Lecture 6: Overlay Networks

CS 598: Advanced Internetworking
Matthew Caesar
February 15, 2011
Overlay networks: Motivations

- Protocol changes in the network happen very slowly

- Why?
  - Internet is shared infrastructure; need to achieve consensus
  - Many proposals require to change a large number of routers (e.g. IP Multicast, QoS); otherwise end-users won’t benefit

- Proposed changes that haven’t happened yet on large scale:
  - More addresses (IPv6, 1991)
  - Security (IPSEC, 1993); Multicast (IP multicast, 1990)
Overlay networks: Motivations

• Also, “one size does not fit all”

• Applications need different levels of
  – Reliability
  – Performance (latency
  – Security
  – Access control (e.g., who is allowed to join a multicast group)
Overlay networks: Goals

- Make it easy to deploy new functionalities in the network → Accelerate the pace of innovation
- Allow users to customize their service
Solution

- Build a computer network on top of another network
  - Individual hosts autonomously form a “virtual” network on top of IP
  - Virtual links correspond to inter-host connections (e.g., TCP sessions)
Example:
Resilient Overlay Networks

- Premise: by building an application-layer overlay network, can increase performance and reliability of routing
- Install N computers at different Internet locations
- Each computer acts like an overlay network router
  - Between each overlay router is an IP tunnel (logical link)
  - Logical overlay topology is all-to-all ($N^2$ total links)
- Run a link-state routing algorithm over the overlay topology
  - Computers measure each logical link in real time for packet loss rate, throughput, latency → these define link costs
  - Route overlay traffic based on measured characteristics
Motivating example: a congested network
Solution: an "overlay" network

Machines remember overlay topology, probe links, advertise link quality

A

Path taken by TCP sessions

C

Establish TCP sessions ("overlay links") between hosts

Loss=1%

Loss=2%

Loss=25%

Machines remember overlay topology, probe links, advertise link quality

Establish TCP sessions ("overlay links") between hosts
Benefits of overlay networks

• **Performance:**
  – Difficult to provide QoS at network-layer due to deployment hurdles, lack of incentives, application-specific requirements
  – Overlays can probe faster, propagate more routes

• **Flexibility:**
  – Difficult to deploy new functions at IP layer
  – Can perform multicast, anycast, QoS, security, etc
New problem: scalability

Problems:
Number of links increases with $O(n^2)$
Link-state overhead increases with $O(n^3)$!
Alternative: replace full-mesh with logical ring

Problem:
- Stretch increases with O(n)
- Still requires O(n) state per node
Alternative:
replace full-mesh with ring

Problem:
Stretch: $O(n)$

State:
still requires $O(n)$ state per node
Improvement:
Stretch: reduces to O(lg n)
State: reduces to O(lg n)
Scaling overlay networks with Distributed Hash Tables (DHTs)

• Assign each host a numeric identifier
  – Randomly chosen, hash of node name, public key, etc

• Keep pointers (fingers) to other nodes
  – Goal: maintain pointers so that you can reach any destination in few overlay hops
  – Choosing pointers smartly can give low delay, while retaining low state

• Can also store objects
  – Insert objects by “consistently” hashing onto id space

• Forward by making progress in id space
Different kinds of DHTs

- Different topologies give different bounds on stretch (delay penalty)/state, different stability under churn, etc. Examples:

  - **Chord**
    - Pointers to immediate successor on ring, nodes spaced $2^k$ around ring
    - Forward to numerically closest node without overshooting

  - **Pastry**
    - Pointers to nodes sharing varying prefix lengths with local node, plus pointer to immediate successor
    - Forward to numerically closest node

  - **Others:** Tapestry (like Pastry, but no successor pointers), CAN (like Chord, but torus namespace instead of ring)
The Chord DHT

“Successors” maintained for correctness

Each node assigned numeric identifier from circular id-space

“Fingers” maintained for performance

504+16

504+8

504+4

504+3

504+2

504+1

504
Chord Example: Forwarding a lookup

Dest = 802

Cuts namespace-distance in half per hop
You can divide any integer N in half at most log(N) times
= logarithmic stretch
Example: Joining a new node

1. Joining node must be aware of a “bootstrap” node in DHT. Joining node sends join request through bootstrap node towards the joining node’s ID

2. Bootstrap forwards message towards joining node’s ID, causing message to resolve to joining node’s future successor

3. Successor informs predecessor of its new successor, adds joining node as new predecessor
To improve robustness, each node can maintain more than one successor
   - E.g., maintain the K>1 successors immediately adjacent to the node

In the notify() message, node A can send its k-1 successors to its predecessor B

Upon receiving the notify() message, B can update its successor list by concatenating the successor list received from A with A itself
Chord: Discussion

• Query can be implemented
  – Iteratively
  – Recursively

• Performance: routing in the overlay network can be more expensive than routing in the underlying network
  – Because usually no correlation between node ids and their locality; a query can repeatedly jump from Europe to North America, though both the initiator and the node that store them are in Europe!
  – Solutions: can maintain multiple copies of each entry in their finger table, choose closest in terms of network distance
Goal: fill each “pointer table” entry with topologically-nearby nodes (1320 points to 2032 instead of 2211, even though they both fit in this position)

1320’s pointer table (base=4, digits=4)

<table>
<thead>
<tr>
<th>Increasing digit</th>
<th>Increasing prefix length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*: 0002</td>
<td>1*: 2032</td>
</tr>
<tr>
<td>10*: 1023</td>
<td>11*: 1103</td>
</tr>
<tr>
<td>130*: 131* 1310</td>
<td>132*: 1322*</td>
</tr>
<tr>
<td>1320* 1321 1321</td>
<td>1323 1323</td>
</tr>
</tbody>
</table>

No node fits here, leave blank
My own ID matches, can leave blank (next row down will have more specific match anyway)
Fixes one digit per hop 
+ logarithmic number of digits per hop 
= logarithmic stretch
3233’s pointer table (base=4, digits=4)

<table>
<thead>
<tr>
<th>0*:</th>
<th>0002</th>
<th>1*:</th>
<th>1320</th>
<th>2*:</th>
<th>2032</th>
<th>3*:</th>
</tr>
</thead>
<tbody>
<tr>
<td>30*:</td>
<td>31*:</td>
<td>3103</td>
<td>32*:</td>
<td>3211</td>
<td>33*:</td>
<td></td>
</tr>
<tr>
<td>320*:</td>
<td>321*:</td>
<td>3211</td>
<td>322*:</td>
<td>3221</td>
<td>323*:</td>
<td>3233</td>
</tr>
<tr>
<td>3230*:</td>
<td>3231*:</td>
<td>3232*:</td>
<td>3233*:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Join (dst=3231)
Content Addressable Network (CAN)

• Associate to each node and item a unique id in a d-dimensional space

• Properties
  – Routing table size $O(d)$
  – Guarantees that a file is found in at most $d \times n^{1/d}$ steps, where $n$ is the total number of nodes
CAN Example:
Two dimensional space

- Space divided between nodes
- All nodes cover the entire space
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
- Example:
  - Assume space size (8x8)
  - Node n1:(1,2) first node that joins
    - Cover the entire space
CAN Example:
Two dimensional space

- Node n2: (4,2) joins → space is divided between n1 and n2
CAN Example:
Two dimensional space

- Node n2:(4,2) joins \( \rightarrow \) space is divided between n1 and n2
• Nodes n4:(5,5) and n5:(6,6) join
CAN Example:
Two dimensional space

- **Nodes:**
  - n1:(1,2)
  - n2:(4,2)
  - n3:(3,5)
  - n4:(5,5)
  - n5:(6,6)

- **Items:**
  - f1(2,3)
  - f2(5,1)
  - f3:(2,1)
  - f4(7,5)
CAN Example: Two dimensional space

- Each item is stored at the node who owns the mapping in its space
CAN Example: Two dimensional space

• Query example:
• Each node knows its neighbors in the d-space
• Forward query to the neighbor that is closest to the query id
• Example: assume n1 queries f4
Preserving consistency

• What if a node fails?
  – Solution: probe neighbors to make sure alive, proactively replicate objects

• What if node joins in wrong position?
  – Solution: nodes check to make sure they are in the right order
  – Two flavors: weak stabilization, and strong stabilization
Chord Example: weak stabilization

Check: if my successor's predecessor is a better match for my successor

```python
def stabilize():
    x = successor.predecessor
    if (x in (n, successor)):
        successor = x
        successor.notify(n)
```

Tricky case: zero position on ring
Example where weak stabilization fails

```python
n.stabilize():
x = successor.predecessor;
if (x in (n, successor))
    successor = x
    successor.notify(n)
```
# Comparison of DHT geometries

<table>
<thead>
<tr>
<th>Geometry</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring</td>
<td>Chord, Symphony</td>
</tr>
<tr>
<td>Hypercube</td>
<td>CAN</td>
</tr>
<tr>
<td>Tree</td>
<td>Plaxton</td>
</tr>
<tr>
<td>Hybrid = Tree + Ring</td>
<td>Tapestry, Pastry</td>
</tr>
<tr>
<td>XOR d(id1, id2) = id1 XOR id2</td>
<td>Kademlia</td>
</tr>
</tbody>
</table>
## Comparison of DHT algorithms

<table>
<thead>
<tr>
<th></th>
<th>Node Degree</th>
<th>Dilation</th>
<th>Congestion</th>
<th>Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chord</td>
<td>$\log(n)$</td>
<td>$\log(n)$</td>
<td>$\log(n)/n$</td>
<td>hypercube</td>
</tr>
<tr>
<td>Tapestry</td>
<td>$\log(n)$</td>
<td>$\log(n)$</td>
<td>$\log(n)/n$</td>
<td>hypercube</td>
</tr>
<tr>
<td>CAN</td>
<td>$D$</td>
<td>$D^*(n^{1/D})$</td>
<td>$D^*(n^{1/D})/D$</td>
<td>D-dim torus</td>
</tr>
<tr>
<td>Small World</td>
<td>$O(1)$</td>
<td>$\log^2 n$</td>
<td>$(\log^2 n)/n$</td>
<td>Cube connected cycle</td>
</tr>
<tr>
<td>Viceroy</td>
<td>7</td>
<td>$\log(n)$</td>
<td>$\log(n)/n$</td>
<td>Butterfly</td>
</tr>
</tbody>
</table>

- **Node degree**: The number of neighbors per node
- **Dilation**: Length of longest path that any packet traverses in the network
  - **Stretch**: Ratio of longest path to shortest path through the underlying topology
- **Congestion**: maximum number of paths that use the same link
Security issues

• Sybil attacks
  – Malicious node pretends to be many nodes
  – Can take over large fraction of ID space, files

• Eclipse attacks
  – Malicious node intercepts join requests, replies with its cohorts as joining node’s fingers

• Solutions:
  – Perform several joins over diverse paths, PKI, leverage social network relationships, audit by sharing records with neighbors
Hashing in networked software

• Hash table: maps identifiers to keys
  – Hash function used to transform key to index (slot)
  – To balance load, should ideally map each key to different index

• Distributed hash tables
  – Stores values (e.g., by mapping keys and values to servers)
  – Used in distributed storage, load balancing, peer-to-peer, content distribution, multicast, anycast, botnets, BitTorrent’s tracker, etc.
Background: hashing

keys

Ahmed
Yan
John
Viraj

function

hashes

00
01
02
03
04
05
08
Example: Sum ASCII digits, mod number of bins
Problem: _______________
Solution: Consistent Hashing

- Hashing function that reduces churn
- Addition or removal of one slot does not significantly change mapping of keys to slots
- Good consistent hashing schemes change mapping of K/N entries on single slot addition
  - K: number of keys
  - N: number of slots
- E.g., map keys and slots to positions on circle
  - Assign keys to closest slot on circle
Solution: Consistent Hashing

- Slots have IDs selected randomly from [0,100]
- Hash keys onto same space, map key to closest bin
- Less churn on failure → more stable system

**Function**

- **Ahmed**
  - \[ 65 + 72 + 77 + 69 + 68 = 351 \]
  - \[ 351 \mod 100 = 51 \]

- **Yan**
  - \[ 89 + 65 + 78 = 232 \]
  - \[ 232 \mod 100 = 32 \]

- **John**
  - \[ 74 + 79 + 72 + 78 = 303 \]
  - \[ 303 \mod 100 = 3 \]

- **Viraj**
  - \[ 86 + 73 + 82 + 65 + 74 = 380 \]
  - \[ 380 \mod 100 = 80 \]