

# Internet Technology

## 04. Peer-to-Peer Applications

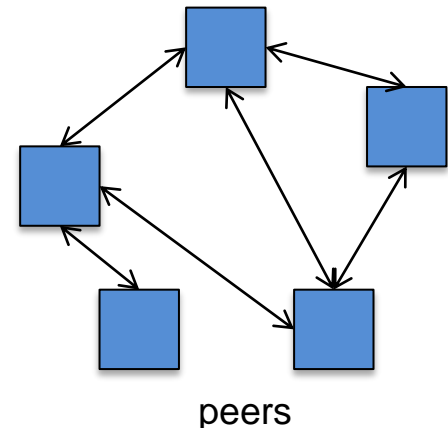
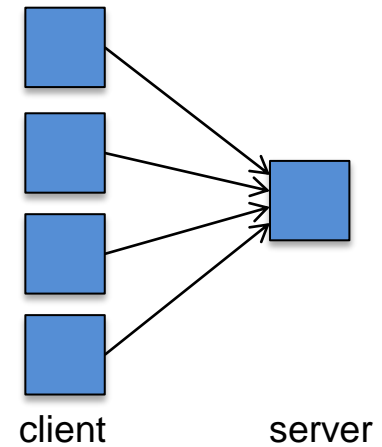
Paul Krzyzanowski

Rutgers University

Spring 2016

# Peer-to-Peer (P2P) Application Architectures

- No reliance on a central server
- Machines (peers) communicate with each other
- Pools of machines (peers) provide the service
- Goals
  - **Robustness**
    - Expect that some systems may be down
  - **Self-scalability**
    - The system can handle greater workloads as more peers are added



# Peer-to-Peer networking

“If a million people use a web site simultaneously, doesn't that mean that we must have a heavy-duty remote server to keep them all happy?

No; we could move the site onto a million desktops and use the Internet for coordination.

Could amazon.com be an itinerant horde instead of a fixed central command post? Yes.”

– David Gelernter  
*The Second Coming – A Manifesto*

See <http://edge.org/conversation/the-second-coming-a-manifesto>

# Peer to Peer applications

- P2P targets diverse solutions
  - Cooperative computation
  - Communications (e.g., Skype)
  - Exchanges, digital currency (bitcoin)
  - DNS (including multicast DNS)
  - Content distribution (e.g., BitTorrent)
  - Storage distribution
- P2P can be a distributed server
  - Lots of machines spread across multiple datacenters

Today, we'll focus on file distribution

# Four key primitives

- **Join/Leave**
  - How do you join a P2P system?
  - How do you leave it?
  - Who can join?
- **Publish**
  - How do you advertise content?
- **Search**
  - How do you find a file?
- **Fetch**
  - How do you download the file?



Strategies:

- Central server
- Flood the query
- Route the query

# Example: Napster

- **Background**
  - Started in 1999 by 19-year-old college dropout Shawn Fanning
  - Built only for sharing MP3 files
  - Stirred up legal battles with \$15B recording industry
  - Before it was shut down in 2001:
    - 2.2M users/day, 28 TB data, 122 servers
    - Access to contents could be slow or unreliable
- **Big idea:** Central directory, distributed contents
  - Users register files in a directory for sharing
  - Search in the directory to find files to copy

# Napster: Overview

---

Napster is based on a **central directory**

- **Join**
  - On startup, a client contacts the central server
- **Publish**
  - Upload a list of files to the central server
  - These are the files you are sharing and are on your system
- **Search**
  - Query the sever
  - Get back one or more peers that have the file
- **Fetch**
  - Connect to the peer and download the file

# Napster: Discussion

---

- **Pros**

- Super simple
- Search is handled by a single server
- The directory server is a single point of control
  - Provides definitive answers to a query

- **Cons**

- Server has to maintain state of all peers
- Server gets all the queries
- The directory server is a single point of control
  - No directory server, no Napster!



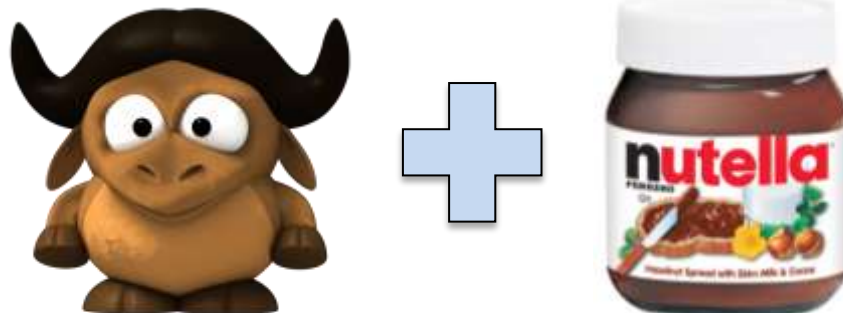
# Example: Gnutella

- **Background**

- Created by Justin Frankel and Tom Pepper (authors of Winamp)
- AOL acquired their company, Nullsoft in 1999
- In 2000, accidentally released gnutella
- AOL shut down the project but the code was released

- **Big idea:** create fully distributed file sharing

- Unlike Napster, you cannot shut down gnutella



# Gnutella: Overview

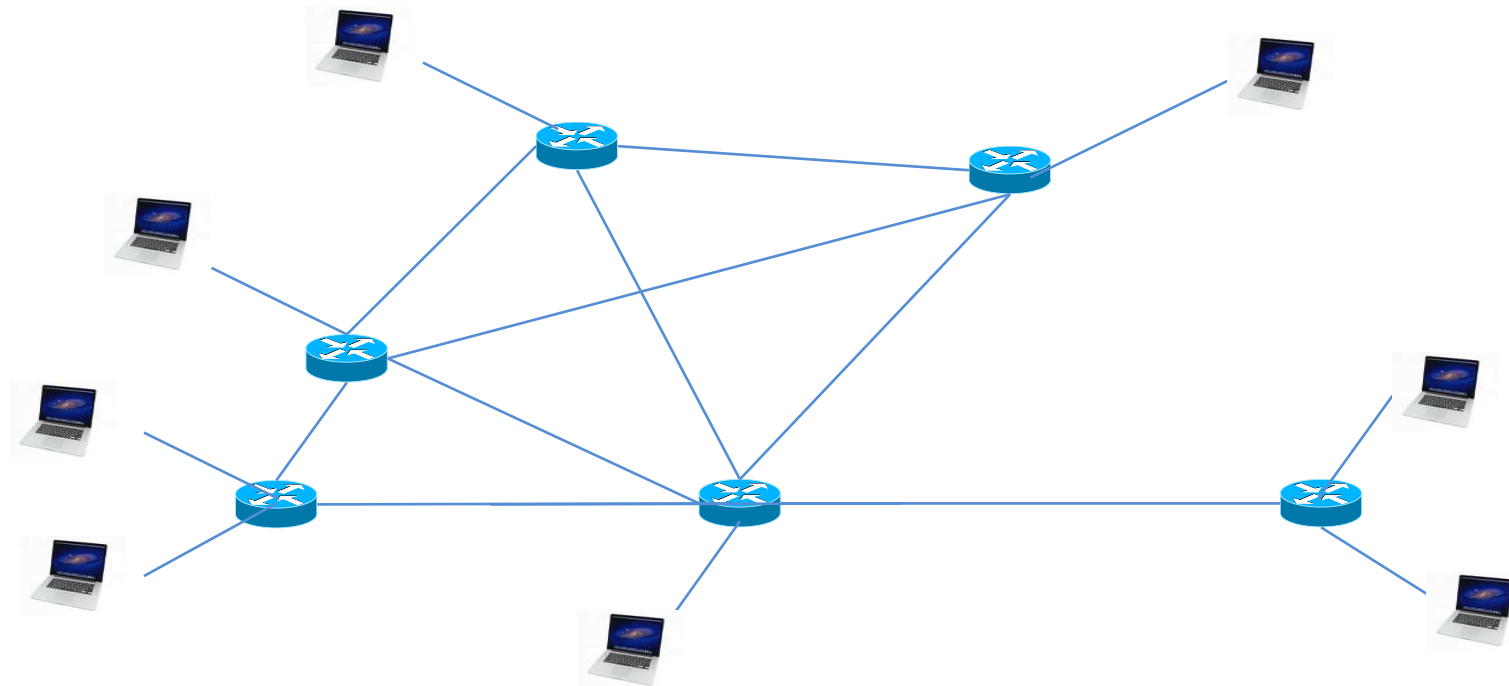
## Gnutella is based on **query flooding**

- **Join**
  - On startup, a node (peer) contacts at least one node
    - Asks who its friends are
  - These become its “connected nodes”
- **Publish**
  - No need to publish
- **Search**
  - Ask connected nodes. If they don't know, they will ask their connected nodes, and so on...
  - Once/if the reply is found, it is returned to the sender
- **Fetch**
  - The reply identifies the peer; connect to the peer via HTTP & download

# Overlay network

An **overlay network** is a virtual network formed by **peer connections**

- Any node might know about a small set of machines
- “Neighbors” might not be physically close to you – they’re just who you know

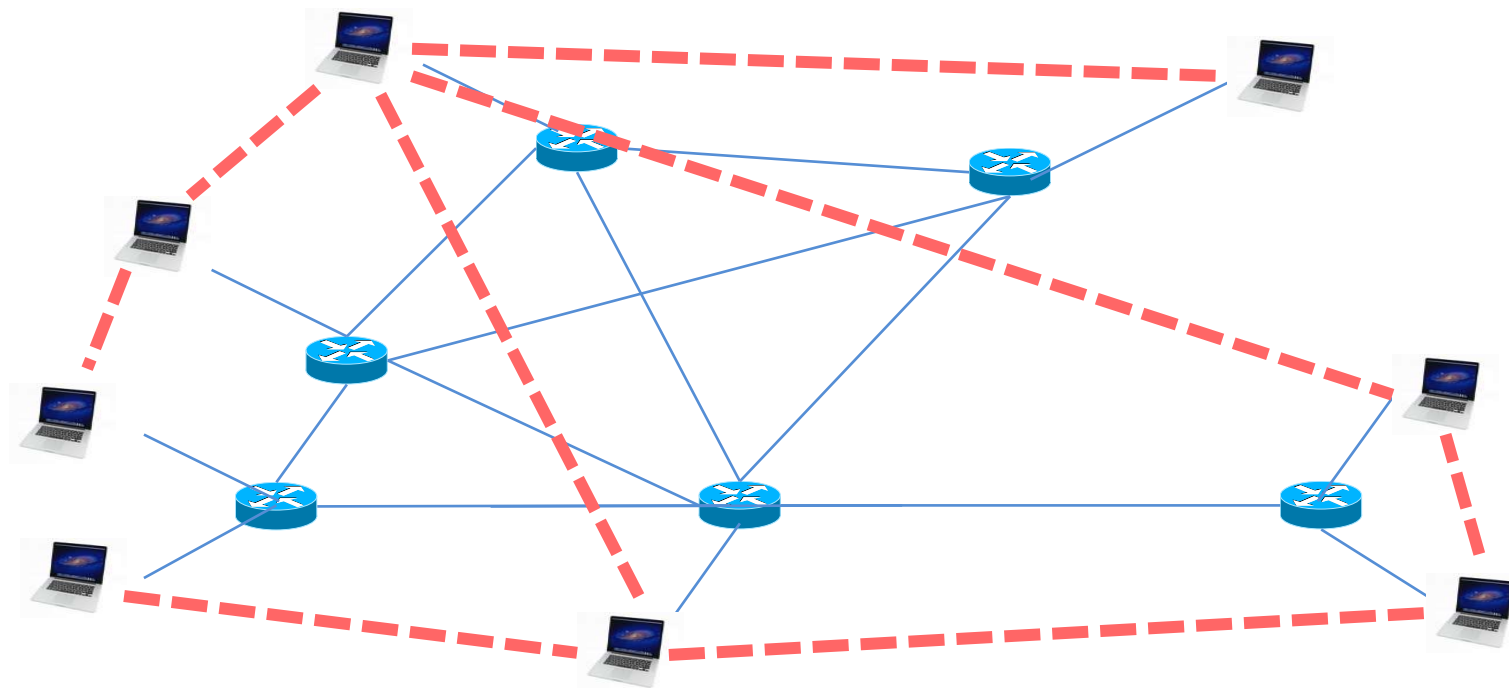


Underlying IP Network

# Overlay network

An **overlay network** is a virtual network formed by **peer connections**

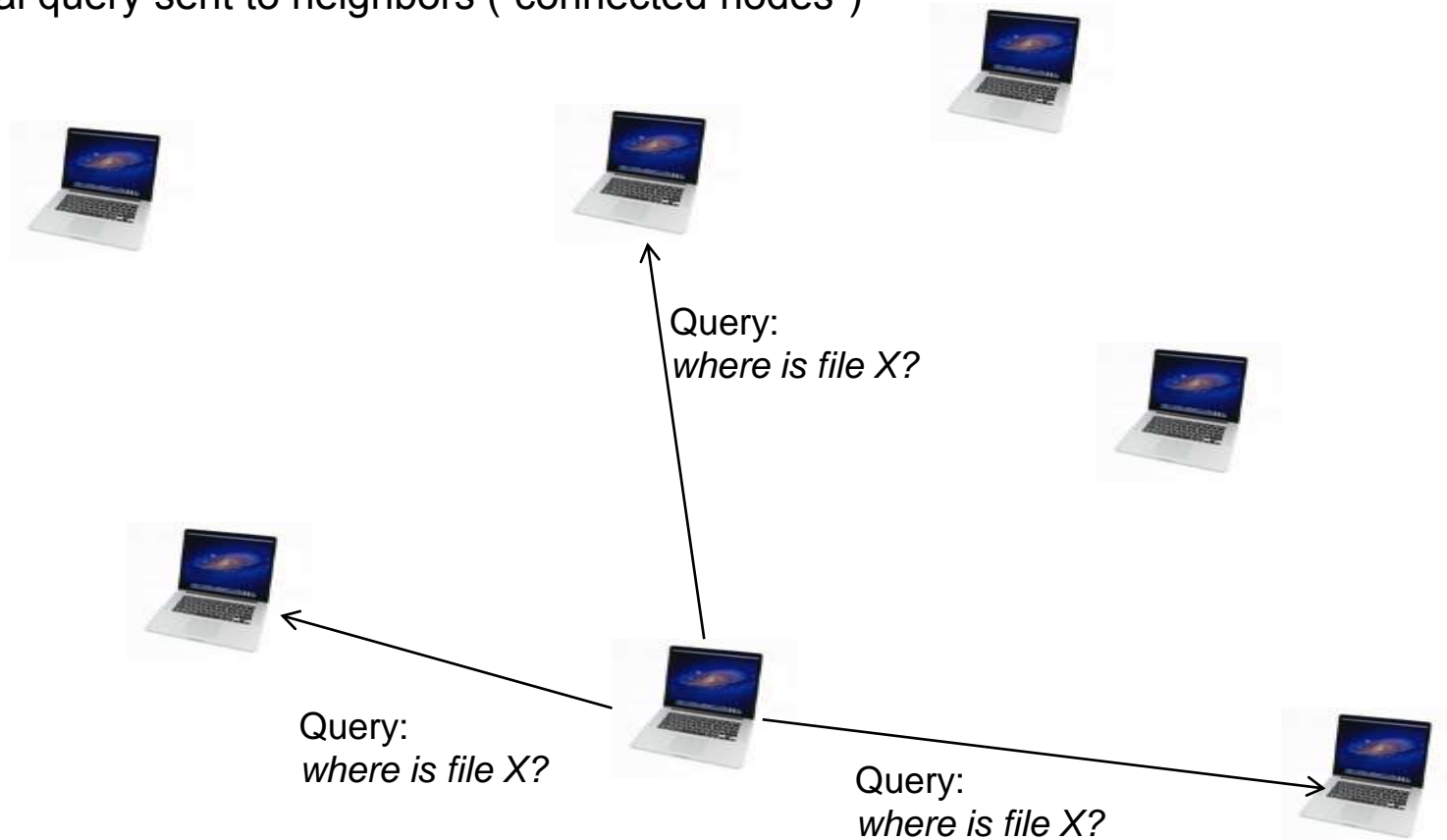
- Any node might know about a small set of machines
- “Neighbors” might not be physically close to you – they’re just who you know



Overlay Network

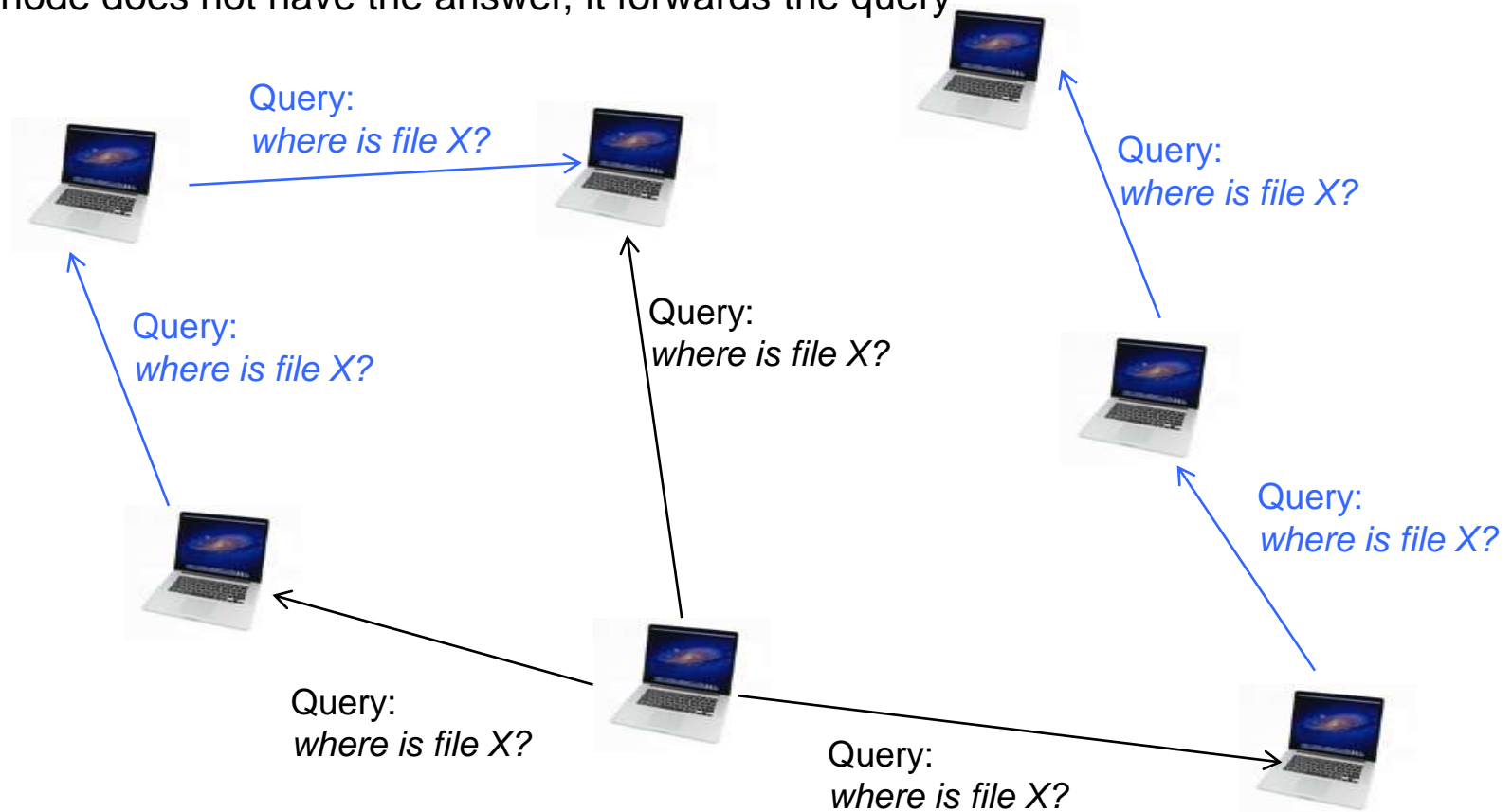
# Gnutella: Search

Initial query sent to neighbors (“connected nodes”)



# Gnutella: Search

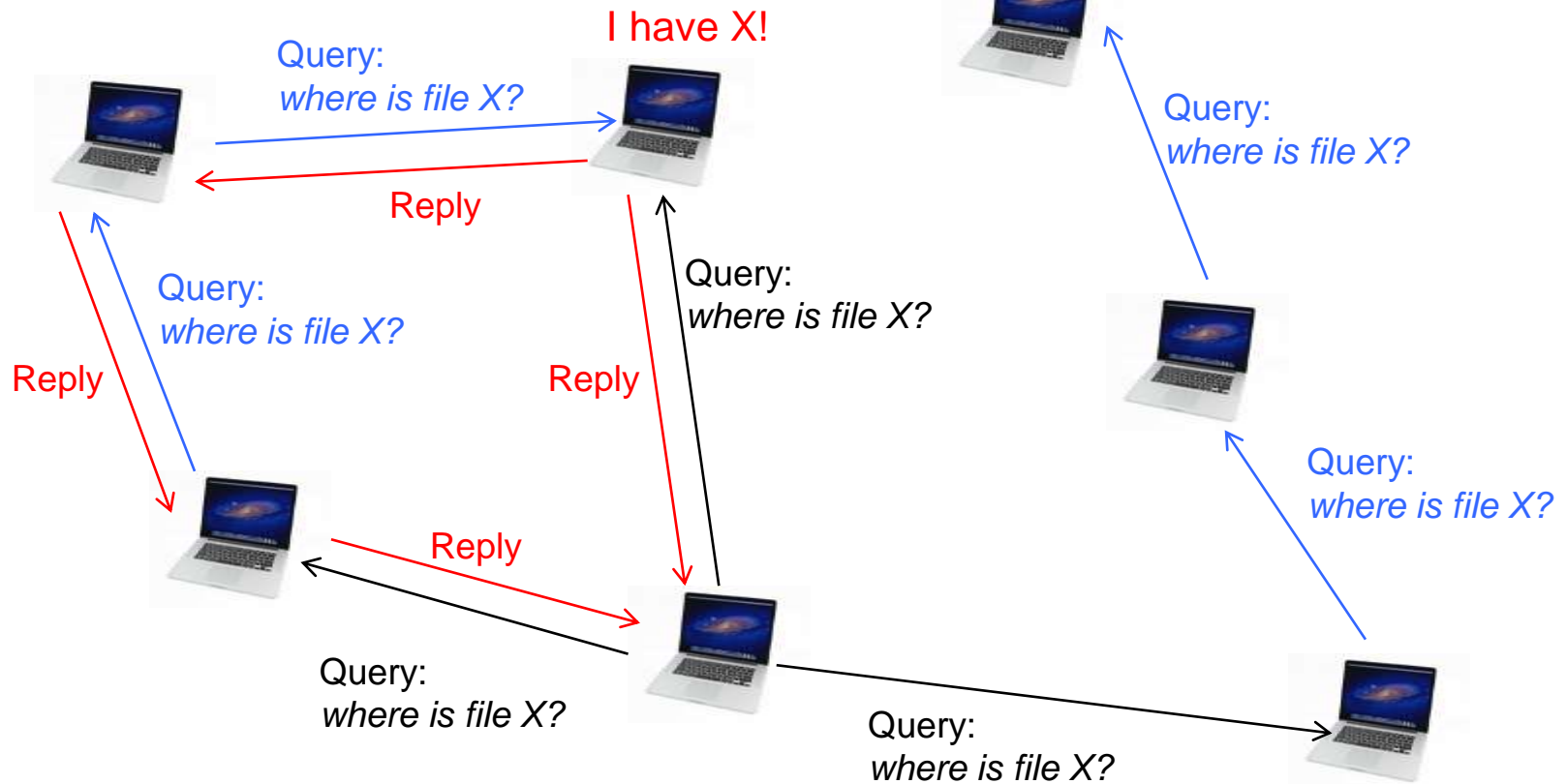
If a node does not have the answer, it forwards the query



Queries have a hop count (time to live) – so we avoid **forwarding loops**

# Gnutella: Search

If a node has the answer, it replies – replies get forwarded



# Gnutella: Search

- Original protocol
  - **Anonymous**: you didn't know if the request you're getting is from the originator or the forwarder
  - Replies went through the same query path
- Downloads
  - Node connects to the server identified in the reply
  - If a connection is not possible due to firewalls, the requesting node can send a **push request** for the remote client to send it the file



# Peers do not have equal capabilities

---

- Network upstream and downstream bandwidth
- Connectivity costs (willingness to participate)
- Availability
- Compute capabilities

# Gnutella: Enhancements

- **Optimizations**

- Requester's IP address sent in query to optimize reply
- Every node is no longer equal
  - Leaf nodes & **Ultrapeers**
  - Leaf nodes connect to a small number of ultrapeers
  - Ultrapeers are connected to  $\geq 32$  other ultrapeers
  - Route search requests through ultrapeers

- **Downloads**

- Node connects to the server identified in the reply
- If a connection is not possible due to firewalls, the requesting node can send a *push request* for the remote client to send it the file

# Gnutella: Summary

- **Pros**

- Fully decentralized design
- Searching is distributed
- No control node – cannot be shut down
- Open protocol

- **Cons**

- Flooding is inefficient:
  - Searching may require contacting a lot of systems; limit hop count
- Well-known nodes can become highly congested
- In the classic design, if nodes leave the service, the system is crippled

# Example: FastTrack/Kazaa

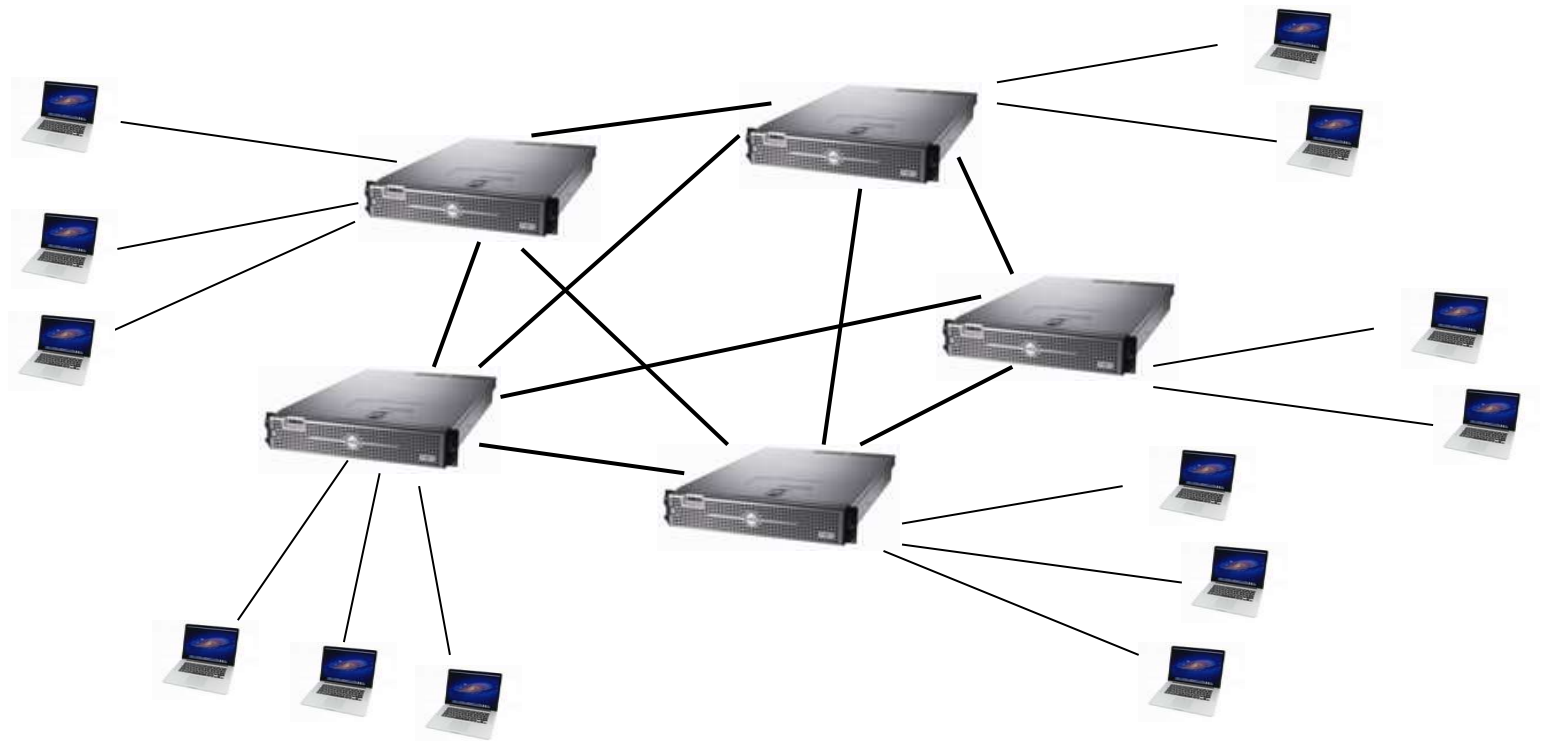
- **Background**

- Kazaa & FastTrack protocol created in 2001
- Team of Estonian programmers – same team that will later create Skype
- Post-Napster and a year after Gnutella was released
- **FastTrack**: used by others (Grokster, iMesh, Morpheus)
  - Proprietary protocol; Several incompatible versions

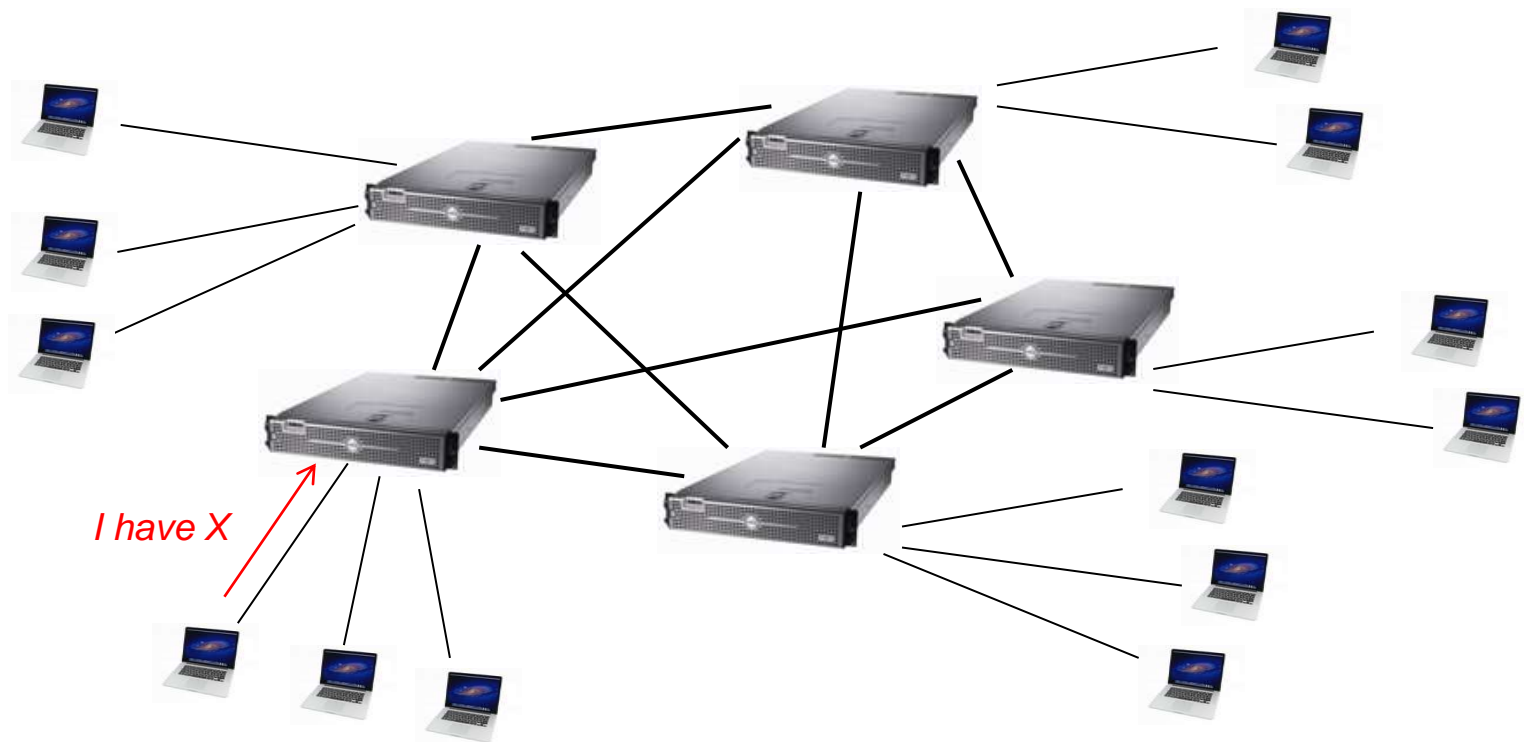
- **Big idea**: Some nodes are better than others

- A subset of client nodes have fast connectivity, lots of storage, and fast processors
- These will be used as **supernodes** (similar to gnutella's ultrapeers)
- Supernodes:
  - Serve as indexing servers for slower clients
  - Know other supernodes

# Kazaa: Supernodes

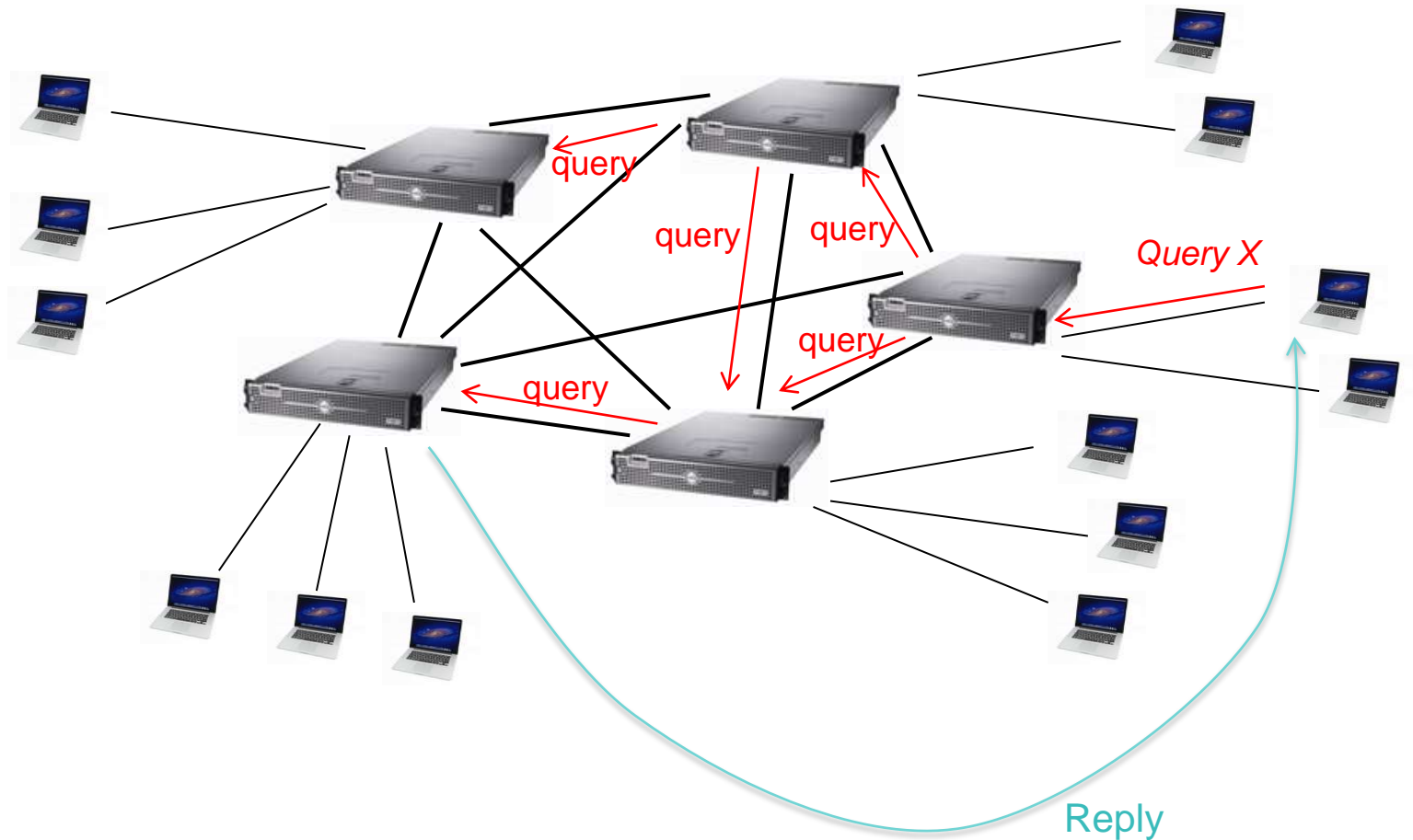


# Kazaa: publish a file



# Kazaa: search

Supernodes answer for all their peers (ordinary nodes)



# Kazaa: Discussion

---

## Selective flooding of queries

- **Join**
  - A peer contacts a supernode
- **Publish**
  - Peer sends a list of files to a supernode
- **Search**
  - Send a query to the supernode
  - Supernodes flood the query to other supernodes
- **Fetch**
  - Download the file from the peer with the content



# Kazaa: Summary

---

- **Pros**

- Similar to improved Gnutella
- Efficient searching via supernodes
- Flooding restricted to supernodes

- **Cons**

- Can still miss files
- Well-known supernodes provide opportunity to stop service

# BitTorrent

- **Background**
  - Introduced in 2002 by Bram Cohen
  - Motivation
    - Popular content exhibits temporal locality: *flash crowds*
      - E.g., slashdot effect, CNN on 9/11, new movies, new OS releases
- **Big idea:** allow others to download from you while you are downloading
  - Efficient fetching, not searching
  - Single publisher, many downloaders

# BitTorrent: Overview

## Enable downloads from peers

- **Join**
  - No need to join  
(seed registers with tracker server; peers register when they download)
- **Publish**
  - Create a torrent file; give it to a *tracker server*
- **Search**
  - Outside the BitTorrent protocol
  - Find the tracker for the file you want, contact it to get a list of peers with files
- **Fetch**
  - Download chunks of the file from our peers
  - At the same time, other peers may request chunks from you

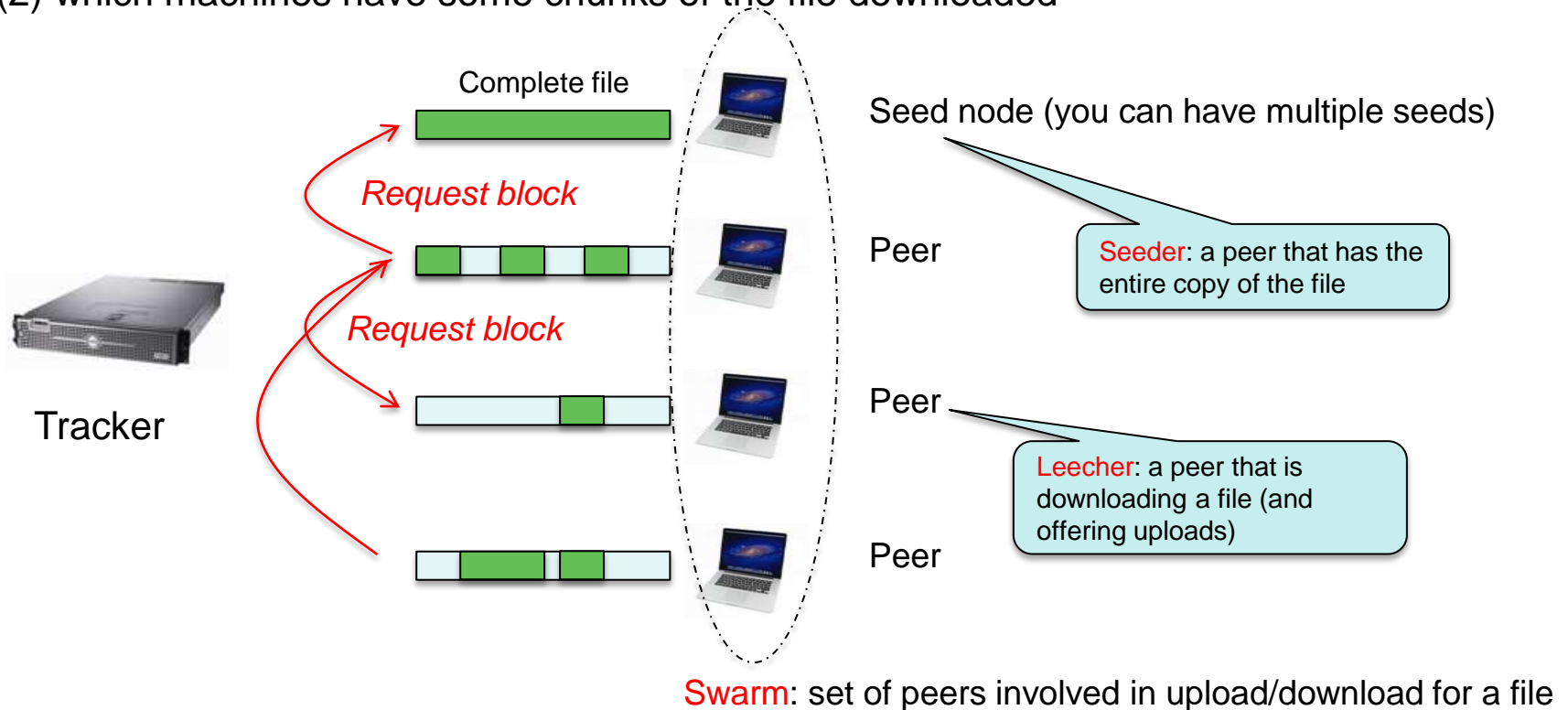
# BitTorrent: Publishing & Fetching

- To distribute a file
  - Create a **.torrent** file
  - Contains
    - name, size, hash of each chunk, address of a tracker server.
  - Start a **seed node**: initial copy of the full file
  - Start the **tracker** for the file
    - Tracker manages uploading & downloading of the content
- **To get a file**
  - Get a .torrent file
  - Contact **tracker** named in the file
    - Get the list of seeders and other nodes with portions of the file
    - Tracker will also announce you to others
  - Contact a random node for a list of file chunk numbers
  - Request a random block of the file

# BitTorrent: Downloading a file in chunks

Tracker identifies:

- (1) initial system(s) that has 100% of the file (the seed)
- (2) which machines have some chunks of the file downloaded



When a peer finished downloading a file, it may become a seed and remain online without downloading any content.

# BitTorrent Summary

---

- **Pros**

- Scales well; performs well when many participants
- Gives peers an incentive to share
  - It is sometimes not possible to download without offering to upload

- **Cons**

- Search is not a part of the protocol; relies on torrent index servers
- Files need to be large for this to work well
- Rare files do not offer distribution
- A tracker needs to be running to bootstrap the downloads

# Distributed Hash Tables

# Locating content

- Our discussion on peer-to-peer applications focused on content distribution
  - Content was fully distributed
- How do we find the content?

Napster	Central server (hybrid architecture)
Gnutella & Kazaa	Network flooding Optimized to flood supernodes ... but it's still flooding
BitTorrent	Nothing! It's somebody else's problem

- Can we do better?



# What's wrong with flooding?

- Some nodes are not always up and some are slower than others
  - Gnutella & Kazaa dealt with this by classifying some nodes as “supernodes” (called “ultrapeers” in Gnutella)
- Poor use of network (and system) resources
- Potentially high latency
  - Requests get forwarded from one machine to another
  - Back propagation (e.g., Gnutella design), where the replies go through the same chain of machines used in the query, increases latency even more

# Hash tables

- Remember hash functions & hash tables?
  - Linear search:  $O(N)$
  - Tree:  $O(\log N)$
  - Hash table:  $O(1)$

# What's a hash function? (refresher)

- Hash function
  - A function that takes a variable length input (e.g., a string) and generates a (usually smaller) fixed length result (e.g., an integer)
  - Example: hash strings to a range 0-6:
    - $\text{hash}(\text{"Newark"}) \rightarrow 1$
    - $\text{hash}(\text{"Jersey City"}) \rightarrow 6$
    - $\text{hash}(\text{"Paterson"}) \rightarrow 2$
- Hash table
  - Table of *(key, value)* tuples
  - Look up a key:
    - Hash function maps *keys* to a range  $0 \dots N-1$   
table of  $N$  elements  
`i = hash(key)`  
`table[i]` contains the item
  - No need to search through the table!

# Considerations with hash tables (refresher)

- Picking a good hash function
  - We want uniform distribution of all values of *key* over the space  $0 \dots N-1$
- Collisions
  - Multiple keys may hash to the same value
    - `hash("Paterson")` → 2
    - `hash("Edison")` → 2
  - `table[i]` is a **bucket (slot)** for all such (*key, value*) sets
  - Within `table[i]`, use a linked list or another layer of hashing
- Think about a hash table that grows or shrinks
  - If we add or remove buckets → need to rehash keys and move items

# Distributed Hash Tables (DHT)

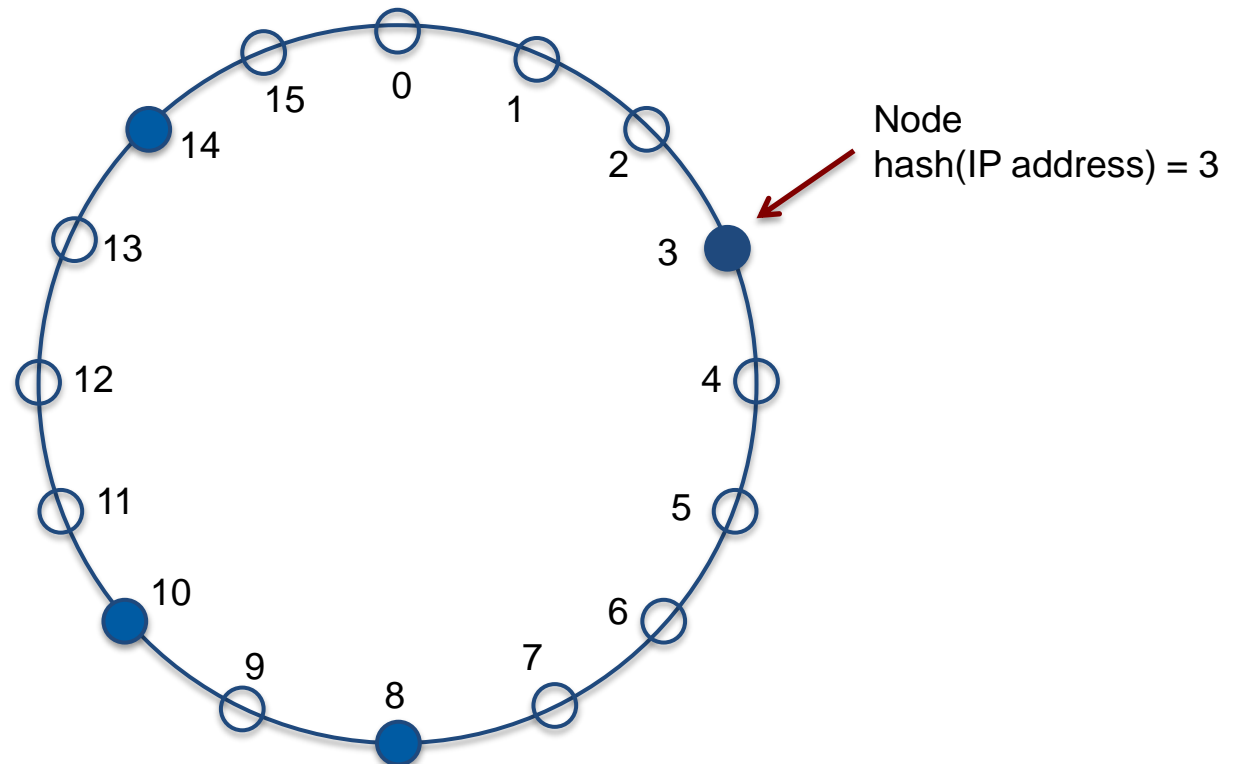
- Create a peer-to-peer version of a (*key, value*) database
- How we want it to work
  1. A peer queries the database with a key
  2. The database finds the peer that has the value
  3. That peer returns the (*key, value*) pair to the querying peer
- Make it efficient!
  - A query should not generate a flood!
- We'll look at one DHT implementation called **Chord**

# The basic idea

- Each node (peer) is identified by an integer in the range  $[0, 2^n-1]$ 
  - $n$  is a big number, like 160 bits
- Each key is hashed into the range  $[0, 2^n-1]$ 
  - E.g., SHA-1 hash
- Each peer will be responsible for a range of keys
  - A key is stored at the closest **successor** node
  - Successor node = first node whose ID  $\geq$  hash(key)
- If we arrange the peers in a **logical ring** (incrementing IDs) then a peer needs to know only of its successor and predecessor
  - This limited knowledge of peers makes it an **overlay network**

# Chord & consistent hashing

- A key is hashed to an  $m$ -bit value:  $0 \dots 2^m - 1$
- A logical ring is constructed for the values  $0 \dots 2^m - 1$
- Nodes are placed on the ring at  $hash(IP\ address)$



# Key assignment

- Example:  $n=16$ ; system with 4 nodes (so far)
- Key, value data is stored at a **successor**
  - a node whose value is  $\geq \text{hash}(\text{key})$

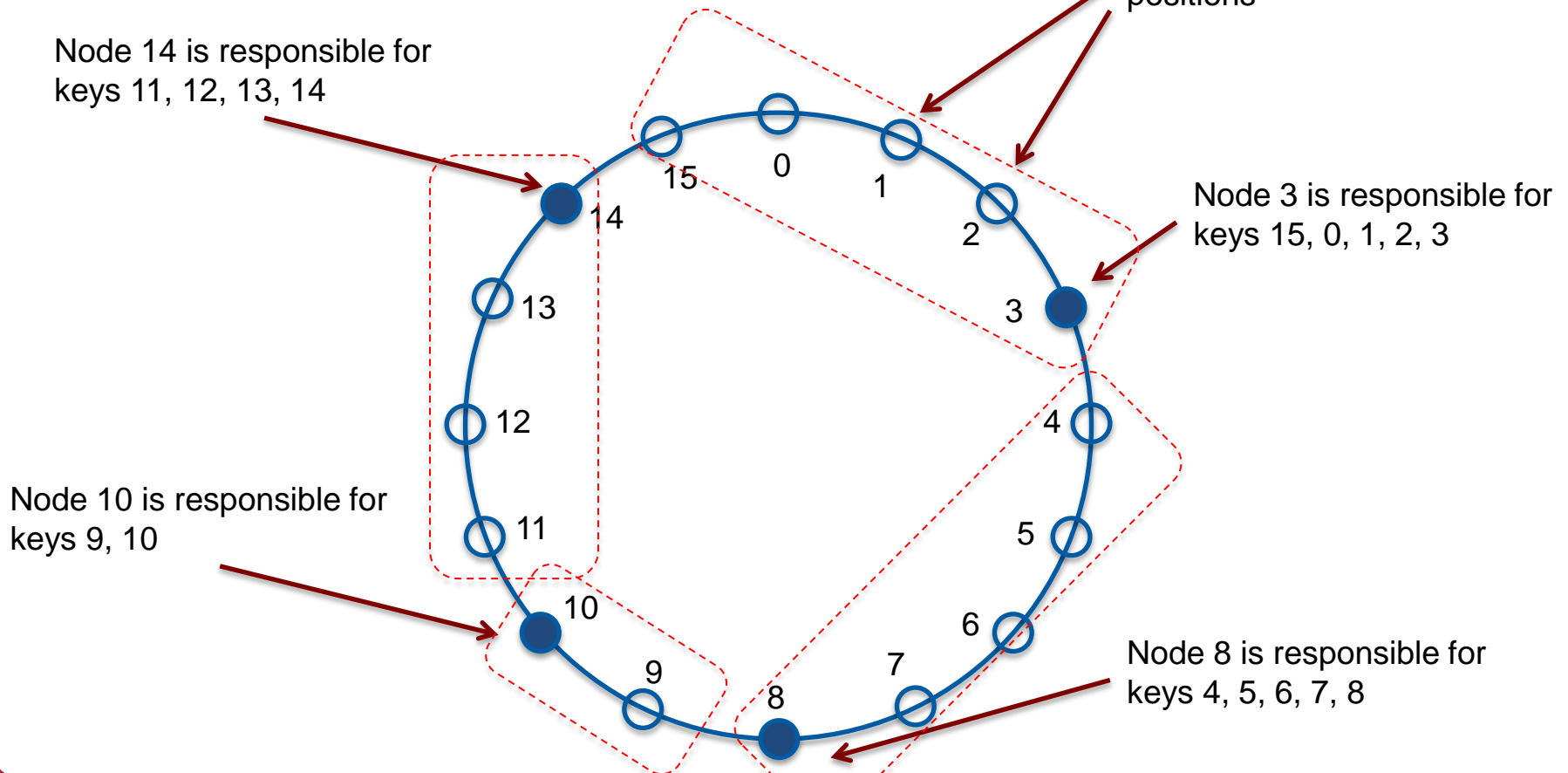
Node 14 is responsible for keys 11, 12, 13, 14

Node 10 is responsible for keys 9, 10

No nodes at these empty positions

Node 3 is responsible for keys 15, 0, 1, 2, 3

Node 8 is responsible for keys 4, 5, 6, 7, 8







# Let's figure out three more things

---

1. Adding/removing nodes
2. Improving lookup time
3. Fault tolerance

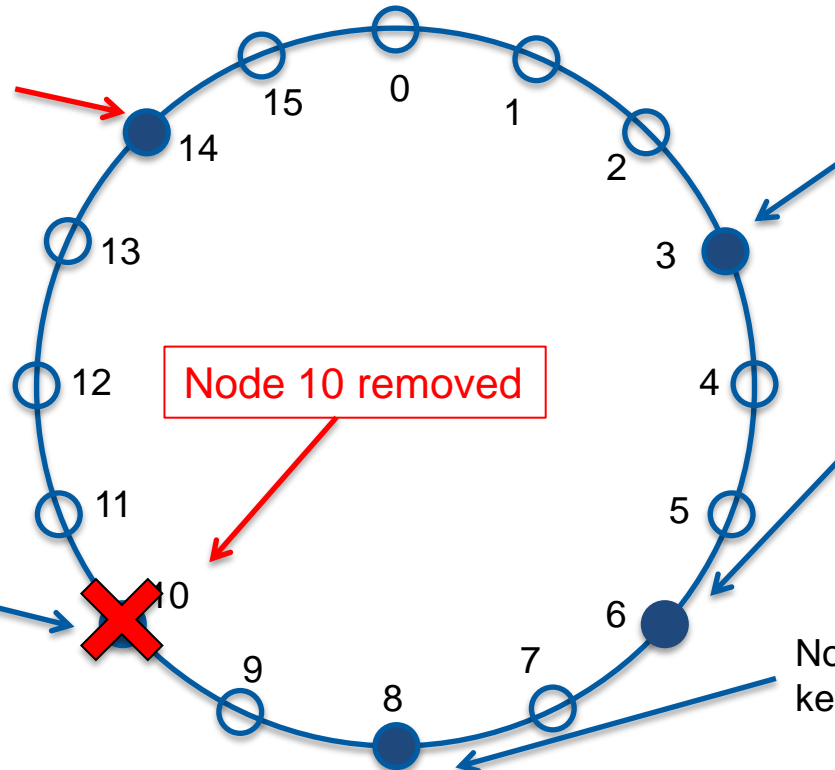


# Removing a node

- Keys are reassigned to the node's successor
- Data for those *(key, value)* pairs must be moved to the successor

Node 14 was responsible for keys 11, 12, 13, 14

Node 14 is now responsible for keys 9, 10, 11, 12, 13, 14



Node 3 is responsible for keys 15, 0, 1, 2, 3

Node 6 is responsible for keys 4, 5, 6

Node 10 was responsible for keys 9, 10

Node 8 is responsible for keys 7, 8

# Performance

- We're not thrilled about  $O(N)$  lookup
- Simple approach for great performance
  - Have all nodes know about each other
  - When a peer gets a node, it searches its table of nodes for the node that owns those values
  - Gives us  $O(1)$  performance
  - Add/remove node operations must inform everyone
  - Not a good solution if we have millions of peers (huge tables)

# Finger tables

- Compromise to avoid huge per-node tables
  - Use **finger tables** to place an upper bound on the table size
- Finger table = partial list of nodes
- At each node,  $i^{\text{th}}$  entry in finger table identifies node that succeeds it by at least  $2^{i-1}$  in the circle
  - finger\_table[0]: immediate (1<sup>st</sup>) successor
  - finger\_table[1]: successor after that (2<sup>nd</sup>)
  - finger\_table[2]: 4<sup>th</sup> successor
  - finger\_table[3]: 8<sup>th</sup> successor
  - ...
- $O(\log N)$  nodes need to be contacted to find the node that owns a key  
... not as great as  $O(1)$  but way better than  $O(N)$

# Fault tolerance

- Nodes might die
  - (key, value) data would need to be replicated
  - Create  $R$  replicas, storing each one at  $R-1$  successor nodes in the ring
- It gets a bit complex
  - A node needs to know how to find its successor's successor (or more)
    - Easy if it knows all nodes!
  - When a node is back up, it needs to check with successors for updates
  - Any changes need to be propagated to all replicas

The end