Chord and Peer-to-Peer systems

cs739

Notes from reviews

- Is "future work" a flaw?
  - They know it cannot do certain things. Your job is to figure out the importance, and if it is a flaw in the implementation or they just didn't do everything
  - E.g. security, partitions
- Physical network locality ?
  - How would you handle?
  - Have more fingers (e.g. two) per entry?
- Malicious nodes?
  - How much damage could a limited set of attackers do to a large ring?
  - Sybil attack
  - Privacy of participants
- Is O(log n) scalable enough?

What is a peer-to-peer system?

- Distributed systems without any centralized control or hierarchical organization (decentralized)
- All software at each node is equivalent in functionality (symmetric)
- Participants are unmanaged volunteers
  - Frequently come and go (churn)
  - May not be trusted (limit damage)

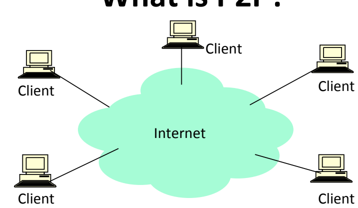
Peer 2 peer challenges

- no centralization,
  - Nobody has "ground truth"
- frequent churn,
  - Users could be laptops turned on occasionally
- Untrusted machines
  - No physical security, network security
- Extreme heterogeneity
  - Nodes range from big servers to small laptops
- Geographic diversity
  - All over the world, varying network connectivity

Early Peer-to-Peer

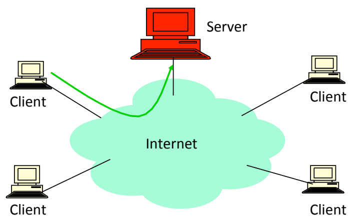
- File systems:
  - Sun NFS, Windows CIFS
    - Any machine can be a server, any machine can access files on any other machine
  - **These provide content but not lookup**
- Napster:
  - Provides central lookup database
  - Host-to-host HTTP data transfer
- Overlay networks
  - Idea: build a network of nodes that pass messages between each other
  - Examples: Gnutella, freenet
  - Lookup: flood requests / cache responses
  - Data transfer:
    - Gnutella: Host-to-host HTTP data transfer
    - Freenet: deliver on overlay with caching
- Swarming:
  - Idea: transfer data from multiple hosts simultaneously to provide better bandwidth
  - Examples: bittorrent
  - Does only data transfer, no lookup

What is P2P?



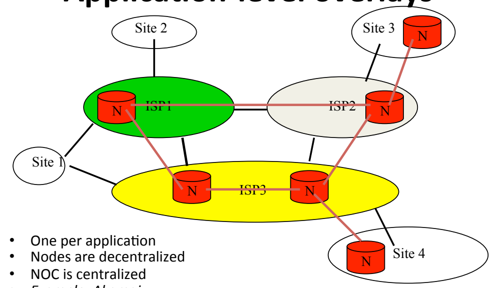
- A distributed system architecture:
  - No centralized control
  - Nodes are symmetric in function
- Typically many nodes, but unreliable and heterogeneous

Traditional distributed computing: client/server



- Successful architecture, and will continue to be so
- Tremendous engineering necessary to make server farms scalable and robust

Application-level overlays



- One per application
- Nodes are decentralized
- NOC is centralized
- *Example: Akamai*

P2P systems are overlay networks without central control

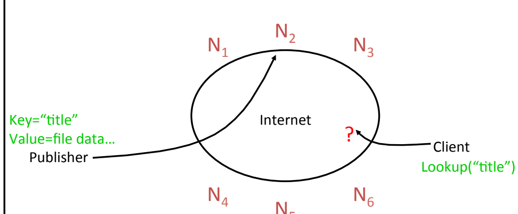
(Potential) P2P advantages

- Allows for scalable incremental growth
- Aggregate tremendous amount of computation and storage resources
- Contributory computing
  - Use idle storage capacity (file systems, backup), compute capacity (seti@home), network capacity (bittorrent)
- Tolerate faults or intentional attacks
  - Highly distributed, redundant
- Compare to DNS: hierarchical client/server
  - Owners for delegated pieces of content
  - Lookups proportional to name length
  - Suffers from hotspots (e.g. root)

Real P2P advantages

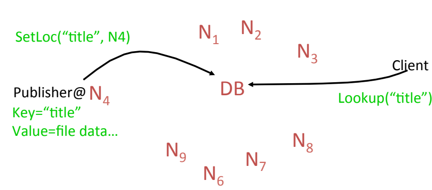
- Distribute large amounts of content without purchasing a lot of bandwidth
  - Bittorrent
  - BBC Iplayer
- Censorship resistance
  - No centralized services to take out
- Anonymity
  - Overlay routing conceals source of requests, data

Example P2P problem: lookup



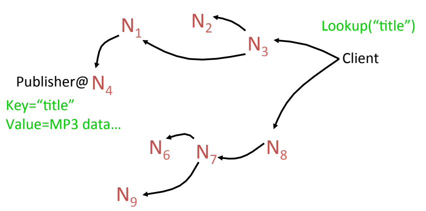
- At the heart of all P2P systems

Centralized lookup (Napster)



Simple, but O(N) state and a single point of failure

Flooded queries (Gnutella)



Robust, but worst case O(N) messages per lookup

Basic P2P service: lookup

- Finding things in a centralized system:
  - DNS: look at root, it delegates down tree to leaves
  - Web: ask google/baidu/bing
  - email: talk to your mail server, use DNS
- Finding things in a decentralized system:
  - You may not know all the nodes in the system
  - Nodes you knew may have left, new ones may have joined
- What can you do with lookup?
  - Find other clients of a service (e.g. skype users)
  - Find content (e.g. servers storing a file, clients willing to share a file)
- Other p2p services:
  - content distribution: reuse unmetered client bandwidth
- QUESTION: Why do you want a decentralized service?
  - Can deploy new service without paying an ISP!

Core ideas with Chord

- Chord service:
  - Given an object name, find a server
  - O(log n) state per node (don't need to know all members)
- Naming:
  - Objects & servers named with 160 bit SHA-1 hash
    - Note: using IP address for server prevents them from selecting their name to obtain data for specific objects
- Basic lookup: consistent hashing
  - Object stored at closest "successor" on ring of addresses
  - Nodes store next node in ring (successor)

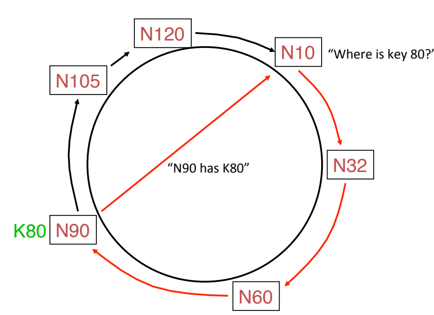
Chord goals

- Minimize state at client
  - Why? Is it because of storage space?
  - Or because it may change as the network changes (less state means less inconsistent state)
- Real tradeoff:
  - More state -> faster lookups, more overhead from churn
  - Less state -> slower lookups, less work on churn

Simple lookup

- Lookup(key) at any node n'
  - If key == n'
    - return "me"
  - if key > n' && key <= successor(n')
    - return successor(n')
  - else go ask successor(n')
- Linked list search – linear in number of nodes

Basic lookup (like Dynamo)



Faster lookup with Finger Tables

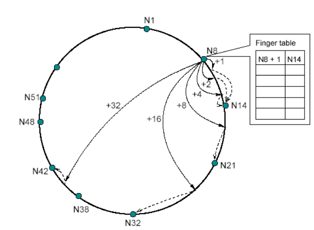
- Keep pointers to distant locations on the ring
  - Forward requests to a node preceding the region the answer is in
    - it will have more detailed information
  - Each forward cuts the portion of the ring left
- ith entry in the table at node n contains the first node s that succeds n by at least 2i-1
- s = successor (n + 2i-1)
- s is called the ith finger of node n

The Chord algorithm – Scalable node localization

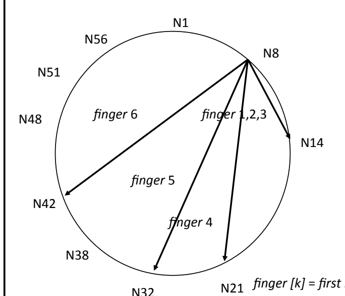
- Additional routing information to accelerate lookups
- Each node n contains a routing table with up to m entries (m: number of bits of the identifiers) => finger table
- ith entry in the table at node n contains the first node s that succeds n by at least 2i-1
- s = successor (n + 2^{i-1})
- s is called the i^{th} finger of node n

The Chord algorithm – Scalable node localization

Finger table:
finger[i] = successor (n + 2^{i-1})



Scalable Lookup Scheme



Finger Table for N8

N8+1	N14
N8+2	N14
N8+4	N14
N8+8	N21
N8+16	N32
N8+32	N42

finger [k] = first node that succeeds $(n+2^{k-1})\bmod 2^m$

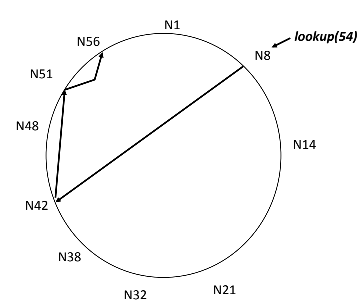
Scalable Lookup Scheme

```
// ask node n to find the successor of id
n.find_successor(id)
    if (id belongs to (n, successor])
      return successor;
    else
      n0 = closest preceding node(id);
      return n0.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
    for i = m downto 1
      if (finger[i] belongs to (n, id))
          return finger[i];
    return n;
```

Key: find predecessor of key, then ask its successor
QUESTION: Why?
- Only predecessor know truth about who is next in ring

Lookup Using Finger Table



Iterative vs Recursive lookup

- Iterative:
  - Lookup at a server, it returns a referral to a closer server
  - Client then contacts the next server
- Recursive:
  - Lookup at server, and it makes call to next server
- Choice:
  - Recursive: lower latency, but more load at servers and more failure prone (cannot easily retry)

Scalable Lookup Scheme

- Each node forwards query at least halfway along distance remaining to the target
- Theorem: With high probability, the number of nodes that must be contacted to find a successor in a N-node network is O(log N)

QUESTION: Consistency

- What is the Chord consistency model?
  - Any guarantees of eventual consistency or session consistency

Three steps in join

Step 0: all nodes maintain predessor & successor links (doubly linked list) to make it easy to insert

Step 1. Initialize predecessor and fingers of the new node. (finger[1]=successor)

Step 2. Update the predecessor and the fingers of the existing nodes. (Thus notify nodes that must include N20 in their table. N110[1] = N20, not N32.

Step 3. Transfer objects to the new node as appropriate.

The Join procedure

The new node id asks a gateway node n to find the successor of id

n.(find_successor(id)

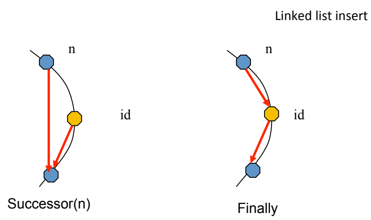
```
if    id ∈ (n, successor]
      then    return successor
      else    forward the query around the circle
fi
```

Needs O(n) messages. This is slow.

A More Efficient Join

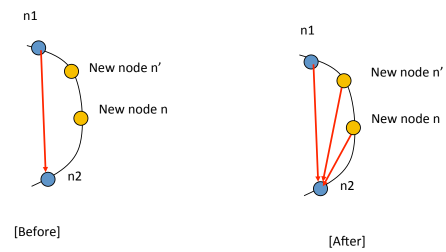
- Maintain predecessor pointers so nodes form a doubly linked list
  - so can insert in the middle O(1)
- new node N asks any node N' for its successor and fingers
  - standard lookup works here
  - Update successor's predecessor
- Ask other nodes to update their finger tables
  - Look for nodes 2^i before on ring, ask them to update their tables, scan backwards until no update needed

Steps in join



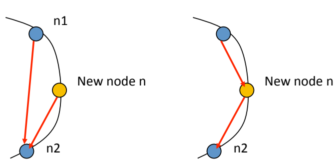
New node can tell N about itself.
N can ask its successor who its predecessor is
But the transition does not happen immediately

Concurrent Join



Stabilization

Periodic stabilization is needed to integrate the new node into the network and restore the invariant.
Key point: easier to stabilize than guarantee always correct like Chubby repair from failure



Predecessor.successor(n1) ≠ n1, so n1 adopts predecessor.successor(n1) = n as its new successor

Node Failures

- Key step in failure recovery is maintaining correct successor pointers
- To help achieve this, each node maintains a *successor-list* of its *r* nearest successors on the ring
- If node *n* notices that its successor has failed, it replaces it with the first live entry in the list
- Successor lists are stabilized as follows:
  - node n reconciles its list with its successor s by copying s's successor list, removing its last entry, and prepending s to it.
  - If node n notices that its successor has failed, it replaces it with the first live entry in its successor list and reconciles its successor list with its new successor.

Performance under failure

- Without replication, results show that lookups fail at same rate as nodes fail
  - small number of failures don't lead to larger failures (no single point of failure)
- While nodes join, some lookups fail
  - Not as robust as Dynamo
- Partitioning of Chord ring:
  - Results show that the network was not partitioned (if ring was completely severed, all lookups in other partition would fail)
- Partitioning of network:
  - Not addressed. How often is the Internet partitioned? What do partitions look like?

Replication

- What if a node you wanted to lookup goes down?
  - Replicate and next R nodes along ring (same as Dynamo)
  - Store next R successors at each node
- Load balancing
  - Don't have to follow finger table; can choose nodes from finger table that are geographically/latency closest

What can you build with Chord

- DHash: distributed hash table
  - Chord finds the node, DHash has get/put for data
- CFS: read-only file system
  - Store blocks of data using Chord and DHash, a DHT
- IVY: a read-write file system
  - Store logs of modifications using Chord & DHash
  - Client retrieves data by replaying client logs
- DNS: store record in Chord/DHash
- Arpeggio: search
  - Maintain distributed index over metadata terms
  - Store index for each "term" in dhash
  - Client joins using AND for conjunctive searches, OR,
  - Client sends terms to server, which compares metadata for file (only need to query one server now), OR,
  - Precompute results for up to K terms (canonical order), and do search that way

Challenges to Contributory P2P

- High churn means storing large amounts of data is expensive
  - Have to keep copying data if used for backup
- Requires external security mechanism
  - E.g. public key certificates for authenticating content
- Best suited as a dedicated piece of infrastructure with low admin costs

Security problems in P2P

- As noted in paper:
  - Attacker can generate content that hashes to a specific node to overload it or exploit a vulnerability (placement attacks)
  - Attacker can generate identity to take a place in the ring
    - Can choose a location that interesting content is stored on- and subvert the content
  - Attacker can create arbitrary many identities
    - One host could have 1000's of nodes in a Chord ring
    - How could you tell these are real nodes?
      - "Sybil attack"

What is the value of P2p

- Academically interesting exercise
  - High challenges to having:
    - no centralization,
    - frequent churn,
    - Untrusted machines
    - Extreme heterogeneity
    - Geographic diversity
  - Pieces useful in data center settings
    - Dynamo vs. chord – use table for lookup instead
    - Scalable Key-value stores – dynamo, memcached, etc.
    - Akamai content distribution – send data via overlay rather than directly

Willy Zaenepoel on P2P

- Decentralized is harder/more complex than centralized
  - P2P tries to make this a feature, yet few real applications demand true decentralization except illegality
  - But this yields more research papers
- P2P problems
  - Hard to find data (Chord)
  - Hard to secure (Sybil attack, no root of trust)
  - Hard to write programs
  - These all lead to more papers
- Real benefits of P2P: content distribution
  - Solved by BitTorrent, not P2P research
- P2P has low impact
  - What are the natural uses of decentralized systems?
  - Which of the goals of P2P are actually best solved by P2P?