Today

- Overlay networks and Peer-to-Peer
Motivation

- Suppose you want to write a routing protocol to replace IP
  - But your network administrator prevents you from writing arbitrary data on your network
- What can you do?
  - You have a network that can send packets between arbitrary hosts (IP)
- You could...
  - Pretend that the point-to-point paths in the network are *links* in an overlay network...
Overlay Networks

• Users want innovation

• Change is *very* slow on the Internet (e.g. IPv6!)
  – Require consensus (IETF)
  – Lots of money sunk in existing infrastructure

• Solution: don’t require change in the network!
  – Use IP paths, deploy your own processing among nodes
Why would you want that anyway?

• Doesn’t the network provide you with what you want?
  – What if you want to teach a class on how to implement IP? (IP on top of UDP… sounds familiar?)
  – What if Internet routing is not ideal?
  – What if you want to test out new multicast algorithms, or IPv6?

• Remember…
  – The Internet started as an overlay over ossified telephone networks!
Case Studies

• Resilient Overlay Network
• Peer-to-peer systems
• Others (won’t cover today)
  – Email
  – Web
  – End-system Multicast
  – Your IP programming assignment
  – VPNs
  – Some IPv6 deployment solutions
  – …
Resilient Overlay Network - RON

- **Goal:** increase performance and reliability of routing
- **How?**
  - Deploy N computers in different places
  - Each computer acts as a router between the N participants
- **Establish IP tunnels between all pairs**
- **Constantly monitor**
  - Available bandwidth, latency, loss rate, etc…
- **Route overlay traffic based on these measurements**
RON

Default IP path determined by BGP & OSPF

Reroute traffic using red alternative overlay network path, avoid congestion point

Acts as overlay router

Picture from Ion Stoica
RON

• **Does it scale?**
  – Not really, only to a few dozen nodes (NxN)

• **Why does it work?**
  – Route around congestion
  – In BGP, policy trumps optimality

• **Example**
  – 2001, one 64-hour period: 32 outages over 30 minutes
  – RON routed around failure in 20 seconds

Peer-to-Peer Systems

• How did it start?
  – A killer application: file distribution
  – Free music over the Internet! (not exactly legal…)

• Key idea: share storage, content, and bandwidth of individual users
  – Lots of them

• Big challenge: coordinate all of these users
  – In a scalable way (not NxN!)
  – With changing population (aka churn)
  – With no central administration
  – With no trust
  – With large heterogeneity (content, storage, bandwidth,…)

3 Key Requirements

• **P2P Systems do three things:**
  • Help users *determine what they want*
    – Some form of search
    – P2P version of Google
  • **Locate that content**
    – Which node(s) hold the content?
    – P2P version of DNS (map name to location)
  • **Download the content**
    – Should be efficient
    – P2P form of Akamai
Napster (1999)
Napster
Napster
Napster

• Search & Location: central server
• Download: contact a peer, transfer directly
• Advantages:
  – Simple, advanced search possible
• Disadvantages:
  – Single point of failure (technical and … legal!)
  – The latter is what got Napster killed
Gnutella: Flooding on Overlays (2000)

- Search & Location: flooding (with TTL)
- Download: direct

An “unstructured” overlay network
Gnutella: Flooding on Overlays
Gnutella: Flooding on Overlays
Gnutella: Flooding on Overlays
KaZaA: Flooding w/ Super Peers (2001)

- Well connected nodes can be installed (KaZaA) or self-promoted (Gnutella)
Say you want to make calls among peers

• You need to find who to call
  – Centralized server for authentication, billing

• You need to find where they are
  – Can use central server, or a decentralized search, such as in KaZaA

• You need to call them
  – What if both of you are behind NATs? (only allow outgoing connections)
  – You could use another peer as a relay…
Skype

• Built by the founders of KaZaA!
• Uses Superpeers for registering presence, searching for where you are
• Uses regular nodes, outside of NATs, as decentralized relays
  – This is their killer feature
• This morning, from my computer:
  – 25,456,766 people online
Lessons and Limitations

• **Client-server performs well**
  – But not always feasible

• **Things that flood-based systems do well**
  – Organic scaling
  – Decentralization of visibility and liability
  – Finding popular stuff
  – Fancy *local* queries

• **Things that flood-based systems do poorly**
  – Finding unpopular stuff
  – Fancy *distributed* queries
  – Vulnerabilities: data poisoning, tracking, etc.
  – Guarantees about anything (answer quality, privacy, etc.)
BitTorrent (2001)

- One big problem with the previous approaches
  - Asymmetric bandwidth
- BitTorrent (original design)
  - Search: independent search engines (e.g. PirateBay, isoHunt)
    - Maps keywords -> .torrent file
  - Location: centralized tracker node per file
  - Download: chunked
    - File split into many pieces
    - Can download from many peers
BitTorrent

• How does it work?
  – Split files into large pieces (256KB ~ 1MB)
  – Split pieces into subpieces
  – Get peers from tracker, exchange info on pieces

• Three-phases in download
  – Start: get a piece as soon as possible (random)
  – Middle: spread pieces fast (rarest piece)
  – End: don’t get stuck (parallel downloads of last pieces)
BitTorrent

- Self-scaling: incentivize sharing
  - If people upload as much as they download, system scales with number of users (no free-loading)
- Uses *tit-for-tat*: only upload to who gives you data
  - *Choke* most of your peers (don’t upload to them)
  - Order peers by download rate, choke all but P best
  - Occasionally unchoke a random peer (might become a nice uploader)

- Optional reading:
Structured Overlays: DHTs

- Academia came (a little later)...
- **Goal:** Solve *efficient decentralized location*
  - Remember the second key challenge?
  - Given ID, map to host
- **Remember the challenges?**
  - Scale to millions of nodes
  - Churn
  - Heterogeneity
  - Trust (or lack thereof)
    - Selfish and malicious users
DHTs

• IDs from a flat namespace
  – Contrast with hierarchical IP, DNS

• Metaphor: hash table, but distributed

• Interface
  – Get(key)
  – Put(key, value)

• How?
  – Every node supports a single operation:
    Given a key, route messages to node holding key
Identifier to Node Mapping Example

- Node 8 maps [5, 8]
- Node 15 maps [9, 15]
- Node 20 maps [16, 20]
- ...
- Node 4 maps [59, 4]

- Each node maintains a pointer to its successor

Example from Ion Stoica
Remember Consistent Hashing?

- But each node only knows about a small number of other nodes (so far only their successors)
• Each node maintains its successor

• Route packet (ID, data) to the node responsible for ID using successor pointers
Stabilization Procedure

• Periodic operation performed by each node N to handle joins

N: periodically:
    STABILIZE \rightarrow N\.successor;

M: upon receiving STABILIZE from N:
    NOTIFY(M\.predecessor) \rightarrow N;

N: upon receiving NOTIFY(M') from M:
    if (M' between (N, N\.successor))
    N\.successor = M';
Joining Operation

- Node with id=50 joins the ring
- Node 50 needs to know at least one node already in the system
  - Assume known node is 15

\[
\begin{align*}
\text{succ}&=58 \\
\text{pred}&=35
\end{align*}
\]
Joining Operation

- Node 50: send join(50) to node 15
- Node 44: returns node 58
- Node 50 updates its successor to 58
- Node 50: send stabilize() to node 58
- Node 58:
  - update predecessor to 50
  - send notify() back
Joining Operation (cont’d)

- Node 44 sends a stabilize message to its successor, node 58.
- Node 58 reply with a notify message.
- Node 44 updates its successor to 50.
- Node 44 sends a stabilize message to its new successor, node 50
- Node 50 sets its predecessor to node 44
Joining Operation (cont’d)

- This completes the joining operation!
Achieving Efficiency: *finger tables*

Finger Table at 80

<table>
<thead>
<tr>
<th>$i$</th>
<th>$ft[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>96</td>
</tr>
<tr>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>3</td>
<td>96</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>5</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
</tr>
</tbody>
</table>

Say $m=7$

...(diagram of finger table with calculations and peer entries)...

The $i$th entry at peer with id $n$ is first peer with id $\geq n + 2^i (\text{mod } 2^m)$
Chord

• There is a tradeoff between routing table size and diameter of the network
• Chord achieves diameter $O(\log n)$ with $O(\log n)$-entry routing tables
Many other DHTs

- **CAN**
  - Routing in n-dimensional space
- **Pastry/Tapestry/Bamboo**
  - (Book describes Pastry)
  - Names are fixed bit strings
  - Topology: hypercube (plus a ring for fallback)
- **Kademlia**
  - Similar to Pastry/Tapestry
  - But the ring is ordered by the XOR metric
  - Used by BitTorrent for distributed tracker
- **Viceroy**
  - Emulated butterfly network
- **Koorde**
  - DeBruijn Graph
  - Each node connects to 2n, 2n+1
  - Degree 2, diameter log(n)
- ...
Discussion

• **Query can be implemented**
  – Iteratively: easier to debug
  – Recursively: easier to maintain timeout values

• **Robustness**
  – Nodes can maintain \((k>1)\) successors
  – Change notify() messages to take that into account

• **Performance**
  – Routing in overlay can be worse than in the underlay
  – Solution: flexibility in neighbor selection
    • Tapestry handles this implicitly (multiple possible next hops)
    • Chord can select any peer between \([2^n,2^{n+1})\) for finger, choose the closest in latency to route through
Where are they now?

- Many P2P networks shut down
  - Not for technical reasons!
  - Centralized systems work well (or better) sometimes

- But…
  - Vuze network: Kademlia DHT, millions of users
  - Skype uses a P2P network similar to KaZaA
Where are they now?

• DHTs allow coordination of MANY nodes
  – Efficient flat namespace for routing and lookup
  – Robust, scalable, fault-tolerant

• If you can do that
  – You can also coordinate co-located peers
  – Now dominant design style in datacenters
    • E.g., Amazon’s Dynamo storage system
  – DHT-style systems everywhere

• Similar to Google’s philosophy
  – Design with failure as the common case
  – Recover from failure only at the highest layer
  – Use low cost components
  – Scale out, not up
Next time

- It’s about the data
  - How to encode it, compress it, send it…