Peer-to-Peer Protocols and Systems

TA: David Murray

15-441 Spring 2006

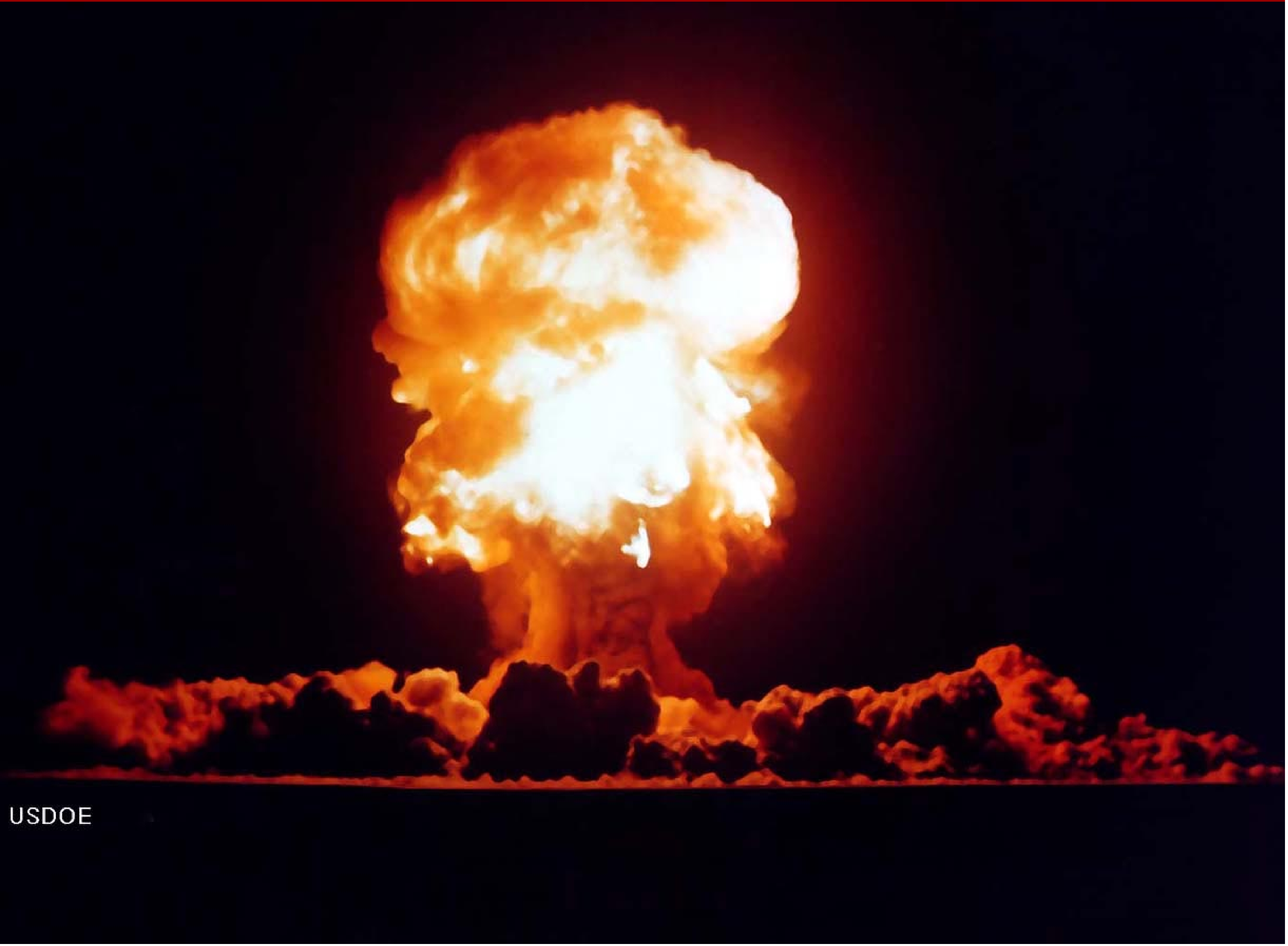
4/19/2006

P2P - Outline

- What is P2P?
- P2P System Types
  - 1) File-sharing
  - 2) File distribution
  - 3) Streaming
- Uses & Challenges

Problem: Scalability

- Hundreds of clients => 1 server
  - OK
- Thousands of clients => 1 server
  - Maybe OK
- Millions/billions of clients => 1 server
  - What happens?...



USDOE

Solution:

Distribute the cost among the end users

Three Classes of P2P Systems

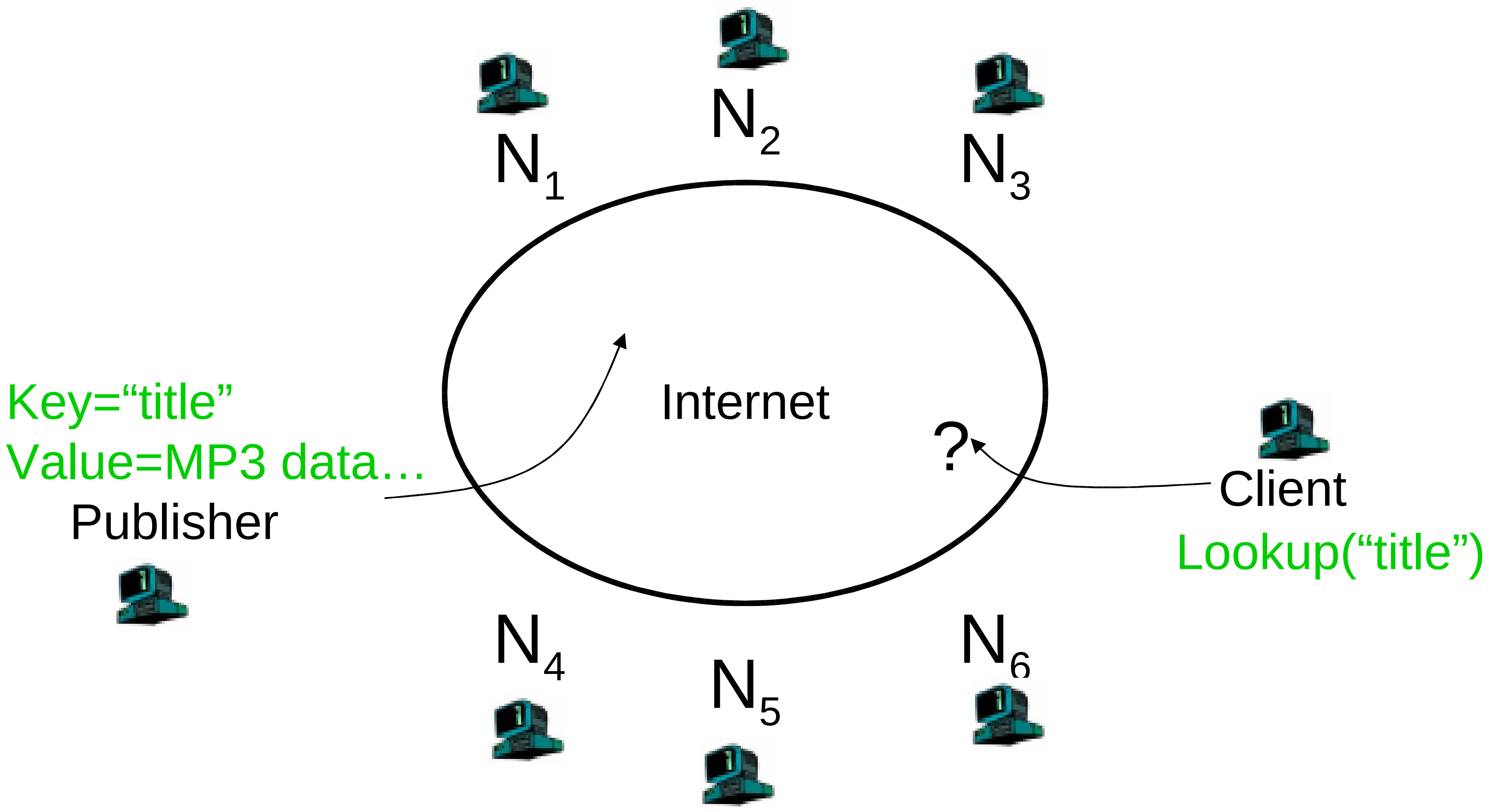
- 1) File-sharing
  - (old) Napster (centralized)
  - Gnutella (flooding)
  - KaZaA (intelligent flooding)
  - DHTs/Chord (structured overlay routing)
- 2) File distribution
  - BitTorrent
- 3) Streaming
  - End System Multicast (a.k.a. Overlay Multicast)

1) P2P File-sharing Systems

1) P2P File-sharing Systems

- **Centralized Database**
  - (old) Napster
- **Query Flooding**
  - Gnutella
- **Intelligent Query Flooding**
  - KaZaA
- **Structured Overlay Routing**
  - Distributed Hash Tables

File searching



File searching

- Needles vs. Haystacks
  - – Searching for top 40, or an obscure punk track from 1981 that nobody’s heard of?
- Search expressiveness
  - – Whole word? Regular expressions? File names? Attributes? Whole-text search?
    - (e.g., p2p gnutella or p2p google?)

File-sharing: Framework

- Common Primitives:
  - **Join**: how do I begin participating?
  - **Publish**: how do I advertise my file?
  - **Search**: how do I find a file?
  - **Fetch**: how do I retrieve a file?

P2P File-sharing Systems

- **Centralized Database**
  - (old) Napster
- **Query Flooding**
  - Gnutella
- **Intelligent Query Flooding**
  - KaZaA
- **Structured Overlay Routing**
  - Distributed Hash Tables

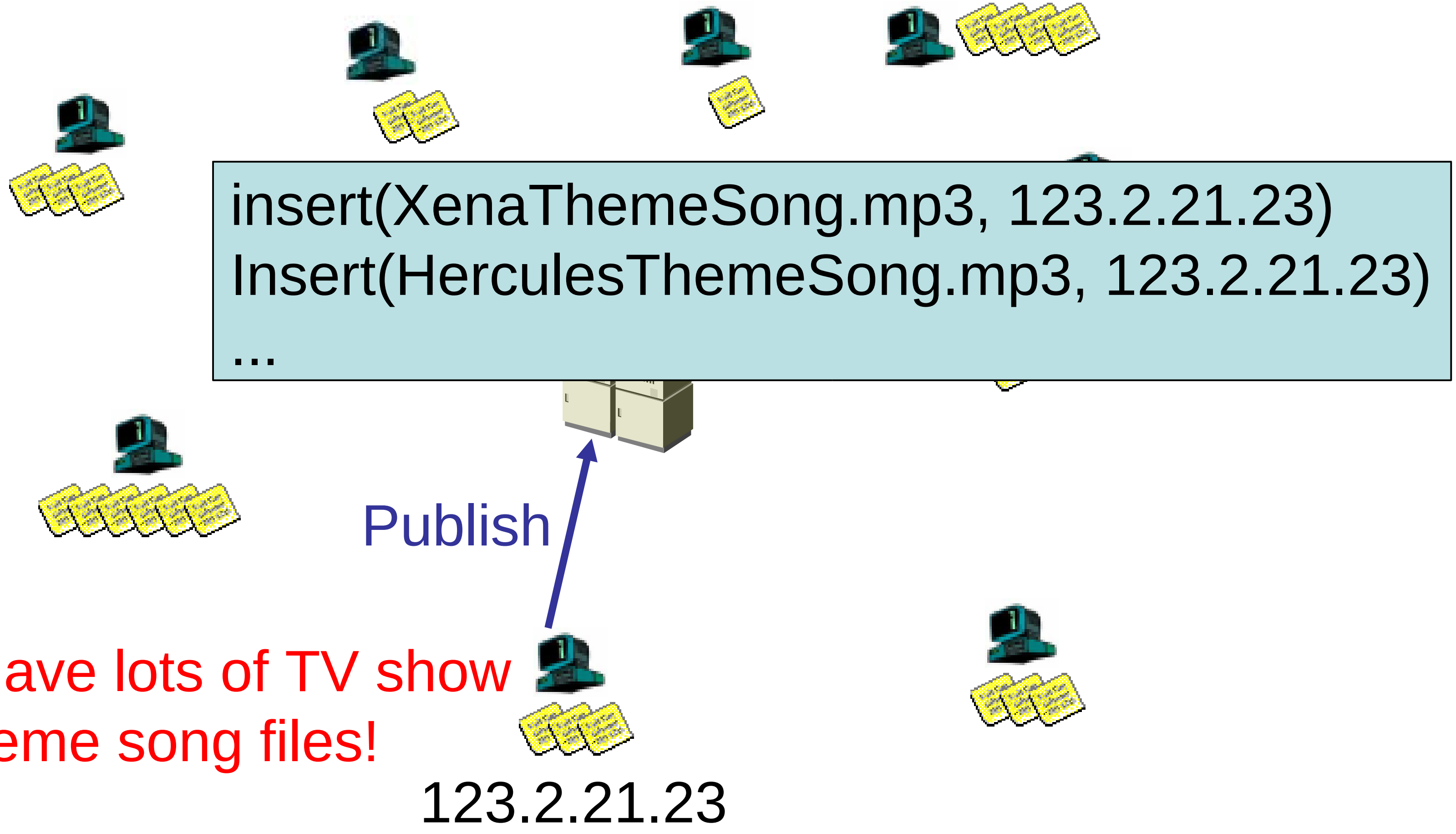
(old) Napster: History

- 1999: Sean Fanning launches Napster
- Peaked at 1.5 million simultaneous users
- Jul 2001: Napster shuts down
- [2003: Napster's name reused for an online music service (no relation)]

(old) Napster: Overview

- Centralized Database:
  - **Join**: on startup, client contacts central server
  - **Publish**: reports list of files to central server
  - **Search**: query the server => return someone that stores the requested file
  - **Fetch**: get the file directly from peer

Napster: Publish



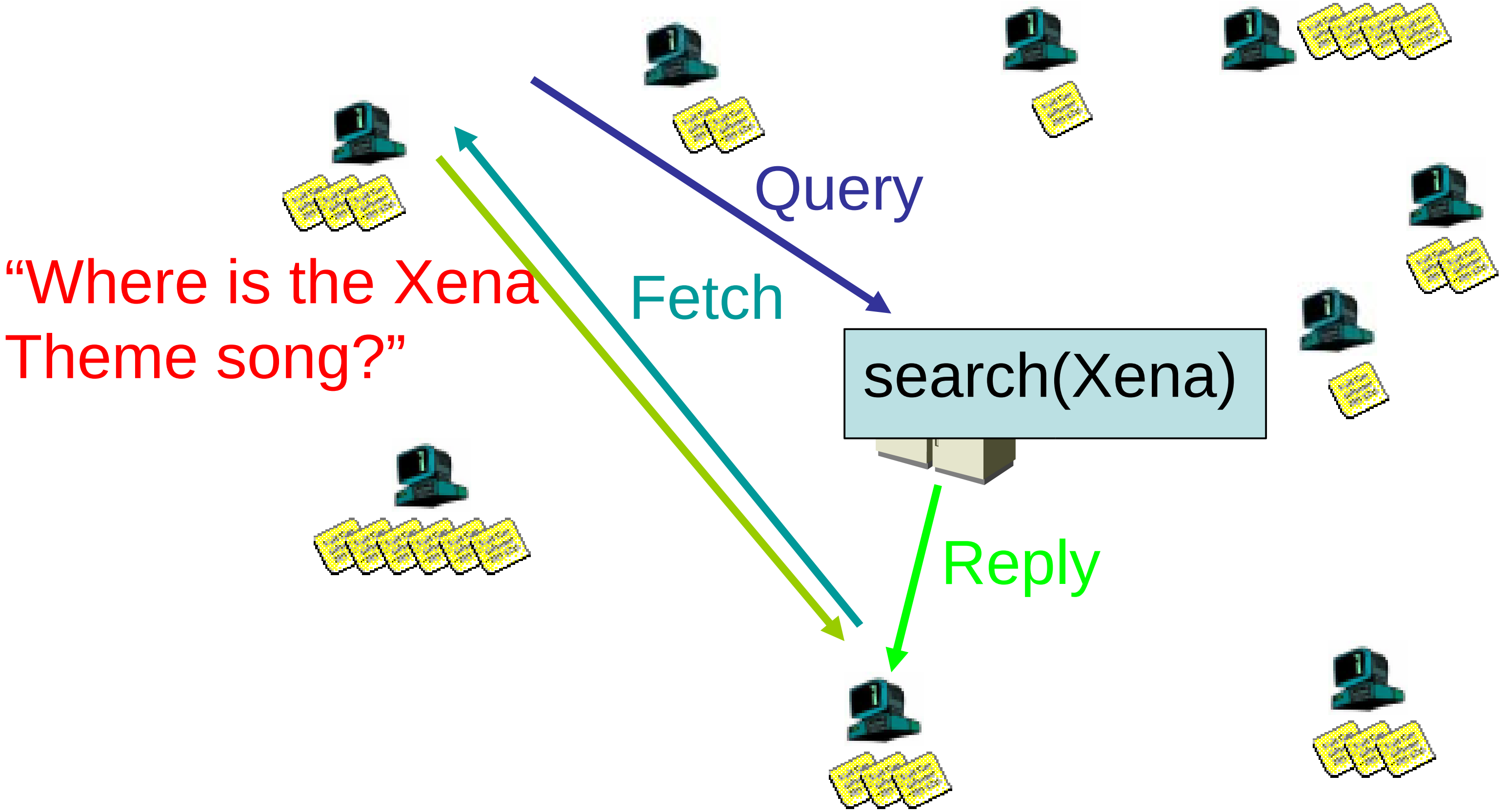
insert(XenaThemeSong.mp3, 123.2.21.23)
Insert(HerculesThemeSong.mp3, 123.2.21.23)
...

Publish

I have lots of TV show theme song files!

123.2.21.23

Napster: Search



Napster: Discussion

- Pros:
  - Simple
  - Search scope is O(1)
  - Controllable (pro or con?)
- Cons:
  - Server maintains O(N) State
  - Server does all processing
  - Single point of failure

P2P File-sharing Systems

- **Centralized Database**
  - (old) Napster
- **Query Flooding**
  - Gnutella
- **Intelligent Query Flooding**
  - KaZaA
- **Structured Overlay Routing**
  - Distributed Hash Tables

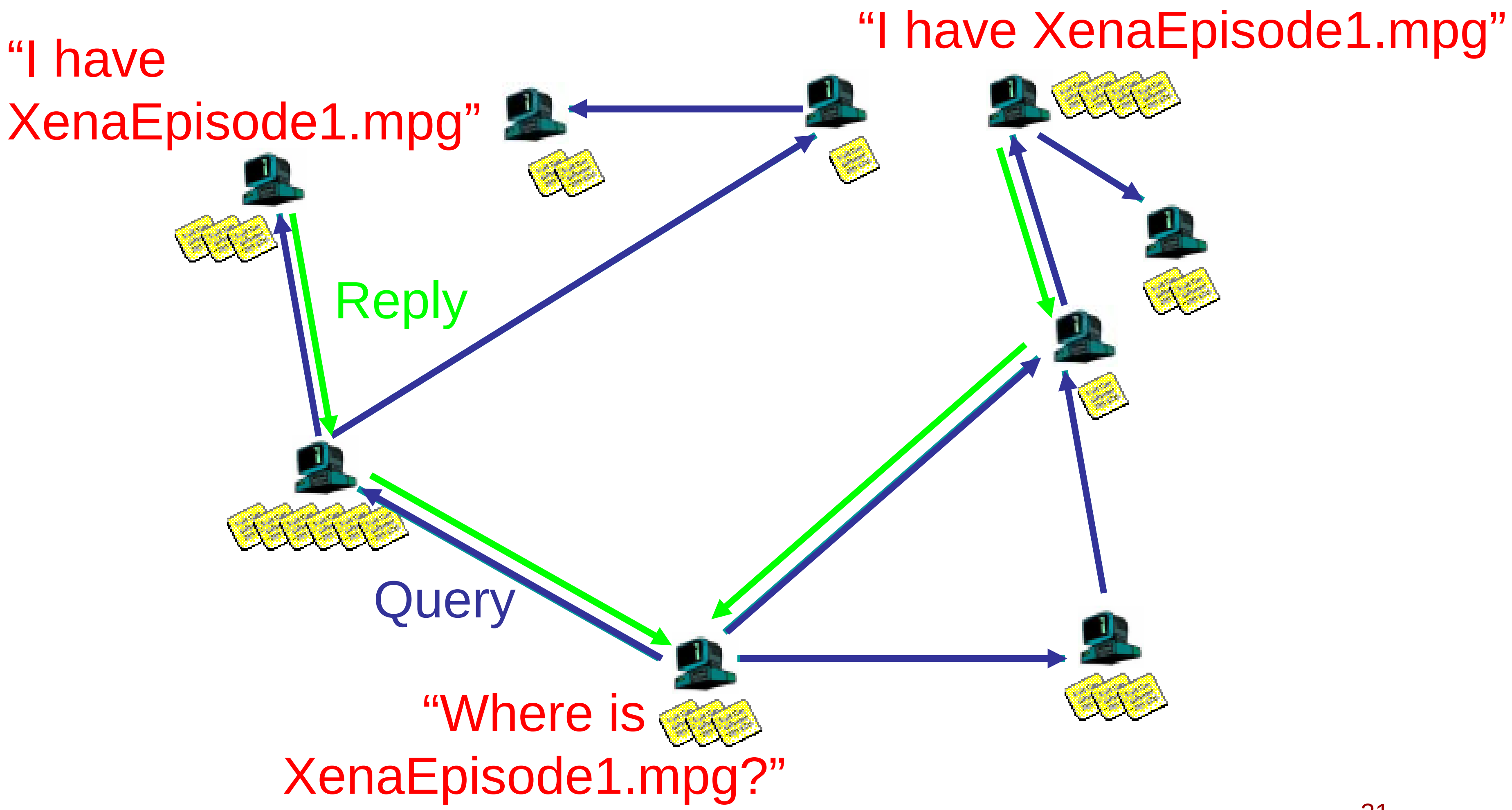
Gnutella: History

- In 2000, J. Frankel and T. Pepper from Nullsoft released Gnutella
- Soon many other clients: Bearshare, Morpheus, LimeWire, etc.
- In 2001, many protocol enhancements including “ultrapeers”

Gnutella: Overview

- Query Flooding:
  - **Join**: on startup, client contacts a few other nodes; these become its “neighbors”
  - **Publish**: (no need)
  - **Search**: ask neighbors, who ask their neighbors, and so on... when/if found, reply to sender.
    - TTL limits propagation
  - **Fetch**: get the file directly from peer

Gnutella: Search


"I have XenaEpisode1.mpg"
"I have XenaEpisode1.mpg"
Reply
Query
"Where is XenaEpisode1.mpg?"

Gnutella: Discussion

- Pros:
  - Fully de-centralized
  - Search cost distributed
  - Processing @ each node permits powerful search semantics
- Cons:
  - Search scope is O(*N*)
  - Search time is O(???)
  - Nodes leave often, network unstable
- TTL-limited search works well for haystacks.
  - For scalability, does NOT search every node. May have to re-issue query later

P2P File-sharing Systems

- Centralized Database
  - (old) Napster
- Query Flooding
  - Gnutella
- **Intelligent Query Flooding**
  - KaZaA
- Structured Overlay Routing
  - Distributed Hash Tables

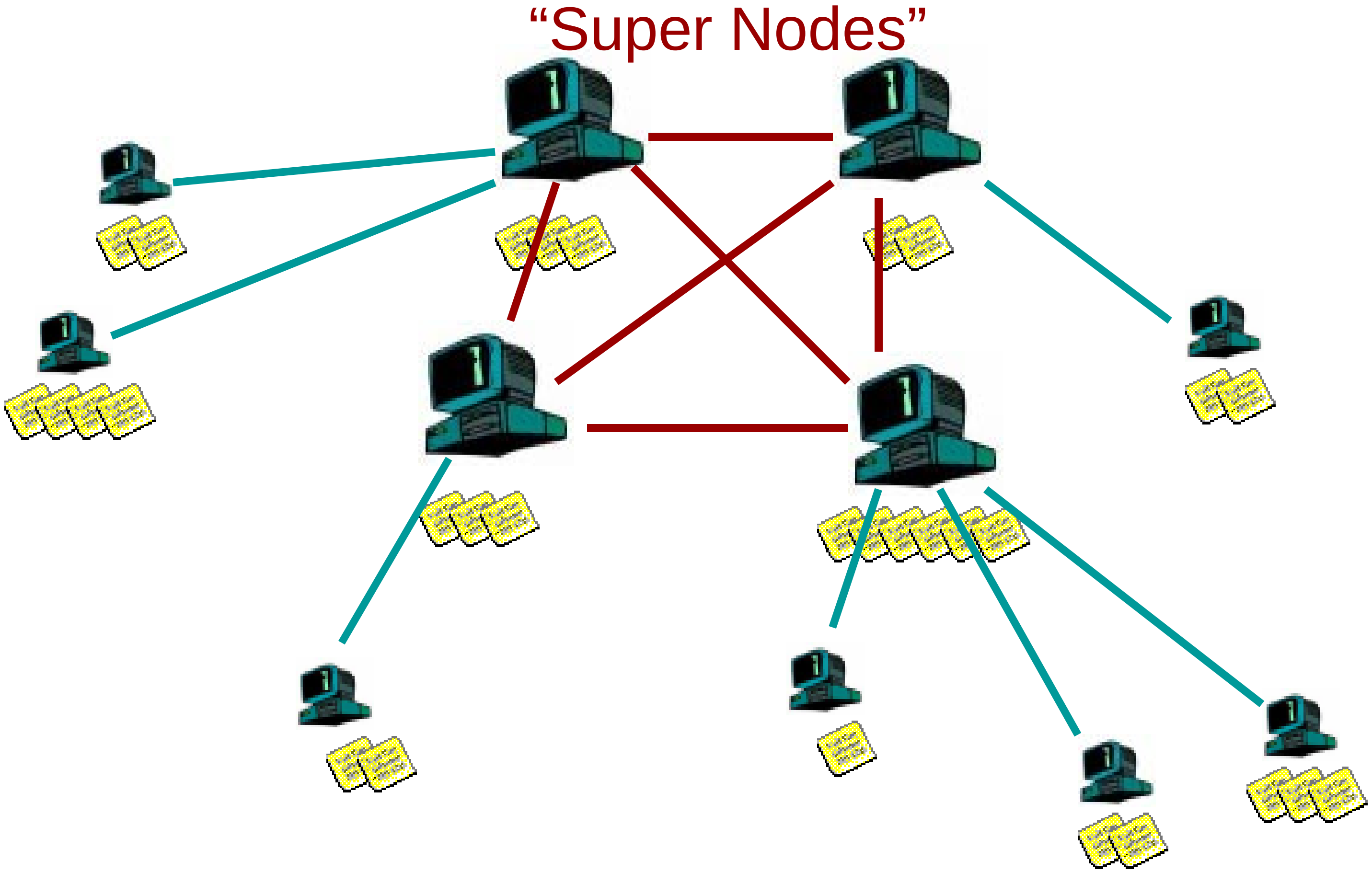
KaZaA: History

- In 2001, KaZaA created by Dutch company Kazaa BV
- Single network called FastTrack used by other clients as well: Morpheus, giFT, etc.
- Eventually protocol changed so other clients could no longer talk to it

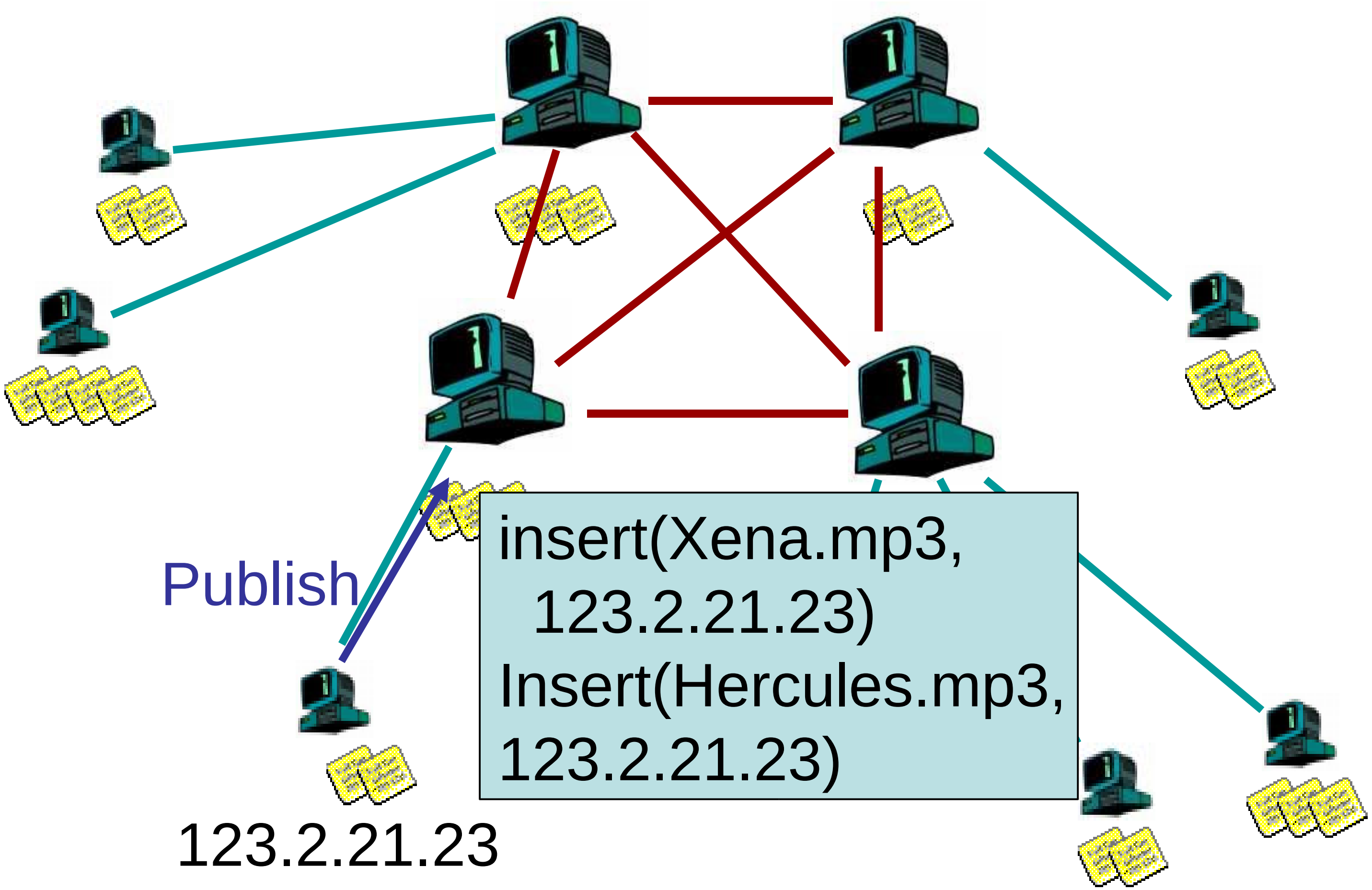
KaZaA: Overview

- “Smart” Query Flooding:
  - **Join**: on startup, client contacts a “supernode” ... may at some point become one itself
  - **Publish**: send list of files to supernode
  - **Search**: send query to supernode, supernodes flood query amongst themselves.
  - **Fetch**: get the file directly from peer(s); can fetch simultaneously from multiple peers

KaZaA: Network Design



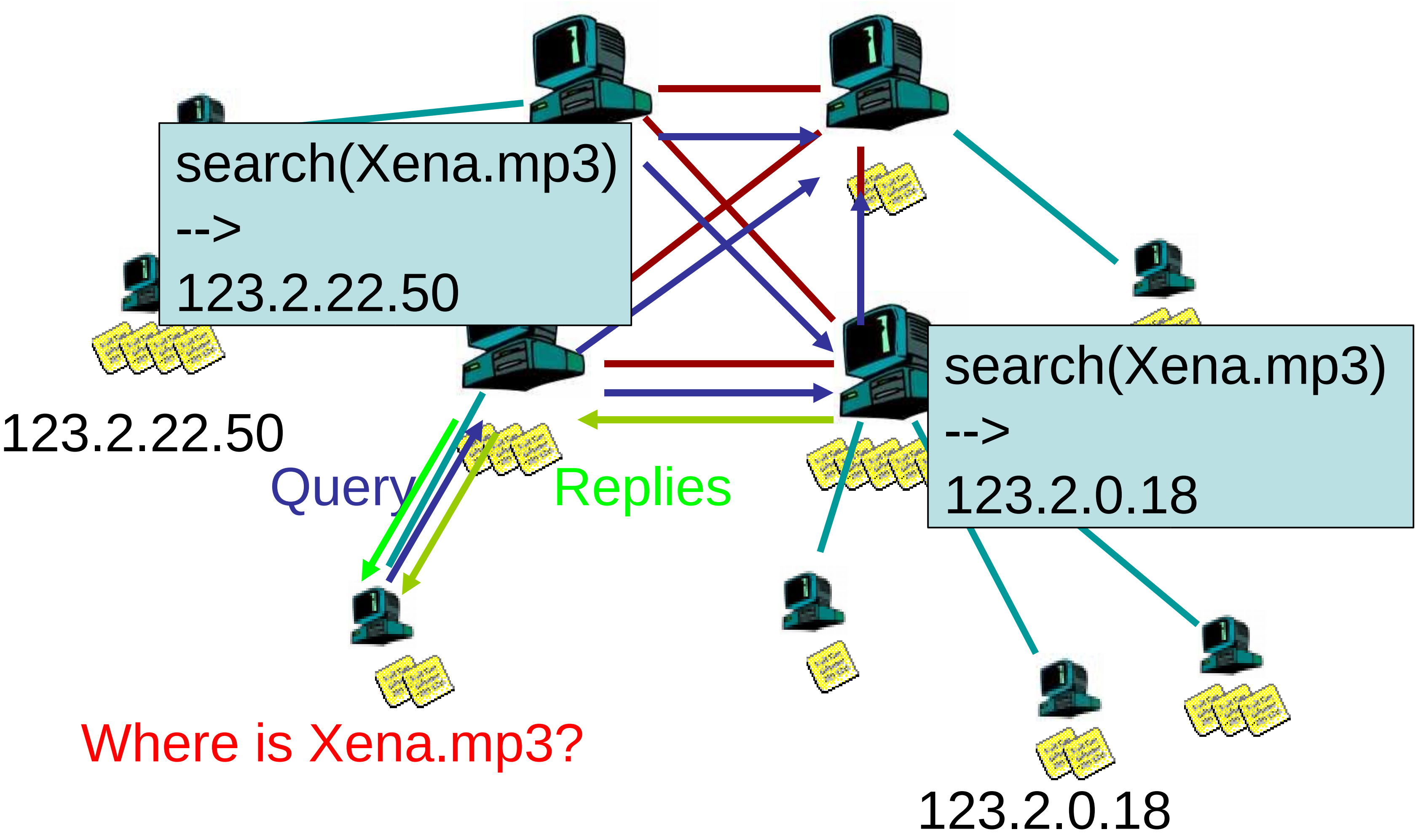
KaZaA: File Insert



insert(Xena.mp3,
123.2.21.23)
Insert(Hercules.mp3,
123.2.21.23)

I have lots of files!

KaZaA: File Search



search(Xena.mp3) --> 123.2.22.50

search(Xena.mp3) --> 123.2.0.18

KaZaA: Fetching

- More than one node may have requested file...
- How to tell?
  - Must be able to distinguish identical files
  - Not necessarily same filename
  - Same filename not necessarily same file...
- Use Hash of file
  - KaZaA uses its own “UUHash”: fast, but not secure
  - Alternatives: MD5, SHA-1
- How to fetch?
  - Get bytes [0..1000] from A, [1001...2000] from B
  - Alternative: Erasure Codes

KaZaA: Discussion

- Pros:
  - Tries to take into account node heterogeneity:
    - Bandwidth
    - Host Computational Resources
    - Host Availability (?)
  - Rumored to take into account network locality
- Cons:
  - Mechanisms easy to circumvent
  - Still no real guarantees on search scope or search time
- Similar behavior to Gnutella, but better.

Stability and Superpeers

- Why supernodes?
  - Query consolidation
    - Many connected nodes may have only a few files
    - Propagating a query to a sub-node would take more b/w than answering it yourself
  - Caching effect
    - Requires network stability
- Supernode selection is time-based
  - How long you've been on is a good predictor of how long you'll be around.

P2P File-sharing Systems

- **Centralized Database**
  - (old) Napster
- **Query Flooding**
  - Gnutella
- **Intelligent Query Flooding**
  - KaZaA
- **Structured Overlay Routing**
  - Distributed Hash Tables

Distributed Hash Tables

- Academic answer to P2P
- Goals
  - Guaranteed lookup success
  - Provable bounds on search time
  - Provable scalability
- Makes some things harder
  - Fuzzy queries / full-text search / etc.
- Read-write, not read-only
- Hot Topic in networking since introduction in ~2000/2001

DHT: Overview

- **Abstraction**: a distributed “hash-table” (DHT) data structure:
  - put(id, item);
  - item = get(id);
- **Implementation**: nodes in system form a distributed data structure
  - Can be Ring, Tree, Hypercube, Skip List, Butterfly Network, ...

DHT: Overview (continued)

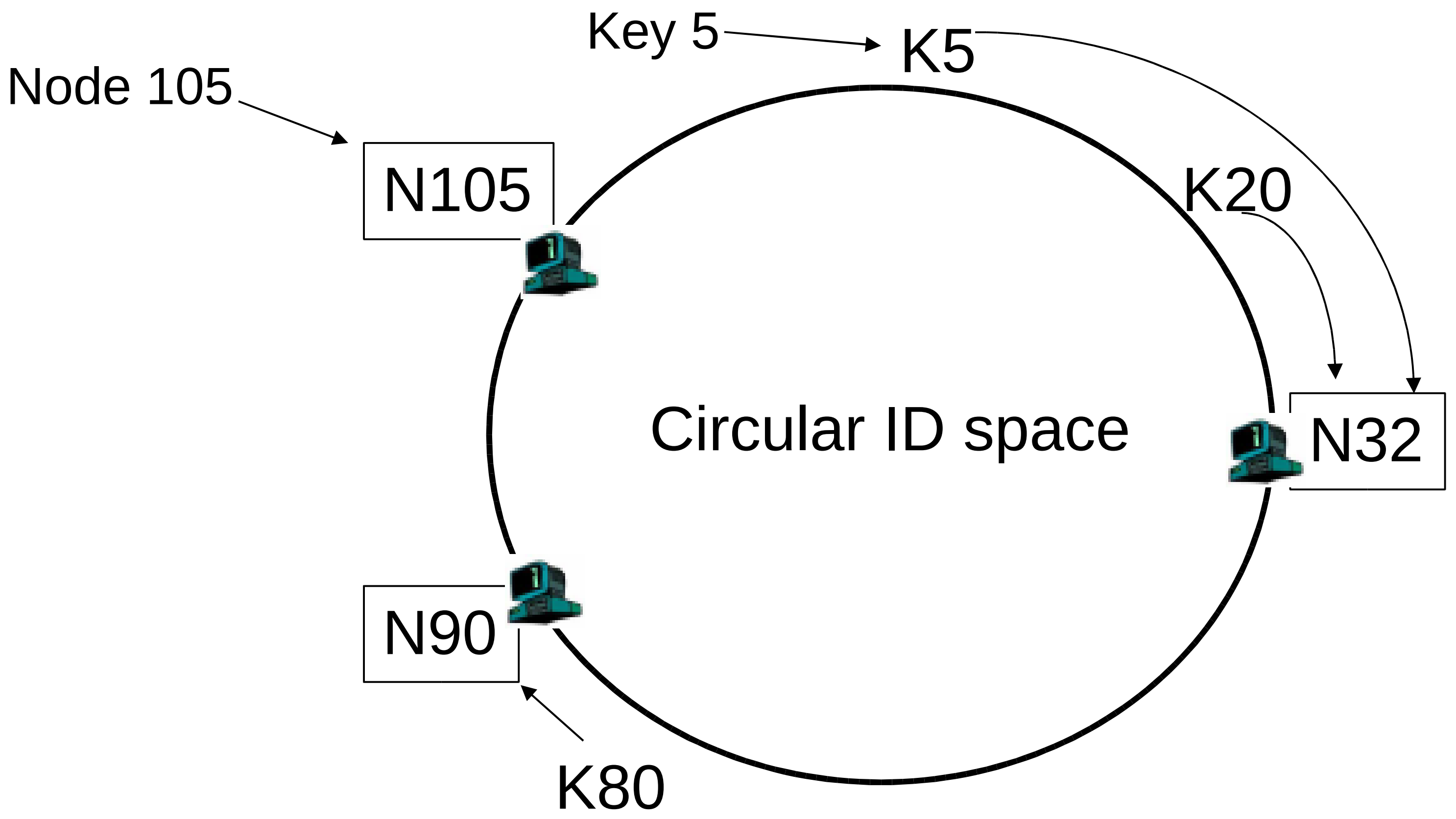
- Structured Overlay Routing:
  - **Join**: On startup, contact a “bootstrap” node and integrate yourself into the distributed data structure; get a *node id*
  - **Publish**: Route publication for *file id* toward a close *node id* along the data structure
  - **Search**: Route a query for file id toward a close node id. Data structure guarantees that query will meet the publication. (Note: cannot do keyword search)
  - **Fetch**: Two options:
    - Publication contains actual file => fetch from where query stops
    - Publication says “I have file X” => query tells you 128.2.1.3 has X, use IP routing to get X from 128.2.1.3

DHT: Example – Chord

- Associate to each node and file a unique *id* in an *uni*-dimensional space (a Ring)
  - E.g., pick from the range [0...2^m]
  - Usually the hash of the file or IP address
- Properties:
  - “It allows a distributed set of participants to agree on a single node as a rendezvous point for a given key, without any central coordination.” (Chord site)
  - Can find data using O(log *N*) messages, where *N* is the total number of nodes
    - (Why? We’ll see…)

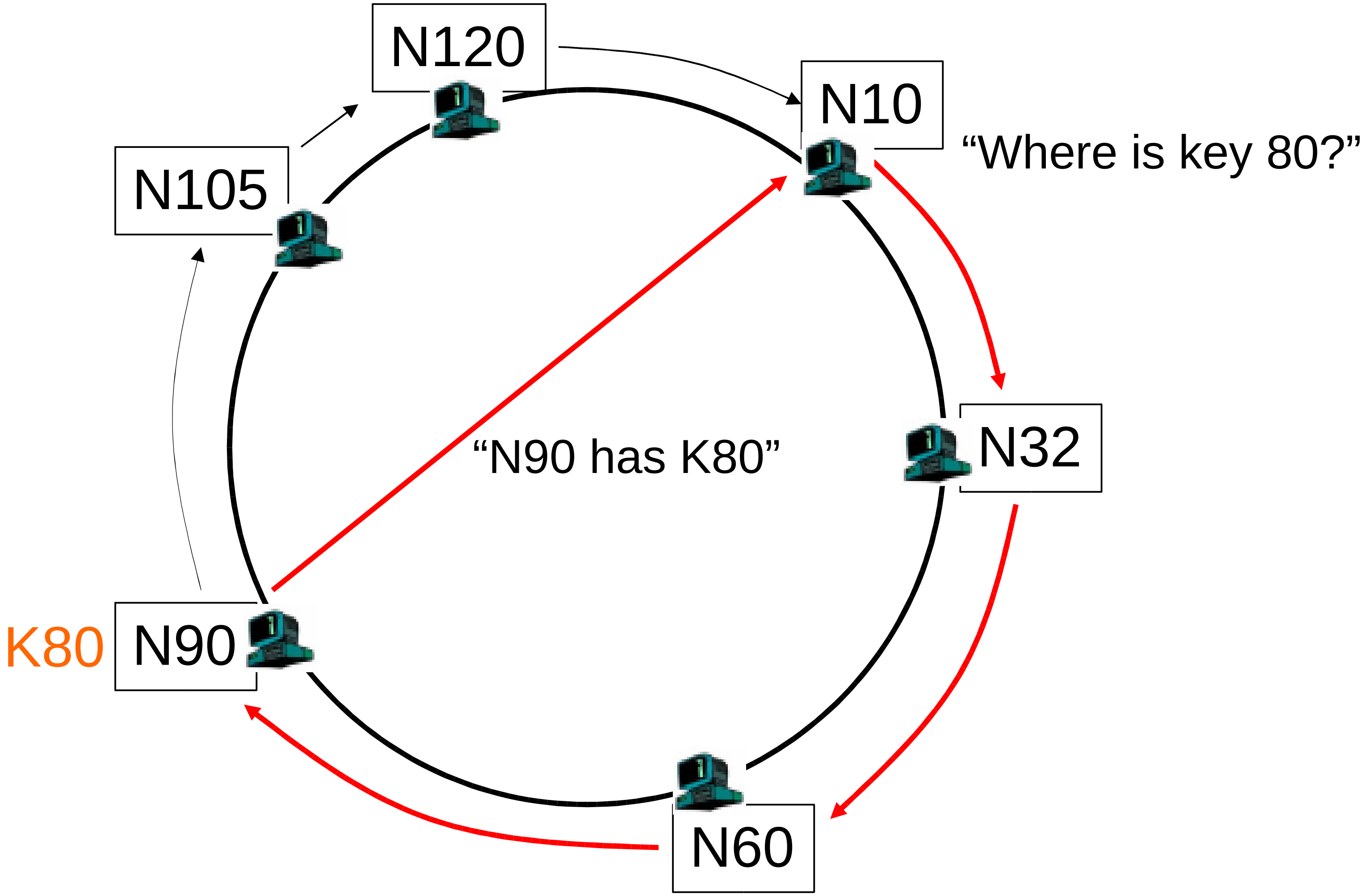
from MIT in 2001

DHT: Consistent Hashing

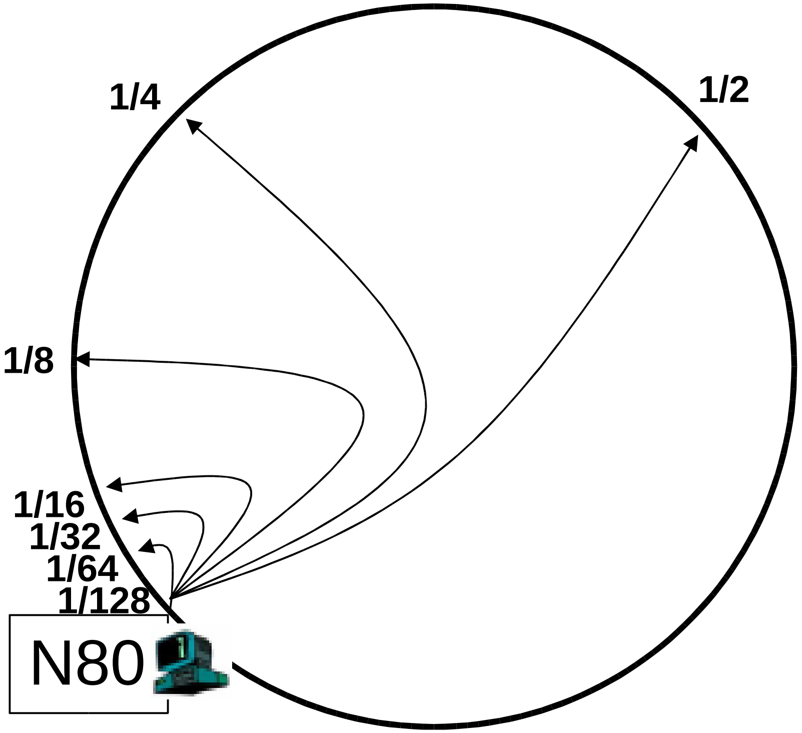


A key is stored at its successor: node with next higher ID

DHT: Chord Basic Lookup



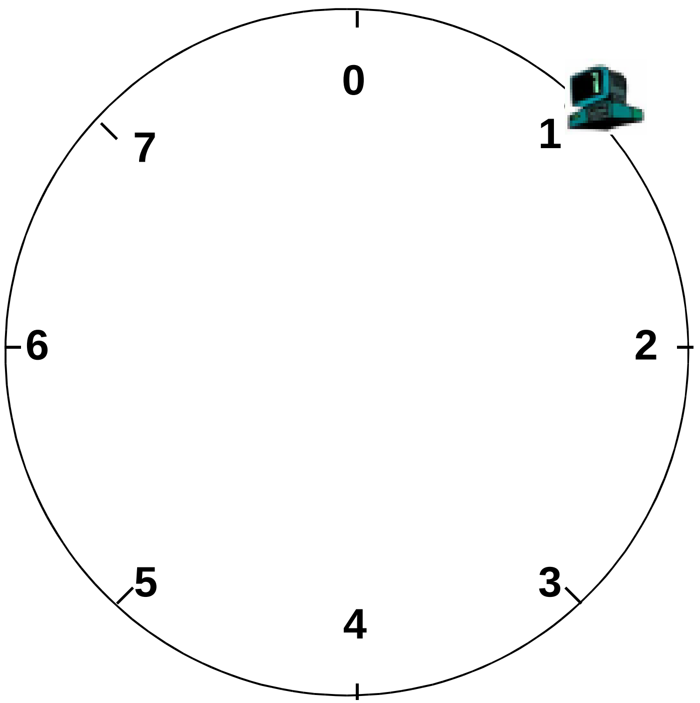
DHT: Chord “Finger Table”



- Entry *i* in the finger table of node *n* is the first node that succeeds or equals *n* + 2^{i}
- In other words, the ith finger points 1/2^{n-i} way around the ring
- (This is what makes O(log N) messages for a retrieval possible!)

DHT: Chord Join

- Assume an identifier space [0..8]
- Node n1 joins

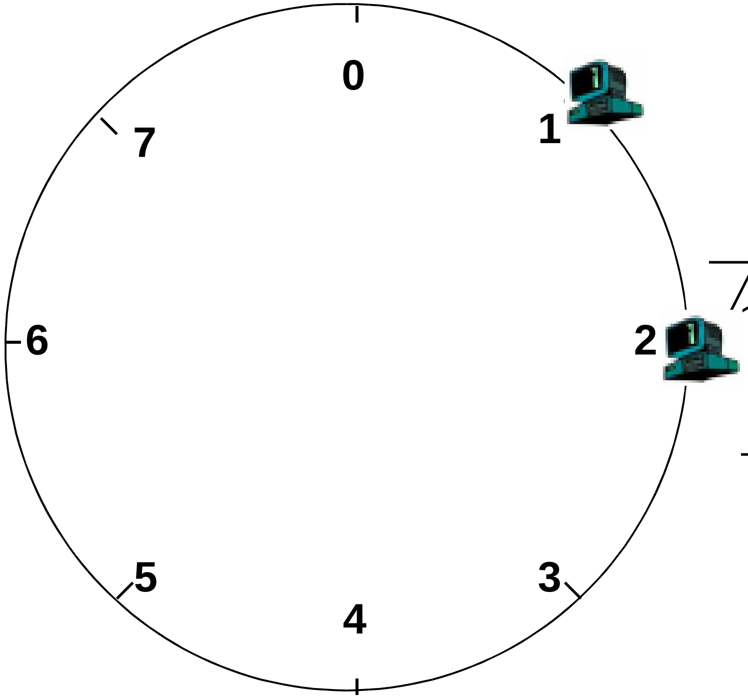


Succ. Table

i	$id+2^i$	succ
0	2	1
1	3	1
2	5	1

DHT: Chord Join

- Node n2 joins



Succ. Table

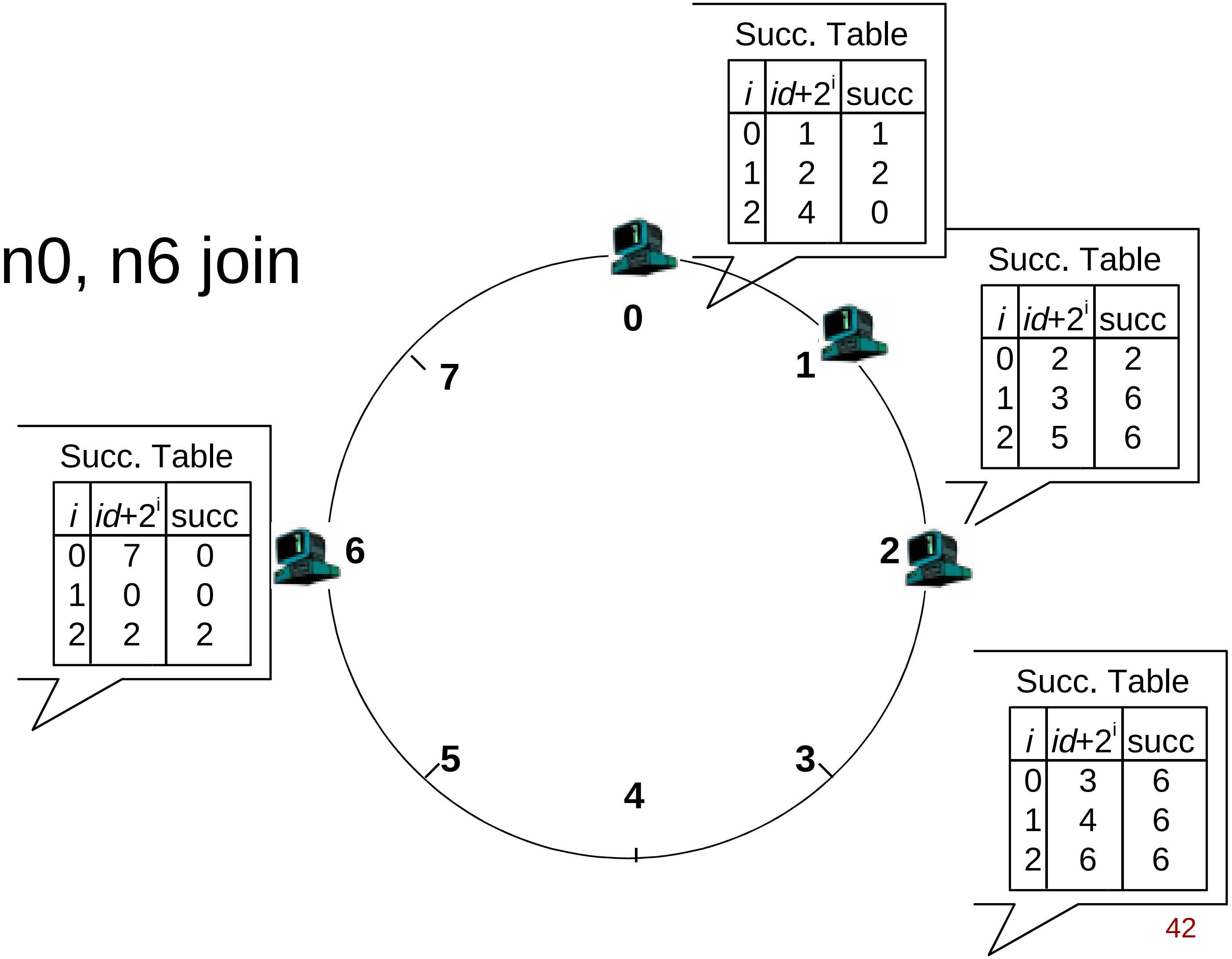
i	*id*+2^i	succ
0	2	2
1	3	1
2	5	1

Succ. Table

i	*id*+2^i	succ
0	3	1
1	4	1
2	6	1

DHT: Chord Join

- Nodes n0, n6 join



Succ. Table

i	*id*+2^i	succ
0	1	1
1	2	2
2	4	0

Succ. Table

i	*id*+2^i	succ
0	2	2
1	3	6
2	5	6

Succ. Table

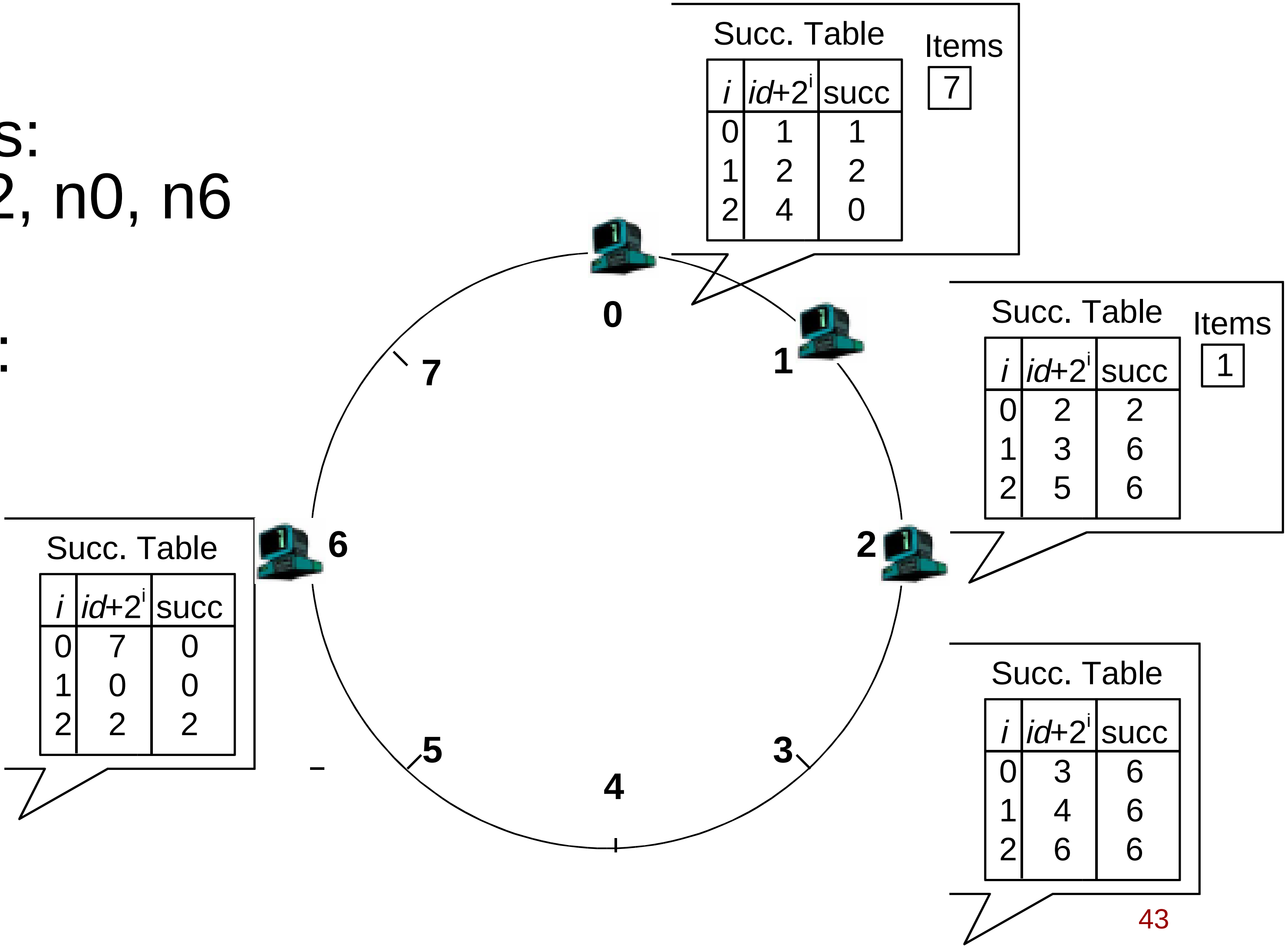
i	*id*+2^i	succ
0	7	0
1	0	0
2	2	2

Succ. Table

i	*id*+2^i	succ
0	3	6
1	4	6
2	6	6

DHT: Chord Join

- Nodes:
  n1, n2, n0, n6
- Items:
  f7, f2



Node 0:

Succ. Table

i	*id*+2^i	succ
0	1	1
1	2	2
2	4	0

Items

7

Node 1:

Succ. Table

i	*id*+2^i	succ
0	2	2
1	3	6
2	5	6

Items

1

Node 2:

Succ. Table

i	*id*+2^i	succ
0	3	6
1	4	6
2	6	6

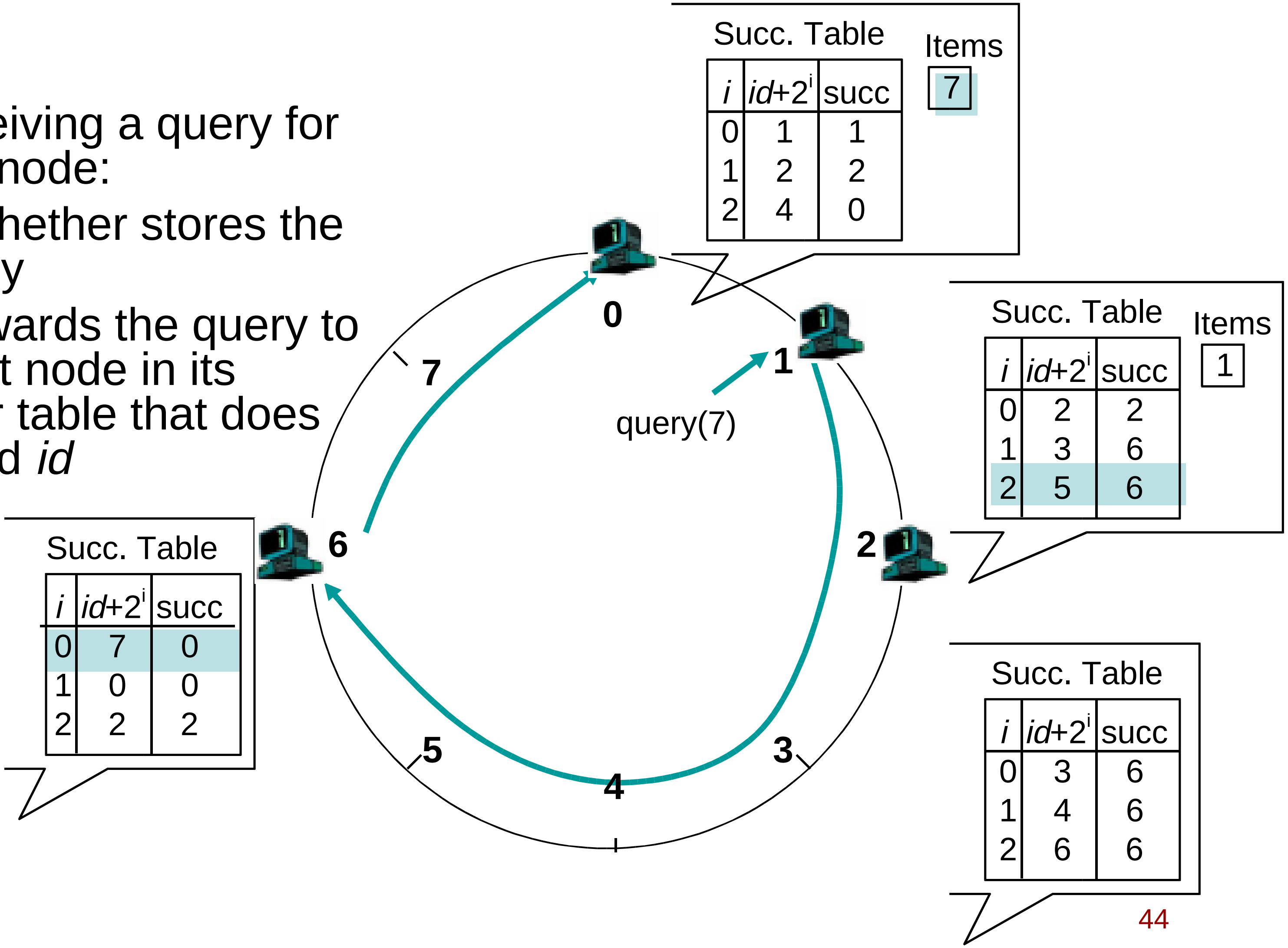
Node 6:

Succ. Table

i	*id*+2^i	succ
0	7	0
1	0	0
2	2	2

DHT: Chord Routing

- Upon receiving a query for item *id*, a node:
- Checks whether stores the item locally
- If not, forwards the query to the largest node in its successor table that does not exceed *id*



Succ. Table

i	*id*+2^i	succ
0	1	1
1	2	2
2	4	0

Items

7

Succ. Table

i	*id*+2^i	succ
0	2	2
1	3	6
2	5	6

Items

1

Succ. Table

i	*id*+2^i	succ
0	3	6
1	4	6
2	6	6

Succ. Table

i	*id*+2^i	succ
0	7	0
1	0	0
2	2	2

DHT: Chord Summary

- Routing table size?
  - Log *N* fingers
- Routing time?
  - Each hop expects to 1/2 the distance to the desired id => expect O(log *N*) hops.

DHT: Discussion

- Pros:
  - – Guaranteed Lookup
  - – O(log *N*) per node state and search scope
- Cons:
  - – Does *not* support keyword search
  - – No one uses them? (only one file sharing app)
  - – Supporting non-exact match search is hard

1) P2P File-sharing: Summary

- Many different styles; remember pros and cons of each
  - centralized, flooding, intelligent flooding, overlay routing
- Lessons learned:
  - Single points of failure are very bad
  - Flooding messages to everyone is bad
  - Underlying network topology is important
  - Not all nodes need be equal
  - Need incentives to discourage freeloading
  - Privacy and security matter
  - Structure can provide theoretical bounds and guarantees

2) P2P File Distribution Systems

(i.e. BitTorrent)

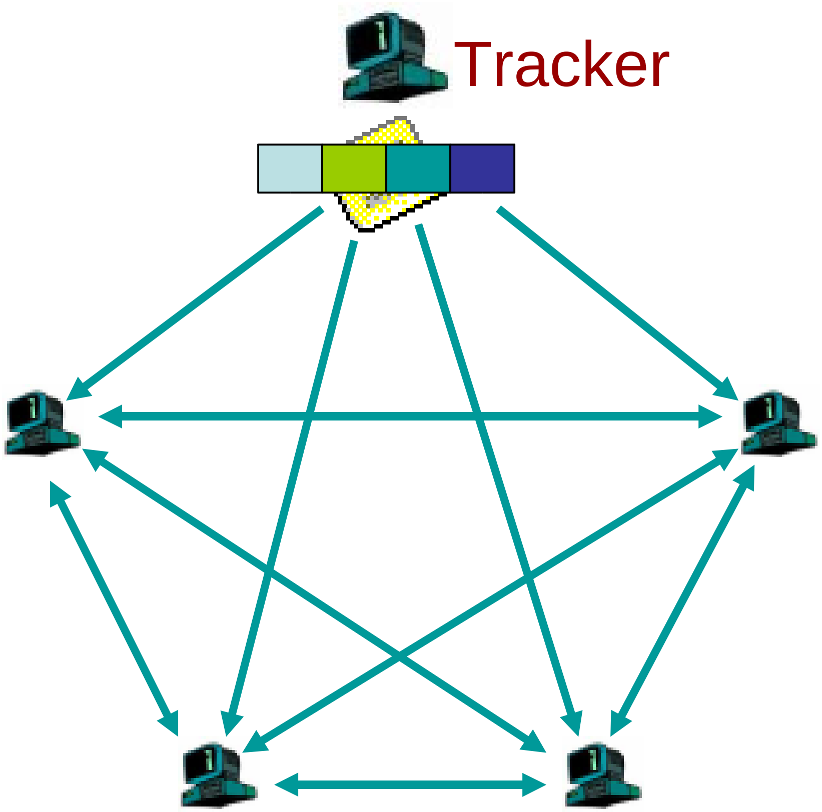
BitTorrent: History

- In 2002, B. Cohen debuted BitTorrent
- Key Motivation:
  - Popularity exhibits temporal locality (Flash Crowds)
  - E.g., Slashdot effect, CNN on 9/11, new movie/game release
- Focused on Efficient *Fetching*, not *Searching*:
  - Distribute the *same* file to all peers
  - Single publisher, multiple downloaders
- Has many “real” publishers:
  - Example: Blizzard Entertainment uses it for World of Warcraft update distribution

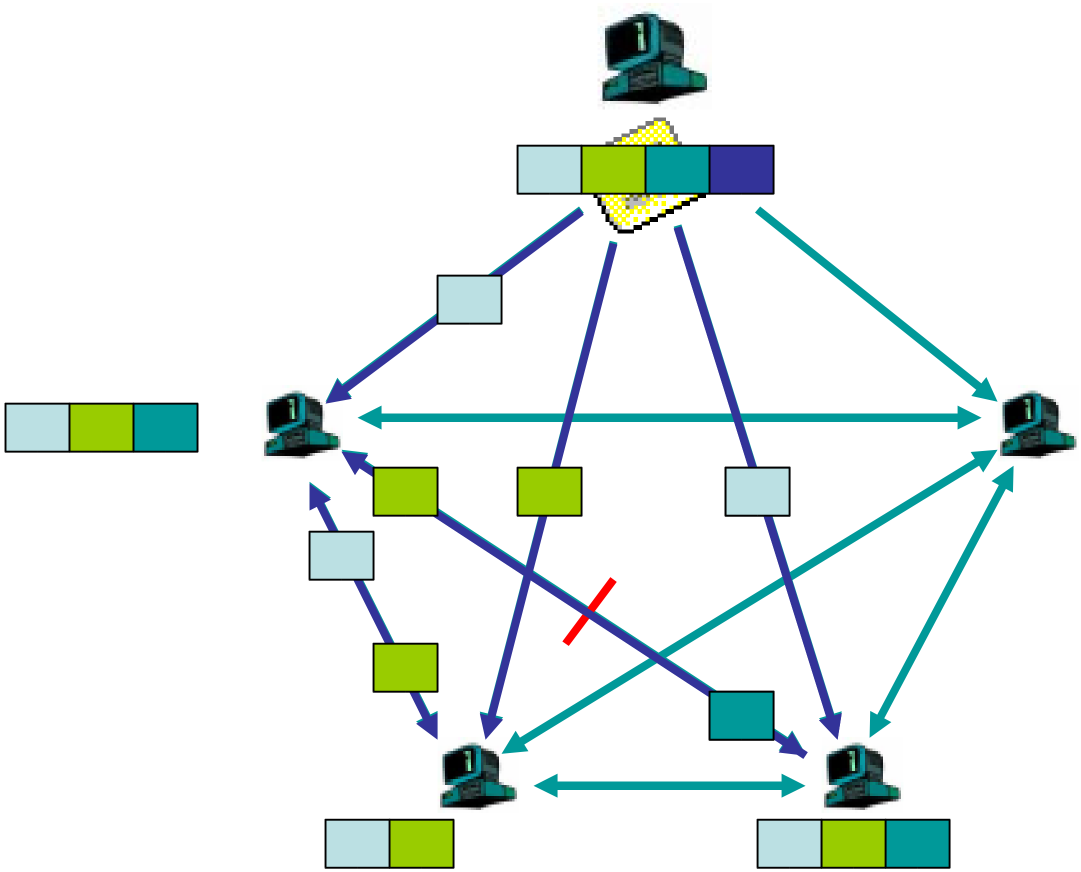
BitTorrent: Overview

- Focused in efficient *fetching*, not searching
- Swarming:
  - **Join**: contact centralized “tracker” server, get a list of peers.
  - **Publish**: Run a tracker server.
  - **Search**: n/a (need to find elsewhere, i.e. Google)
  - **Fetch**: Download chunks of the file from your peers. Upload chunks you have to them.
- Improvements from old Napster-era:
  - Chunk based downloading; few large files
  - Anti-freeloading mechanisms

BitTorrent: Publish/Join



BitTorrent: Fetch



BitTorrent: Summary

- Pros:
  - Works reasonably well in practice
  - Gives peers incentive to share resources; avoids freeloaders
- Cons:
  - No search; only content distribution
  - Central tracker server needed to bootstrap swarm

3) P2P Streaming Systems

(i.e. Overlay Multicast a.k.a. End System Multicast)

Live Broadcast: Pre-Internet

- Tower/Cable/Satellite TV, Radio
- Problems
  - Limitations
    - # of channels
    - Physical reach
  - Cost
    - Content production monopolized by big corps.
    - Content distribution monopolized by big corps.

Live Internet Broadcast: Pre-P2P

- Unicast streaming
  - Small-scale video conferencing
  - Large-scale streaming (AOL Live, etc.)
- Problems
  - Unicast streaming requires lots of bandwidth
    - Example: AOL webcast of Live 8 concert, July 2, 2005: 1500 servers in 90 locations = $$$

Solution...?

Use IP Multicast!...?

Solution: IP Multicast?

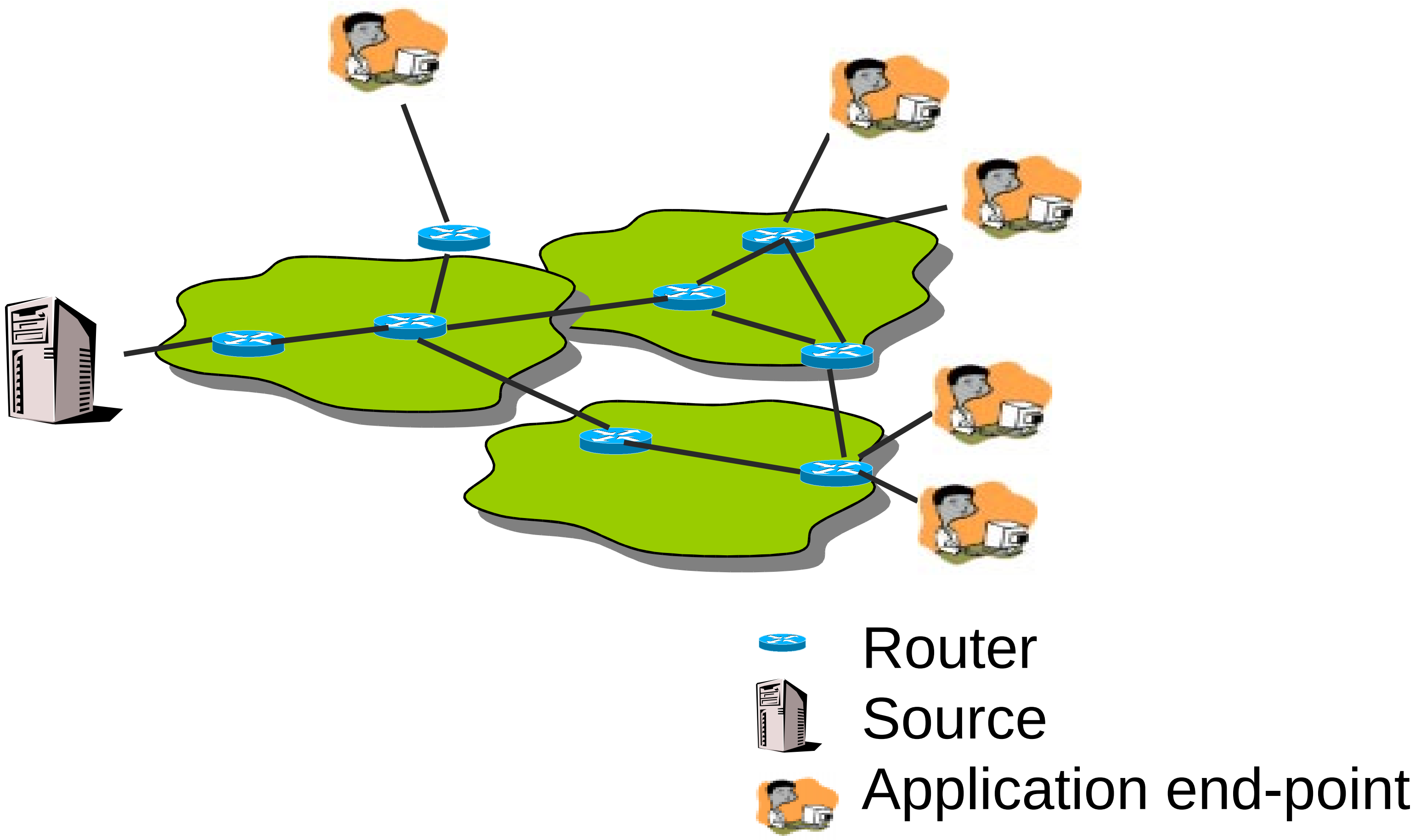
- On a single LAN: GREAT!
  - Can distribute traffic to everyone at once without duplicating streams, set TTL=1, no problem!
- Cross-LAN…Problem □
  - Requires multicast-enabled routers
    - Must allocate resources toward processing multicast packets
    - As a result, *MANY, MANY* computers can’t receive IP multicast packets from outside their LAN

Solution (again!):

Don’t depend on routers…

Distribute the cost among the end users

Internet broadcasting Structure

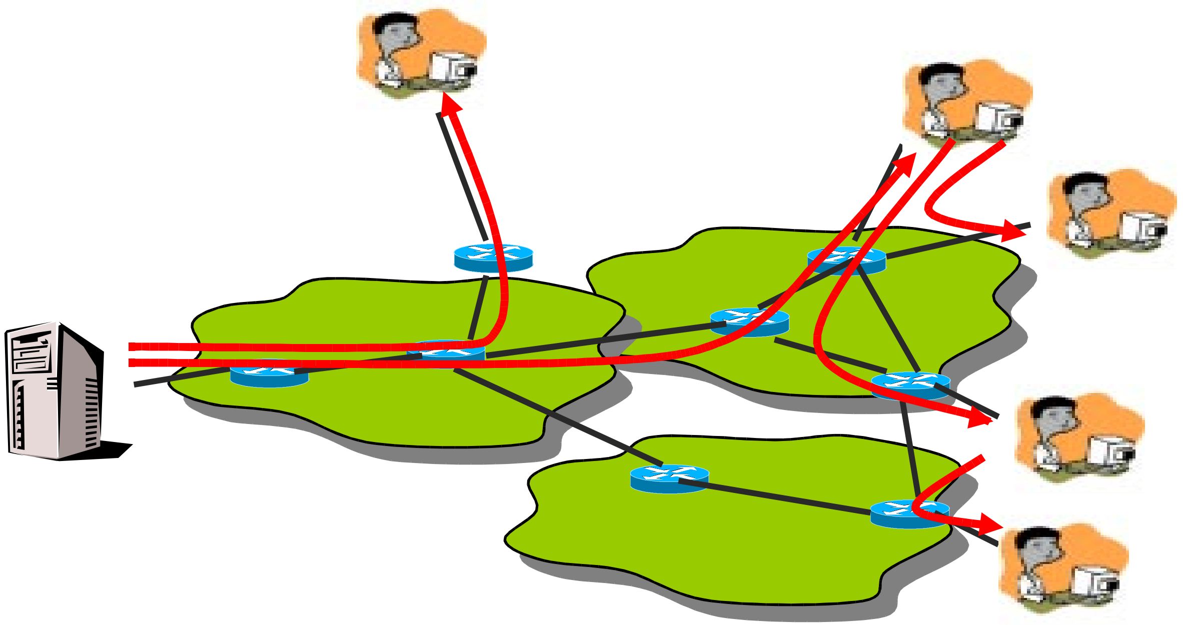


Router

Source

Application end-point

End System Multicast (ESM) (a.k.a Overlay Multicast)



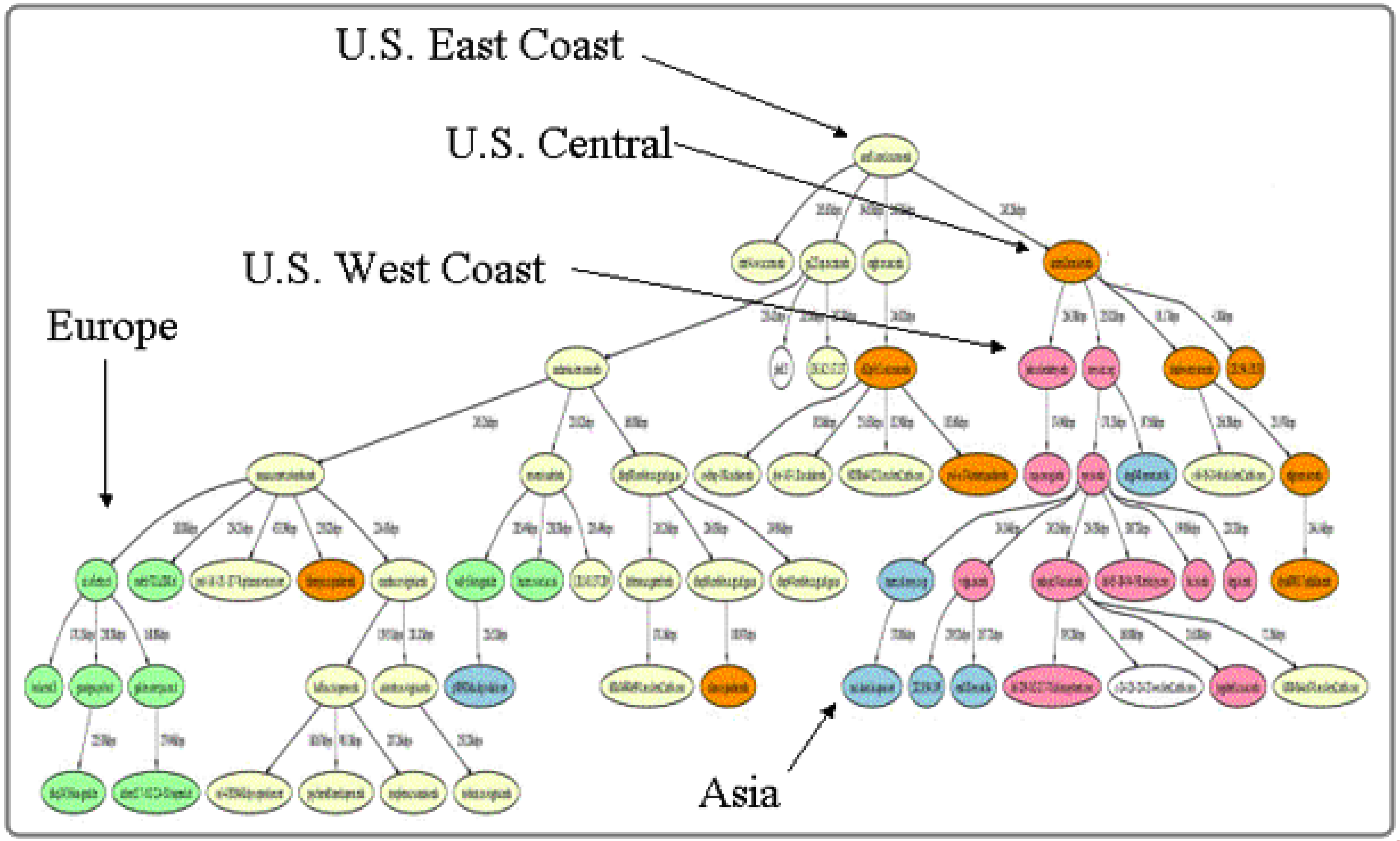
+ Instantly deployable
+ Enables ubiquitous broadcast

Router

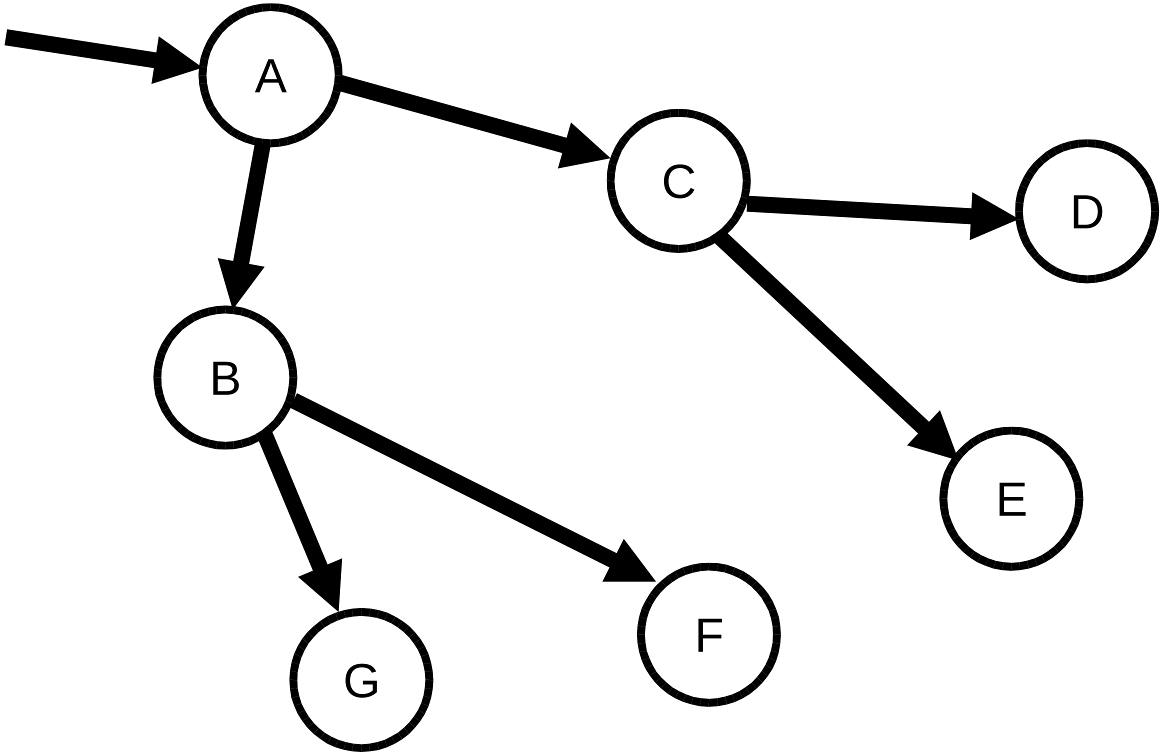
Source

Application end-point

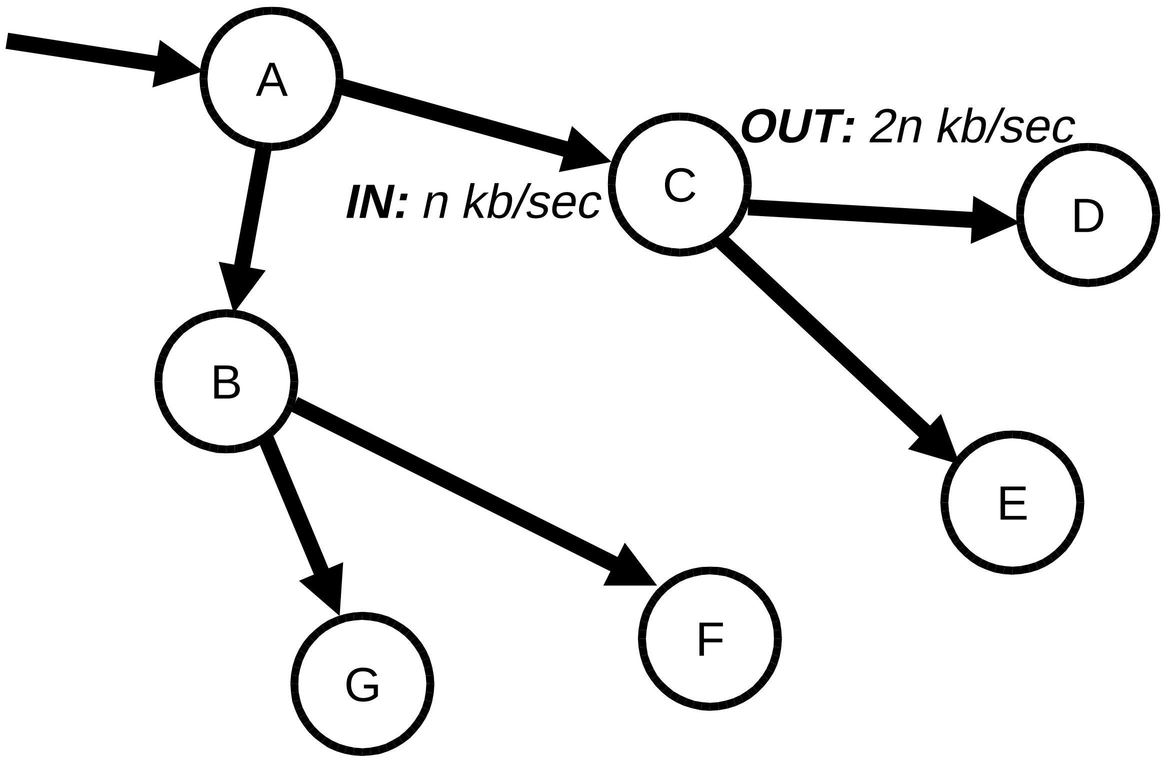
Example ESM Tree http://esm.cs.cmu.edu



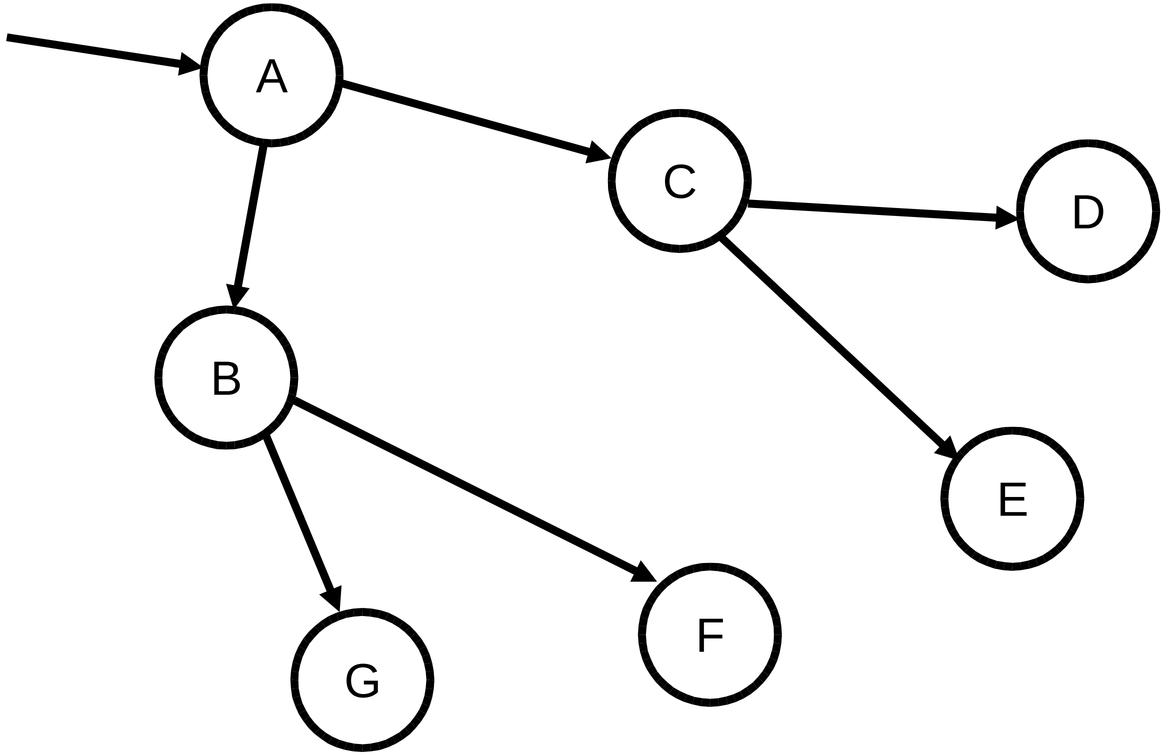
Single Overlay Distribution Tree



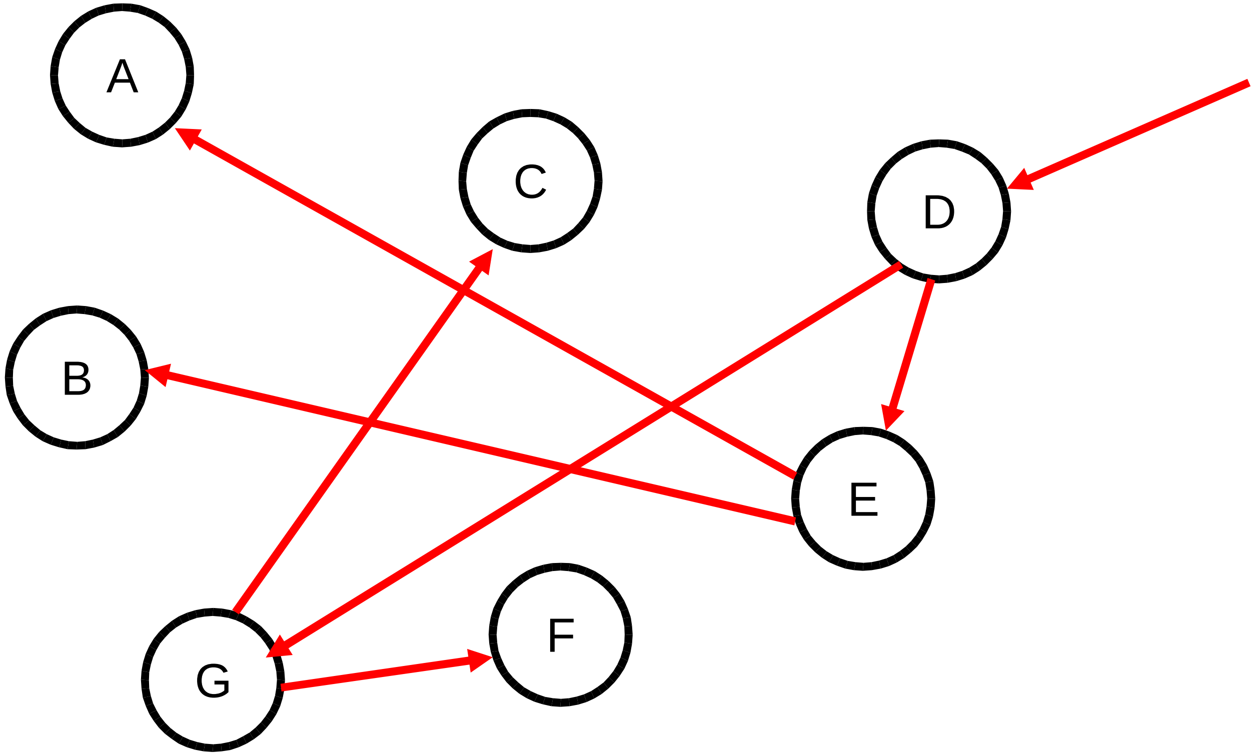
Single Overlay Distribution Tree



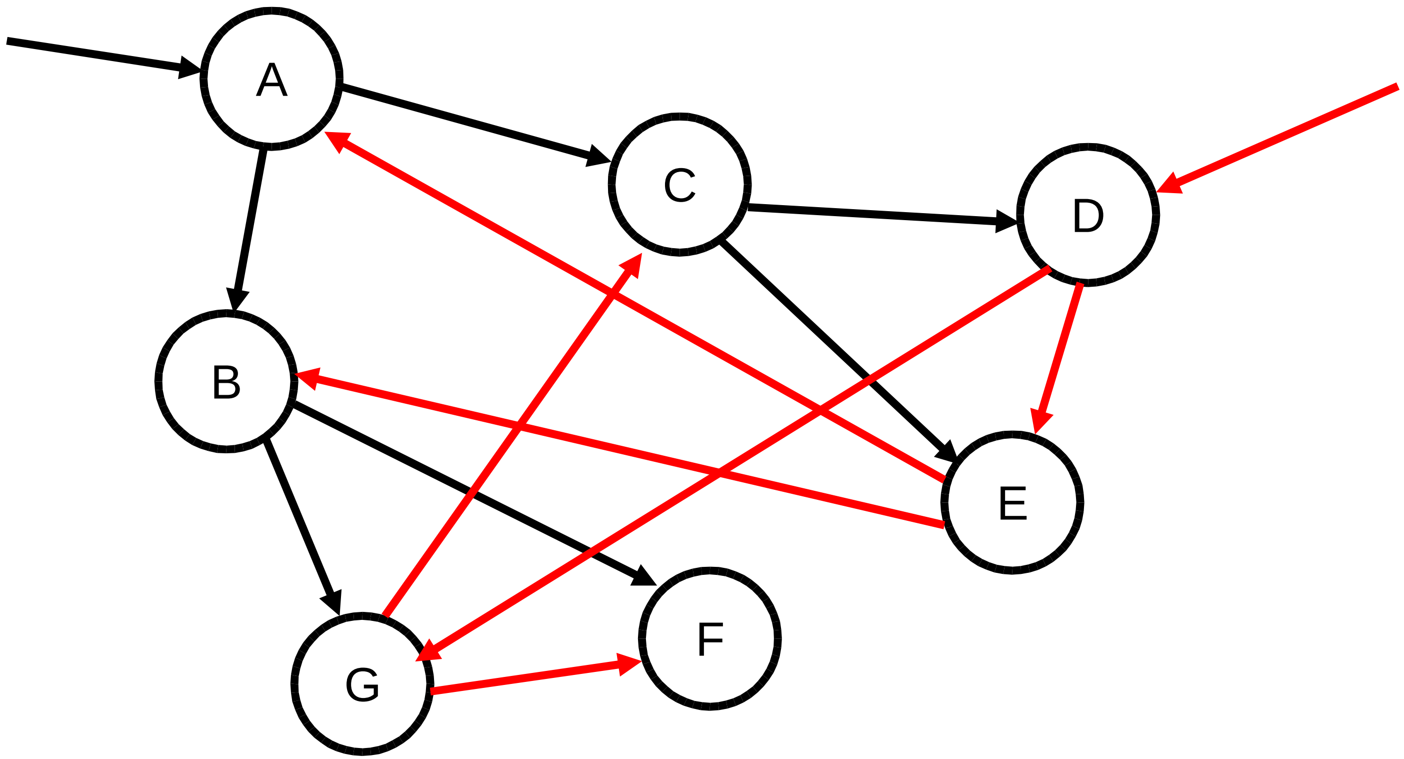
Multiple Overlay Distribution Trees



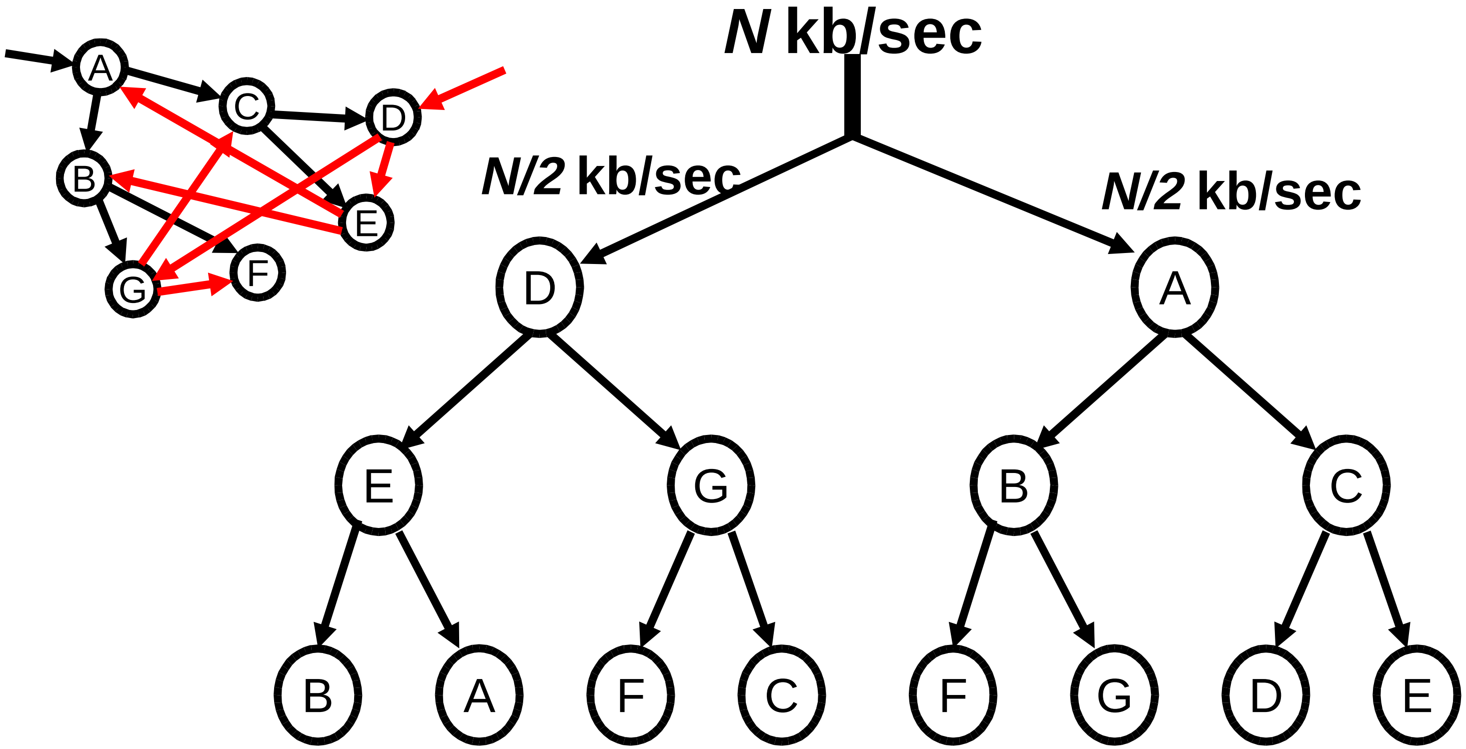
Multiple Overlay Distribution Trees



Multiple Overlay Distribution Trees



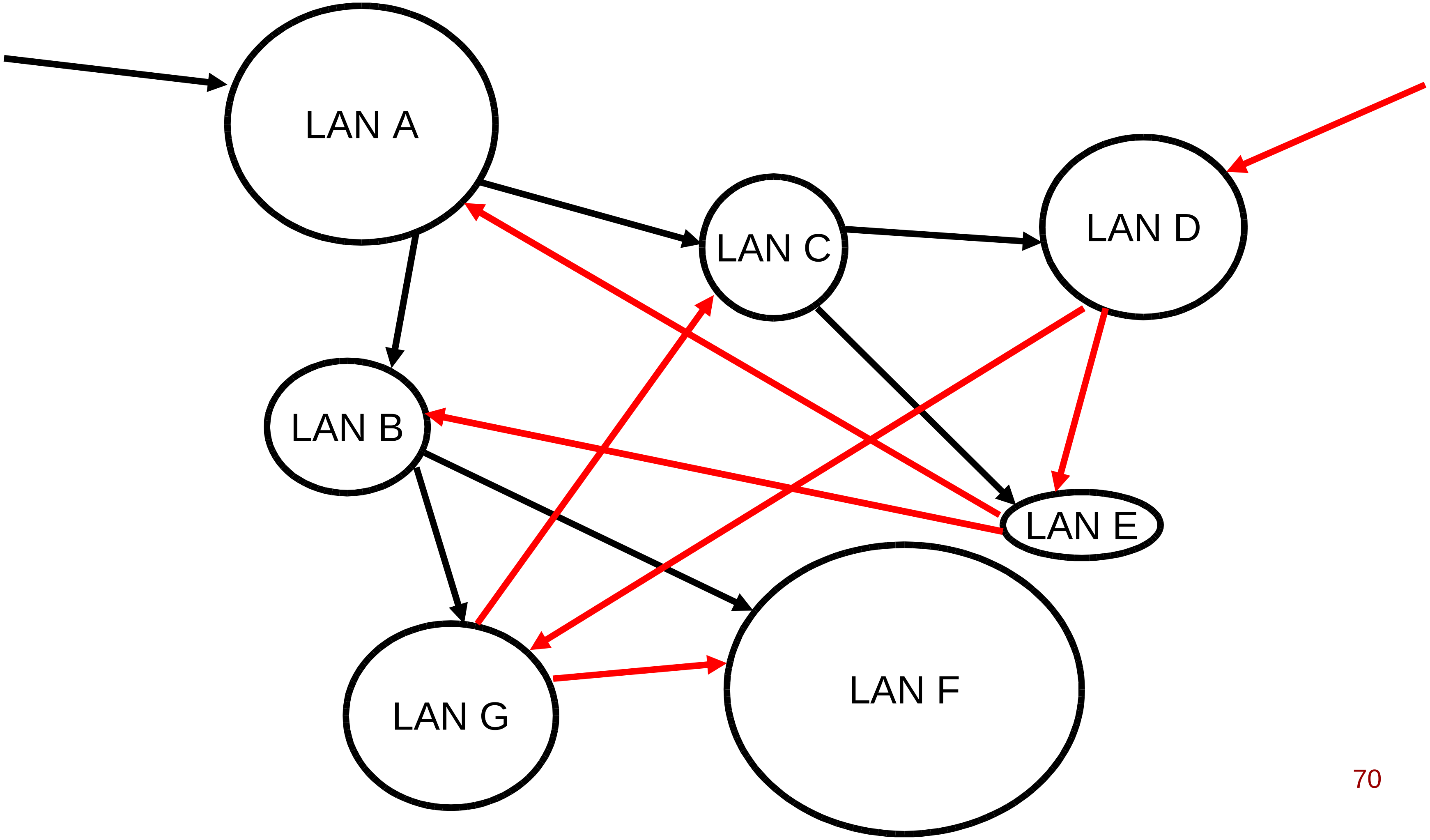
Multiple Overlay Distribution Trees



My Research with ESM

- Can we combine the best parts of multicast with ESM?
  - My solution:
    - Integrate LAN Multicast (i.e. IP Multicast with TTL=1) with ESM
    - Each LAN has 1 or more forwarders from the outside receiving data which gets forwarded on multicast with TTL=1
    - “Nodes” of overlay trees are now LANs instead of individual hosts

ESM + LAN Multicast



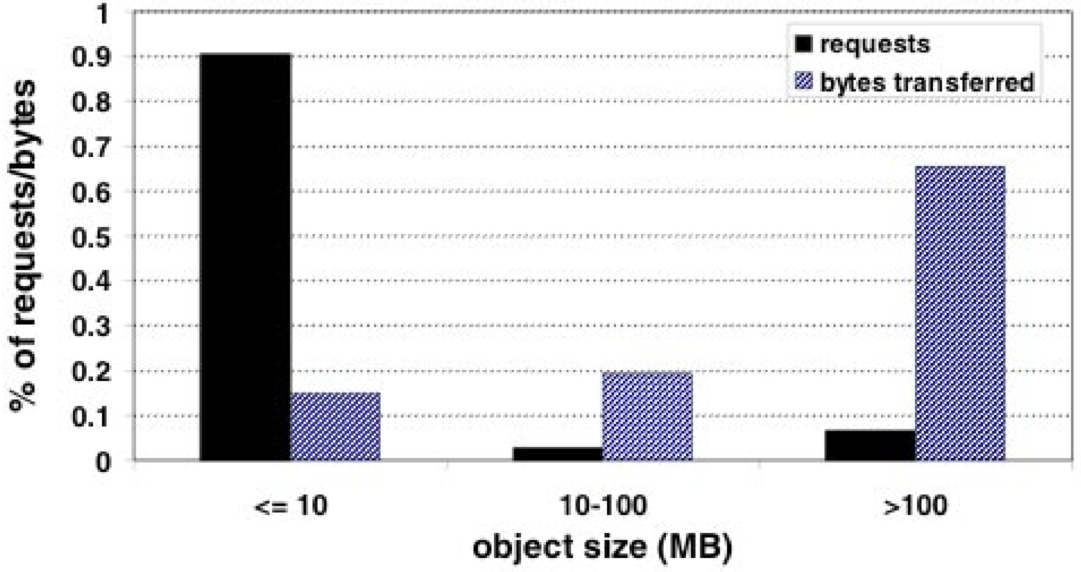
P2P Systems: Summary

- 3 types of P2P systems
  - File-sharing
    - Centralized ((old) Napster), Flooding (Gnutella), Intelligent Flooding (KaZaA) Overlay Routing (DHTs/Chord)
  - File distribution
    - BitTorrent
  - Streaming
    - End System Multicast a.k.a. Overlay Multicast
- Lessons
  - Single points of failure are very bad
  - Underlying network topology is important
  - Not all nodes are equal
  - Can't depend on routers to satisfy all of your networking desires
- Room for growth
  - Privacy & Security
  - Research is ongoing

Extra Slides
(From previous P2P lectures)

KaZaA: Usage Patterns

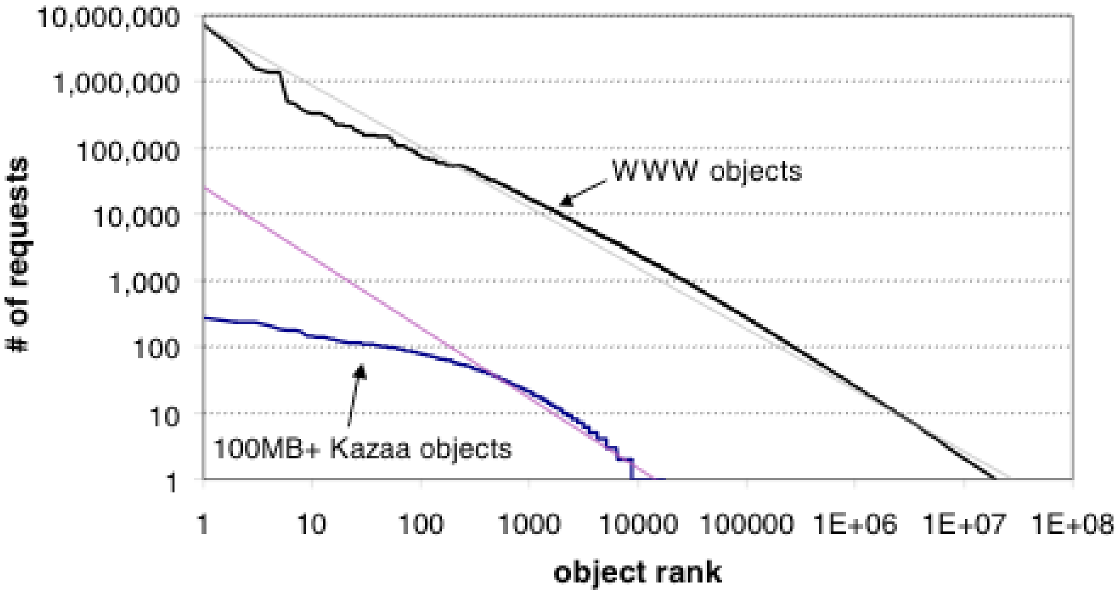
- KaZaA is more than one workload!
  - Many files < 10MB (e.g., Audio Files)
  - Many files > 100MB (e.g., Movies)



from Gummadi *et al.*, *SOSP* 2003

KaZaA: Usage Patterns (2)

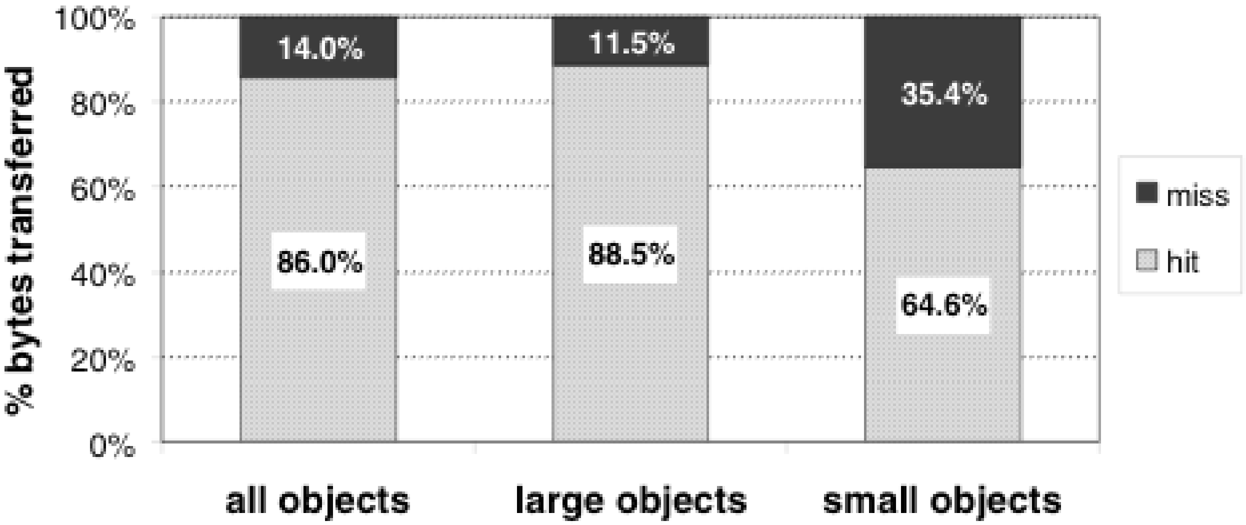
- KaZaA is not Zipf!
  - FileSharing: “Request-once”
  - Web: “Request-repeatedly”



from Gummadi *et al.*, *SOSP* 2003

KaZaA: Usage Patterns (3)

- What we saw:
  - A few big files consume most of the bandwidth
  - Many files are fetched once per client but still very popular
- Solution?
  - Caching!



from Gummadi *et al.*, *SOSP* 2003

Freenet: History

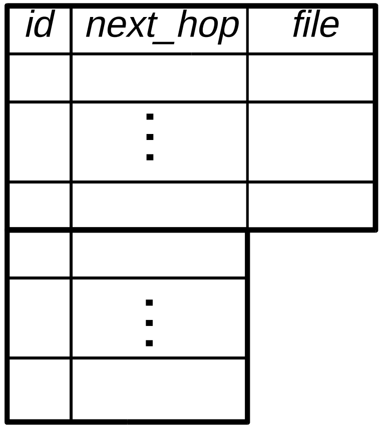
- In 1999, I. Clarke started the Freenet project
- Basic Idea:
  - Employ Internet-like routing on the overlay network to publish and locate files
- Addition goals:
  - Provide anonymity and security
  - Make censorship difficult

Freenet: Overview

- Routed Queries:
  - **Join**: on startup, client contacts a few other nodes it knows about; gets a unique *node id*
  - **Publish**: route file contents toward the *file id*. File is stored at node with *id* closest to *file id*
  - **Search**: route query for *file id* toward the closest *node id*
  - **Fetch**: when query reaches a node containing *file id*, it returns the file to the sender

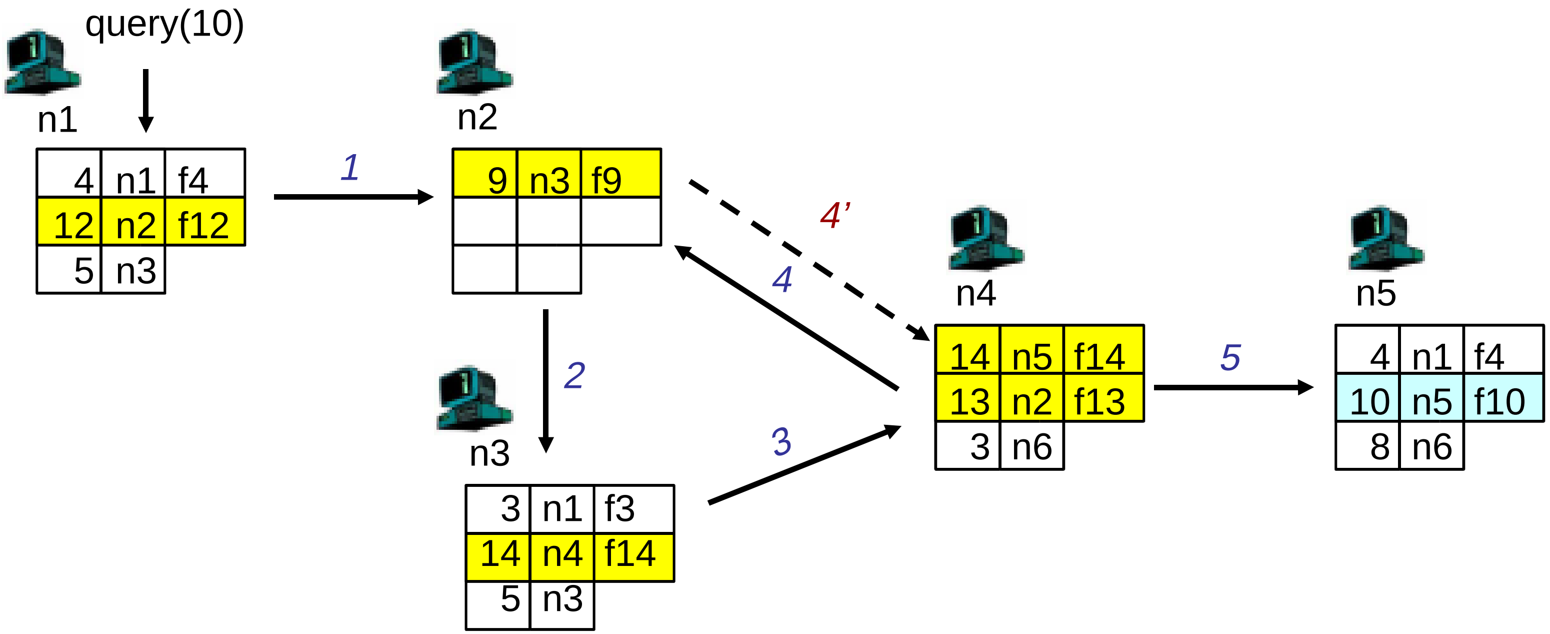
Freenet: Routing Tables

- ***id*** – file identifier (e.g., hash of file)
- ***next_hop*** – another node that stores the file id
- ***file*** – file identified by *id* being stored on the local node



- Forwarding of query for file *id*
  - If file *id* stored locally, then stop
    - Forward data back to upstream requestor
  - If not, search for the “closest” *id* in the table, and forward the message to the corresponding *next_hop*
  - If data is not found, failure is reported back
    - Requestor then tries next closest match in routing table

Freenet: Routing



n1

4	n1	f4
12	n2	f12
5	n3	

n2

9	n3	f9

n3

3	n1	f3
14	n4	f14
5	n3	

n4

14	n5	f14
13	n2	f13
3	n6	

n5

4	n1	f4
10	n5	f10
8	n6	

Freenet: Routing Properties

- “Close” file ids tend to be stored on the same node
  - Why? Publications of similar file ids route toward the same place
- Network tend to be a “small world”
  - Small number of nodes have large number of neighbors (i.e., ~ “six-degrees of separation”)
- Consequence:
  - Most queries only traverse a small number of hops to find the file

Freenet: Anonymity & Security

- Anonymity
  - Randomly modify source of packet as it traverses the network
  - Can use “mix-nets” or onion-routing
- Security & Censorship resistance
  - No constraints on how to choose *ids* for files => easy to have to files collide, creating “denial of service” (censorship)
  - Solution: have a *id* type that requires a private key signature that is verified when updating the file
  - Cache file on the reverse path of queries/publications => attempt to “replace” file with bogus data will just cause the file to be replicated more!

Freenet: Discussion

- Pros:
  - Intelligent routing makes queries relatively short
  - Search scope small (only nodes along search path involved); no flooding
  - Anonymity properties may give you “plausible deniability”
- Cons:
  - Still no provable guarantees!
  - Anonymity features make it hard to measure, debug