# CS162
# Operating Systems and Systems Programming
# Lecture 23

## Peer-to-Peer Systems

April 18, 2011
Anthony D. Joseph and Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

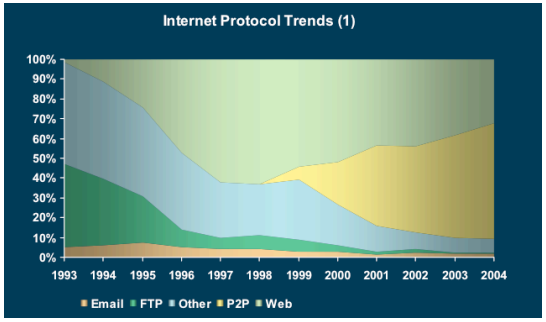
## Three Capstone Lectures

- Peer-to-Peer systems (today)
- Client-Sever (Monday)
- Cloud computing (Wednesday, April 25)
  - Invited lecture: Harry Li (Facebook)



## P2P Traffic

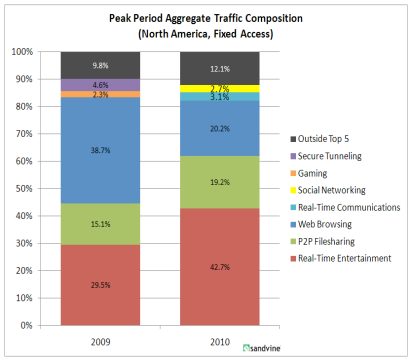
- 2004: some Internet Service Providers (ISPs) claimed > 50% was p2p traffic






## P2P Traffic

- Today, around 20%
- Big chunk now is video entertainment (e.g., Netflix, iTunes)



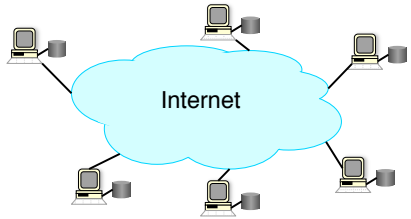
## Peer-to-Peer Systems

- What problem does it try to solve?
  - Provide highly scalable, cost effective (i.e., free!) services, e.g.,
    - » Content distribution (e.g., Bittorrent)
    - » Internet telephony (e.g., Skype)
    - » Video streaming (e.g., Octoshape)
    - » Computation (e.g., SETI@home)
- **Key idea:** leverage "free" resources of users (that use the service), e.g.,
  - Network bandwidth
  - Storage
  - Computation



## How Did it Start?

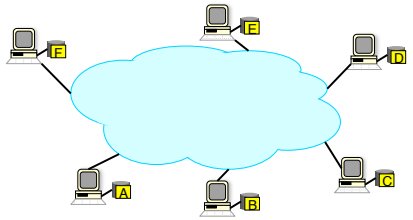
- A killer application: Napster (1999)
  - Free music over the Internet
- Use (home) user machines to store and distribute songs






## Model

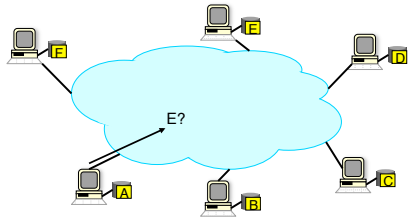
- Each user stores a subset of files
- Each user has access (can download) files from all users in the system






## Main Challenge

- Find a "good" node storing a specified file
- By "good" we mean:
  - Has correct content
  - Can get content fast
  - …



## Other Challenges

- Scale: up to hundred of thousands or millions of machines
- Dynamicity: machines can come and go at any time



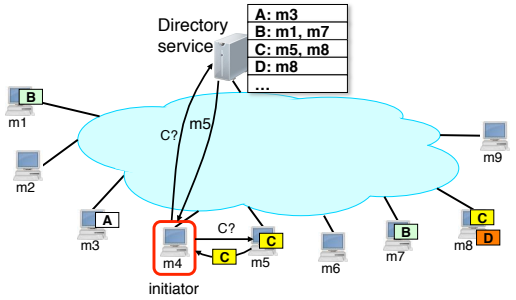
## Napster

- Implements a **centralized** lookup/directory service that maps files (songs) to machines currently in the system
- How to find a file (song)?
  - Query the lookup service → return a machine that stores the required file
    - » Ideally this is the closest/least-loaded machine
  - Download (ftp/http) the file
- Advantages:
  - Simplicity, easy to implement sophisticated search engines on top of a centralized lookup service
- Disadvantages:
  - Robustness, scalability (?)



## Napster: Example

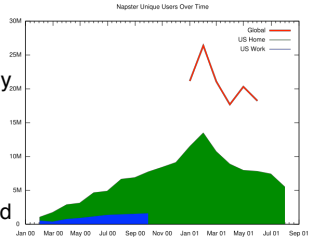
1) A client (initiator) contacts directory service to get file "C"
2) Directory service returns a (possible) close by and lightly loaded peer (m5) storing "C"
3) Client contacts directly m5 to get file "C"






## The Rise and Fall of Napster

- Founded by Shawn Fanning, John Fanning, and Sean Parker
- Operated between June 1999 and July 2001
  - More than 26 million users (February 2001)
- Several high profile songs were leaked before being released:
  - Metallica's "I Disappear" demo song
  - Madonna's "Music" single
- But, also helped made some bands successful (e.g., Radiohead, Dispatch)
- (Reemerged as music store in 2008)




(Source: http://en.wikipedia.org/wiki/File:Napster_Unique_Users.svg)

## The Aftermath

- **"Recording Industry Association of America (RIAA) Sues Music Startup Napster for $20 Billion" –** *December 1999*
- **"Napster ordered to remove copyrighted material" –** *March 2001*
- **Main legal argument:**
  - **Napster owns the lookup service, so it is directly responsible for disseminating copyrighted material**



## Gnutella (2000)

- What problem does it try to solve?
  - Get around the legal vulnerabilities by getting rid of the directory service
- Main idea: Flood the request to peers in the system to find file



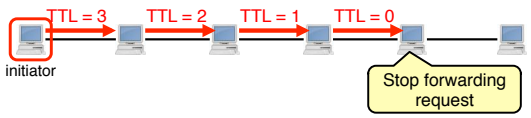
## Gnutella (2000)

- How does request flooding work?
  - Send request to all neighbors
  - Neighbors recursively multicast the request
  - Eventually a machine that has the file receives the request, and it sends back the answer
- Advantages:
  - Totally decentralized, highly robust
- Disadvantages:
  - Not scalable; the entire network can be swamped with requests (to alleviate this problem, each request has a TTL)



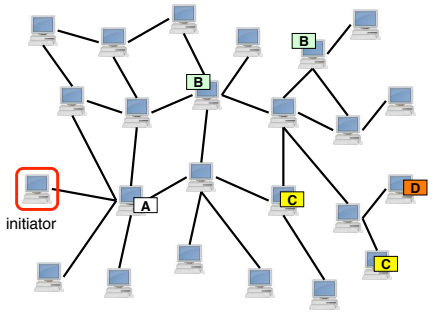
## Gnutella: Time To Live (TTL)

- When the client (initiator) sends a request, it associates a TTL with the request
- When a node forwards the request it decrements the TTL
- When TTL reaches 0, the request is no longer forwarded
- Typically, Gnutella uses TTL = 7
- Example: TTL = 3



## Gnutella: Example

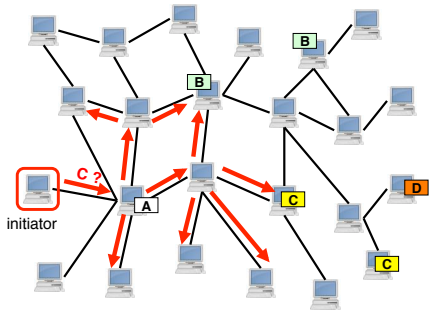
- Assume a client (initiator) asks for file “C”
- Assume TTL=2






## Gnutella: Example

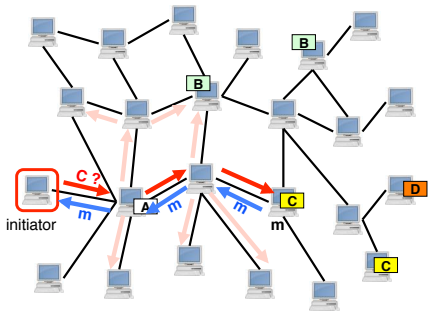
- Initiator send request to its neighbor(s)…
- … which recursively forward the request to their neighbors
- After 3 hops request is dropped






## Gnutella: Example

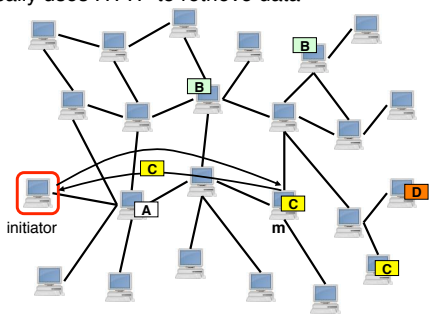
- If node has the requested file it sends a reply back
  - along the reverse path of the request, or
  - directly to initiator






## Gnutella: Example

- Initiator request file “C” from node “m”
  - Initiator may pick one of several machines if receive multiple replies
  - Typically uses HTTP to retrieve data



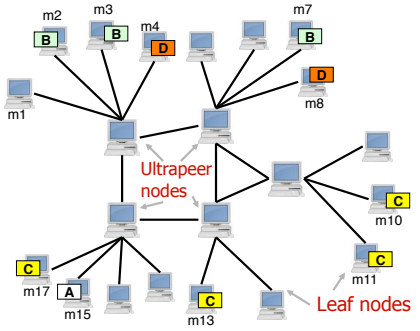
## Two-Level Hierarchy

- What problem does it try to solve?
  - Inefficient search
- Main idea: organize the p2p system in a two level hierarchy
  - Flooding happens only at the top level



## Two-Level Hierarchy

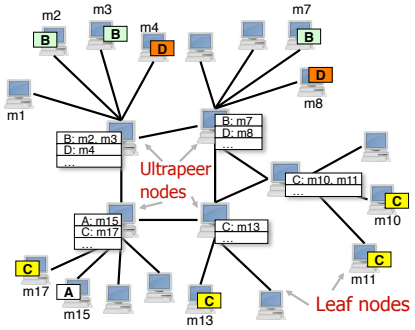
- KaZaa, subsequent versions of Gnutella
- Leaf nodes are connected to a small number of ultrapeers (supernodes)






## Two-Level Hierarchy

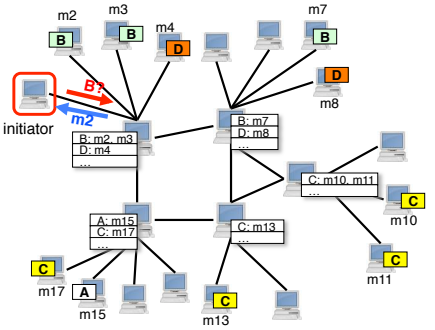
- Each ultra-peer builds a director for the content stored at its peers





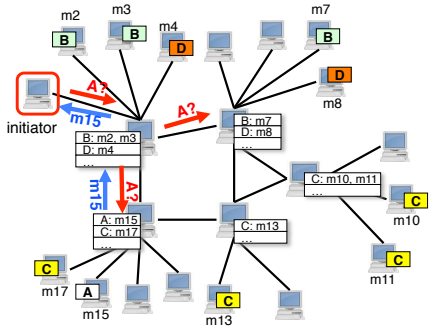

## Gnutella: Example

- Query: A leaf sends query to its ultrapeers
- If ultrapeer has requested content in its directory, the ultrapeer replies immediately



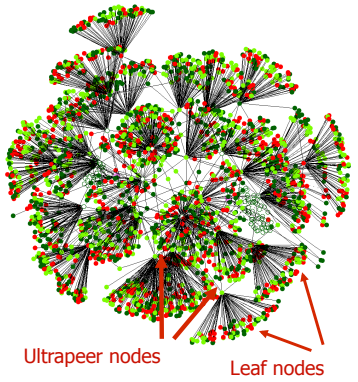
## Gnutella: Example

- Query: A leaf sends query to its ultrapeers
- If ultrapeer doesn't have content in its directory, the ultrapeer floods other ultrapeers






## Example: Oct 2003 Crawl on Gnutella






## Distributed Hash Tables (DHTs)

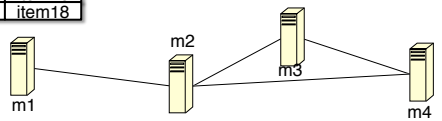
- What problem does it solve?
  - Scalable, low-overhead lookup protocol
  - Guarantee a file is found if it is in the system
- Abstraction: a distributed hash-table data structure
  - *insert(id, item);*
  - *item = query(id);*
  - Note: *item* can be anything: a data object, document, file, pointer to a file…
- Proposals
  - CAN, Chord, Kademlia, Pastry, Viceroy, Tapestry, etc



## DHTs

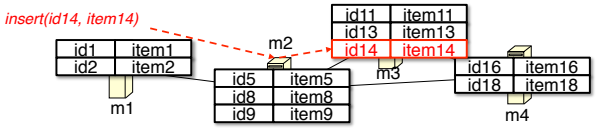
- Partition hash table across nodes

| | |
|---|---|
| id1 | item1 |
| id2 | item2 |
| id5 | item5 |
| id8 | item8 |
| id9 | item9 |
| id11 | item11 |
| id13 | item13 |
| id16 | item16 |
| id18 | item18 |



## DHT: Insertion

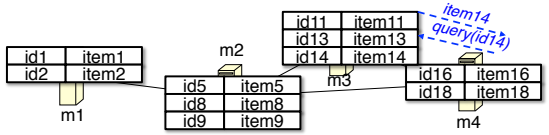
- Call *insert(id4, item14)* at m1
  - Find node responsible for id4, i.e., node m3
  - Insert (id14, item14) at m3






## DHT: Query

- Call *query(id4)* at m4
  - Find node responsible for id4, i.e., m3
  - Return (id14, item14) to m4






## DHT: Lookup Service

- Key primitive: lookup service
  - Given an ID, map it to a host
- Scalability: hundreds of thousands or millions of machines
- One example: Chord (Lecture 15)
- Another example: CAN (next)

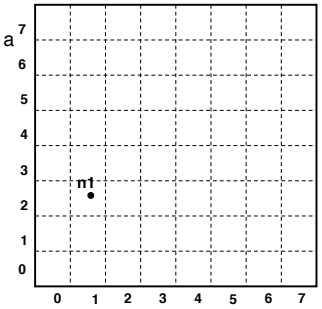


## Content Addressable Network (CAN)

- Associate to each node and item a unique *id* in an *d*-dimensional space, e.g., torus
- Assigning *d*-dimensional ID to content (file):
  - Use *d* hash functions (i.e., $h_1()$, $h_2()$,.., $h_d()$) to compute d coordinates using name or content
  - $id = (id_1, id_2, \ldots, id_d)$, where $id_i = h_i(\text{content})$;
- Assigning *d*-dimensional ID to new node, $n_{new}$:
  1) Pick random $id_{new}$
  2) Find node $n_1$ responsible for $id_{new}$
  3) $n_1$ may pick a new ID for $n_{new,}$ $id'_{new,}$ to better split the ID space

## CAN Example: Two Dimensional Space

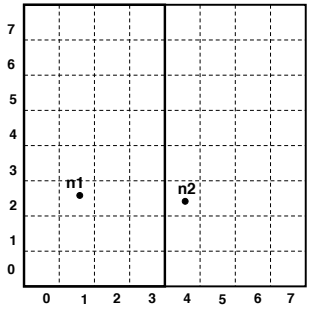
- Space divided between nodes
- All nodes cover the entire space
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
- Example:
  - Assume space size (8 x 8)
  - Node n1:(1, 2) first node that joins → cover the entire space






## CAN Example: Two Dimensional Space

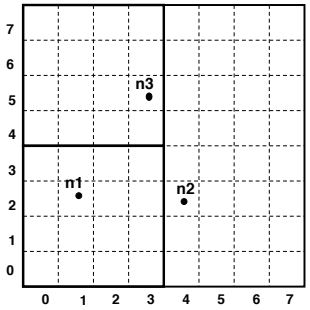
- Node n2:(4, 2) joins → space is divided between n1 and n2






## CAN Example: Two Dimensional Space

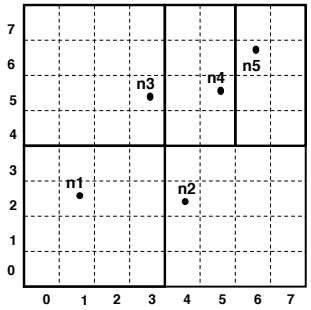
- Node n2:(4, 2) joins → space is divided between n1 and n2





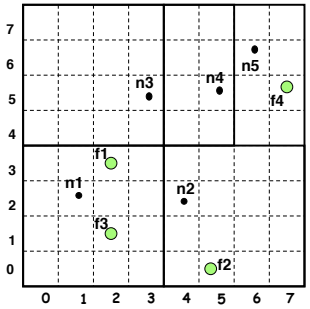

## CAN Example: Two Dimensional Space

- Nodes n4:(5, 5) and n5:(6,6) join



## CAN Example: Two Dimensional Space

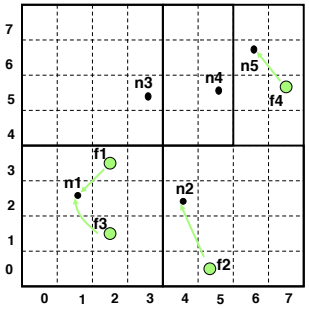
- Nodes: n1:(1, 2); n2:(4,2); n3:(3, 5); n4:(5,5);n5:(6,6)
- Items: f1:(2,3); f2:(5,0); f3:(2,1); f4:(7,5);






## CAN Example: Two Dimensional Space

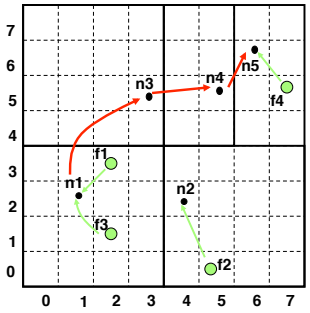
- Each item is stored by the node who owns its mapping in the space






## CAN: Query Example

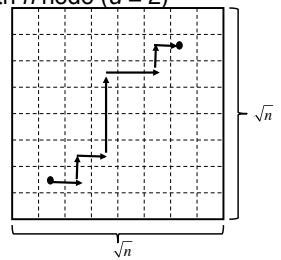
- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: assume n1 queries f4






## Content Addressable Network (CAN)

- Properties
  - Routing table size O(*d*), i.e., each node needs to know about O(*d*) neighbors
  - Guarantees that a file is found in at most $d*n^{1/d}$ steps, where *n* is the total number of nodes
- Example: grid with *n* node (*d* = 2)



## 5min Break



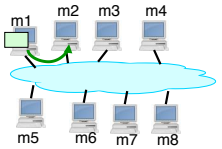
## BitTorrent (2001): The Problem

- What problem does it try to solve?
  - **Efficient** distribution of **large** files (e.g., movies)
- Example:
  - Assume we want to distribute F=1GB file to 8 other nodes
  - Assume all clients are identical and have uplink capacity of C=1 Mbps
  - How long does it take to distribute the file using Napster/ Gnutella/Kazza?

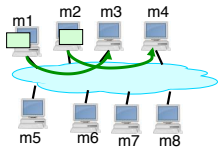


## Efficient Large File Distribution

- Step 1: copy file from m1→m2
  - Duration: F/C = 1GB/1Mbps = 8,000sec



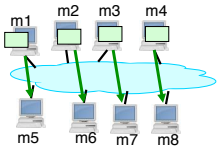

- Step 2: copy files from m1→m3 and m2→m4
  - Duration: 8,000sec






## Efficient Large File Distribution

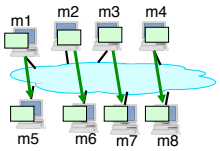
- Step 3: copy files from m1→m5 and m2→m6, m3→m7, m4→m8
  - Duration: 8000sec




- Step k: by end of step k we transfer file to $2^k$ nodes

## Efficient Large File Distribution

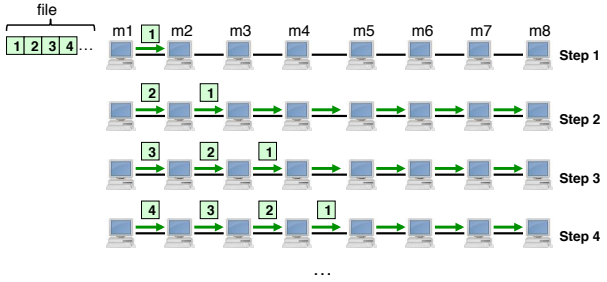
- Total time to transfer file to N machines: $\lceil \log N \rceil \times \frac{F}{C}$
  - If N = 8 → total distribution time = 24,000sec
- Can we do better?
- Yes!
  - Divide file into chunks
  - Pipeline chunk distribution






## Efficient Large File Distribution

- Dive file into blocks of size *B*
- Use chain distribution






## Efficient Large File Distribution


- Example:
  - File size: F=1GB
  - block size: B=1MB
  - # of nodes: N
  - Total time to distribute file:
    - » F/C + (N-1)*B/C =
      = 1GB/1Mbps + (N-1)*1MB/1Mbps = 8,000sec + (N-1)*8sec
    - » If N = 8 → Total distribution time = 8,056sec

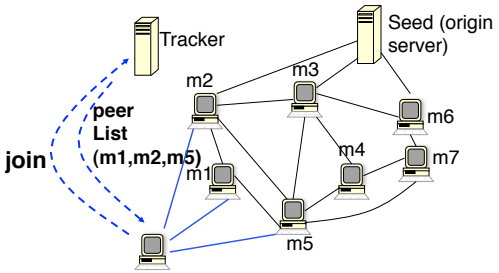


## BitTorrent (2001)

- Allow fast downloads even when sources have low up-link capacity
- How does it work?
  - **Seed** (origin) – site storing the file to be downloaded
  - **Tracker** – server maintaining the list of peers in system
  - Split each file into **pieces** (~ 256 KB each), and each piece into **sub-pieces** (~ 16 KB each)
  - The loader loads one piece at a time
  - Within one piece, the loader can load up to five sub-pieces in **parallel**

## BitTorrent: Join Procedure

1) Peer contacts tracker responsible for file it wants to download
2) Tracker returns a list of peer (20-50) downloading same file
3) Peer connects to peers in the list






## BitTorrent: Download Algorithm

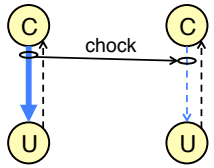
- Download consists of three phases:
- **Start:** get a piece as soon as possible
  - Select a random piece
- **Middle:** spread all pieces as soon as possible
  - Select rarest piece next
- **End:** avoid getting stuck with a slow source, when downloading the last sub-pieces
  - Request in parallel the same sub-piece
  - Cancel slowest downloads once a sub-piece has been received

(For details see: http://bittorrent.org/bittorrentecon.pdf)



## Bittorrent: Handling Freeriders

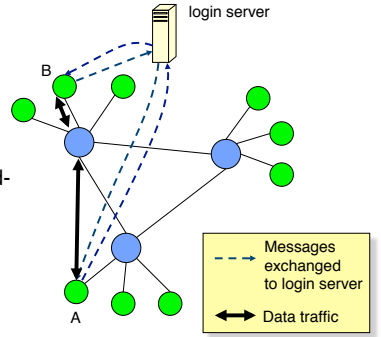
- Free riders: peers that use the network without contributing (with the upstream bandwidth)
- Solution: chocking, a variant of Tit-for-Tat
  - Each peer has a limited number of upload slots
  - When a peer's upload bandwidth is saturated, it exchanges upload bandwidth for download bandwidth
  - If peer U downloads from peer C and doesn't upload in return, C chokes download to U






## Skype (2003)

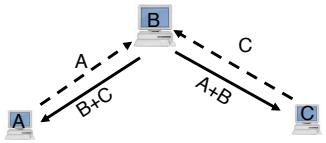
- Peer-to-peer Internet Telephony
- Two-level hierarchy (like KaZaa)
  - Ultrapeers used to route traffic between NATed end-hosts…
  - … plus a login server to
    - » authenticate users
    - » ensure that names are unique across network




(Note*: probable protocol; Skype protocol is not published)

## Skype (cont'd)

- Most powerful machine elected as a host and act as a mixer
- Example:
  - B elected as host/mixer
  - A and C sends their audio streams to B
  - B mixes the missing streams for A and C and sends mixed stream to each of them
- Two-way call: 36 kb/s
- Three-way call: 54 kb/s






## Summary

- The key challenge of building wide area P2P systems is a scalable and robust directory service
- Solutions covered in this lecture
  - Naptser: centralized location service
  - Gnutella: broadcast-based decentralized location service
  - CAN, Chord, Tapestry, Pastry: intelligent-routing decentralized solution
    - » Guarantee correctness
- Bittorrent: efficient distribution of large files
  - Split file into chunks and blocks
  - Parallelize and pipeline transfers