenticity Function $A(doc)$: T or F
  - at expert sites, at all sites?
  - can use signature expert $\rightarrow$ sig(doc) $\rightarrow$ user

• Voting Based
  - authentic is what majority says

• Time Based
  - e.g., oldest version (available) is authentic

Added Challenge: Efficiency

• Example: Current music sharing
  - everyone has authenticity function
  - but downloading files is expensive

• Solution:
  Track peer behavior

How to Track Peer Behavior?

• Trust Vector $[v_1, v_2, v_3, v_4]$  
  $a \ b \ c \ d$

  • Single value between 0 and 1?

• Pair of values
  $[\text{total downloads, good downloads}]$?

Trust Operations

How to Track Peer Behavior?

• Trust Vector $[v_1, v_2, v_3, v_4]$  
  $[1, .9, .5, 0, 0]$

  $[1, 0, 1, .2]$

  $[1, .9, .5, 0, 0]$

  $[1, 1, 0, .3, 1]$

  $[1, 0, 1, 1, .2]$

  $[1, 1, 0, .3, 1]$

  $[1, 1, 0, .3, 1]$

  $[1, 1, 0, .3, 1]$
Issues
- Trust computations in dynamic system
- Overloading good nodes
- Bad nodes can provide good content sometimes
- Bad nodes can build up reputation
- Bad nodes can form collectives

Sample Results
- Fraction of malicious peers

Participation & Incentives
- Autonomous nodes need incentives to work together
  - Forward messages
  - Perform computations
  - Share/store files
  - Provide services
  - Etc.

Incentive types
- Three main kinds of incentives (thus far):
  - Tit for tat
  - Reputation
  - Money/Currency

Tit-for-Tat
- “I do to you what you do for me”
- Example

They need each other to reach more nodes.
⇒ Can retaliate

Reputation and Currency
- “If you do something for me, I will give you reputation/money”
Pros and Cons

- **Tit-for-Tat**
  - Pros: Requires minimal infrastructure and overhead, least prone to cheating
  - Cons: Requires symmetric relationships
- **Currency**
  - Pros: Everyone wants money! In some applications it is required
  - Cons: Requires the heaviest infrastructure
- **Reputation**
  - Applies to most situations, but has some overhead, as well as its own incentive issues

Reputation and Currency

- For these techniques, there are 2 questions:
  - If we have money/reputation scores, how do we use it to give peers incentives?
  - How do we implement money/reputation score in a P2P fashion?

P2P Summary

- **Search**
  - Chord DHT
  - Replicated DHT
  - Gnutella
  - Super-Peers
- **Dealing with Bad Guys**
- **Dealing with Lazy Guys**