CSCI-1680
Web Performance, Content Distribution
P2P

John Jannotti

Based partly on lecture notes by Scott Shenker and Rodrigo Fonseca

Last time

- **HTTP and the WWW**
- **Today: HTTP Performance**
  - Persistent Connections, Pipeline, Multiple Connections
  - Caching
  - Content Distribution Networks

HTTP Performance

- **What matters for performance?**
- **Depends on type of request**
  - Lots of small requests (objects in a page)
  - Some big requests (large download or video)

Small Requests

- **Latency matters**
- **RTT dominates**
- **Two major causes:**
  - Opening a TCP connection
  - Actually sending the request and receiving response
  - And a third one: DNS lookup!

How can we reduce the number of connection setups?

- **Keep the connection open and request all objects serially**
  - Works for all objects coming from the same server
  - Which also means you don’t have to “open” the window each time
- **Persistent connections (HTTP/1.1)**

Browser Request

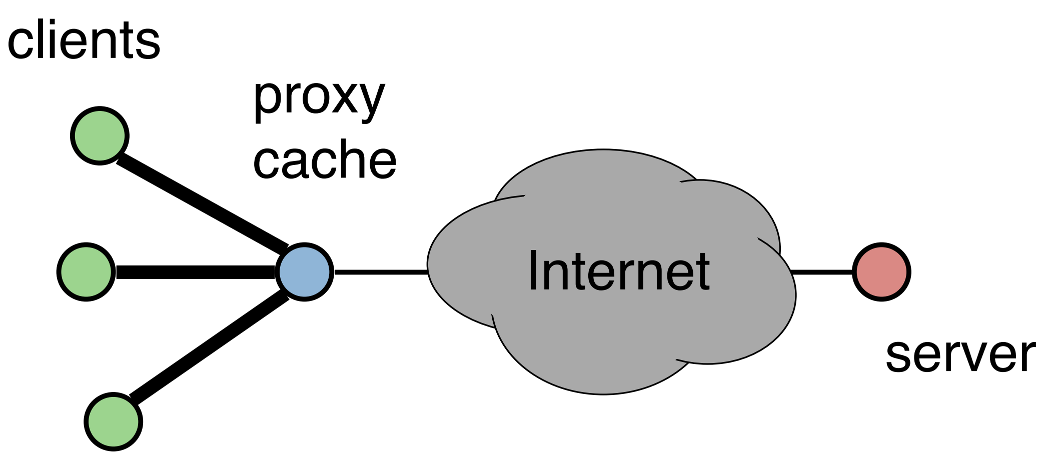
```
GET / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macinto ...
Accept: text/xml,application/xm ...
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

Small Requests (cont)

- **Second problem is that requests are serialized**
  - Similar to stop-and-wait protocols!
- **Two solutions**
  - Pipelined requests (similar to sliding windows)
  - Parallel Connections
    - HTTP standard says no more than 2 concurrent connections per host name
    - Most browsers use more (up to 8 per host, ~35 total)
      - See http://www.browserscope.org/
  - How are these two approaches different?

Larger Objects

- **Problem is throughput in bottleneck link**
- **Solution: HTTP Proxy Caching**
  – Also improves latency, and reduces server load

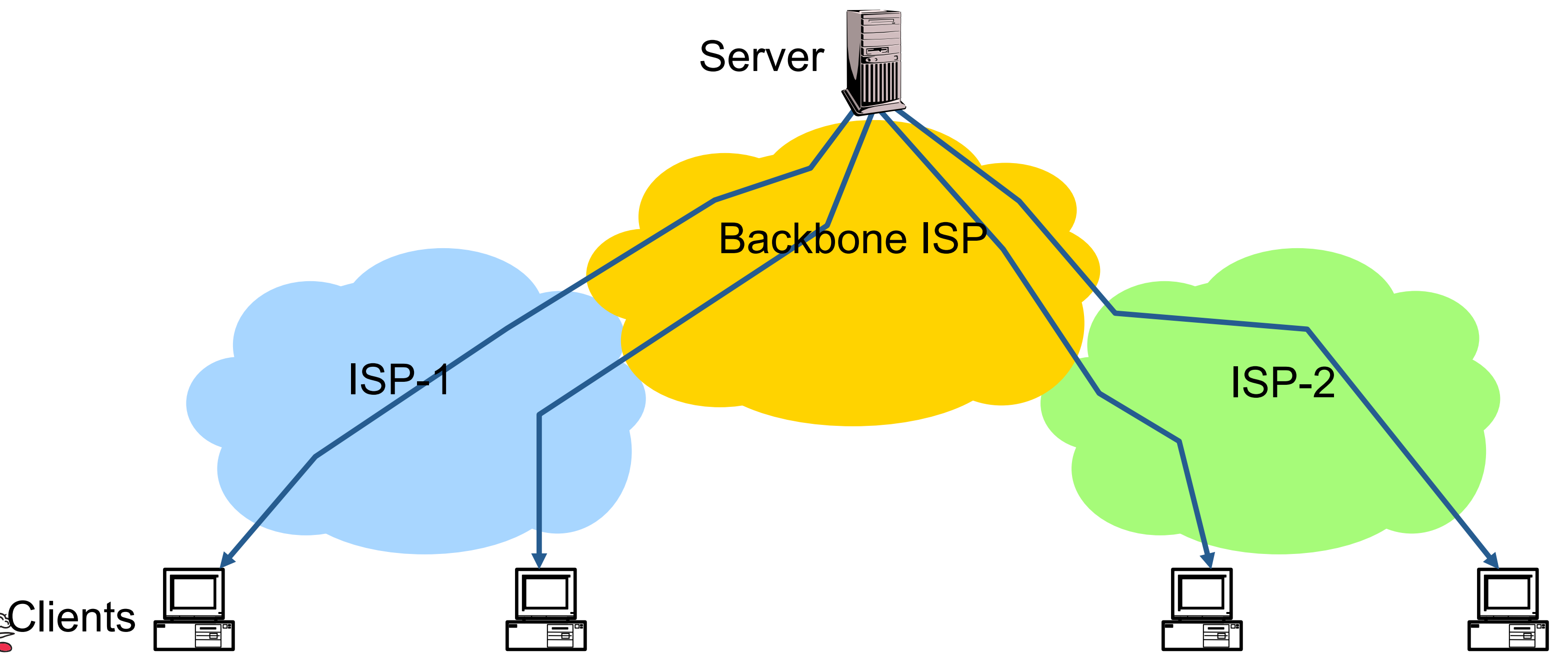


How to Control Caching?

- **Server sets options**
  - **Expires** header
  - No-Cache header
- **Client can do a conditional request:**
  - Header option: if-modified-since
  - Server can reply with 304 NOT MODIFIED

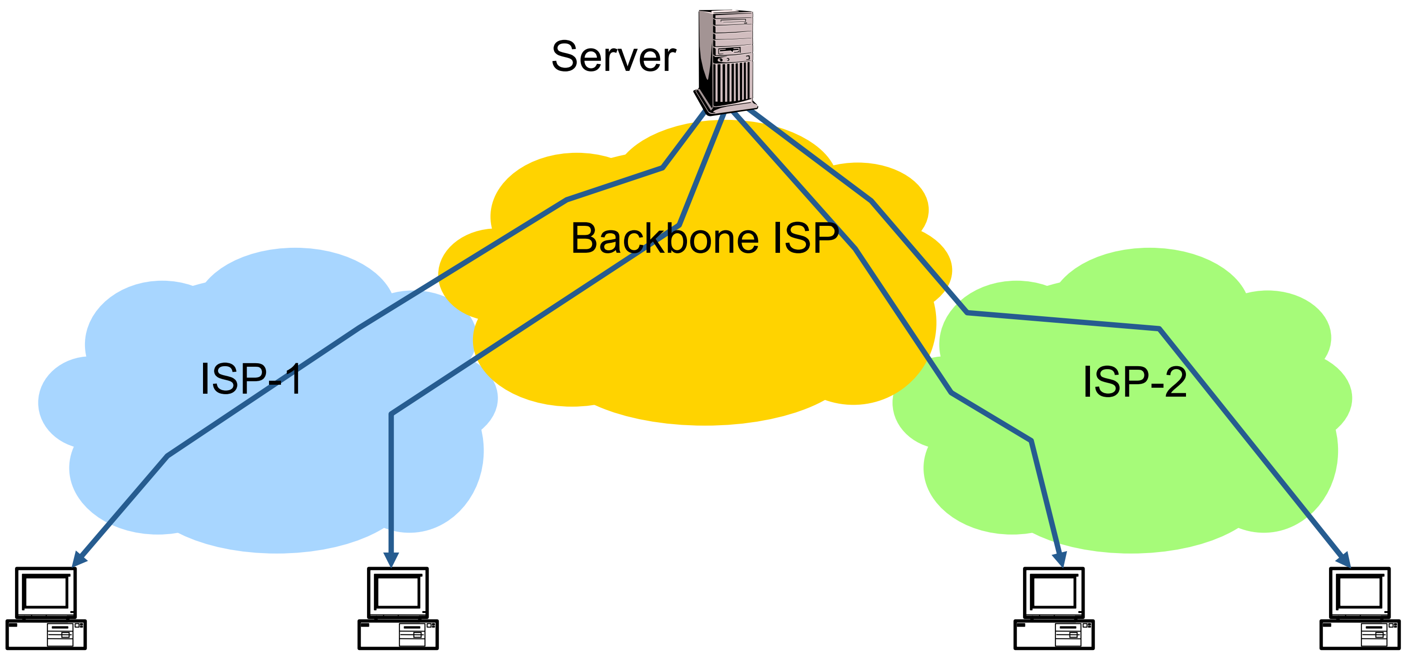
Caching

- **Where to cache content?**
  - Client (browser): avoid extra network transfers
  - Server: reduce load on the server
  - Service Provider: reduce external traffic



Caching

- **Why caching works?**
  - Locality of reference:
    - Users tend to request the same object in succession
    - Some objects are popular: requested by many users



How well does caching work?

- **Very well, up to a point**
  - Large overlap in requested objects
  - Objects with one access place upper bound on hit ratio
  - Dynamic objects not cacheable*
- **Example: Wikipedia**
  - About 400 servers, 100 are HTTP Caches (Squid)
  - 85% Hit ratio for text, 98% for media

* But can cache portions and run special code on edges to reconstruct

HTTP Cache Control

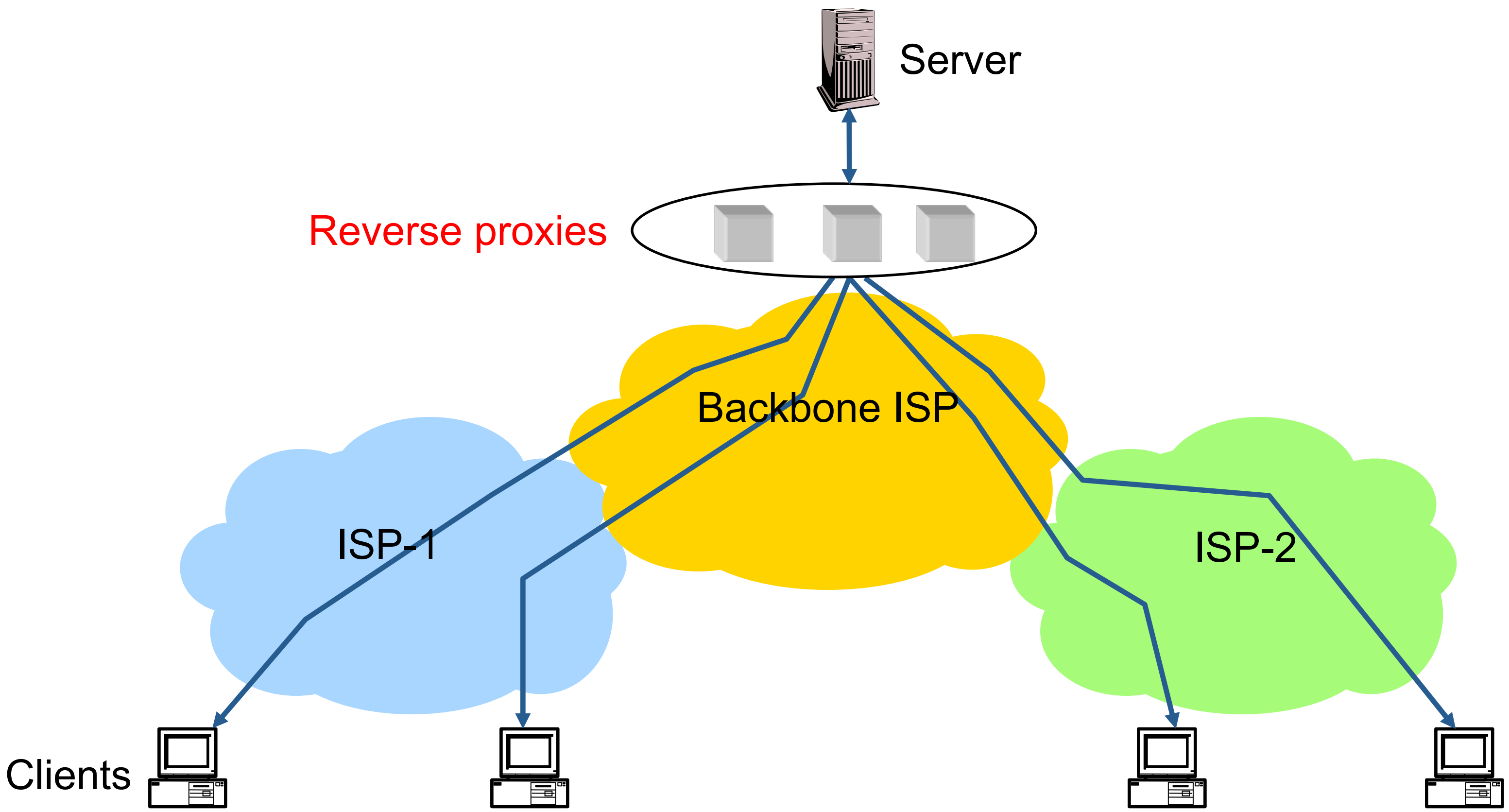
```
Cache-Control   = "Cache-Control" ":" 1#cache-directive
cache-directive = cache-request-directive
| cache-response-directive
cache-request-directive =
  "no-cache"                          ; Section 14.9.1
| "no-store"                          ; Section 14.9.2
| "max-age" "=" delta-seconds         ; Section 14.9.3, 14.9.4
| "max-stale" [ "=" delta-seconds ]   ; Section 14.9.3
| "min-fresh" "=" delta-seconds       ; Section 14.9.3
| "no-transform"                      ; Section 14.9.5
| "only-if-cached"                    ; Section 14.9.4
| cache-extension                     ; Section 14.9.6

cache-response-directive =
  "public"                               ; Section 14.9.1
| "private" [ "=" <"> 1#field-name <"> ] ; Section 14.9.1
| "no-cache" [ "=" <"> 1#field-name <"> ]; Section 14.9.1
| "no-store"                             ; Section 14.9.2
| "no-transform"                         ; Section 14.9.5
| "must-revalidate"                      ; Section 14.9.4
| "proxy-revalidate"                     ; Section 14.9.4
| "max-age" "=" delta-seconds            ; Section 14.9.3
| "s-maxage" "=" delta-seconds           ; Section 14.9.3
| cache-extension                        ; Section 14.9.6

cache-extension = token [ "=" ( token | quoted-string ) ]
```

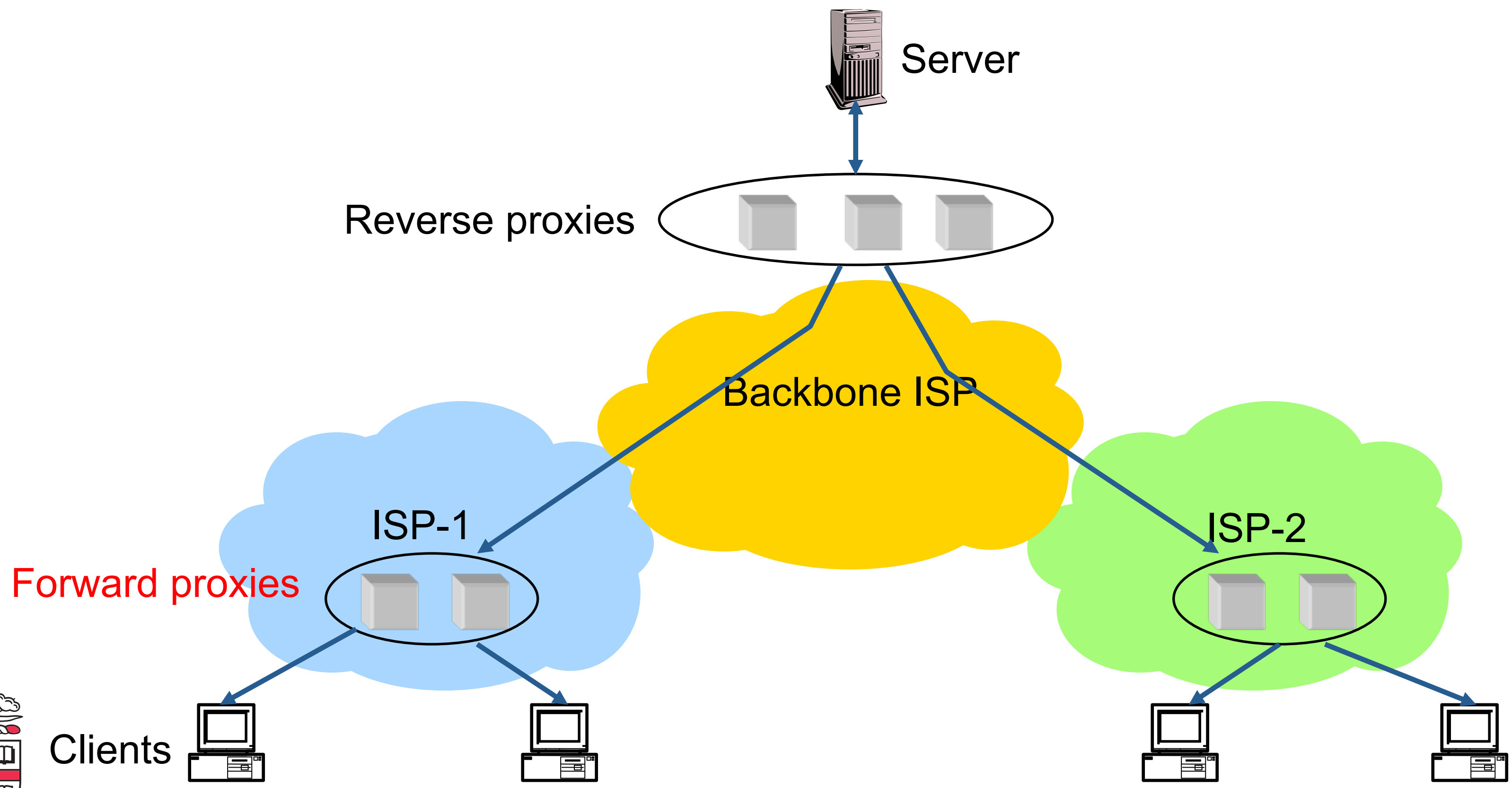
Reverse Proxies

- **Close to the server**
  - Also called Accelerators
  - Only work for static content



Forward Proxies

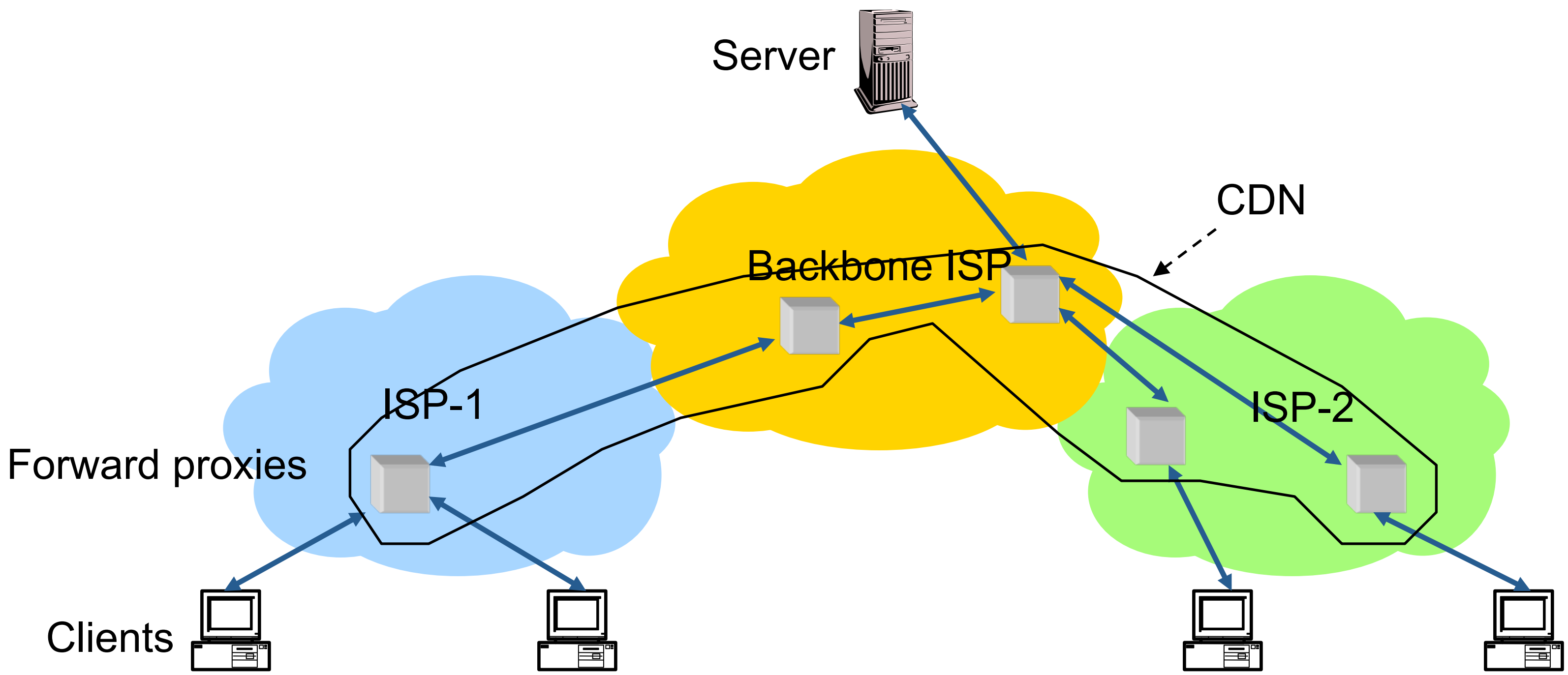
- **Typically done by ISPs or Enterprises**
  - Reduce network traffic and decrease latency
  - May be transparent or configured



Content Distribution Networks

- **Integrate forward and reverse caching**
  - One network generally administered by one entity
  - E.g. Akamai
- **Provide document caching**
  - Pull: result from client requests
  - Push: expectation of high access rates to some objects
- **Can also do some processing**
  - Deploy code to handle some dynamic requests
  - Can do other things, such as transcoding

Example CDN



How Akamai works

- **Akamai has cache servers deployed close to clients**
  - Co-located with many ISPs
- **Challenge: make same domain name resolve to a proxy close to the client**
- **Lots of DNS tricks. BestBuy is a customer**
  - Delegate name resolution to Akamai (via a CNAME)
- **From Brown:**

```
dig www.bestbuy.com
;; ANSWER SECTION:
www.bestbuy.com. 3600     IN   CNAME    www.bestbuy.com.edgesuite.net.
www.bestbuy.com.edgesuite.net. 21600 IN    CNAME    a1105.b.akamai.net.
a1105.b.akamai.net.   20   IN   A    198.7.236.235
a1105.b.akamai.net.   20   IN   A    198.7.236.240
```

  - Ping time: 2.53ms
- **From Berkeley, CA:**

```
a1105.b.akamai.net.   20   IN   A    198.189.255.200
a1105.b.akamai.net.   20   IN   A    198.189.255.207
```

  - Ping time: 3.20ms

DNS Resolution

```
dig www.bestbuy.com
;; ANSWER SECTION:
www.bestbuy.com. 3600     IN   CNAME    www.bestbuy.com.edgesuite.net.
www.bestbuy.com.edgesuite.net. 21600 IN    CNAME    a1105.b.akamai.net.
a1105.b.akamai.net.   20   IN   A    198.7.236.235
a1105.b.akamai.net.   20   IN   A    198.7.236.240
;; AUTHORITY SECTION:
b.akamai.net.     1101 IN   NS   n1b.akamai.net.
b.akamai.net.     1101 IN   NS   n0b.akamai.net.
;; ADDITIONAL SECTION:
n0b.akamai.net.   1267 IN   A    24.143.194.45
n1b.akamai.net.   2196 IN   A    198.7.236.236
```

- **n1b.akamai.net finds an edge server close to the client's local resolver**
  - Uses knowledge of network: BGP feeds, traceroutes. *Their secret sauce…*

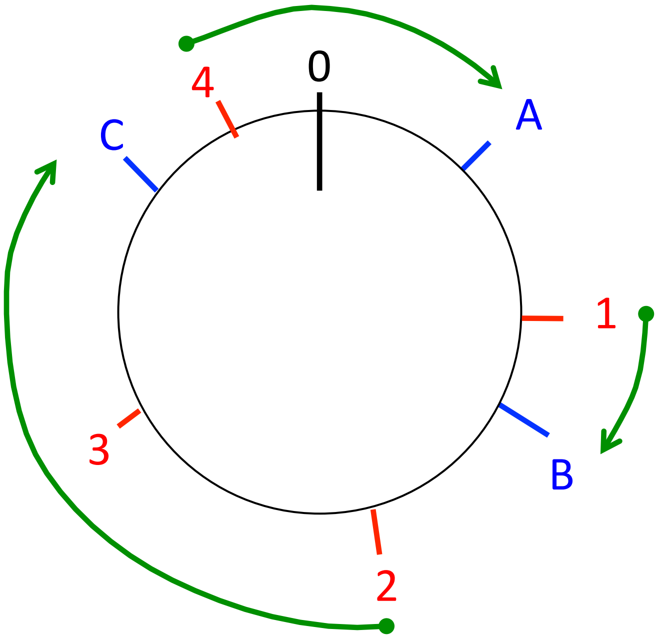
What about the content?

- **Say you are Akamai**
  - Clusters of machines close to clients
  - Caching data from many customers
  - Proxy fetches data from *origin* server first time it sees a URL
- **Choose cluster based on client network location**
- **How to choose server within a cluster?**
- **If you choose based on client**
  - Low hit rate: N servers in cluster means N cache misses per URL

Straw man: modulo hashing

- **Say you have N servers**
- **Map requests to proxies as follows:**
  - Number servers 0 to N-1
  - Compute hash of URL: h = hash (URL)
  - Redirect client to server #p = h mod N
- **Keep track of load in each proxy**
  - If load on proxy #p is too high, try again with a different hash function (or “salt”)
- **Problem: most caches will be useless if you add or remove proxies, change value of N**

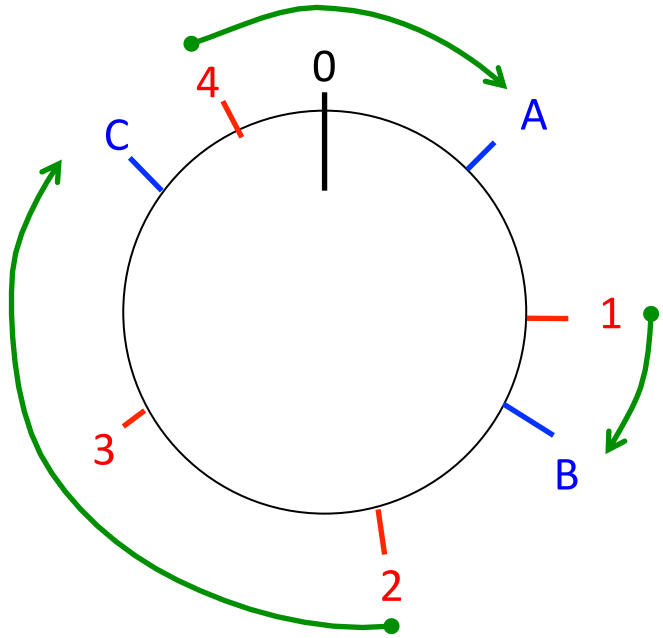
Consistent Hashing [Karger et al., 99]



Object	Cache
1	B
2	C
3	C
4	A

- **URLs and Caches are mapped to points on a circle using a hash function**
- **A URL is assigned to the closest cache clockwise**
- **Minimizes data movement on change!**
  - When a cache is added, only the items in the preceding segment are moved
  - When a cache is removed, only the next cache is affected

Consistent Hashing [Karger et al., 99]



Object	Cache
1	B
2	C
3	C
4	A

- **Minimizes data movement**
  - If 100 caches, add/remove a proxy invalidates ~1% of objects
  - When proxy overloaded, spill to successor
- **Can also handle servers with different capacities. How?**
  - Give bigger proxies more random points on the ring

Summary

- **HTTP Caching can greatly help performance**
  - Client, ISP, and Server-side caching
- **CDNs make it more effective**
  - Incentives, push/pull, well provisioned
  - DNS and Anycast tricks for finding close servers
  - Consistent Hashing for smartly distributing load

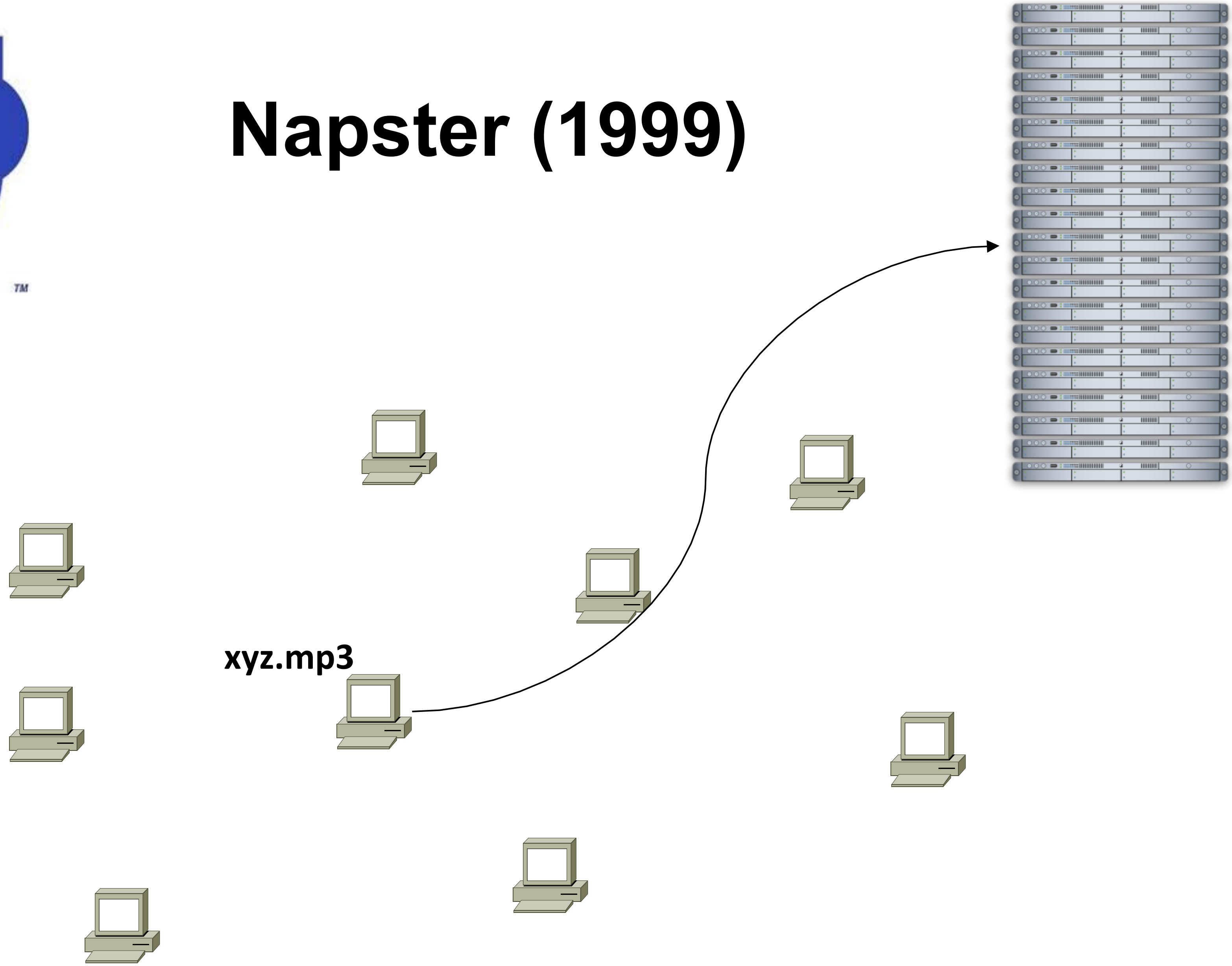
Peer-to-Peer Systems

- **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 little trust
  - With large heterogeneity (content, storage, bandwidth,…)

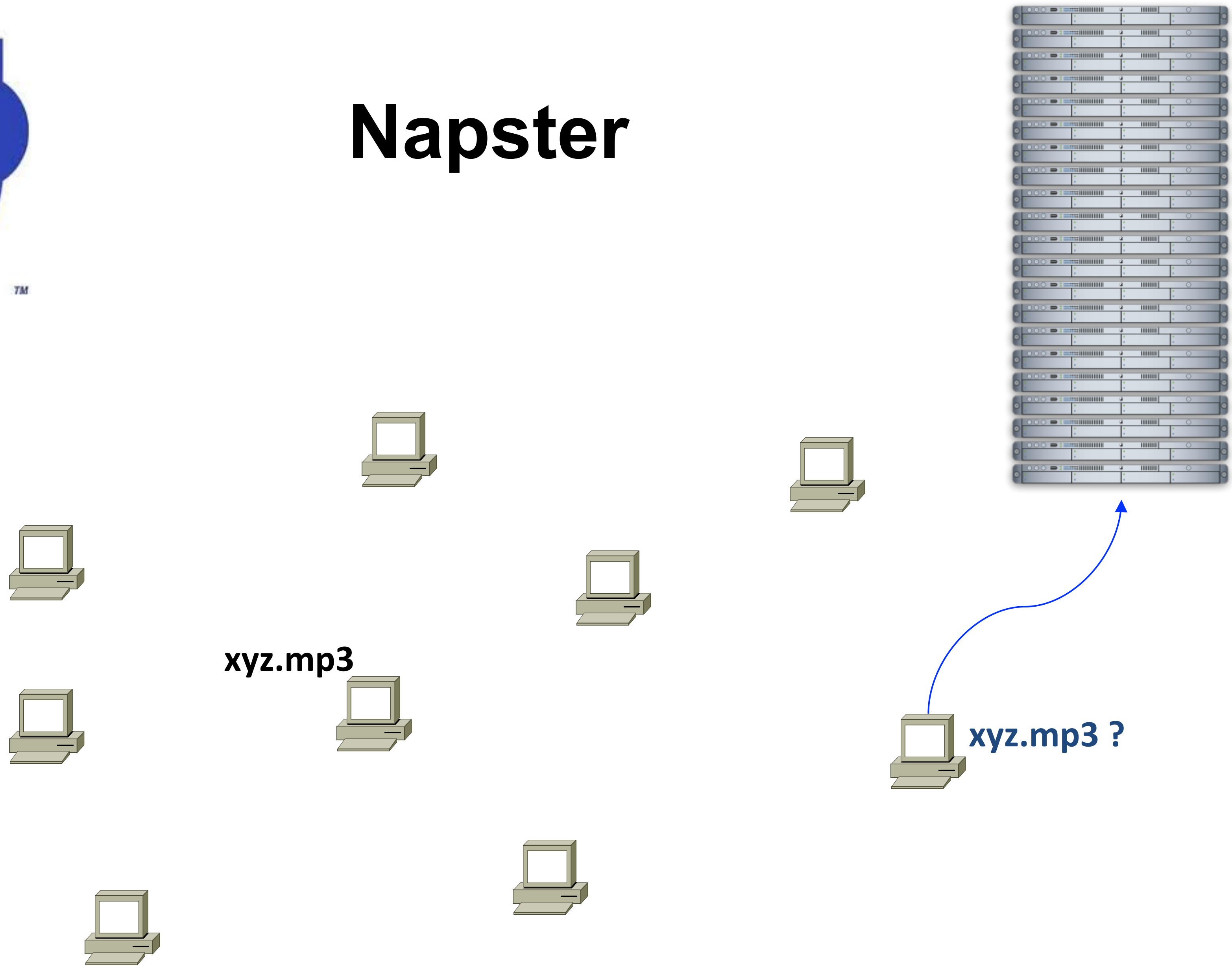
3 Key Requirements

- **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

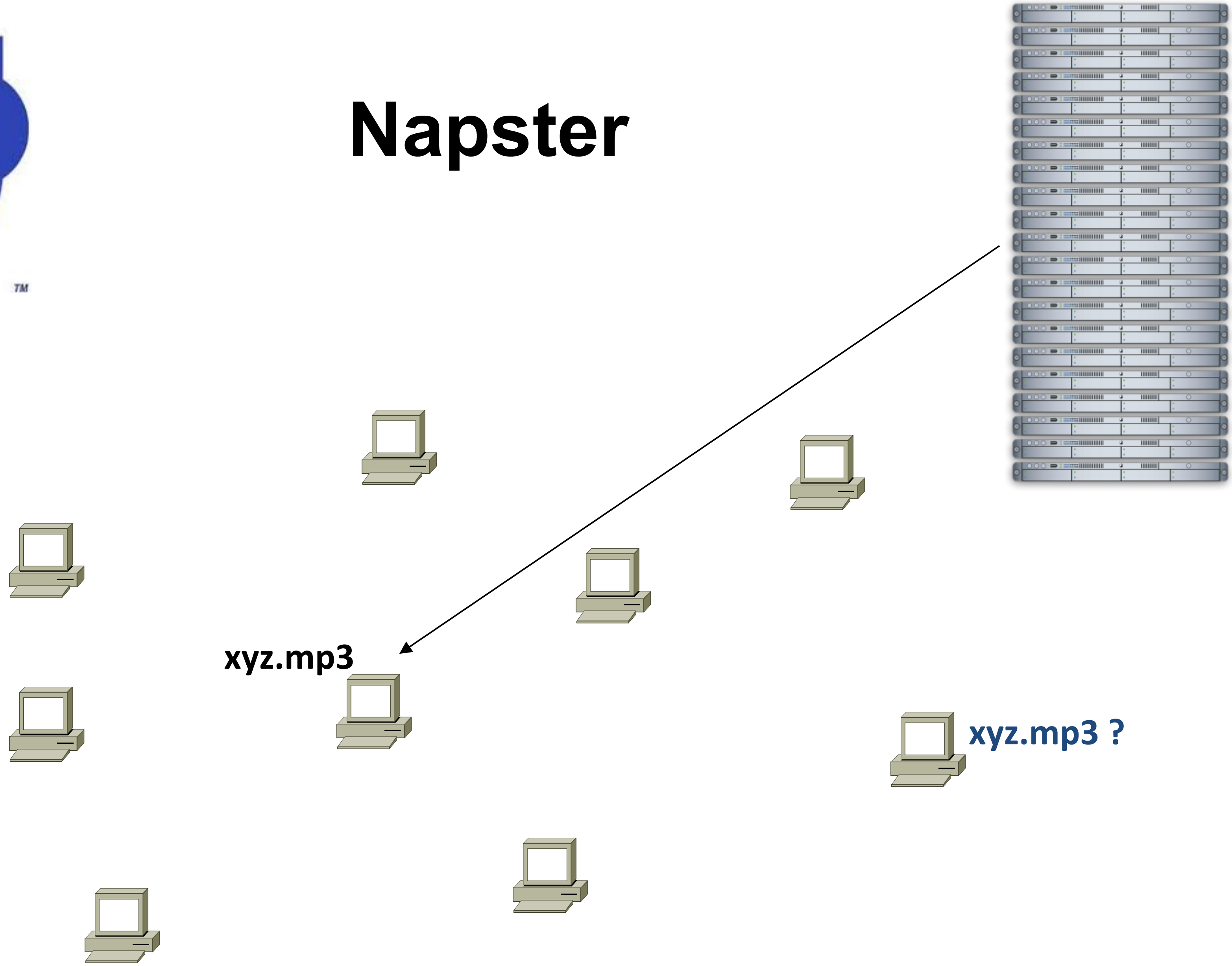
Napster (1999)



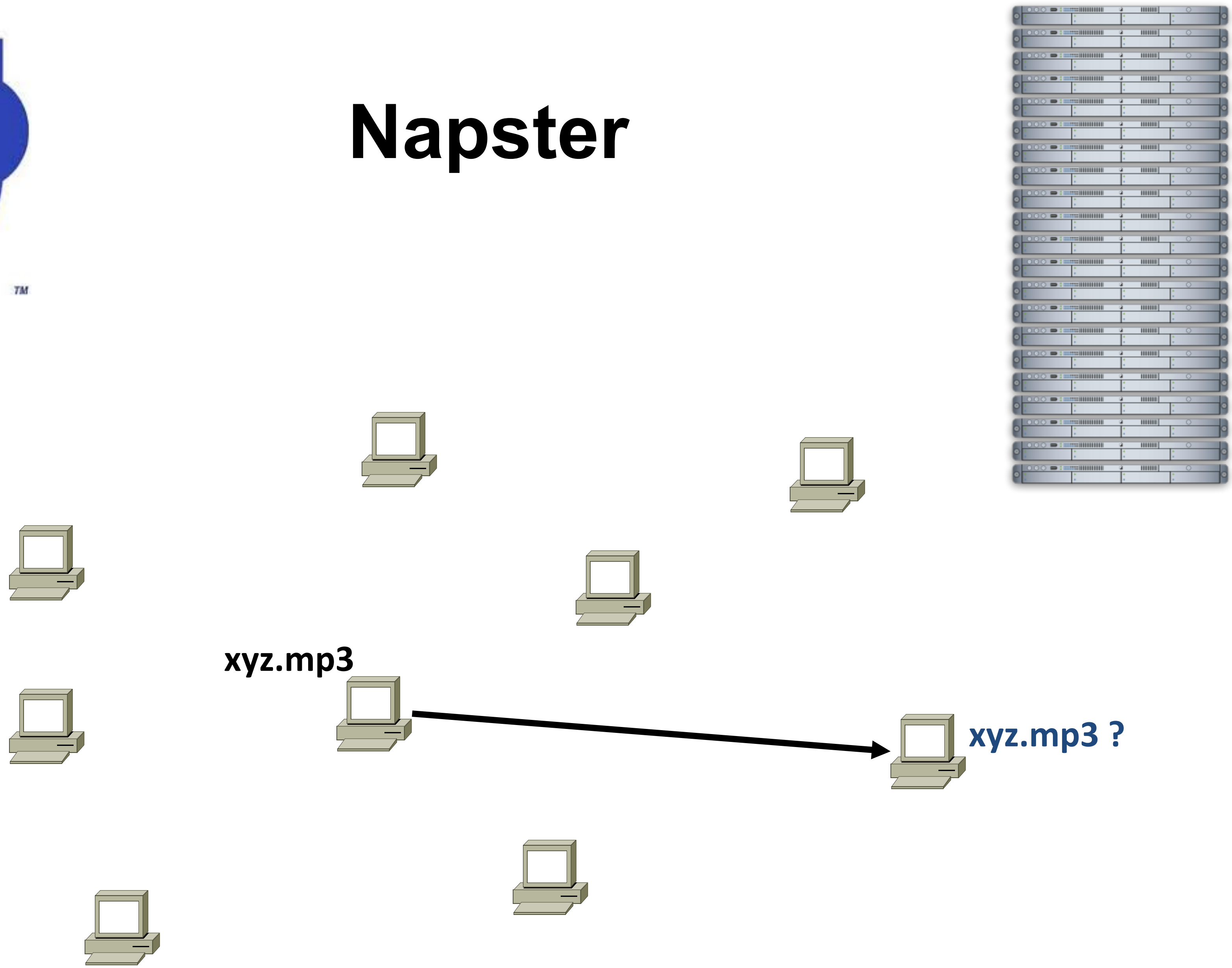
Napster



Napster



Napster

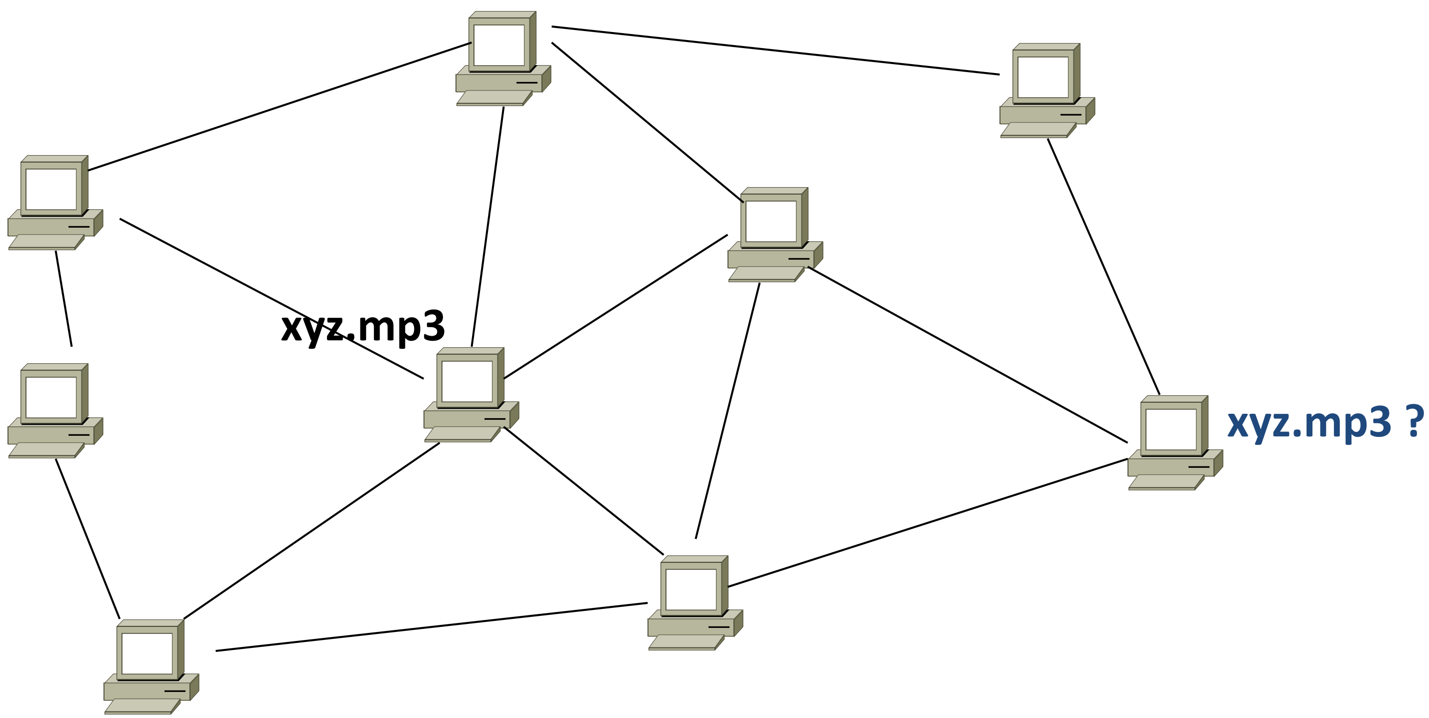


Napster

- **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

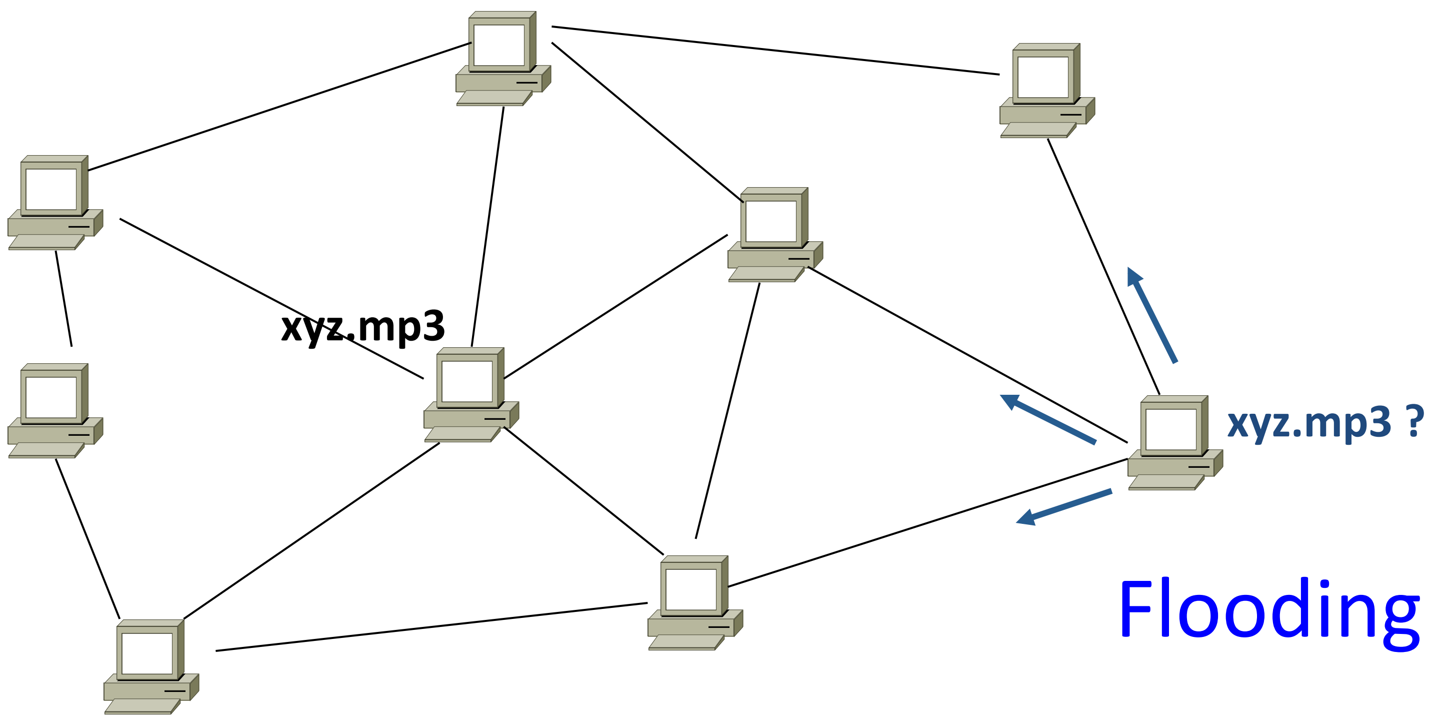
Gnutella: Flooding on Overlays (2000)

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

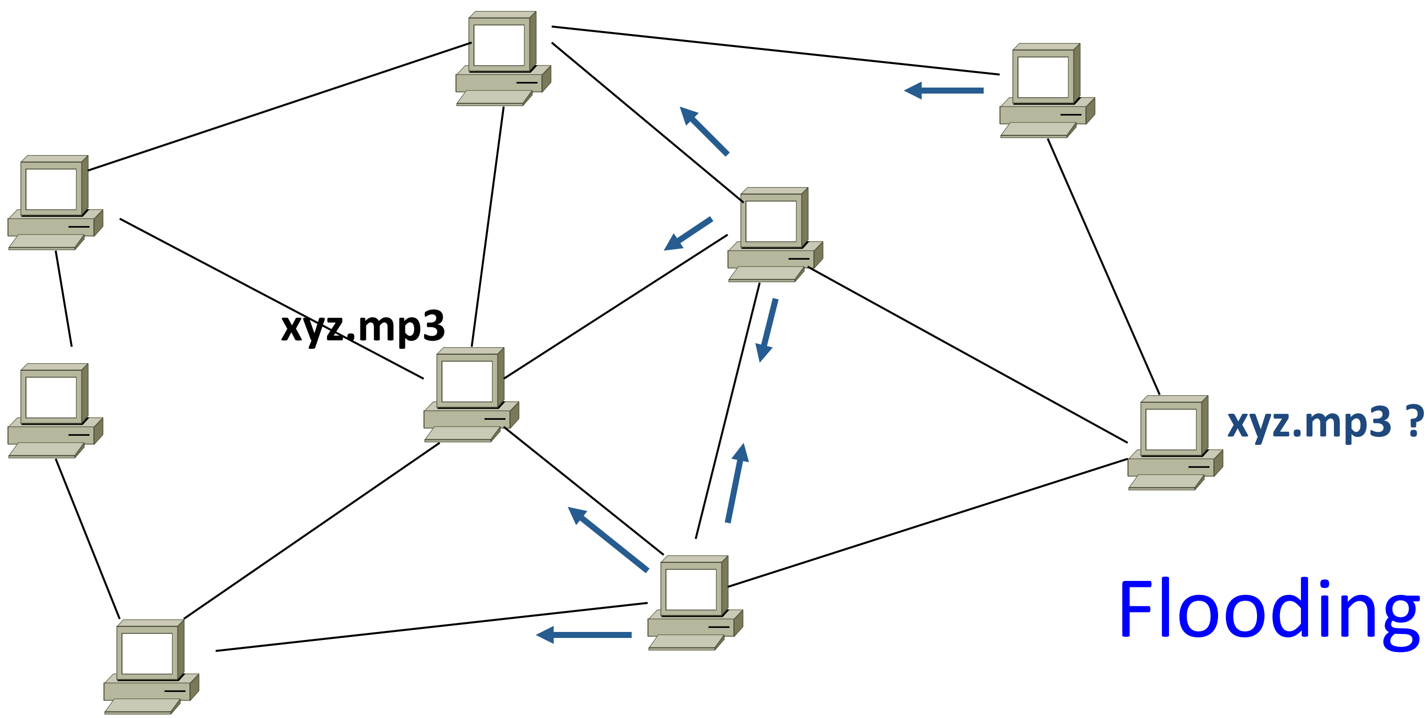


An “unstructured” *overlay network*

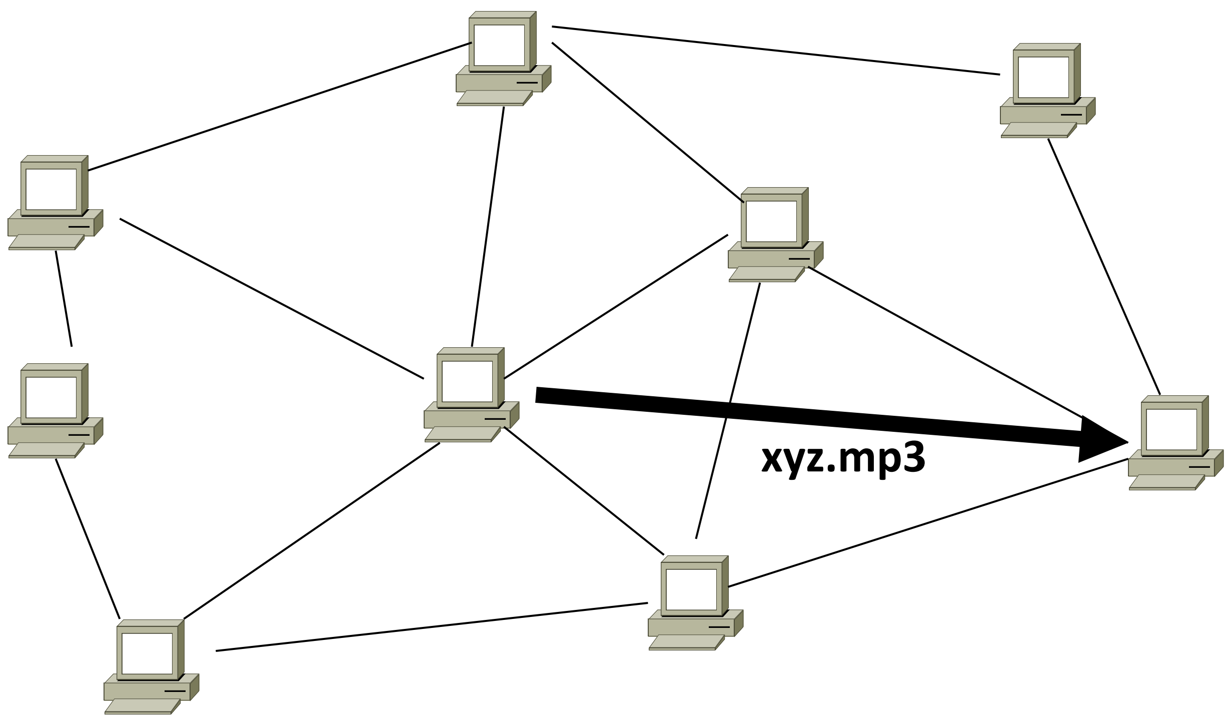
Gnutella: Flooding on Overlays



Gnutella: Flooding on Overlays

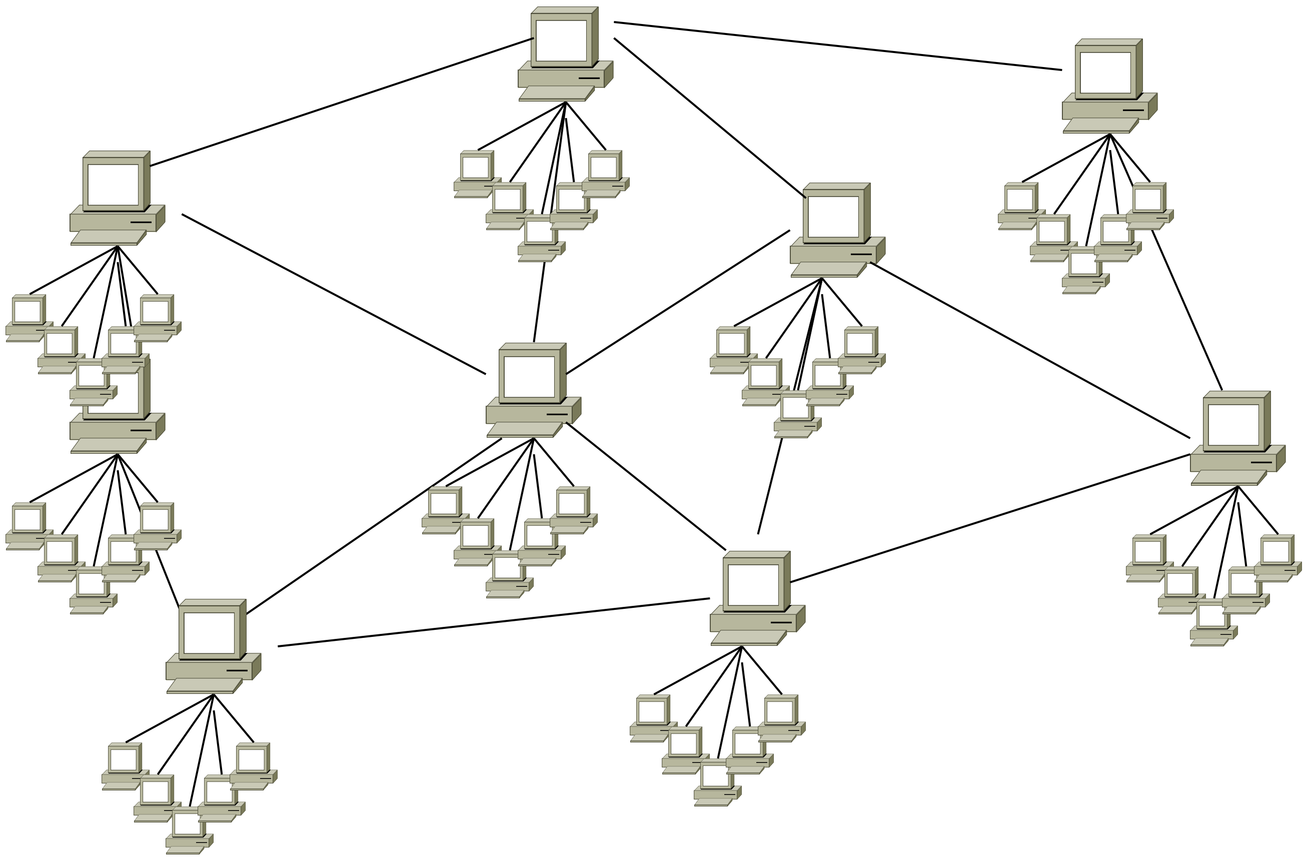


Gnutella: Flooding on Overlays



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
- **One morning, from Rodrigo's computer:**
  - 29,565,560 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:**
  Do Incentives Build Robustness in BitTorrent?
  Piatek et al, NSDI'07

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

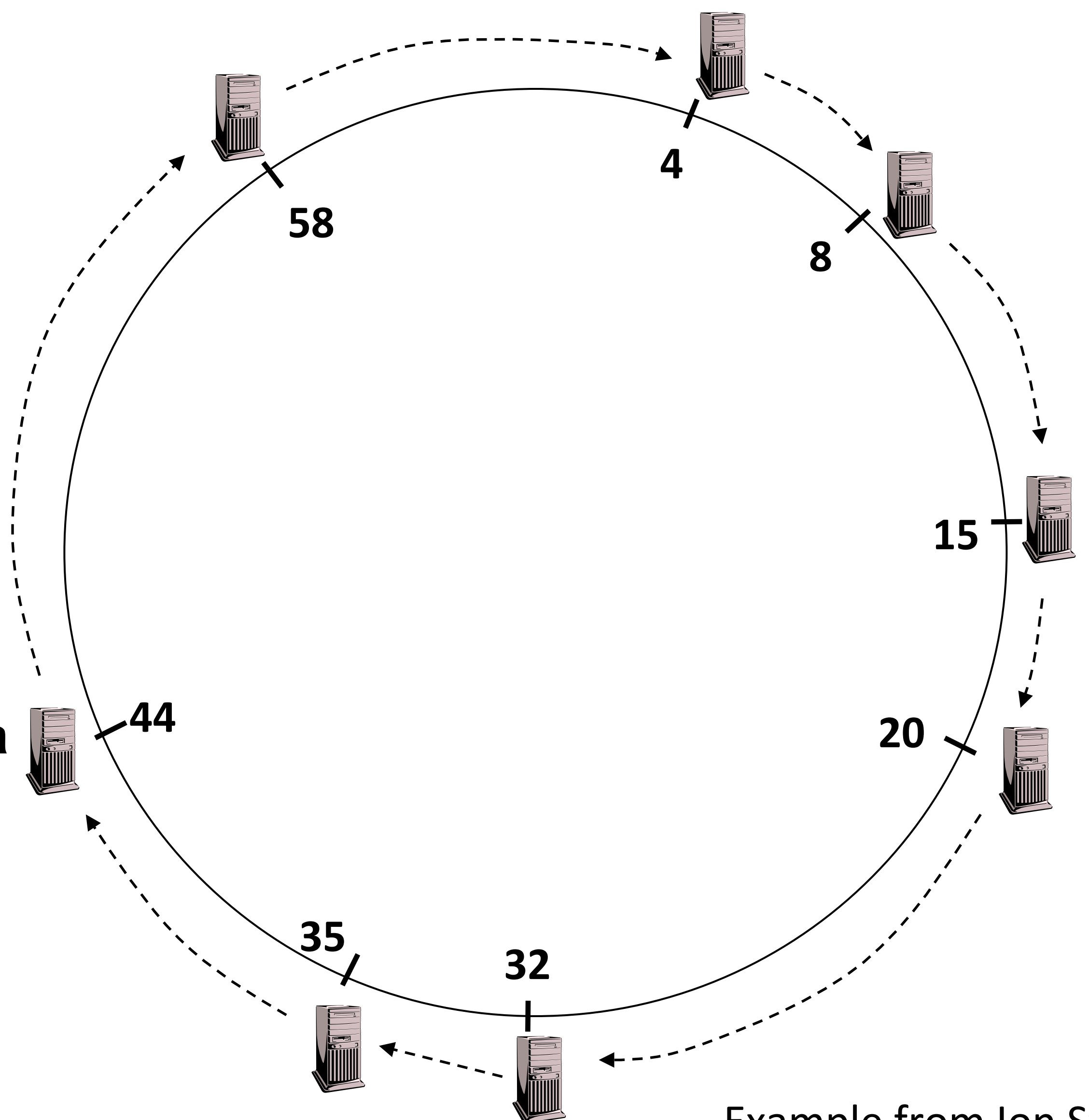
- **IDs from a *flat* namespace**
  - Contrast with hierarchical IP, DNS
- **Metaphor: hash table, but distributed**
- **Interface**
  - Get(key)
  - Put(key, value)
- **How?**
  - Every node supports a single operation:

    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]

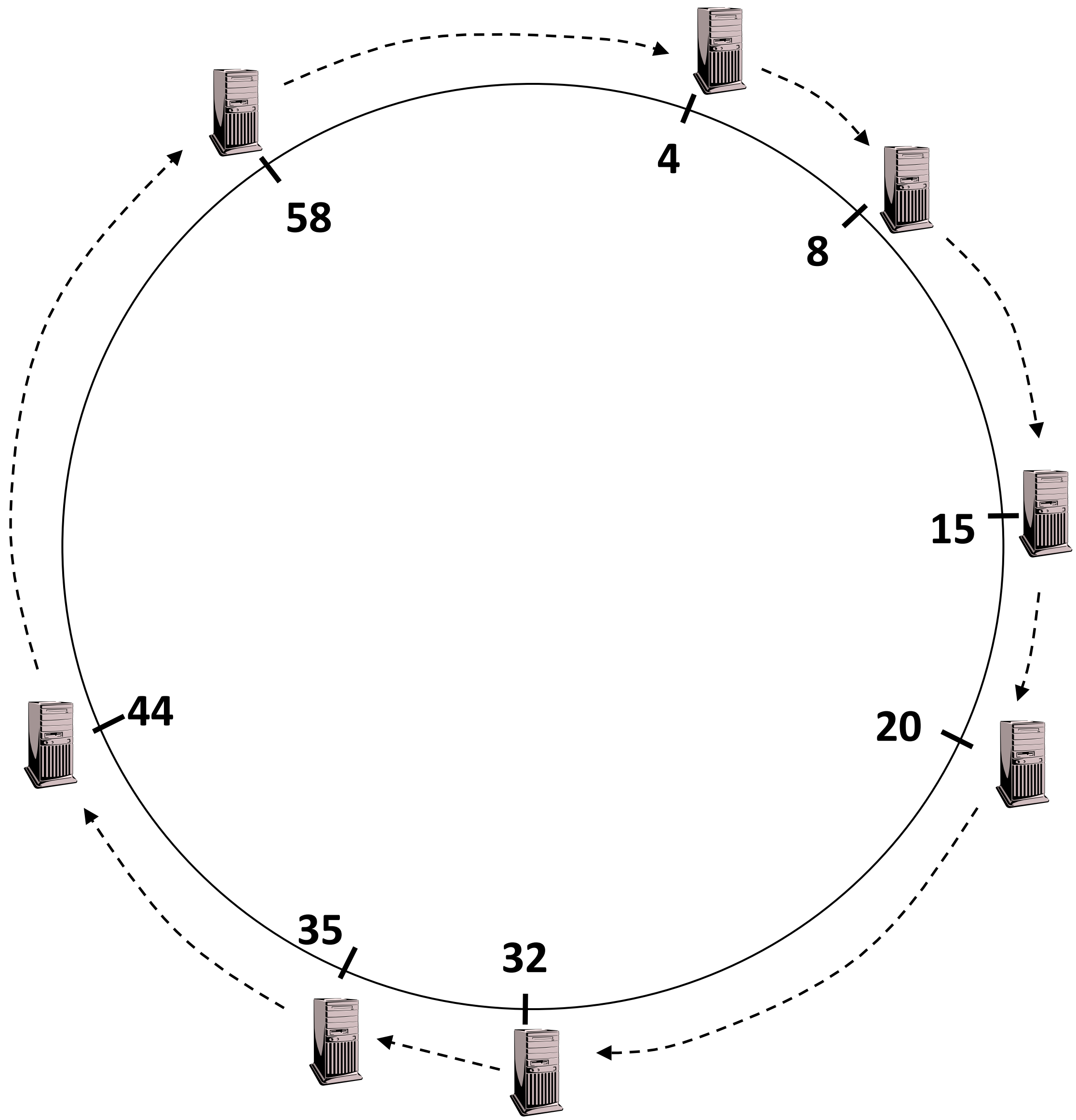
- Each node maintains a pointer to its successor



Example from Ion Stoica

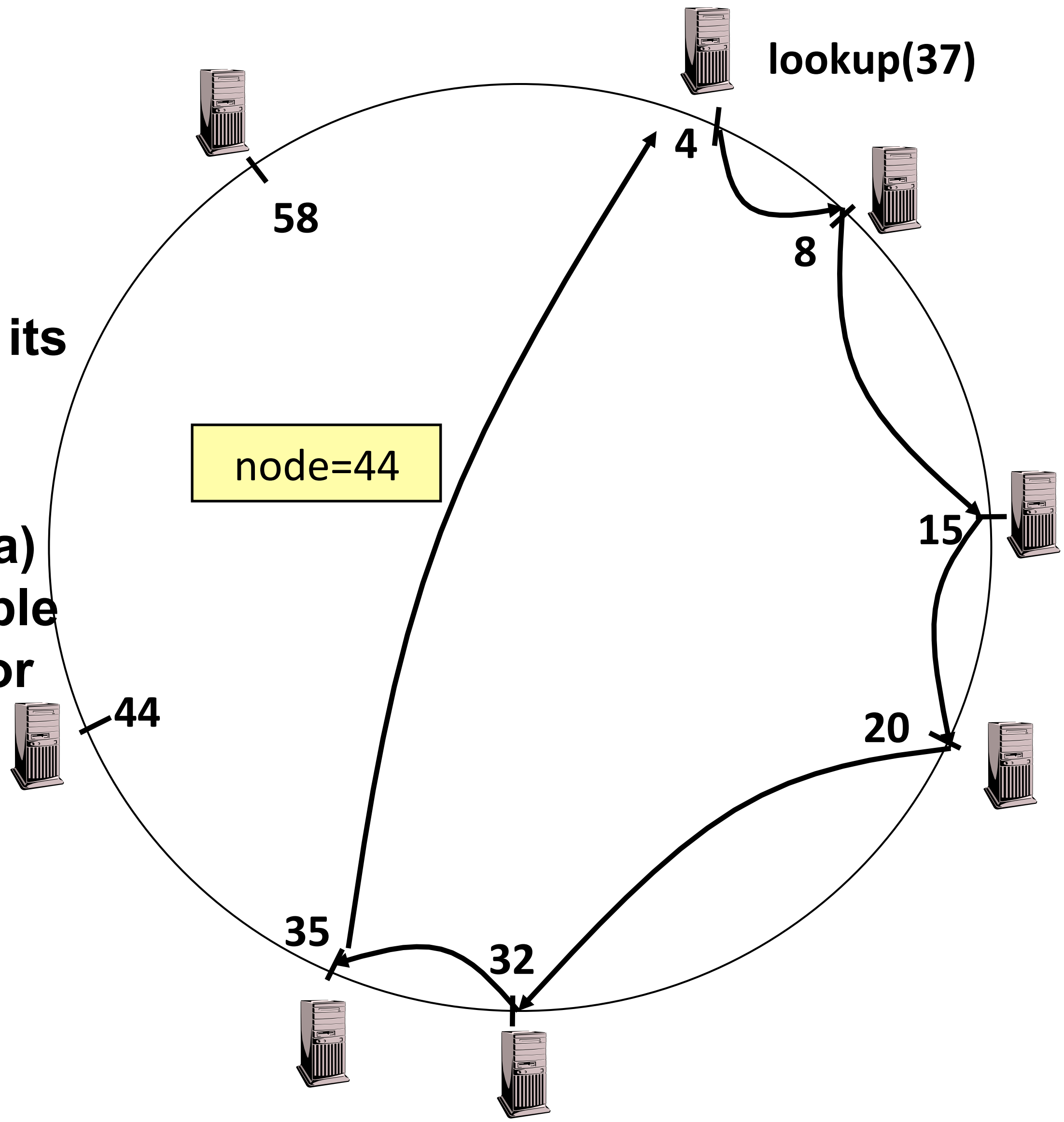
Remember Consistent Hashing?

- But each node only knows about a small number of other nodes (so far only their successors)



Lookup

- **Each node maintains its successor**
- **Route packet (ID, data) to the node responsible for ID using successor pointers**



Optional: DHT Maintenance

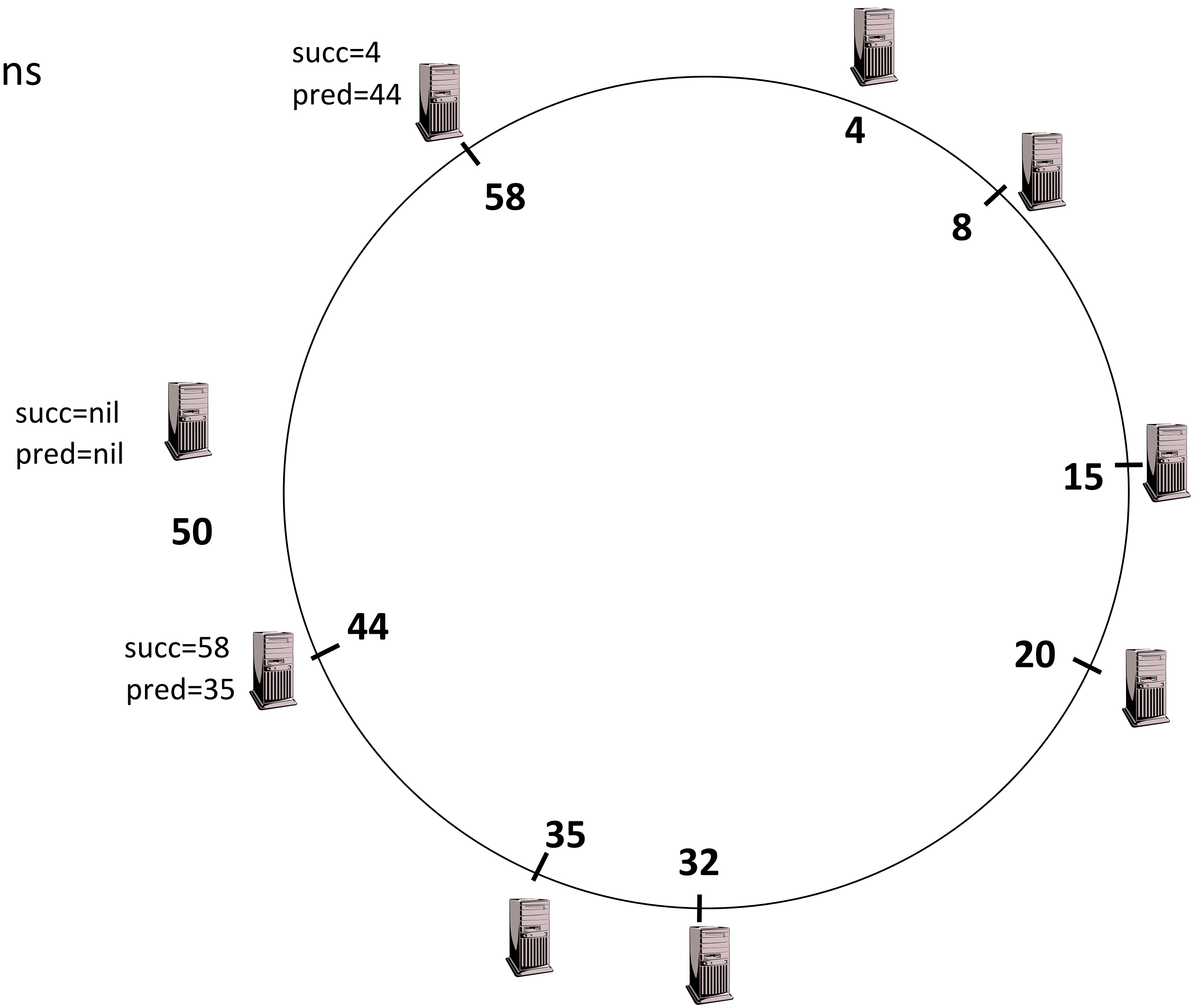
Stabilization Procedure

- **Periodic operations performed by each node N to maintain the ring:**

```
STABILIZE() [N.successor = M]
   N->M: "What is your predecessor?"
   M->N: "x is my predecessor"
   if x between (N,M), N.successor = x
   N->N.successor: NOTIFY()
NOTIFY()
    N->N.successor: "I think you are my successor"
M: upon receiving NOTIFY from N:
    If (N between (M.predecessor, M))
         M.predecessor = N
```

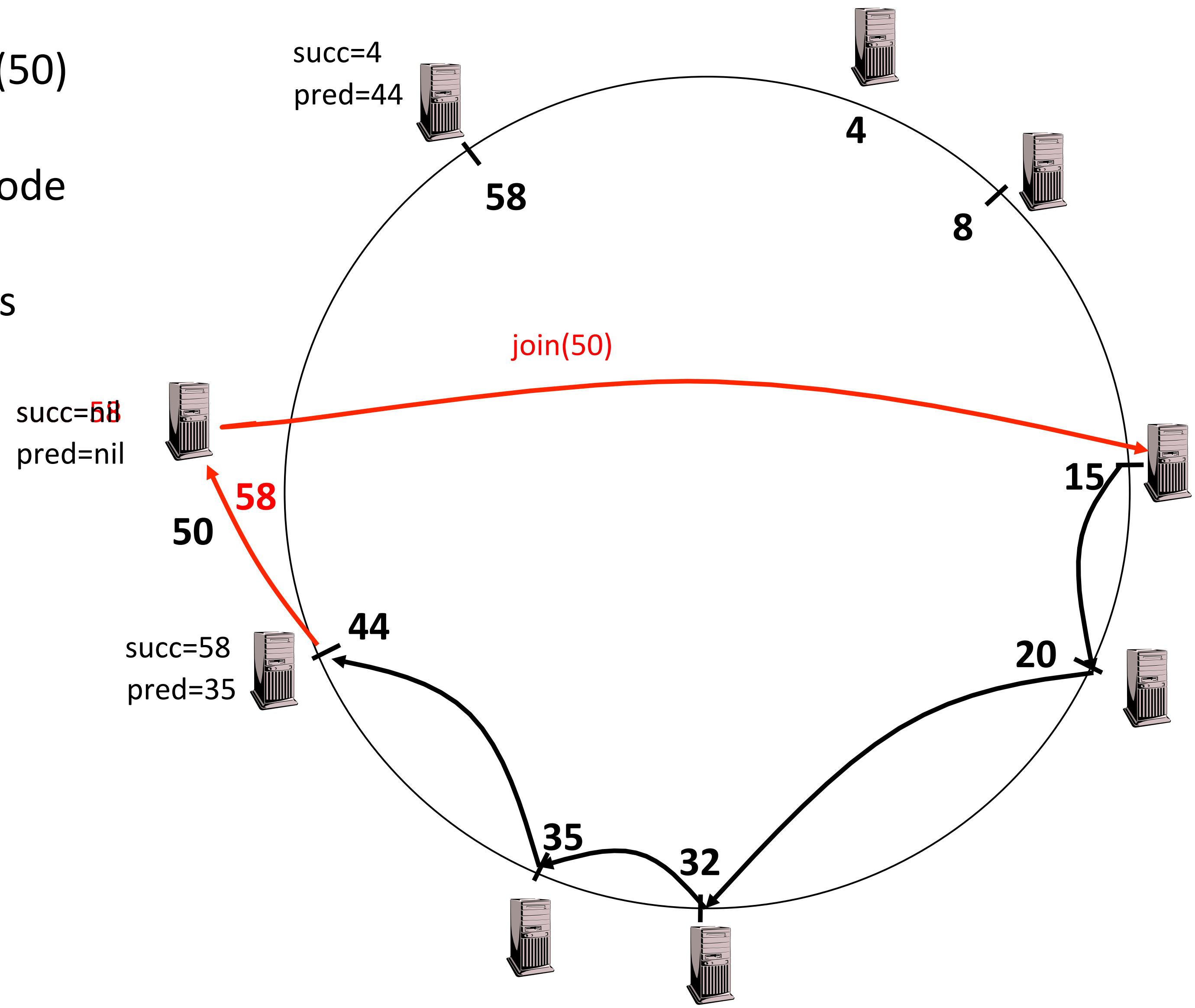
Joining Operation

- 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



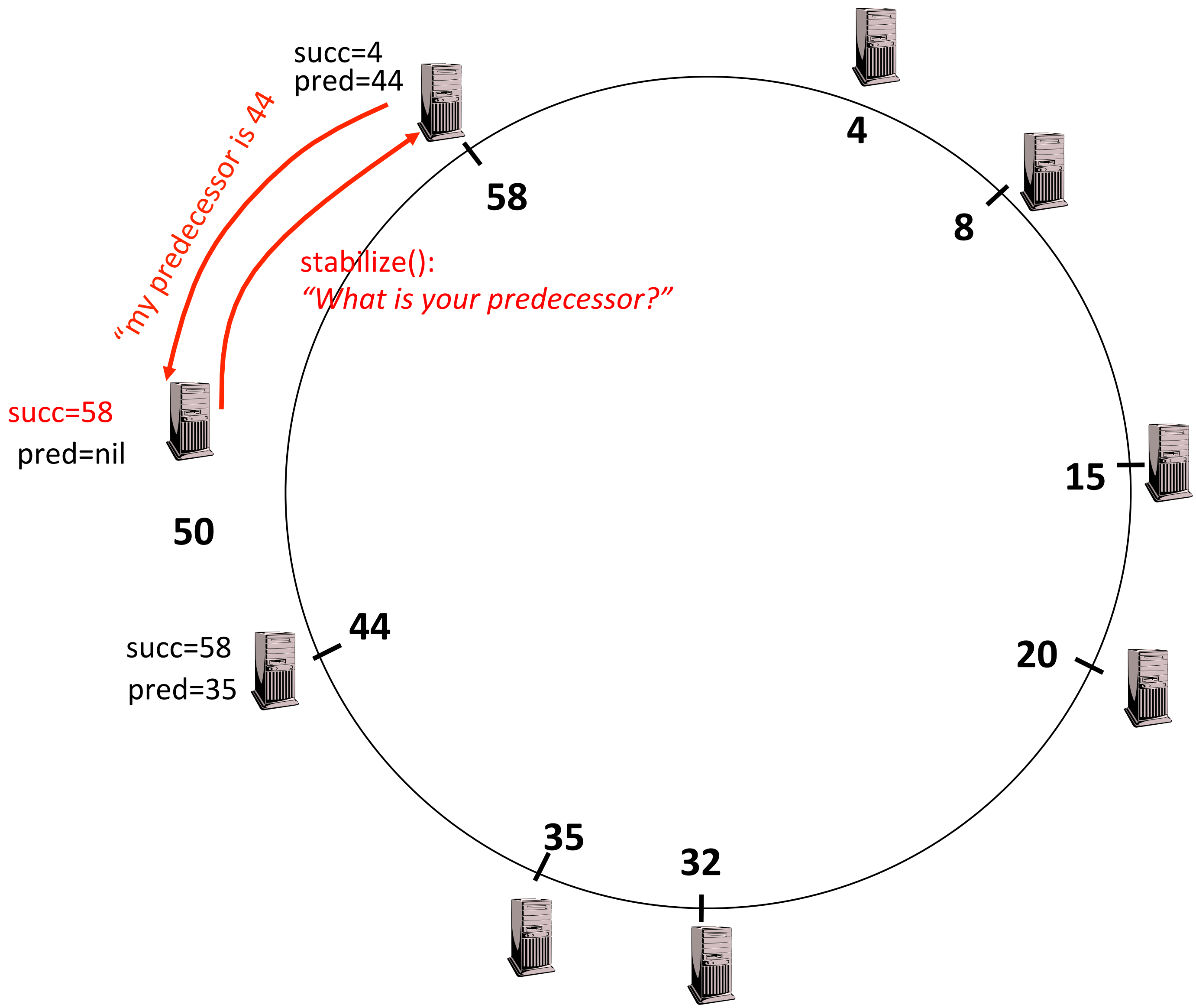
Joining Operation

- Node 50: send join(50) to node 15
- Node 44: returns node 58
- Node 50 updates its successor to 58



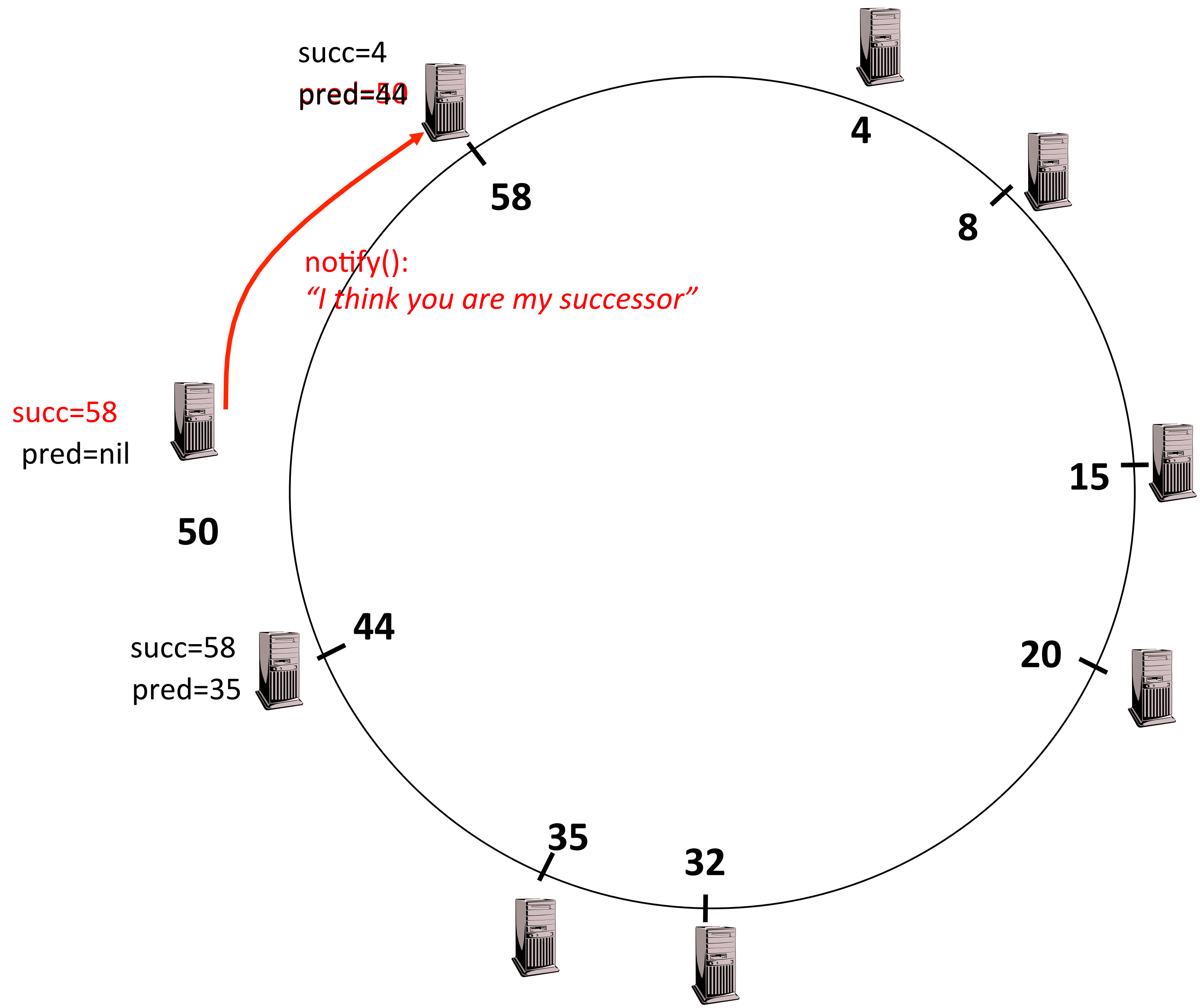
Joining Operation

- Node 50: send stabilize() to node 58
- Node 58:
  - Replies with 44
  - 50 determines it is the right predecessor



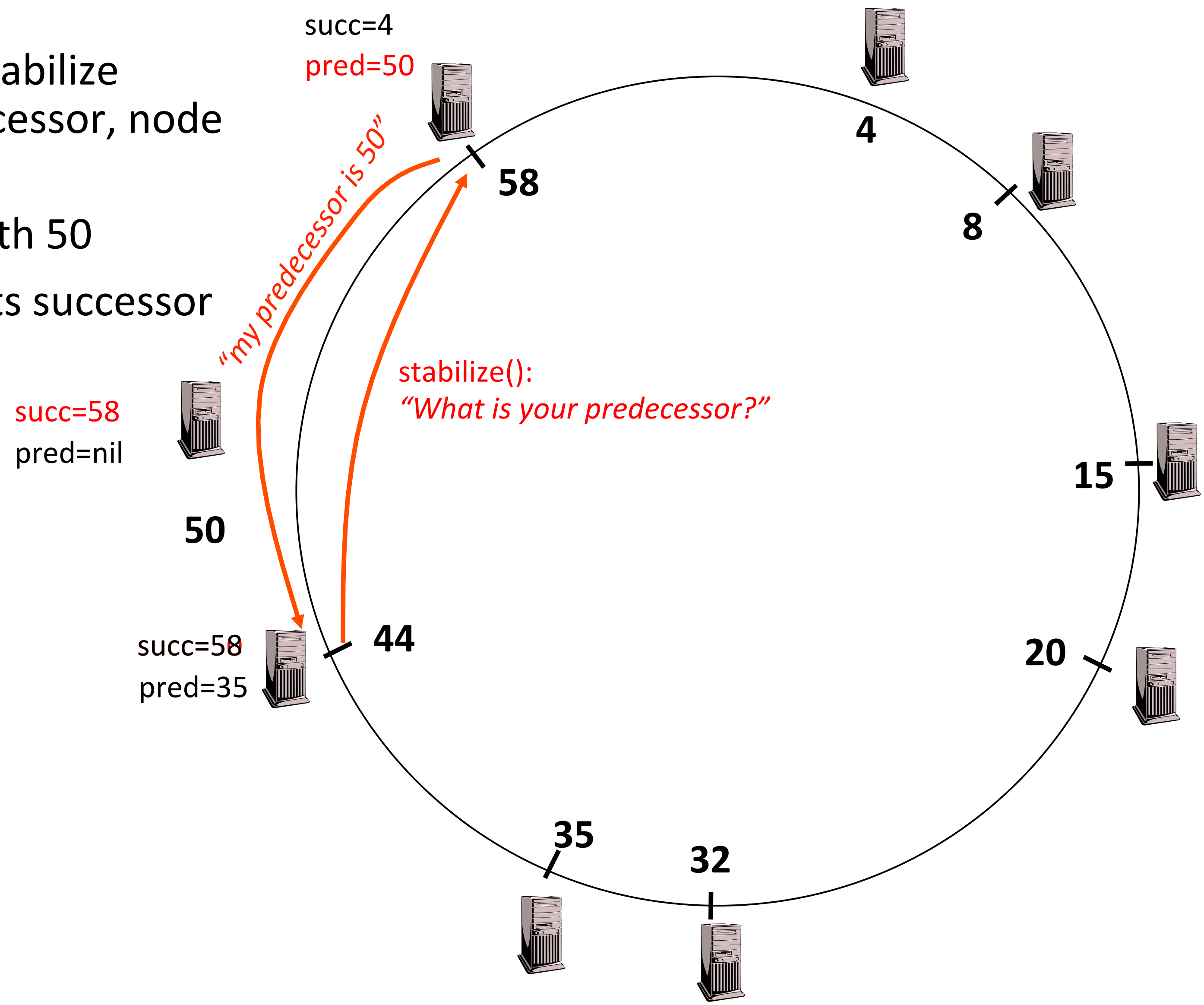
Joining Operation

- Node 50: send notify() to node 58
- Node 58:
  - update predecessor to 50



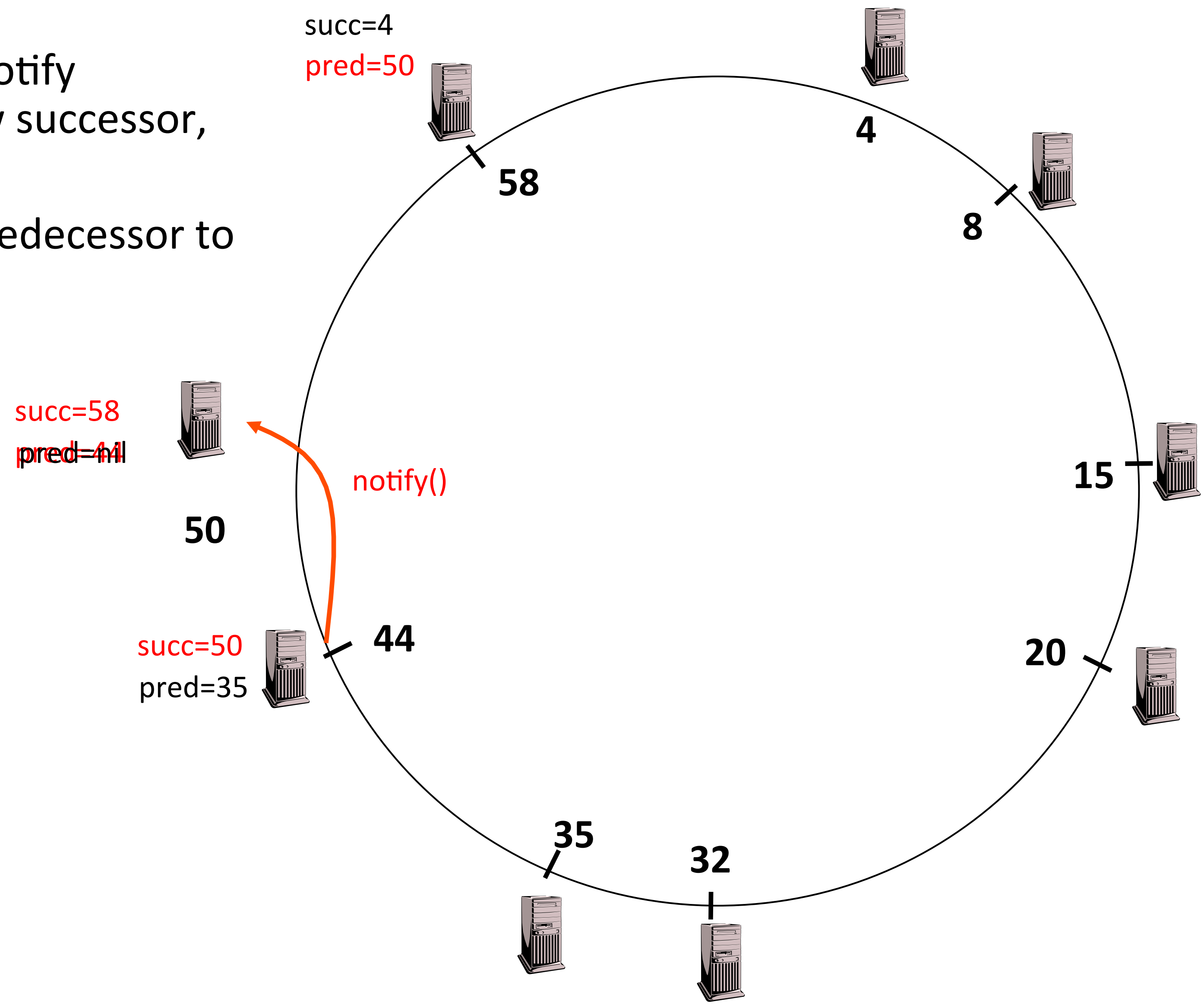
Joining Operation

- Node 44 sends a stabilize message to its successor, node 58
- Node 58 replies with 50
- Node 44 updates its successor to 50



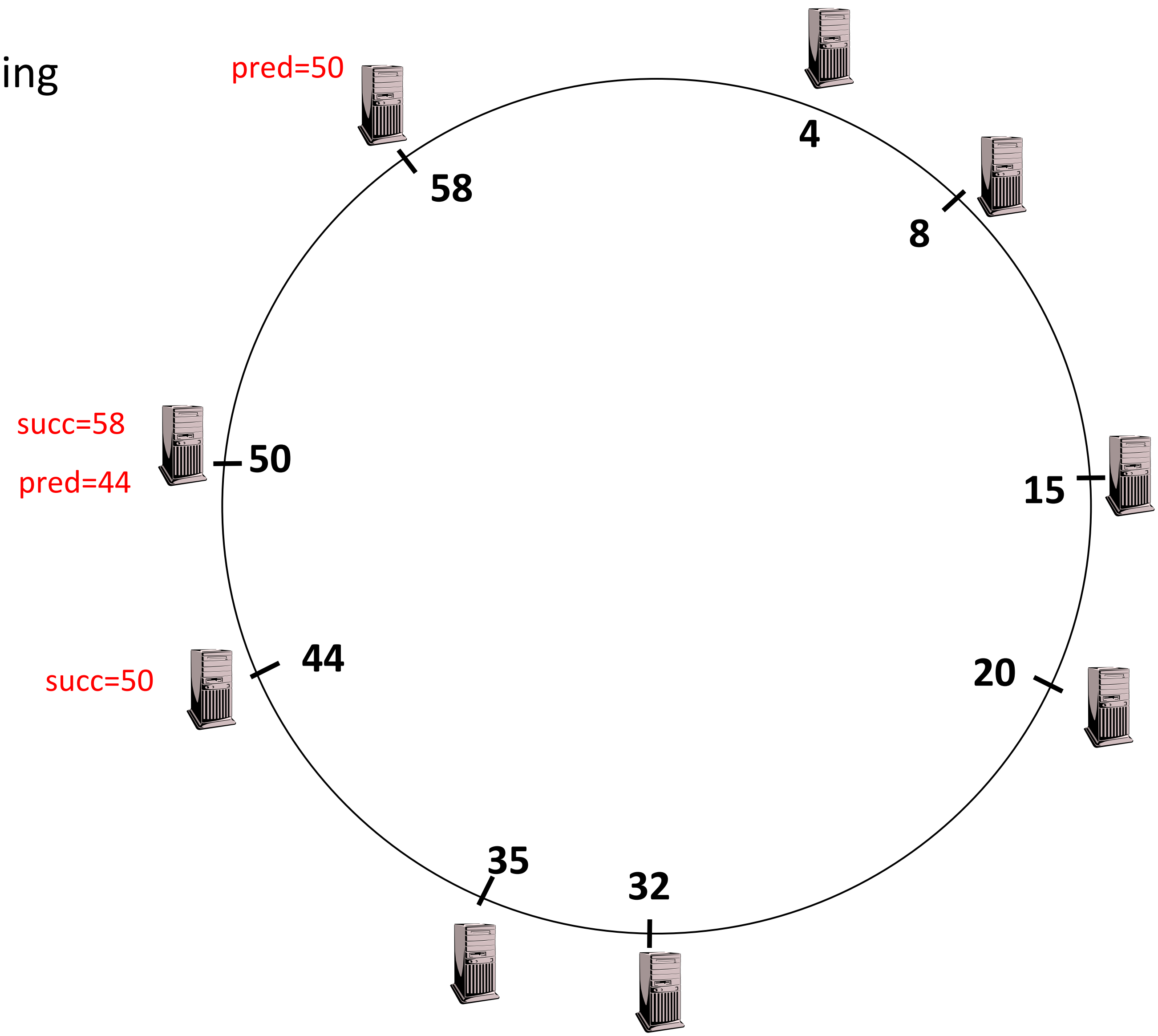
Joining Operation

- Node 44 sends a notify message to its new successor, node 50
- Node 50 sets its predecessor to node 44



Joining Operation (cont'd)

- This completes the joining operation!

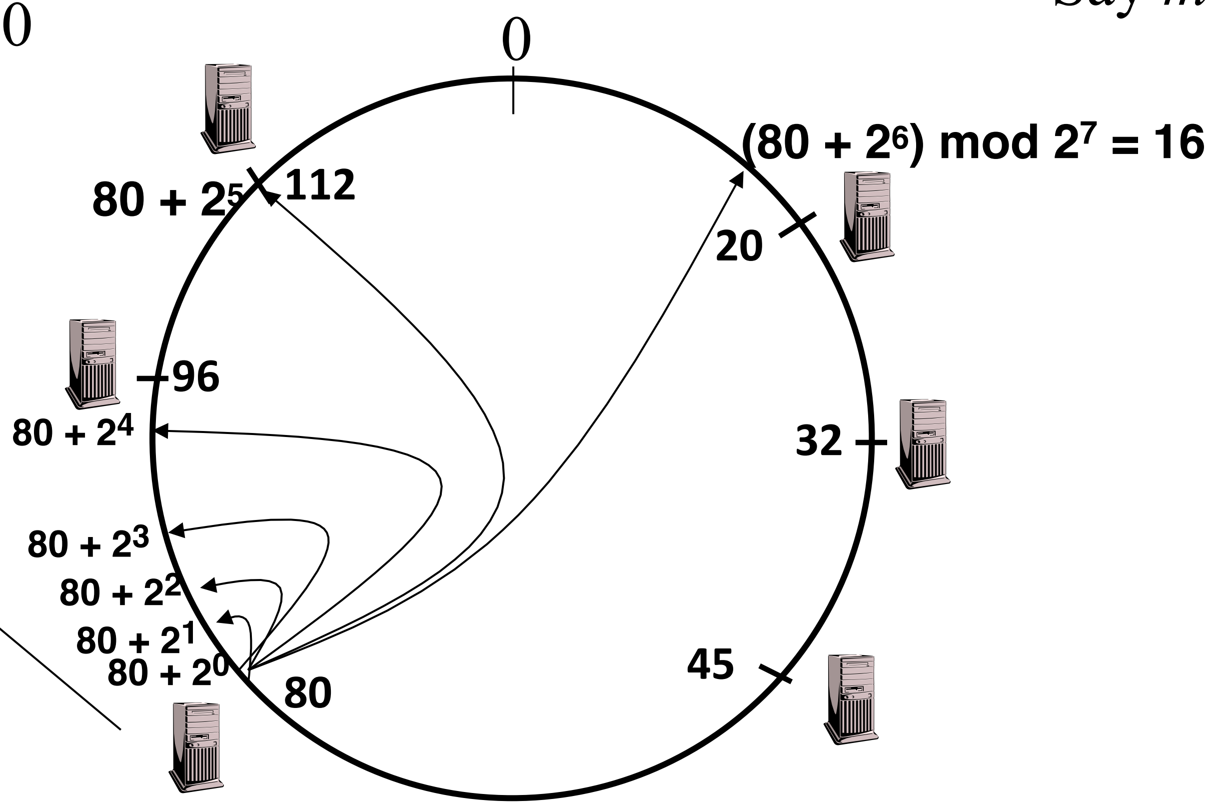


Achieving Efficiency: *finger tables*

Say m=7

Finger Table at 80

i	$ft[i]$
0	96
1	96
2	96
3	96
4	96
5	112
6	20



ith entry at peer with id n is first peer with id >= $n + 2^i \pmod{2^m}$

Chord

- There is a tradeoff between routing table size and diameter (number of hops for lookup) 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