Ubiquitous Computing

Peer-to-peer systems and overlays

Marco Aiello

m.aiello@rug.nl

K. 568 Bernoulliborg

Summary

- A bit of history
- Overlay networks
- Structured P2P systems
  - Pastry, Tapestry, Chord
  - Skip-graphs
- Unstructured P2P systems
  - Freenet, Gnutella, KaZaA, eDonkey
  - End-host based multicast: BitTorrent

Structured Peer-to-Peer Systems

Marco Aiello - RuG

With material (1) Chapter 10 Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edition 4, (2) Chapter 25 Birman: Reliable Distributed Systems (3) Jari Mäntylä, Scalability of Peer-to-Peer Systems (4) Aspens Shah SKip Graphs

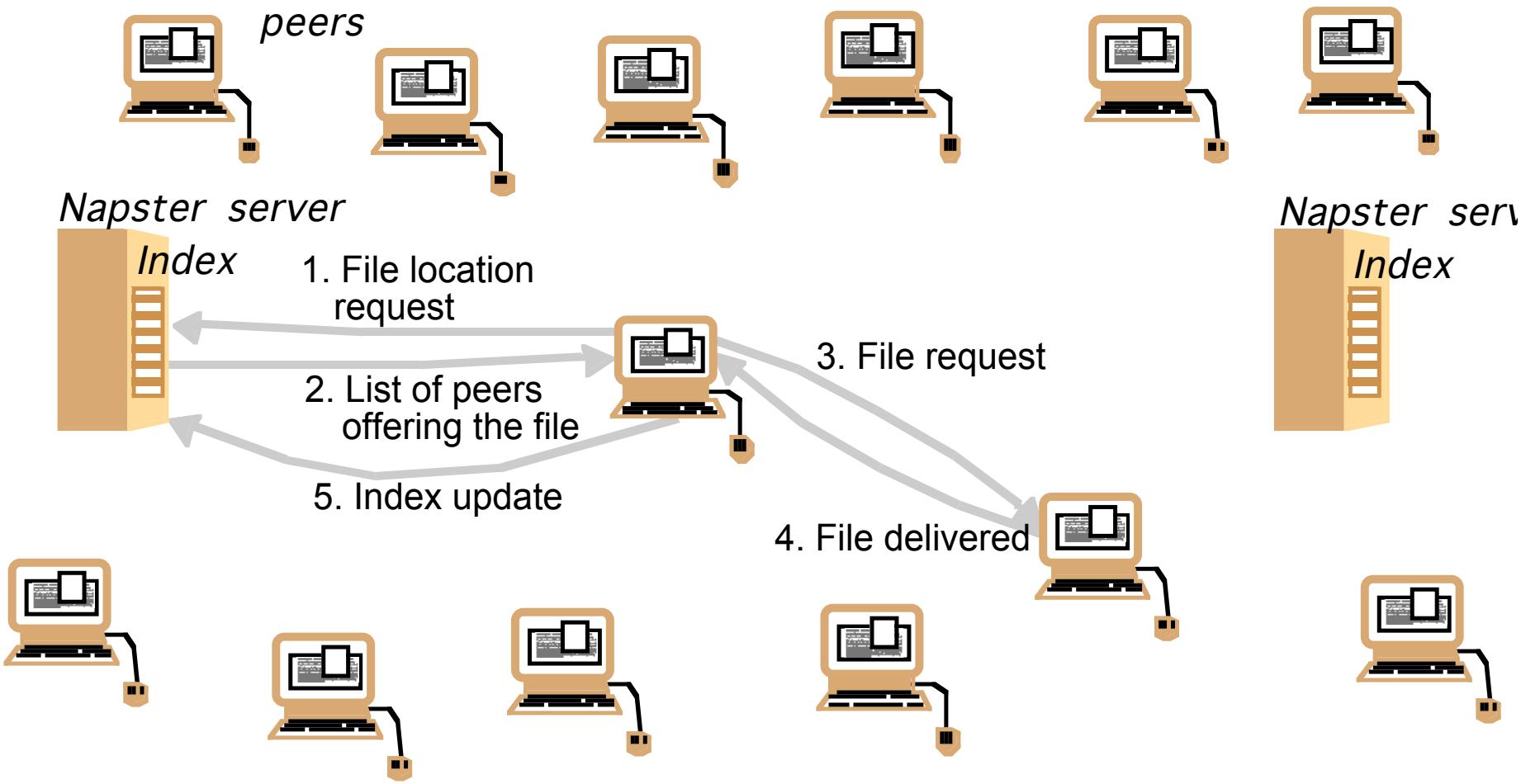
Peer to peer

- support useful distributed services without a centralized control
- typically all nodes share some resources (e.g., computational power or files)
- in the topology all nodes are equal (unlike the client server architecture)
- requires additional application-level routing

Some early P2P systems

- Seti@Home project
  - analysis of radiotelescopic data
  - by august 2002, 3.91 M computers, 27.36 teraflops in one year.
- Napster
  - file system sharing
  - centralized indexes, but shared files
  - no anonymity is guaranteed to users
  - no need for updates to the files (music files are static)
  - availability of resources is not critical
  - Napster server maintains list of regional servers
  - Easy to abuse the cooperative model

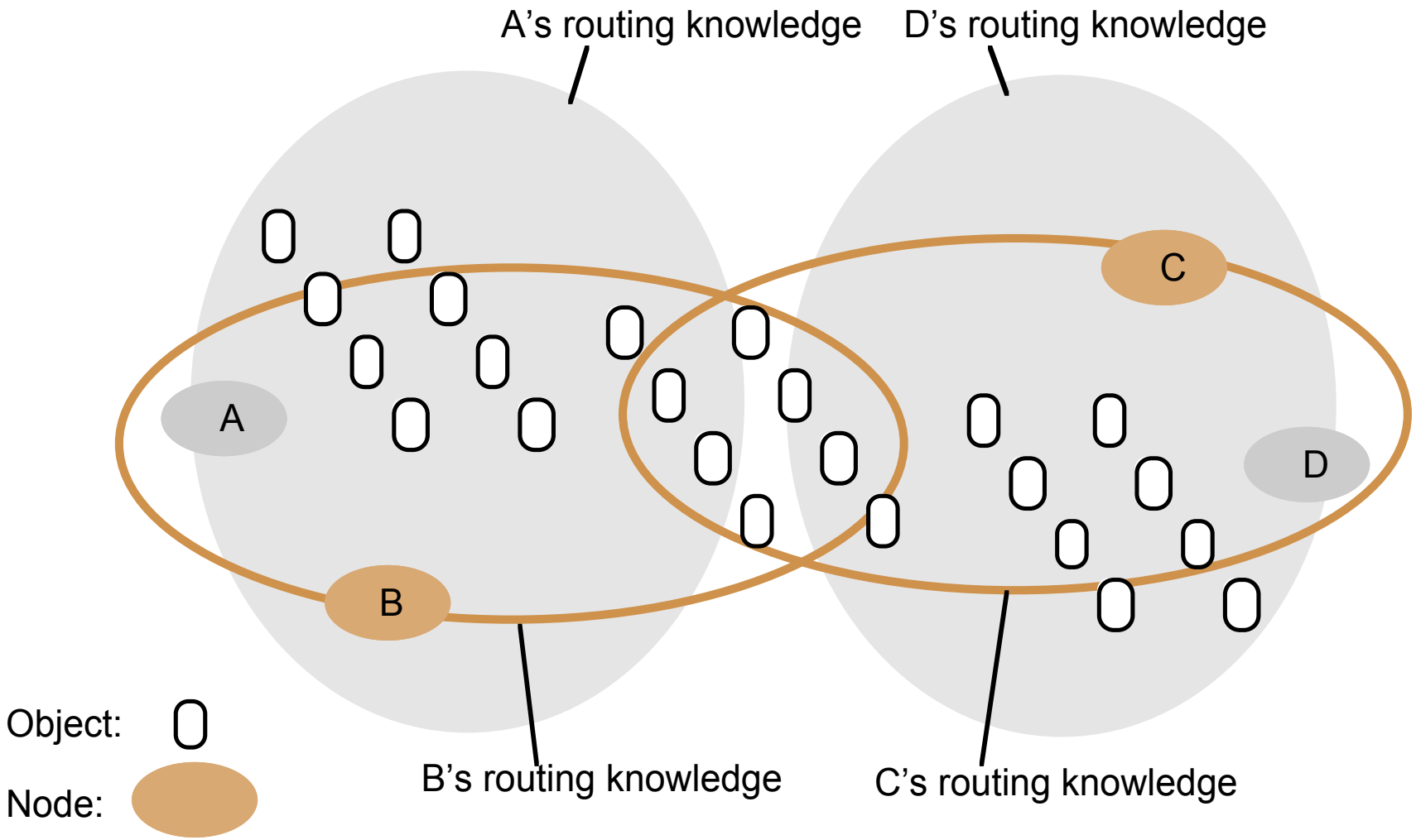
Napster: peer-to-peer file sharing with a centralized, replicated index



Requirements:

- Services implemented across multiple nodes
- Global scalability
- Load balancing (unsupervised)
- Optimization for local interactions between neighboring peers
- Accommodating to highly dynamic host availability
- Security of data in an environment with heterogeneous trust
- Anonymity, deniability and resistance to censorship

Distribution of information in a routing overlay



Routing overlays

- A *routing overlay* is a distributed algorithm to locate nodes and objects
- It is a logical network built on top of another (physical) one.

- Tasks:
  - route a message to an object given its globally unique identifier (e.g., to one of the replicas)
  - management of appearance of new objects on nodes
  - management of removal of objects from nodes
  - management of availability of (new) nodes

Globally Unique IDentifiers

- GUID need to be computed by nodes when making available a new resource
- A hash function used to generate the GUID
  - uniqueness guaranteed by searching the network for the generated GUID
- Distributed Hash Table to find objects
  - (i.e., overlay routing = computing DHT value for an object)
  - a data item with GUID *n* is stored at the node whose GUID is numerically closest to *n* and the *r* hosts with GUIDs closest to *n*
  - Pastry is an example
- Distributed Object Location and Routing (DOLR)
  - explicit mapping between objects and their location(s)
  - Tapestry is an example

Basic programming interface for a distributed hash table (DHT) as implemented by the PAST API over Pastry

put(GUID, data)

The *data* is stored in replicas at all nodes responsible for the object identified by *GUID*

remove(GUID)

Deletes all references to *GUID* and the associated data

value = get(GUID)

The data associated with *GUID* is retrieved from one of the nodes responsible it

Basic programming interface for distributed object location and routing (DOLR) as implemented by Tapestry

publish(GUID)

GUID can be computed from the object (or some part of it, e.g. its name). This function makes the node performing a *publish* operation the host for the object corresponding to *GUID*.

unpublish(GUID)

Makes the object corresponding to *GUID* inaccessible.

sendToObj(msg, GUID, [n])

Following the object-oriented paradigm, an invocation message is sent to an object in order to access it. This might be a request to open a TCP connection for data transfer or to return a message containing all or part of the object's state. The final optional parameter *[n]*, if present, requests the delivery of the same message to n replicas of the object.

Routing

- *Prefix routing:* look at an increasing number of bits for each hop (used in Pastry and Tapestry)
- *Difference routing:* difference between the destination GUID and the neighbouring nodes (Chord)
- *Distance routing:* based on the euclidean distance among nodes in a hyperspace of nodes (CAN)
- *XOR routing:* based on the XOR of pairs of GUIDs (Kademilia)

The last two establish a symmetric relation among nodes

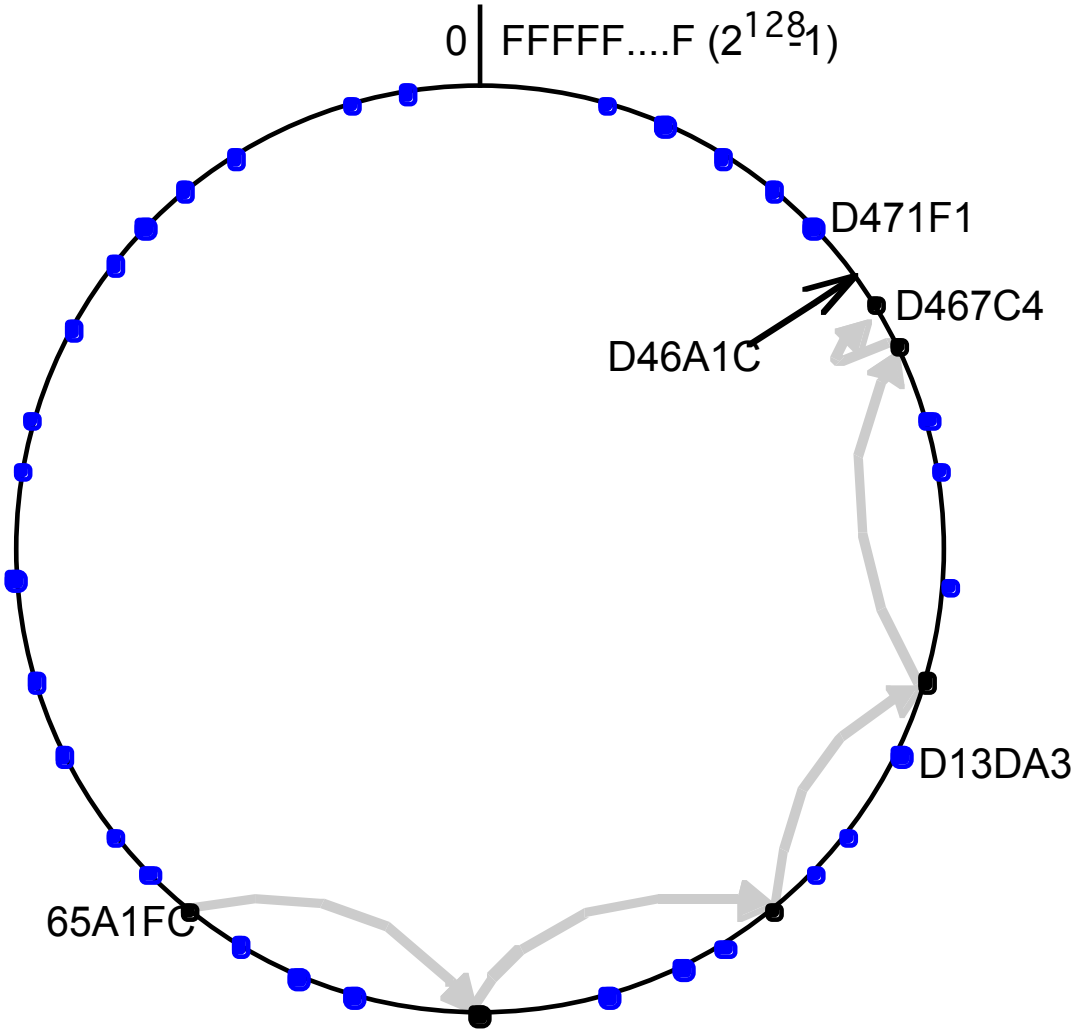
GUIDs are not human readable. BitTorrent stores the mappings names on Web pages.

PASTRY

- A message routing infrastructure
- Prefix routing approach
- 128 bit GUIDs to nodes and objects computed as a secure hash function of the public key
- clashes are extremely unlikely and detected upon GUID creation
- Messages usually transported via UDP
- Message routed to the closest active node to the GUID (the node itself, if active)

Circular routing alone is correct but inefficient

Based on Rowstron and Druschel [2001]



The dots depict live nodes. The space is considered as circular: node 0 is adjacent to node (2128-1). The diagram illustrates the routing of a message from node 65A1FC to D46A1C using leaf set information alone, assuming leaf sets of size 8 (l = 4). This is a degenerate type of routing that would scale very poorly; it is not used in practice.

PASTRY routing

- Basic algorithm: each node has a set of leaf nodes of size 2 *l*
- The set of addresses is considered circular
- Each node has the routing information of *l* nodes preceding it and *l* following it
- What is the complexity of delivering a message when there are *N nodes*?

$$N/(2\ l)$$

- The actual routing makes use of routing tables

Pastry's routing algorithm

To handle a message M addressed to a node D (where $R[p,i]$ is the element at column i, row p of the routing table):

1. If ($L_{-l} < D < L_l$) { // the destination is within the leaf set or is the current node.
2. Forward M to the element L_i of the leaf set with GUID closest to D or the current node A.
3. } else { // use the routing table to despatch M to a node with a closer GUID
4. find p, the length of the longest common prefix of D and A. and i, the $(p+1)^{th}$ hexadecimal digit of D.
5. If ($R[p,i] \neq null$) forward M to $R[p,i]$ // route M to a node with a longer common prefix.
6. else { // there is no entry in the routing table
7. Forward M to any node in L or R with a common prefix of length i, but a GUID that is numerically closer.

}

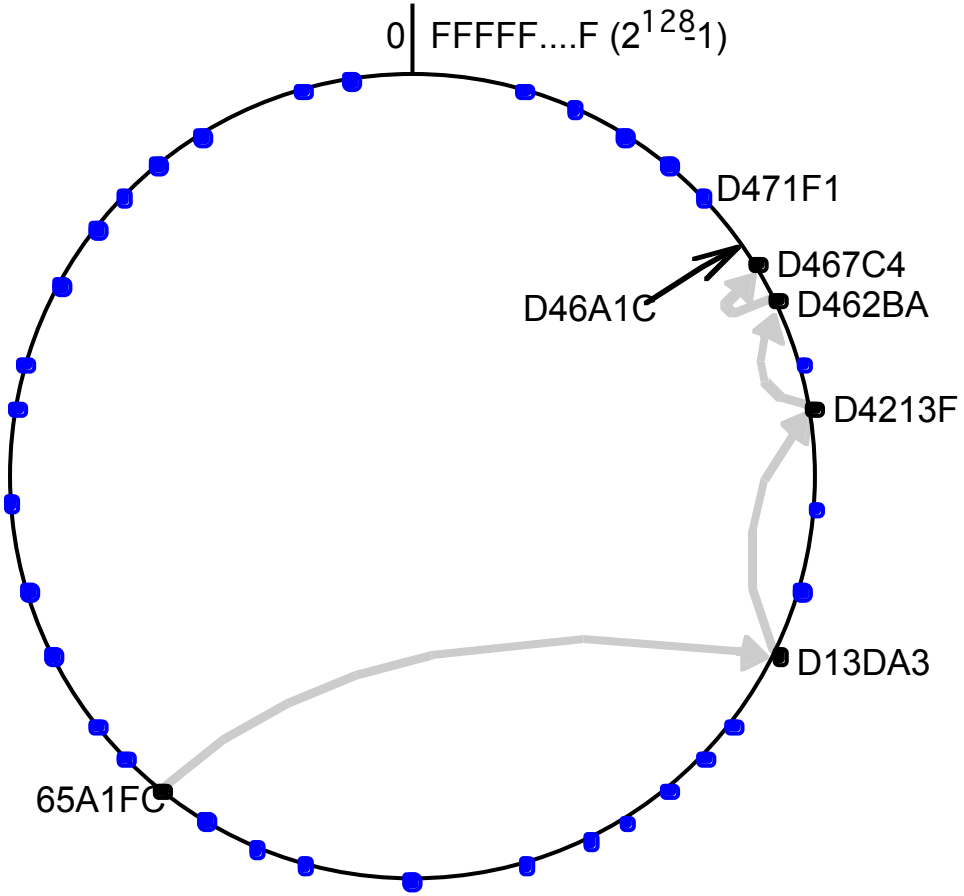
}

First four rows of a Pastry routing table

p =	*GUID prefixes and corresponding nodehandles n*															
0	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
	n	*n*	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*
1	*60*	*61*	*62*	*63*	*64*	*65*	*66*	*67*	*68*	*69*	*6A*	*6B*	*6C*	*6F*	*6E*	*6F*
	n	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*
2	*650*	*651*	*652*	*653*	*654*	*655*	*656*	*657*	*658*	*659*	*65A*	*65B*	*65C*	*65D*	*65E*	*65F*
	n	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*
3	*65A0*	*65A1*	*65A2*	*65A3*	*65A4*	*65A5*	*65A6*	*65A7*	*65A8*	*65A9*	*65AA*	*65AB*	*65AC*	*65AD*	*65AE*	*65AF*
	n		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*

The routing table is located at a node whose GUID begins 65A1. Digits are in hexadecimal. The *n* s represent [GUID, IP address] pairs specifying the next hop to be taken by messages addressed to GUIDs that match each given prefix. Grey-shaded entries indicate that the prefix matches the current GUID up to the given value of *p*: the next row down or the leaf set should be examined to find a route. Although there are a maximum of 128 rows in the table, only $\log_{16} N$ rows will be populated on average in a network with *N* active nodes.

Pastry routing example Based on Rowstron and Druschel [2001]



- What is the complexity using routing tables? $log_{16}(N)$

PASTRY dynamic joining

- New nodes use a joining protocol
- The new node computes a new GUIDs
- Then it contacts nearby nodes (topologically near, computed with round trip calculation)
- Nodes reply with relevant parts of their routing tables
- On initialization, the leaf set is put to the value of the node whose GUID is closest to its own
- Failed node are replaced when discovered by contacting the closest GUID live node. A node is considered failed when it does not communicated

PASTRY

- The routing tables are highly redundant
- Choice among possible routing nodes is made using the Proximity Neighbour Selection algorithm
- The routing is not optimal, but it has been measured to be only 30-50% worse than optimal

- All nodes send heartbeat messages
- at-least-once delivery semantics
- a small randomness is introduced in the routing algorithm to compensate for malicious nodes

MSPastry

- An enhanced version of Pastry tailored at dependability
- Use of acknowledgements
- When no heartbeat from a neighbouring node, replacement procedure is initiated with notification to all nodes in the leaf set
- A Gossip protocol to update the routing tables is executed every 20 minutes

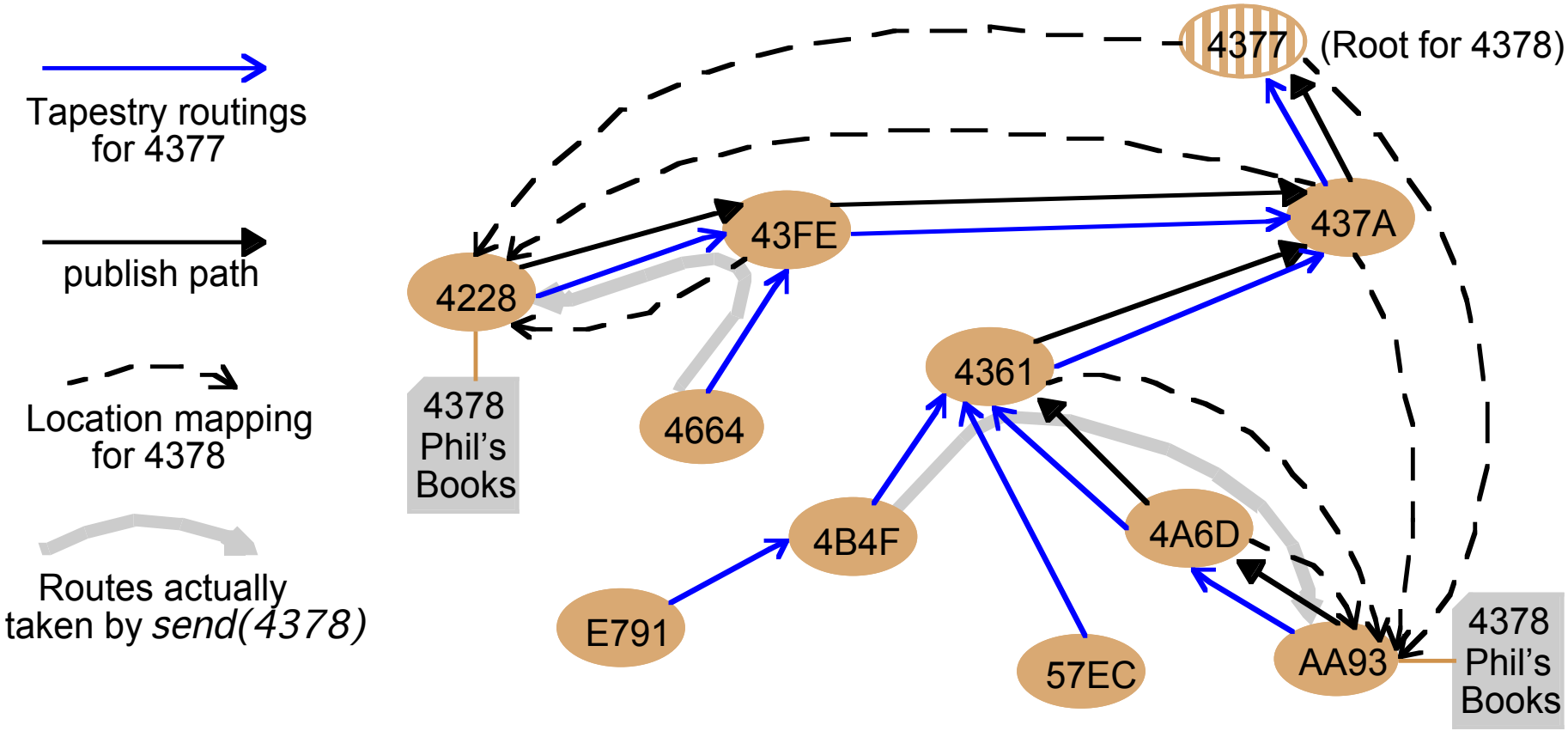
MSPastry evaluation

- *Dependability:* with perfect transport only 0,0015% message loss rate, with 5% transport loss, only 0,0033% loss and 0,0016% delivered to the wrong destination
- *Performance:* measured as relative delay penalty, i.e., the ration between pastry delivery and direct IP delivery. Values between 1.8 and 2.2.
- *Overheads:* of control traffic is less than 2 messages per minute per node (without counting initial setup overhead)

Tapestry

- Similar to Pastry, but uses a Distributed Object Locator and Routing to conceal the location
- Replicated resources are published with the same GUIDs
- Replicas can be placed close to where used
- 160-bit identifiers for nodes and objects
- Each object has a unique root node which is the one with GUID closest to the GUID of the object
- Nodes holding replicas of G, constantly invoke `publish(G)` to get updates of G

Tapestry routing [Zhao et al. 2004]

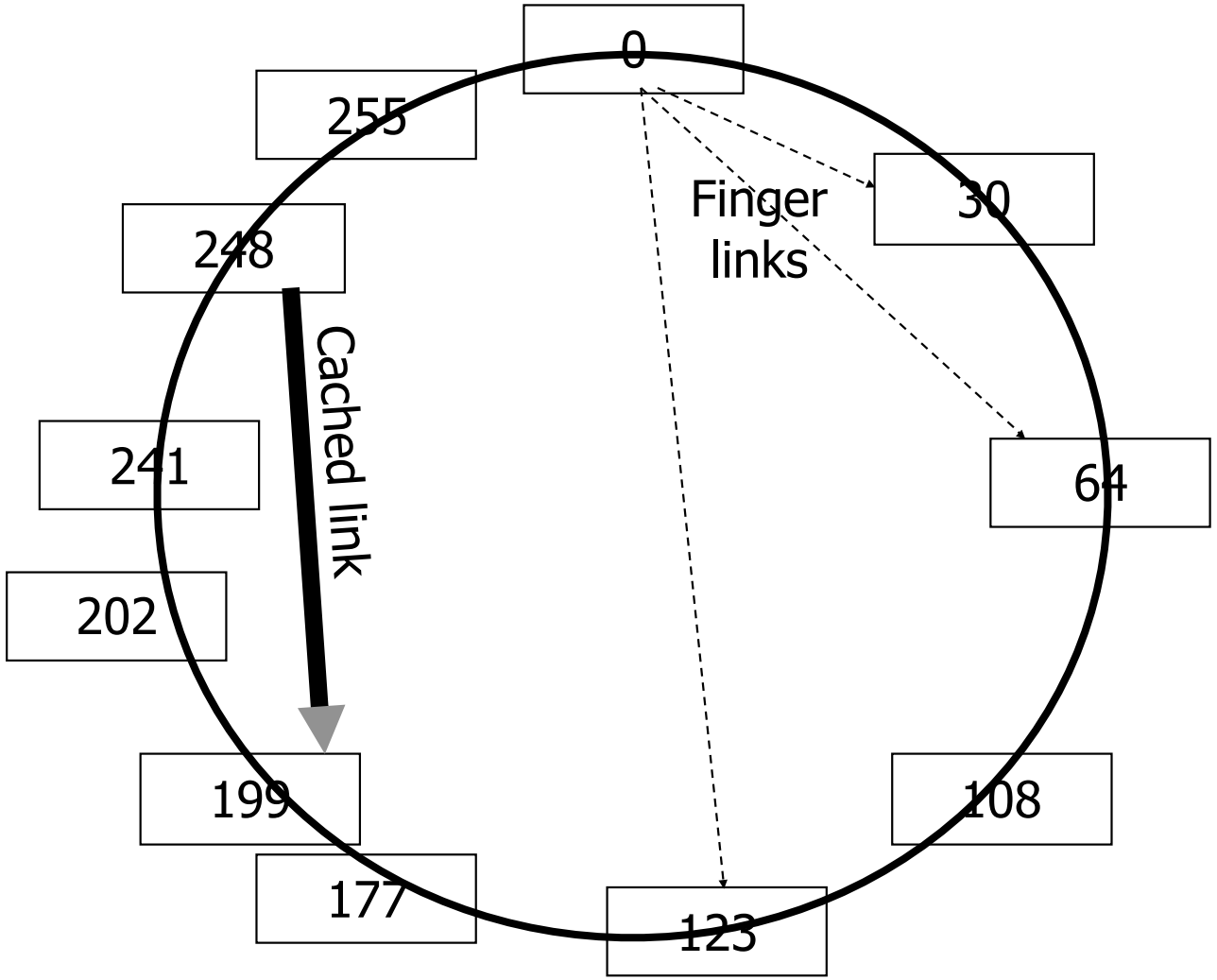


Replicas of the file *Phil's Books* (G=4378) are hosted at nodes 4228 and AA93. Node 4377 is the root node for object 4378. The Tapestry routings shown are some of the entries in routing tables. The publish paths show routes followed by the publish messages laying down cached location mappings for object 4378. The location mappings are subsequently used to route messages sent to 4378.

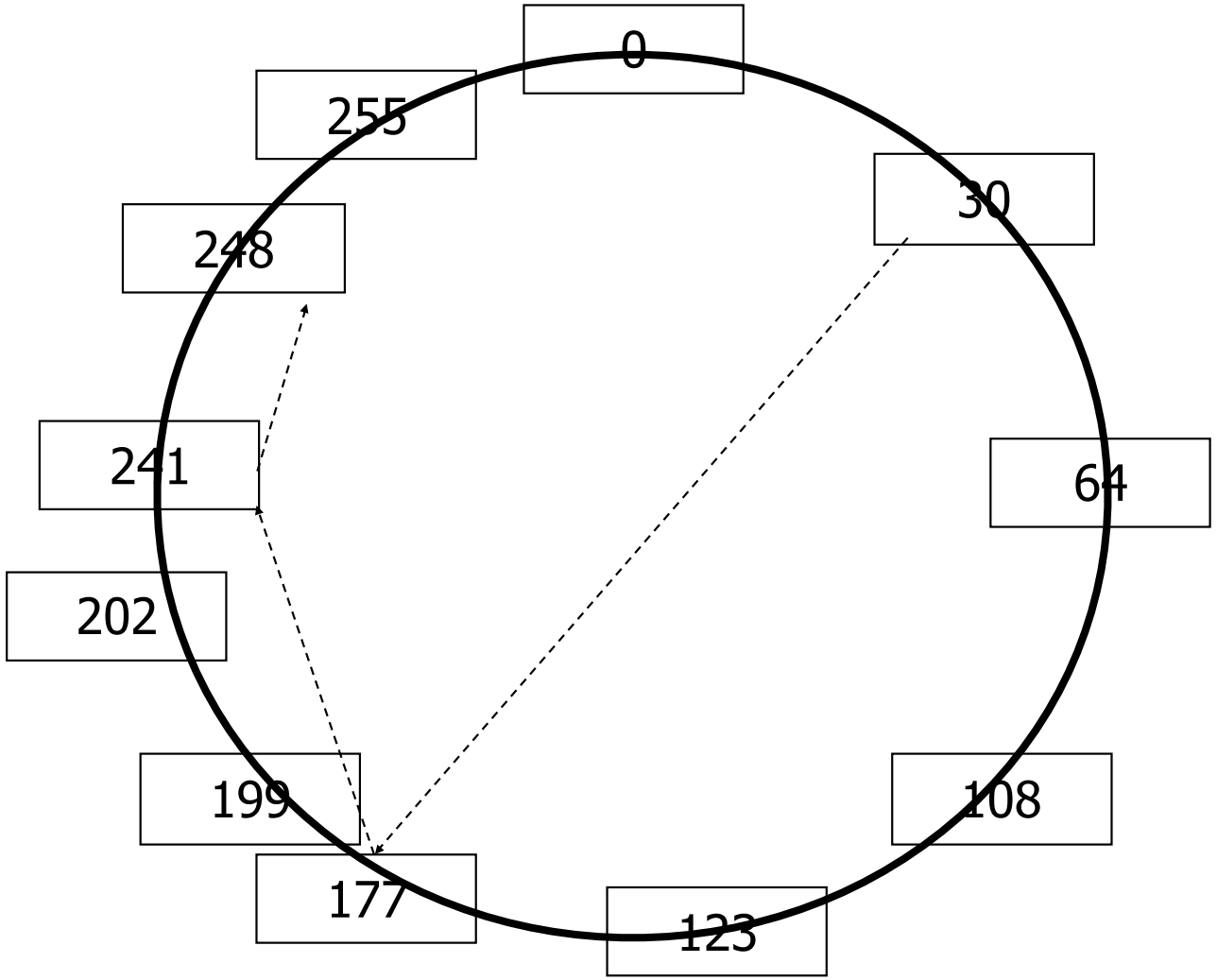
Chord

- MIT
- Data structure mapped to a network
- Each node is given a random ID: by hashing its IP address in a standard way
- Nodes are formed into a ring ordered by ID
- Then each node looks up the node ½ across, ¼ across, 1/8th across, etc.
- We can do binary lookups to get from one node to another now!

Chord picture



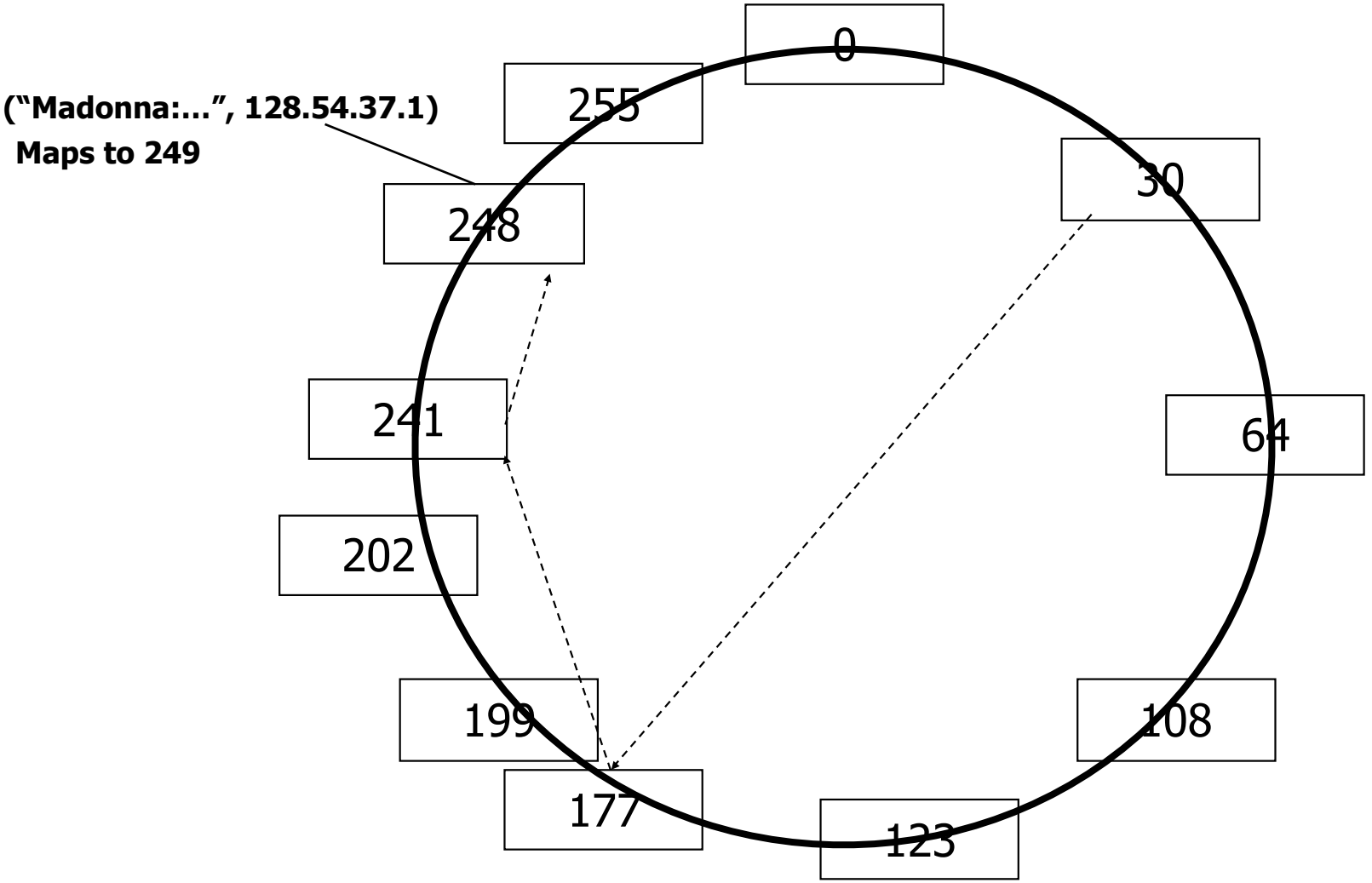
Node 30 searches for 249



OK… so we can look for nodes

- Now, store information in Chord
  - Each “record” consists of
    - A keyword or index
    - A value (normally small, like an IP address)
  - E.g:
    - (“Madonna:I’m not so innocent”, 128.74.53.1)
- We map the index using the hash function and save the tuple at the closest Chord node along the ring

Madonna “maps” to 249



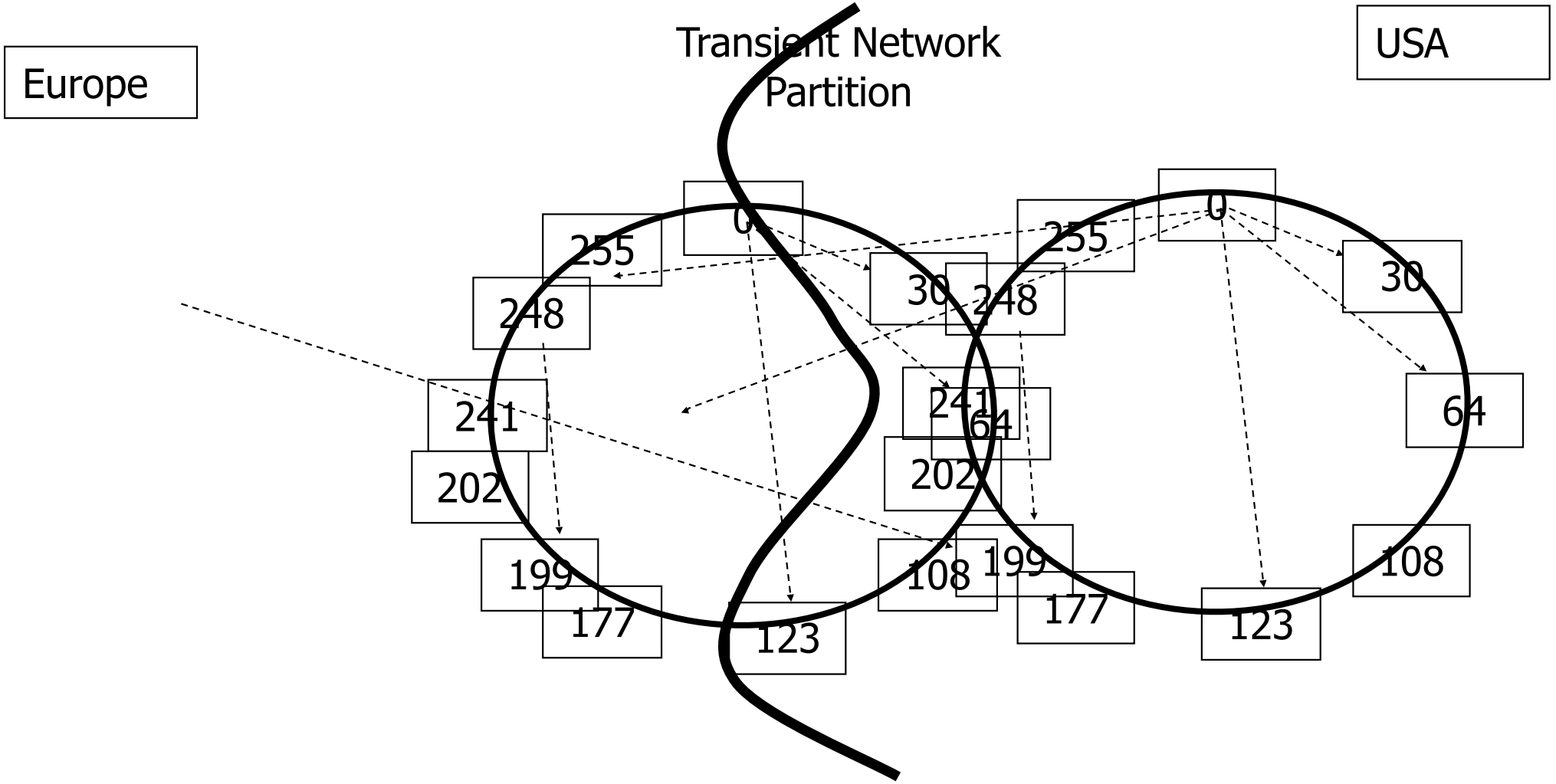
Looking for Madonna

- Take the name we're searching for (needs to be the exact name!)
- Map it to the internal id
- Lookup the closest node
- It sends back the tuple (and you cache its address for speedier access if this happens again!)

Some issues with Chord

- Failures and rejoins are common
  - Called the ***churn*** problem and it leaves holes in the ring
  - Chord has a self-repair mechanism that each node runs, independently, but it gets a bit complex
  - Also need to replicate information to ensure that it won't get lost in a crash

Chord can malfunction if the network partitions…



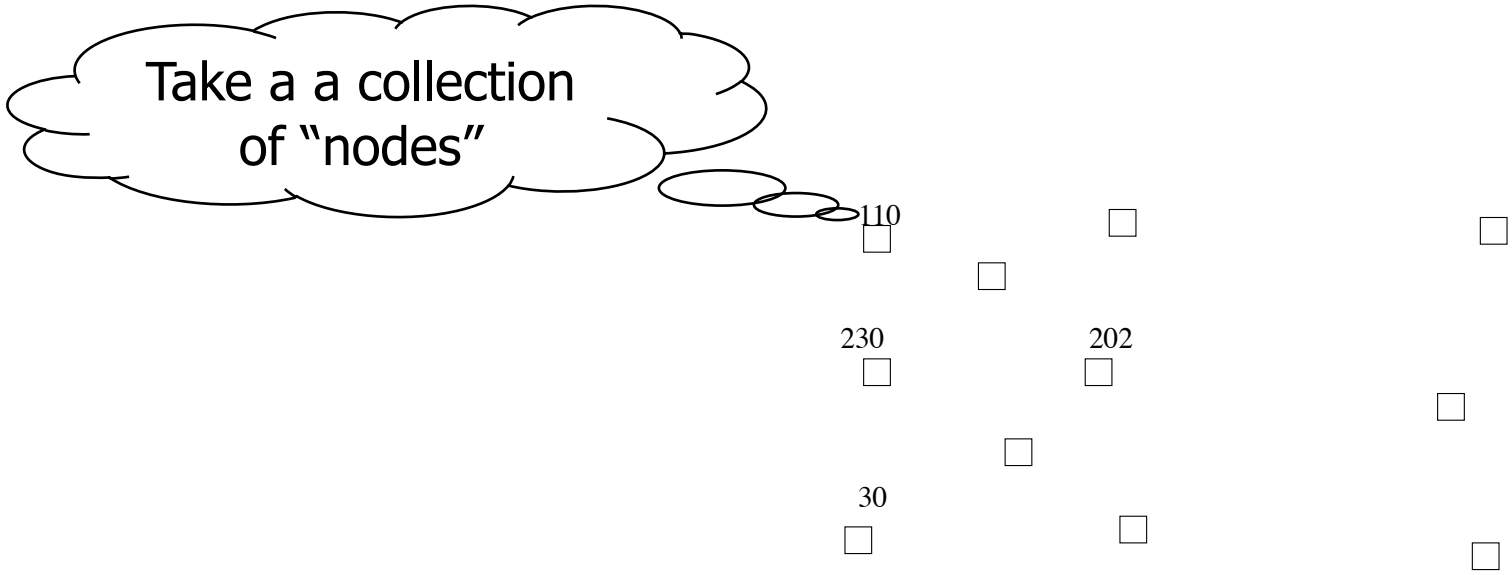
… so, who cares?

- Chord lookups can fail… and it suffers from high overheads when nodes churn
  - Loads surge just when things are already disrupted… quite often, because of loads
  - And can't predict how long Chord might remain disrupted once it gets that way
- Worst case scenario: Chord can become inconsistent and stay that way

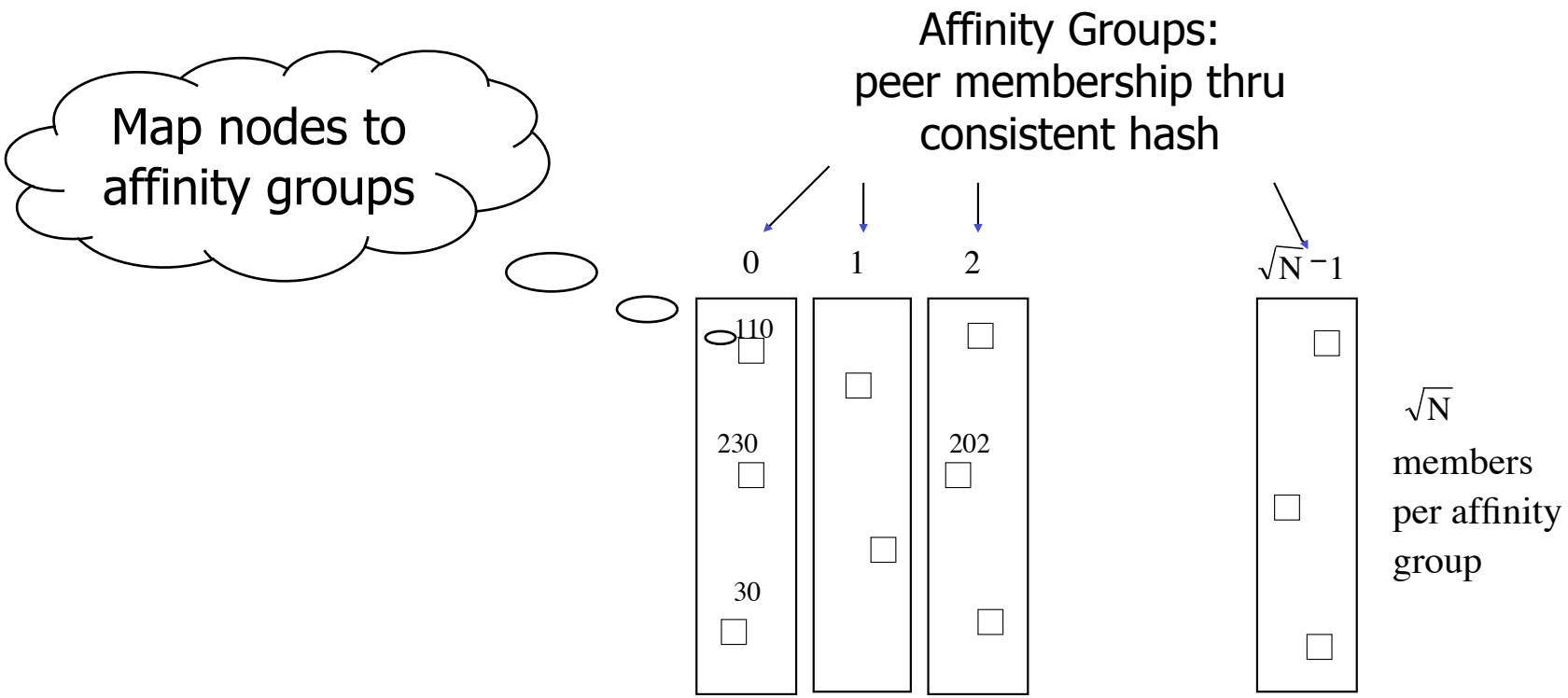
Can we do better?

- Kelips is a Cornell-developed “distributed hash table”, much like Chord
- But unlike Chord it heals itself after a partitioning failure
- It uses gossip to do this…

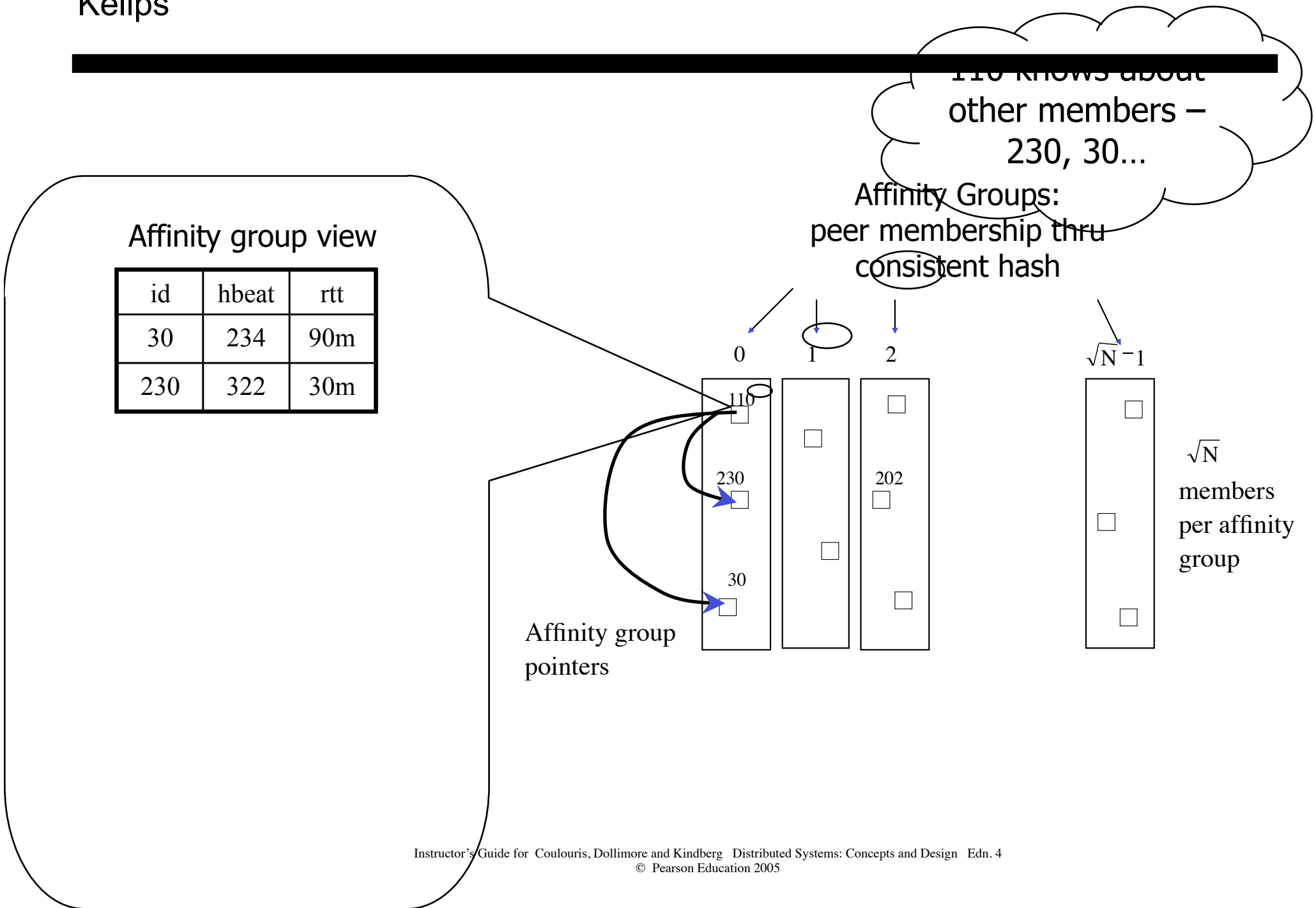
Kelips (Linga, Gupta, Birman)



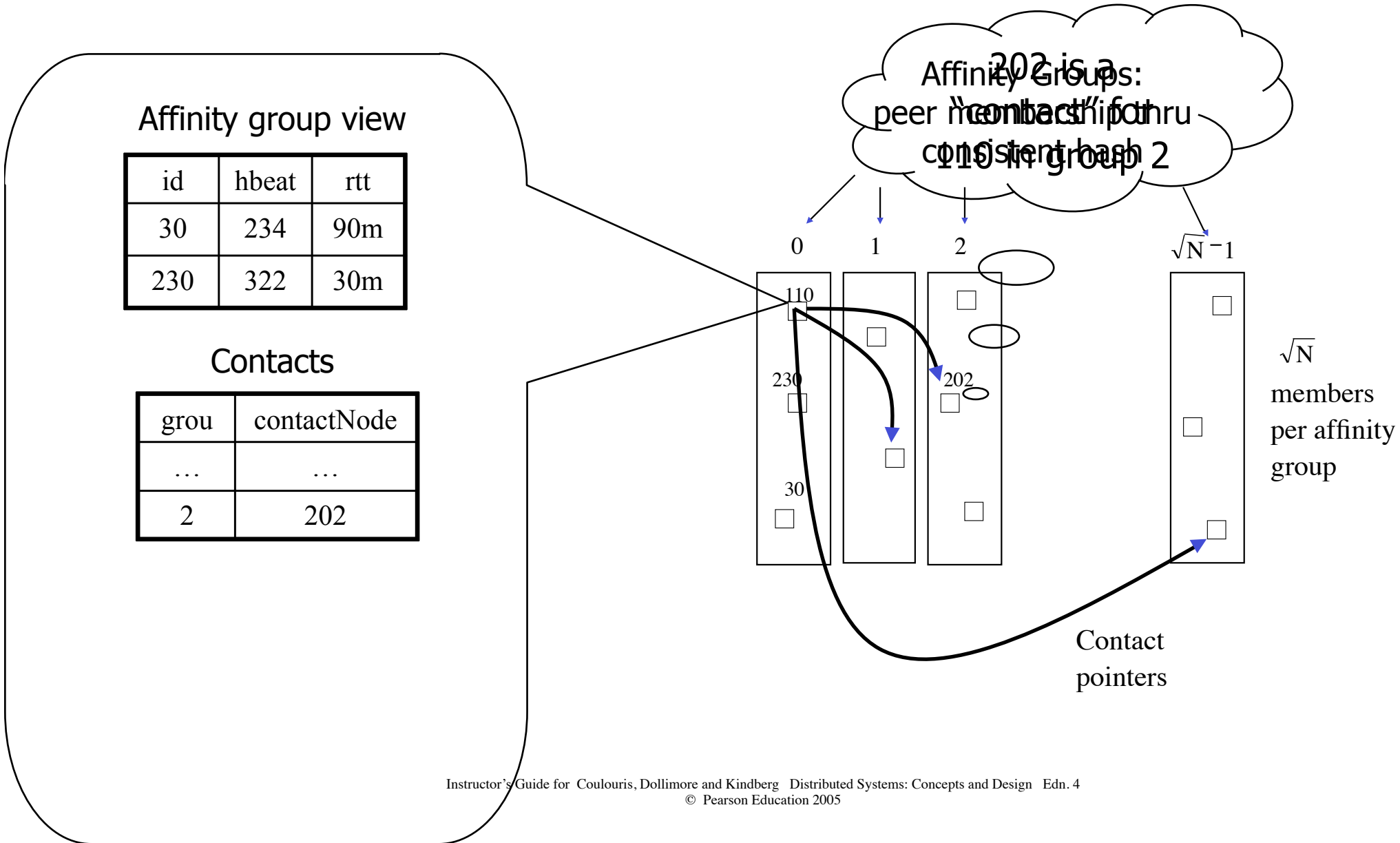
Kelips



Kelips



Kelips



Kelips

"cnn.com" maps to group 2.
So 110 tells group 2 to "route" inquiries about cnn.com to it.

Affinity Groups:
peer membership thru consistent hash

Affinity group view

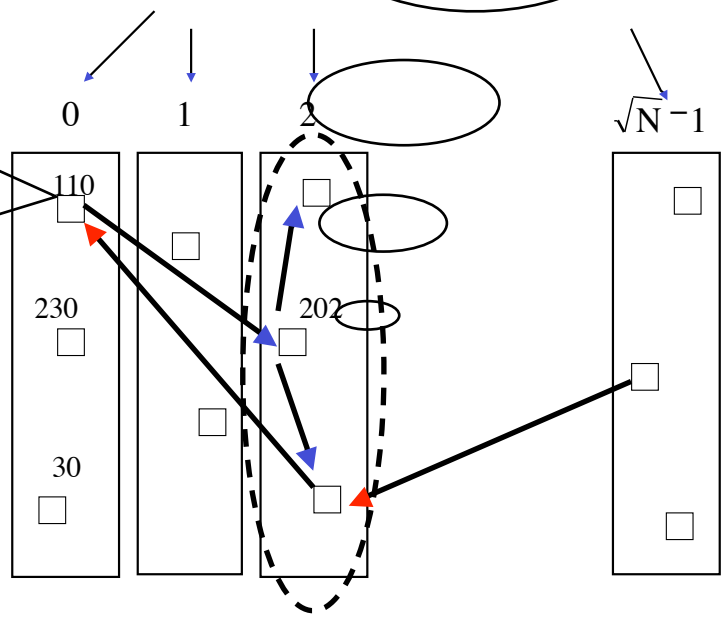
id	hbeat	rtt
30	234	90m
230	322	30m

Contacts

grou	contactNode
…	…
2	202

Resource Tuples

resource	info
…	…
cnn.com	110



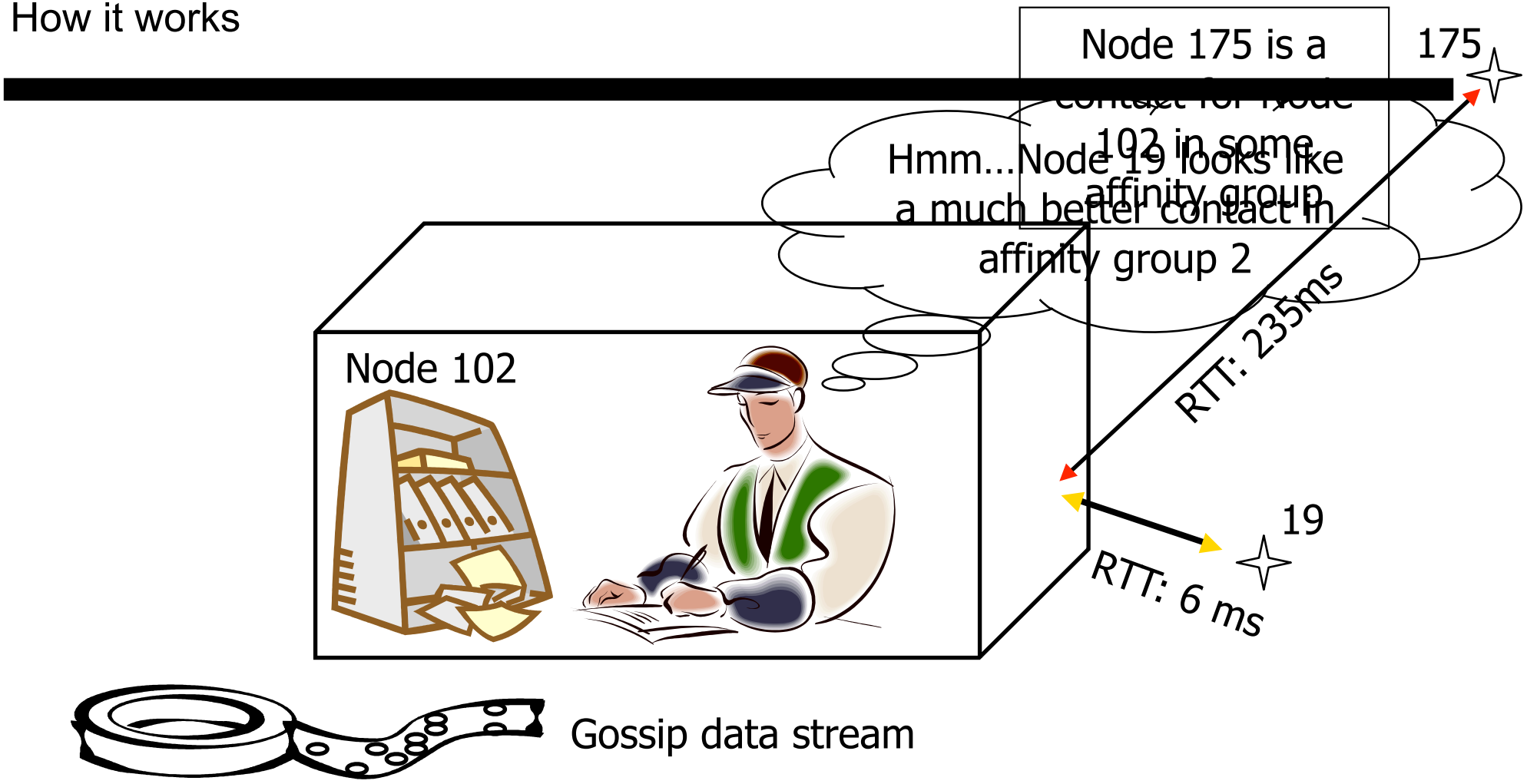
$\sqrt{N}$ members per affinity group

Gossip protocol replicates data cheaply

How it works

- Kelips is entirely gossip based!
  - Gossip about membership
  - Gossip to replicate and repair data
  - Gossip about “last heard from” time used to discard failed nodes
- Gossip “channel” uses fixed bandwidth
  - … fixed rate, packets of limited size

How it works



Gossip data stream

- Heuristic: periodically ping contacts to check liveness, RTT… swap so-so ones for better ones.

Replication makes it robust

- Kelips should work even during disruptive episodes
  - After all, tuples are replicated to $\sqrt{N}$ nodes
  - Query k nodes concurrently to overcome isolated crashes, also reduces risk that very recent data could be missed
- … we often overlook importance of showing that systems work while recovering from a disruption

Squirrel

- A P2P Web caching system exploiting storage and computing resources available on a network
- Based on Pastry
- GET requests can have three outcomes:
  - uncachable, cache miss, or cache hit
- objects are stored with timestamps and time-to-live
- A secure hash function is applied to all objects (URLs) to produce the 128bit GUID
- The node whose GUID is closest becomes the storage for the object: *home node*

Squirrel evaluation

- Tested on 105 and 36.000 clients
- Measured reduction of external bandwidth (based hit ratios of 29-38%)
- Similar to no caching user experience in object access latency
- Overload on machine was of about 0.31 cache requests per minute per machine

- Conclusion: similar performance to a centralized cache, but better fault tolerance and reduced costs

	Lookup	Maintenance	Routing table size
Gnutella	*variable*	*variable*	*number of neighbors*
Tapestry	$O(\log_b(N))$	$O(\log_b(N))$	$b * \log_b(N)$
Pastry	$O(\log_{2^b}(N))$	$O(\log_{2^b}(N))$	$(2^b - 1) * \log_{2^b}(N)$
Chord	$O(\log_2(N))$	$O(\log_2{}^2(N))$	$\log_2(N)$

	Changing system size	Stabilization (nodes joining and disjoining)	Searching (number and kind of requests)	Congestion (popular items, distribution of keys)
Gnutella	*Poor*. The flooding-based routing mechanism does not scale, since the network tends to fill up with messages.	*Good*. The joining is a fairly quick process. Also, Gnutella uses heartbeat messages to determine whether the neighbors are still alive.	*Poor and restricted*. The network is flooded with queries, which do not reach all of the nodes that might have matching objects.	*Poor*. Congested peers are not considered in any way. A node with a popular item might have to send it continuously.
Tapestry	*Good*. Consistent hashing balances the load in the network. Growing network size does not affect the functionality appreciably.	*Fair*. In addition to polling the network to detect link failures, many messages must be sent when node is joining or leaving the network. Backups are kept for each routing table entry to address possibly frequent departures.	*Excellent*. The cost of searching is logarithmic in relation to the network size. Additionally, it performs even under frequent failures due to intelligent algorithm.	*Good*. Tapestry's infrastructure supports replication indirectly. Each node publishes the copy of the object itself. Therefore, The number of congested peers should diminish.
Pastry	*Good*. Consistent hashing balances the load in the network. Growing network size does not affect the functionality appreciably.	*Fair*. Like Tapestry, Pastry polls the network for link failures. Its repair mechanism is a little simpler than that of Tapestry's, but many messages are used there too.	*Excellent*. The cost of searching is logarithmic in relation to the network size. Additionally, it performs even under frequent failures due to intelligent algorithm.	*Good*. Pastry's design supports replicas also. For example PAST [14], a file sharing utility built on Pastry, stores file replicas on a set of nodes.
Chord	*Good*. Consistent hashing balances the load in the network. Growing network size does not affect the functionality appreciably.	*Fair/Good*. Chord is said to have less maintenance overhead than most of the DHT-based systems. At the same time, it is more vulnerable against malicious nodes.	*Excellent*. The cost of searching is logarithmic in relation to the network size. Additionally, it performs even under frequent failures due to intelligent algorithm.	*Fair*. Chord does not provide replication but leaves this to application developers.

Beyond DHT: skip-graphs

- A balanced tree data structure
- Highly resilient to failures
- Very good complexity of performing search, insertion and deletion of nodes
  - Resource location and dynamic node addition and deletion in logarithmic time
  - Each node requires only logarithmic space to store information about neighbors
  - There is no hashing of the resource keys, so related resources are present near each other

- A *skip list* [Pugh 1990], is a randomized balanced tree data structure organized as a tower of increasingly sparse linked lists

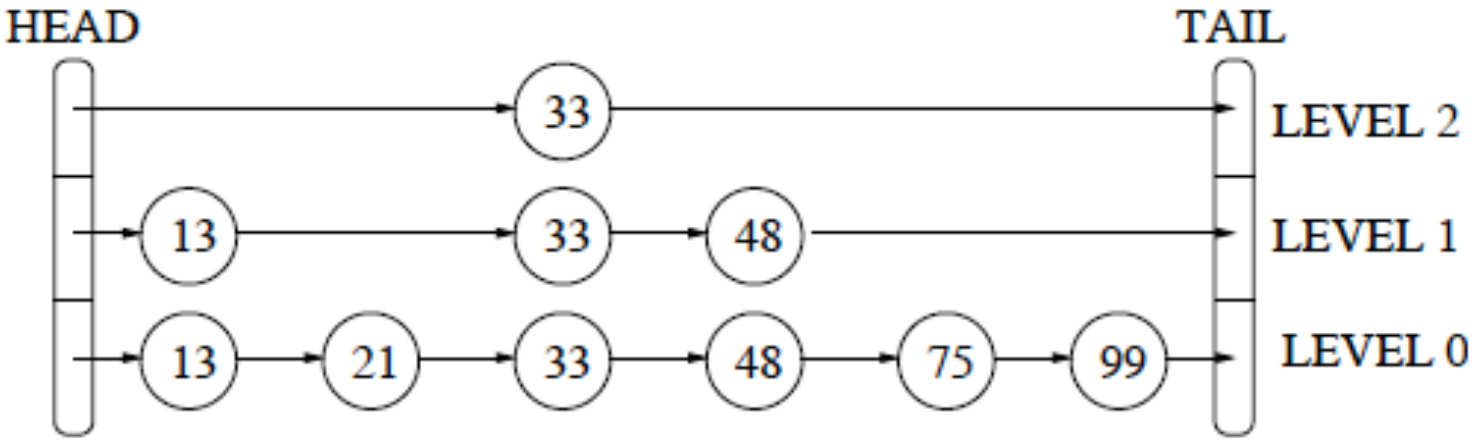


Fig. 1. A skip list with $n = 6$ nodes and $\lceil \log n \rceil = 3$ levels.

- It lacks redundancy

Skip graphs

- A skip graph is a collection of skip lists where the lower levels are shared
- Each node v gets a uniformly random key $m(v) \in \Sigma\infty$ (computed only as needed)
- Nodes v and w are in the same level i list if m(v) and m(w) have the same prefix of length i
- Every node is a highest level node in its own list and all nodes share the same $List_0$.
- Each node needs to keep only two pointers for each level for O(log n) pointers.

- A skip list + membership vector

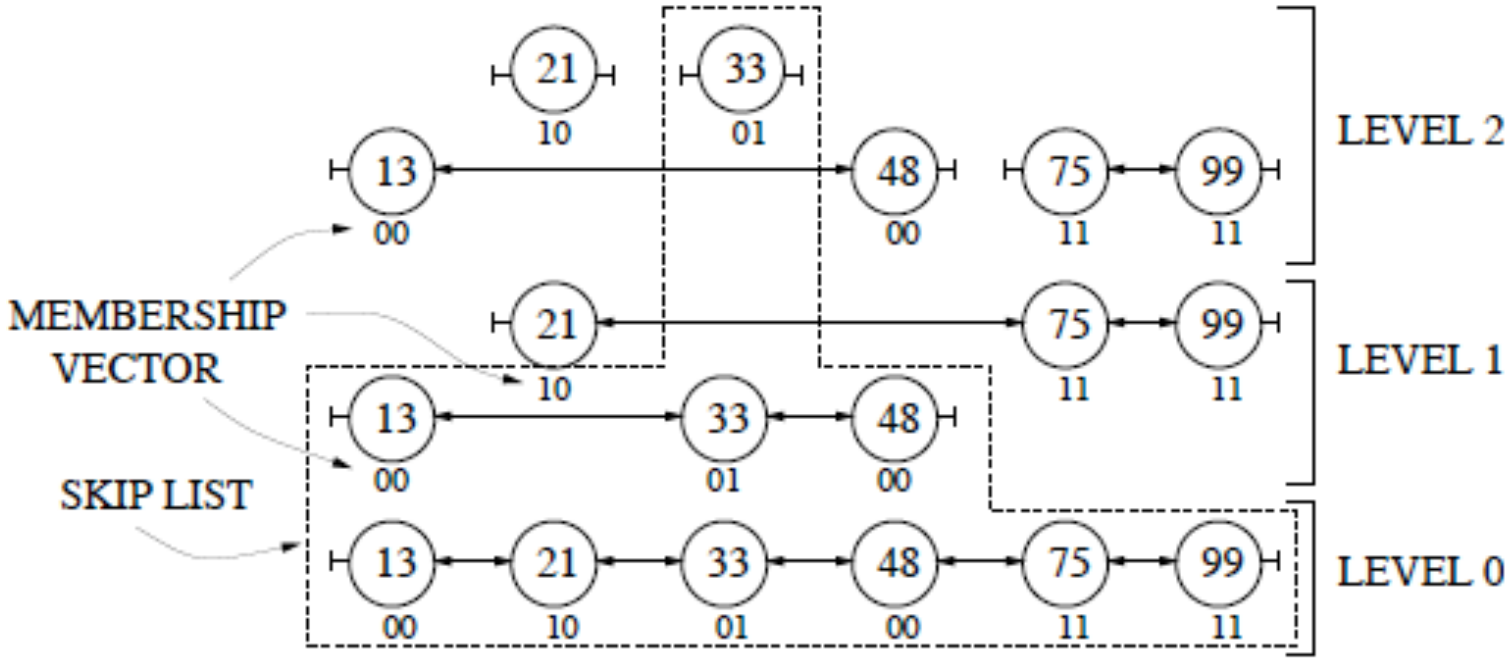
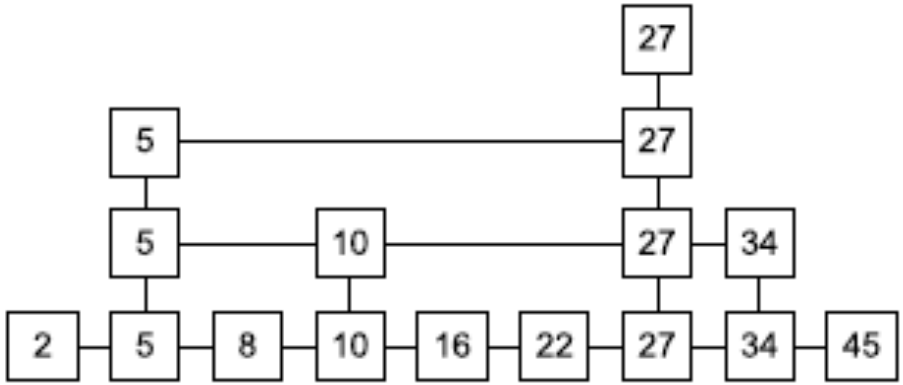
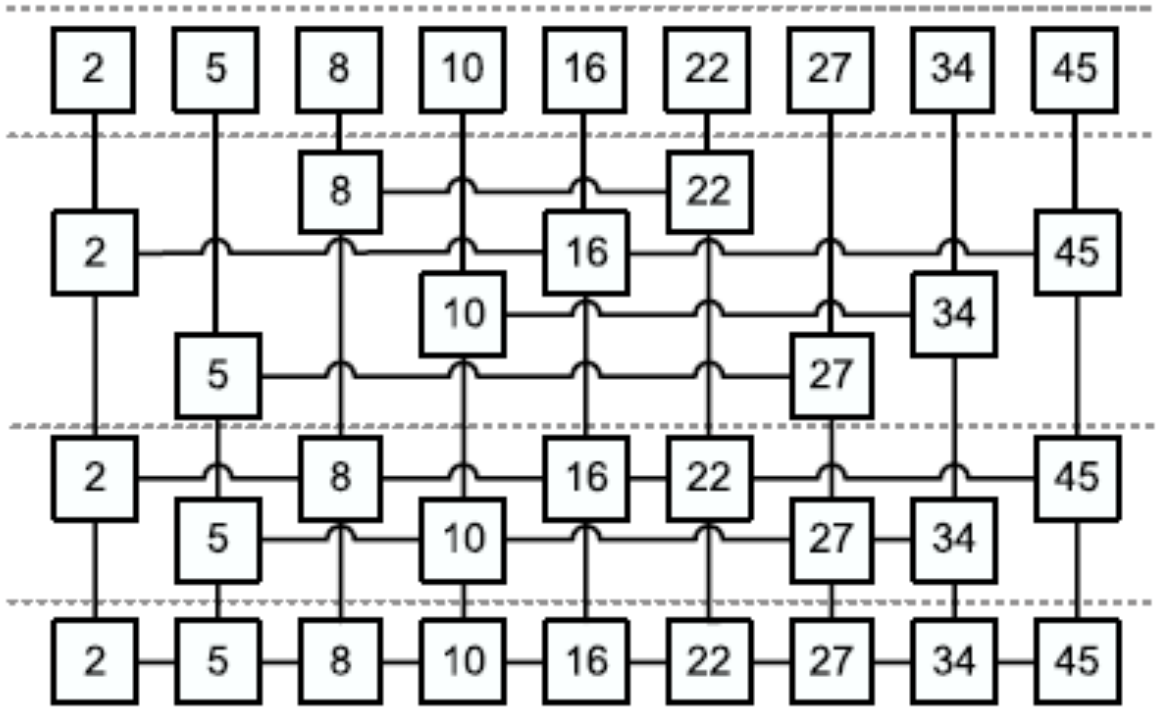


Fig. 2. A skip graph with $n = 6$ nodes and $\lceil \log n \rceil = 3$ levels.





Method	M	$Q(n)$	$U(n)$	$C(n)$
skip graphs/SkipNet [4, 11]	$O(\log n)$	$O(\log n)$ w.h.p.	$O(\log n)$ w.h.p.	$O(\log n/n)$
NoN skip-graphs [14, 15]	$O(\log^2 n)$	$\tilde{O}(\log n/\log\log n)$	$\tilde{O}(\log^2 n)$	$O(\log^2 n/n)$
family trees [20]	$O(1)$	$\tilde{O}(\log n)$	$\tilde{O}(\log n)$	$O(\log n/n)$
deterministic SkipNet [10]	$O(\log n)$	$O(\log n)$	$O(\log^2 n)$	$O(n^{0.32}/n)$
bucket skip graphs [3]	$O(\log n)$	$\tilde{O}(\log n)$	$\tilde{O}(\log n)$	$O(\log^2 n/n)$
skip-webs [2]	$O(\log n)$	$\tilde{O}(\log n/\log\log n)$	$\tilde{O}(\log n/\log\log n)$	$O(\log n/n)$
rainbow skip graphs	$O(1)$	$O(\log n)$ w.h.p.	$O(\log n)$ amort. w.h.p.	$O(\log n/n)$
strong rainbow skip graphs	$O(1)$	$O(\log n)$	$O(\log n)$ amort.	$O(n^{\epsilon}/n)$

Skip graphs facts

- Construction, insertion, searching in O(log n) time
- O(log n) addresses stored per node.
- Robust to adversarial o(log n) node failures
- Localized affect of congestion due to "hot spots"

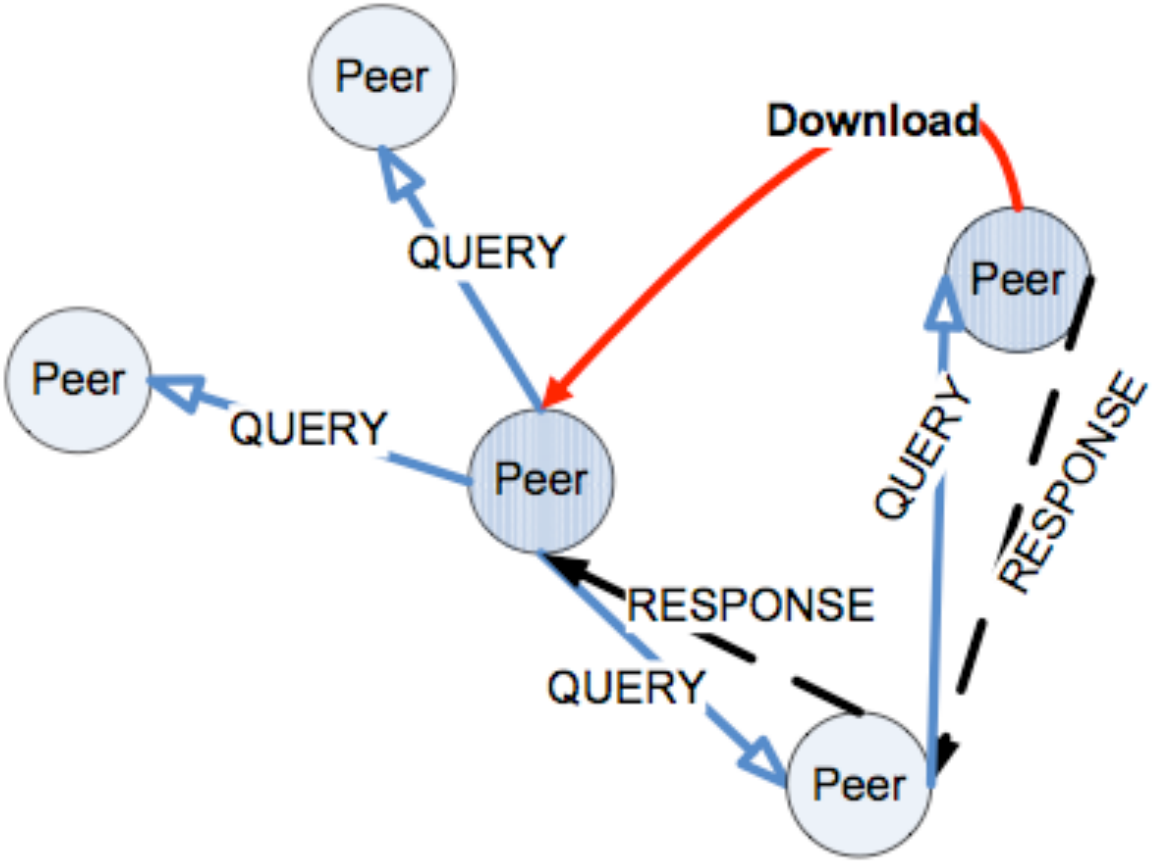
Unstructured P2P

Unstructured P2P systems

- Node does not have a priori knowledge of topology (random graph)
- Flooding as discovery and query mechanism
- Highly replicated data
- Effective for highly popular items, poorly suited for locating rare items
- Scaling may give problems for high aggregates of queries and sudden increase in system size

Gnutella

- Gnutella: all peers absolutely equal
- Flat topology
- Placement of data is not based on any knowledge of the topology
- Typical query method is flooding
- Queries are propagated with a certain radius (time to live of queries based on number of hops)
- To join the network: http://gnutellahosts.com


Peer
Download
QUERY
Peer
Peer
QUERY
Peer
QUERY
RESPONSE
RESPONSE
QUERY
Peer

Guntella API

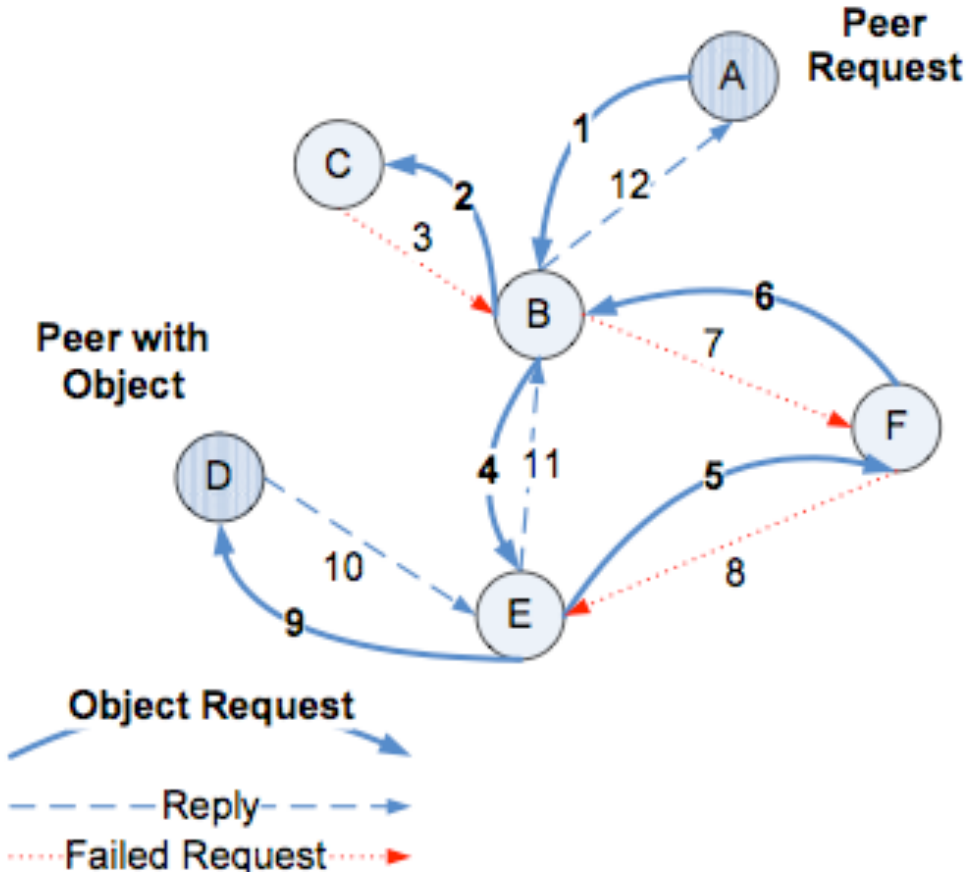
- Group Membership:
  - Ping and Pong
- Search:
  - Query and Query Response
- File Transfer:
  - Get and Push

Gnutella

- Anycast:
  - each user connected to few peers
  - flooding requests to all hosts within designated distance
  - any machine that matches the query responds (if no response increase distance of query)
- Experience shows, that it does not scale well
- Problems with rare files
- Latest version uses notion of super-peer to improve routing performance

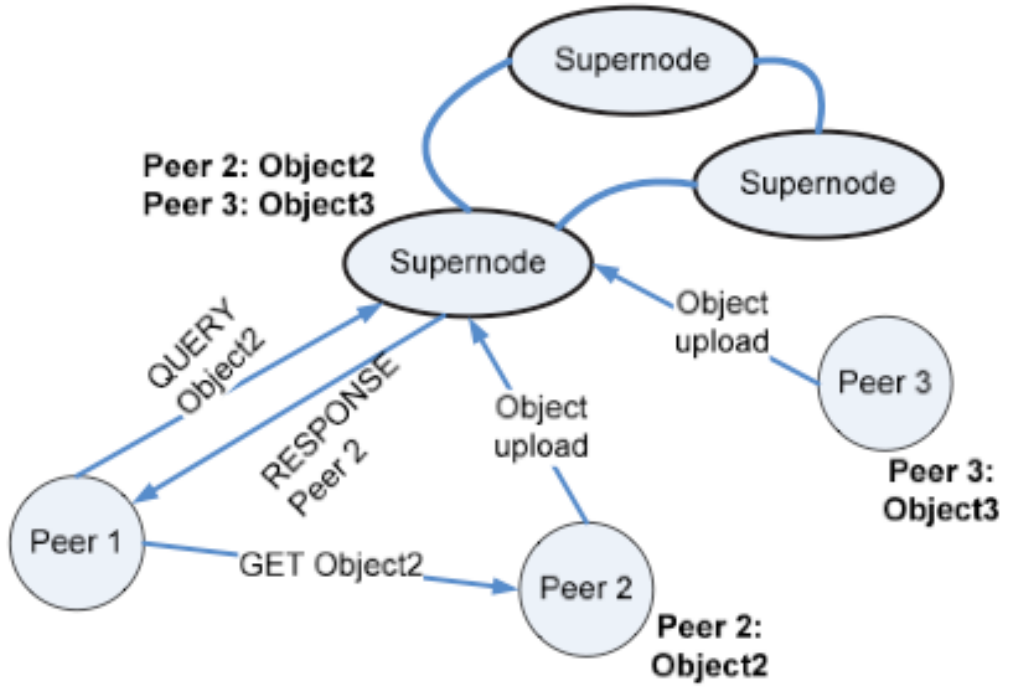
Freenet

- Adaptive p2p network
- Anonymity
- Dynamic routing table maintained by every node
- Data files are encrypted
- User publishes content-key and decryption key
- Suffers from man in the middle and Trojan attacks



KaZaA/FastTrack

- Meta-data searching
- super-peers architecture (hybrid)
- consumes bandwidth to maintain the index at super-peers
- super-peers use broadcast to search with other super-peers (no targeted routing)
- Scale-free network topology (experimental evidence)



eDonkey/Overnet

- Hybrid two-layer P2P
- Publish and retrieve small pieces of data
- To join connect to a known “server” (known IP address)
- uses meta-data for searching

End-Host based Multicast: BitTorrent

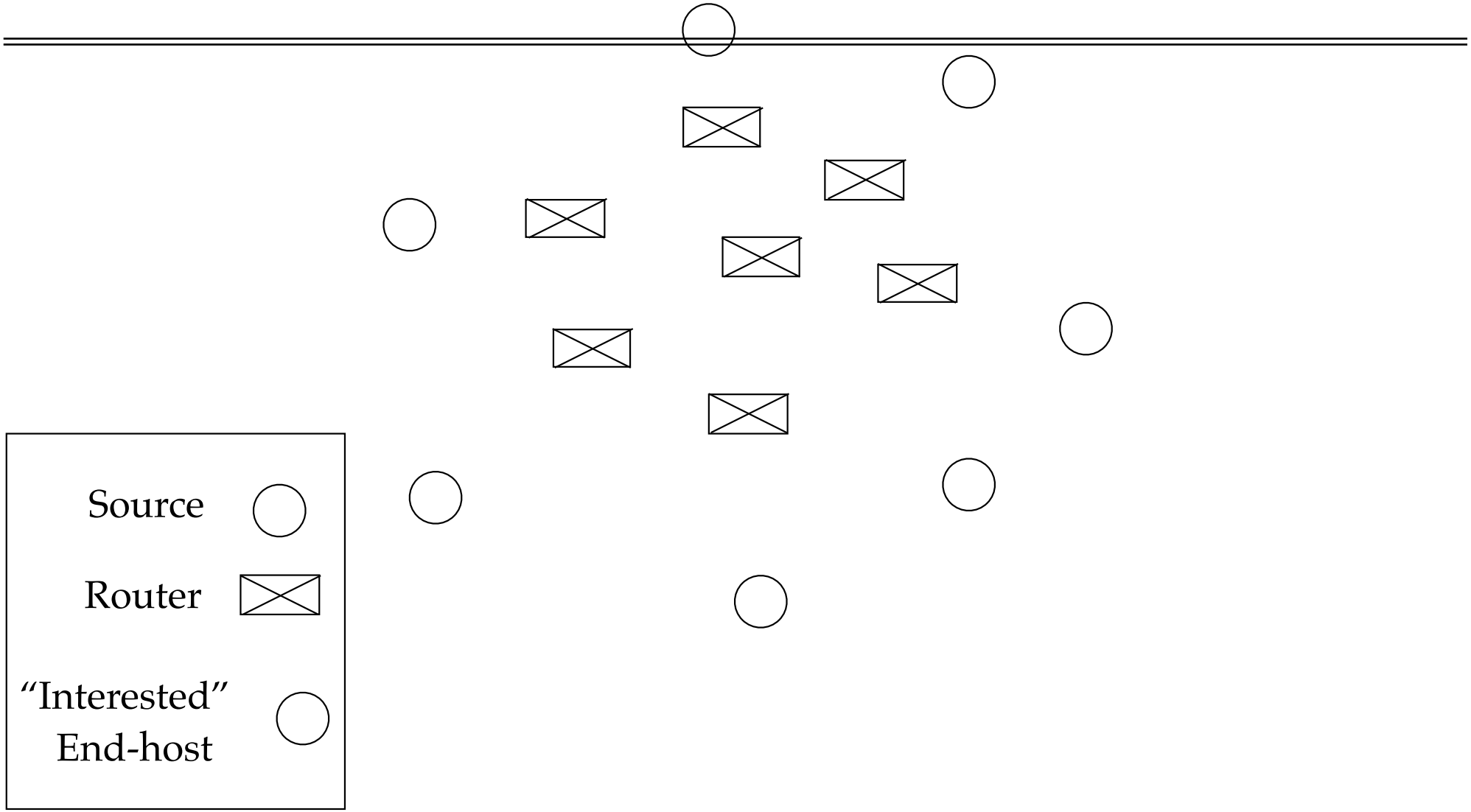
Based on Vivek Vishnumurthy slides @ Cornell

Common Scenario

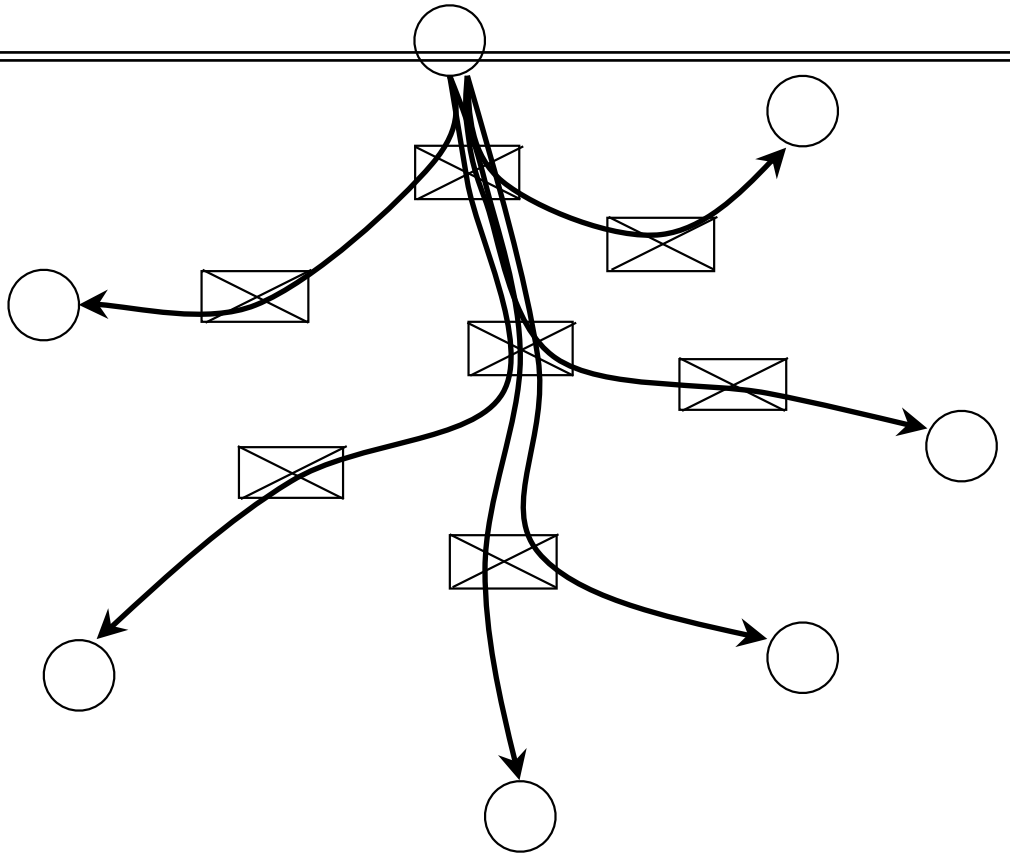
- Millions want to download the same popular huge files (for free)
  - ISO's
  - Media (the real example!)
- Client-server model fails
  - Single server fails
  - Can't afford to deploy enough servers

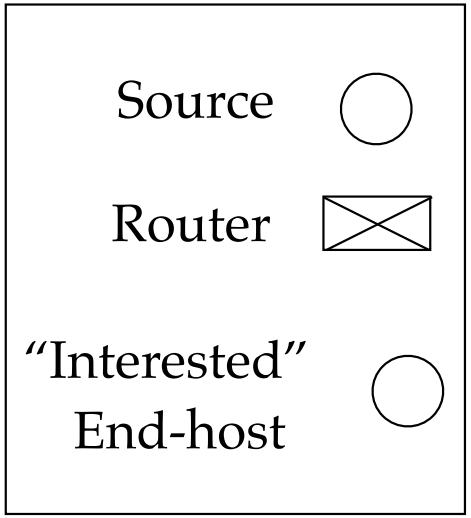
IP Multicast?

- Recall: IP Multicast not a real option in general settings
  - Not scalable
  - Only used in private settings
- Alternatives
  - End-host based Multicast
  - BitTorrent
  - Other P2P file-sharing schemes

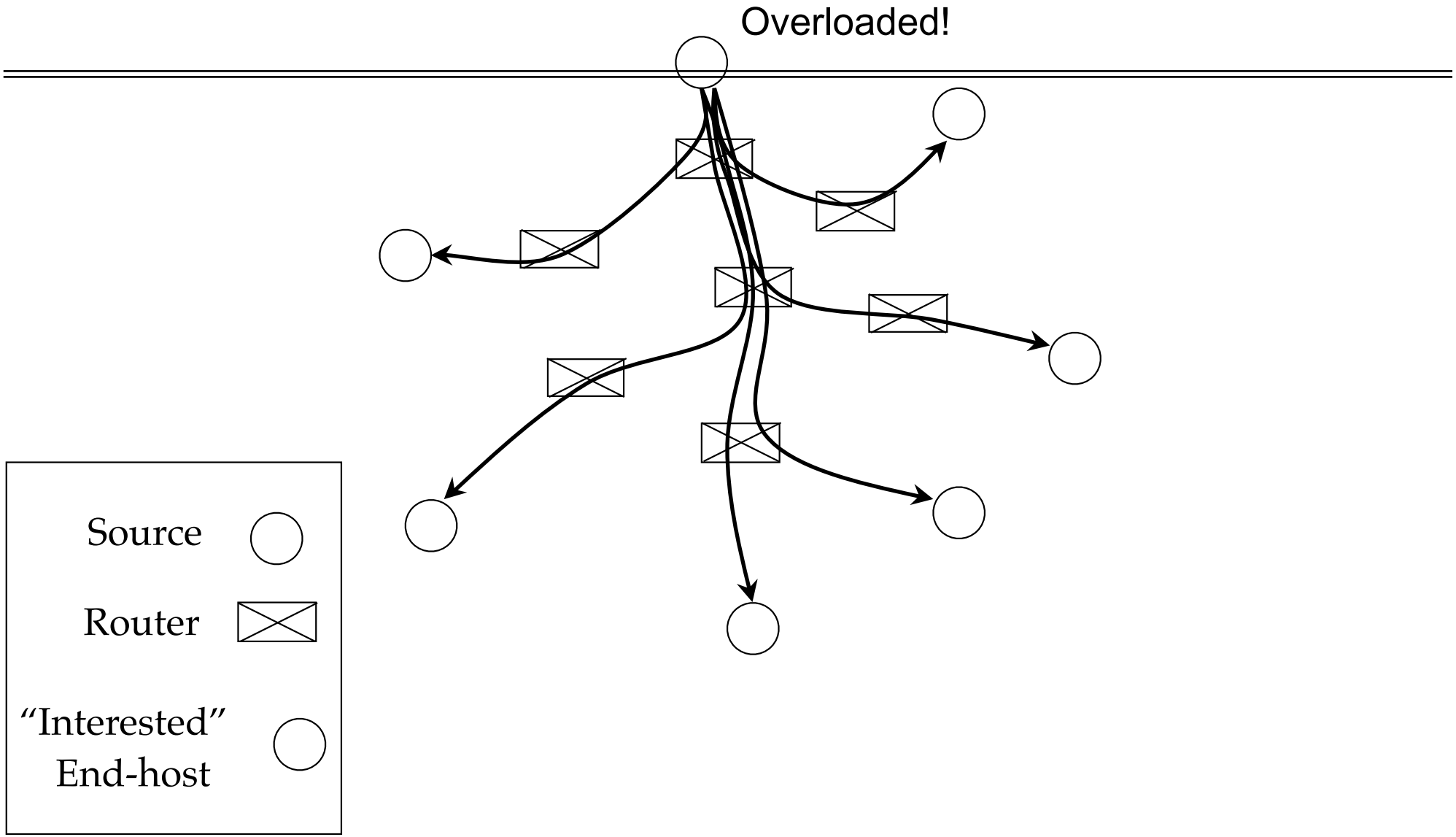

Source
Router
“Interested”
End-host

Client-Server

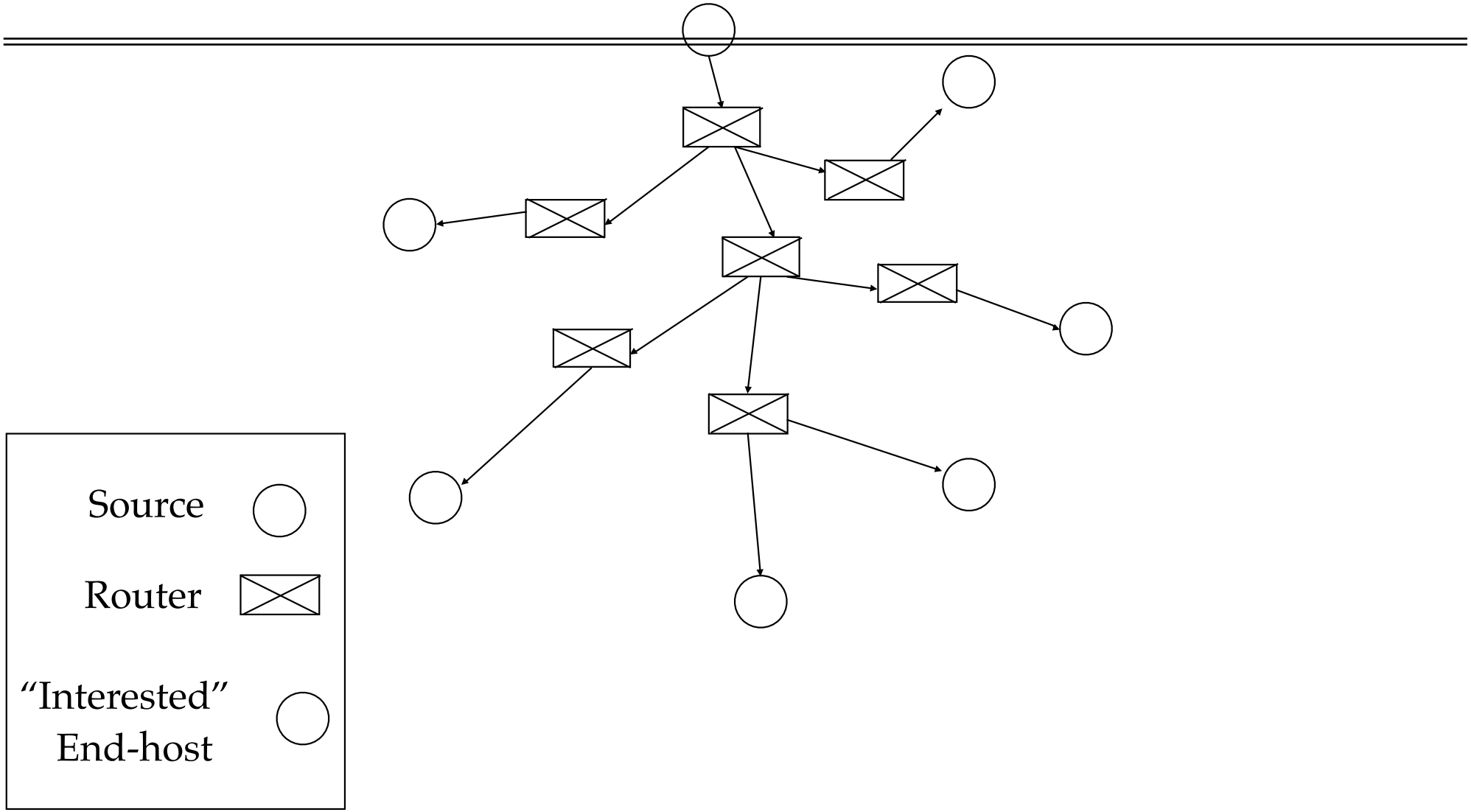




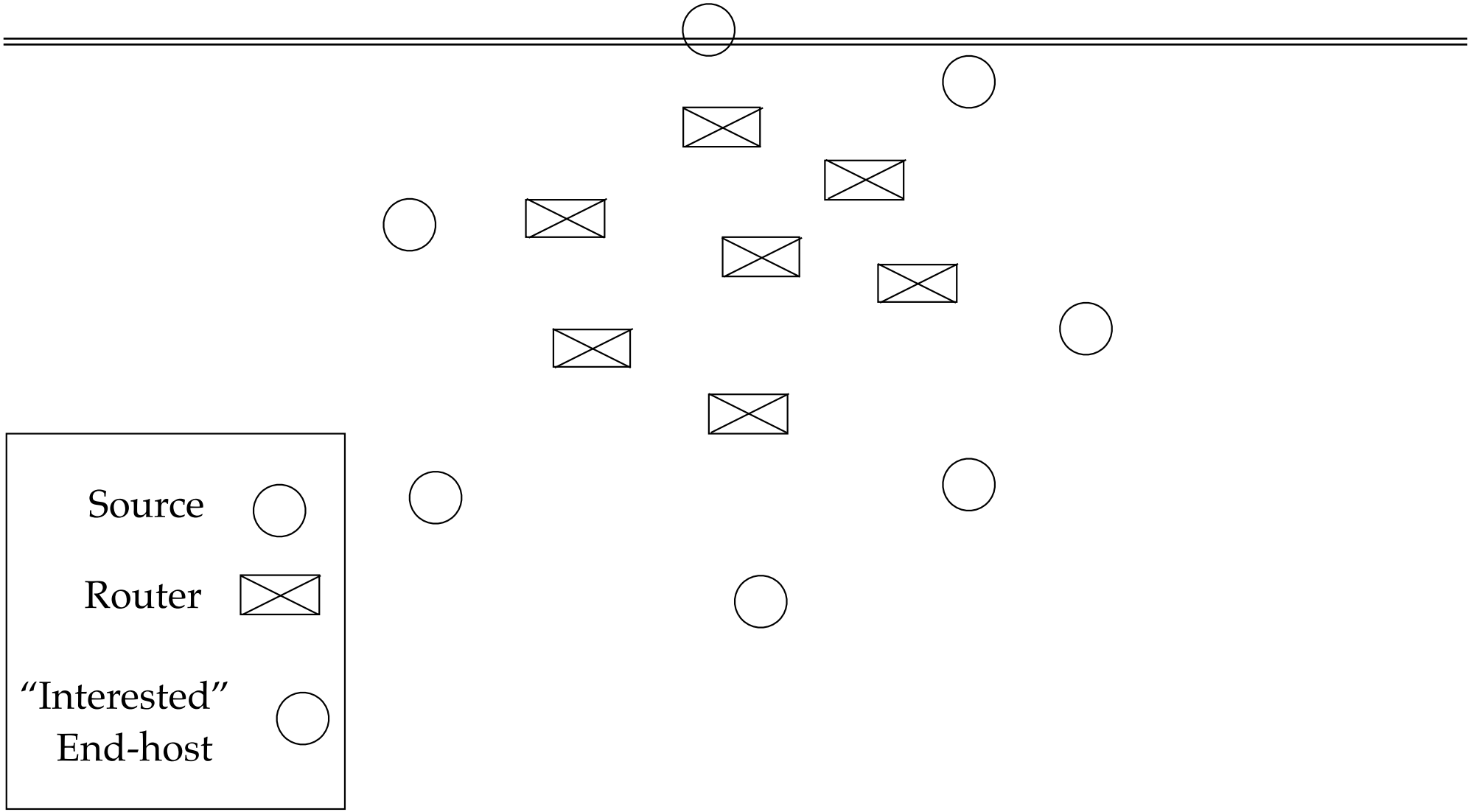
Client-Server



IP multicast



End-host based multicast



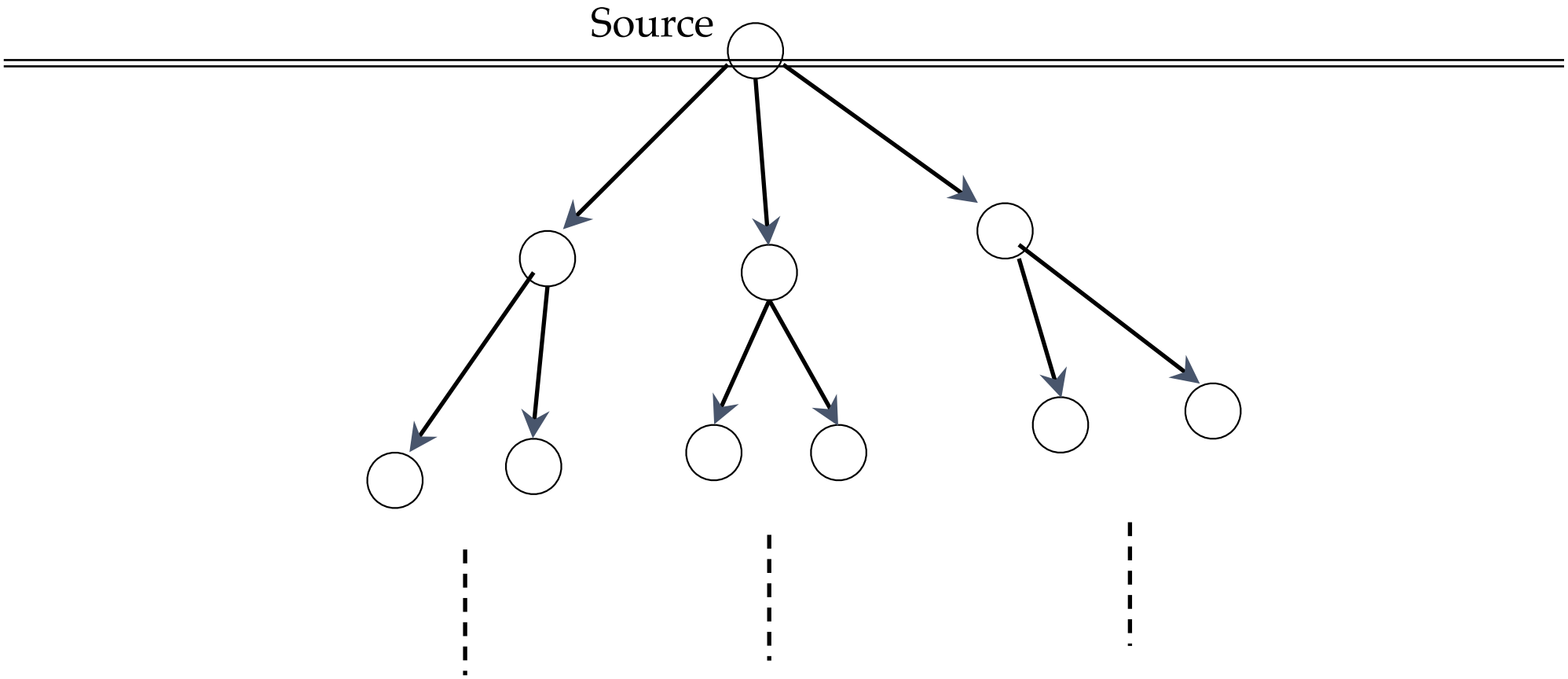
End-host based multicast

- "Single-uploader" ➔ "Multiple-uploaders"
  - Lots of nodes want to download
  - Make use of their *uploading* abilities as well
  - Node that has downloaded (part of) file will then upload it to other nodes.
    - ➢ Uploading costs amortized across all nodes

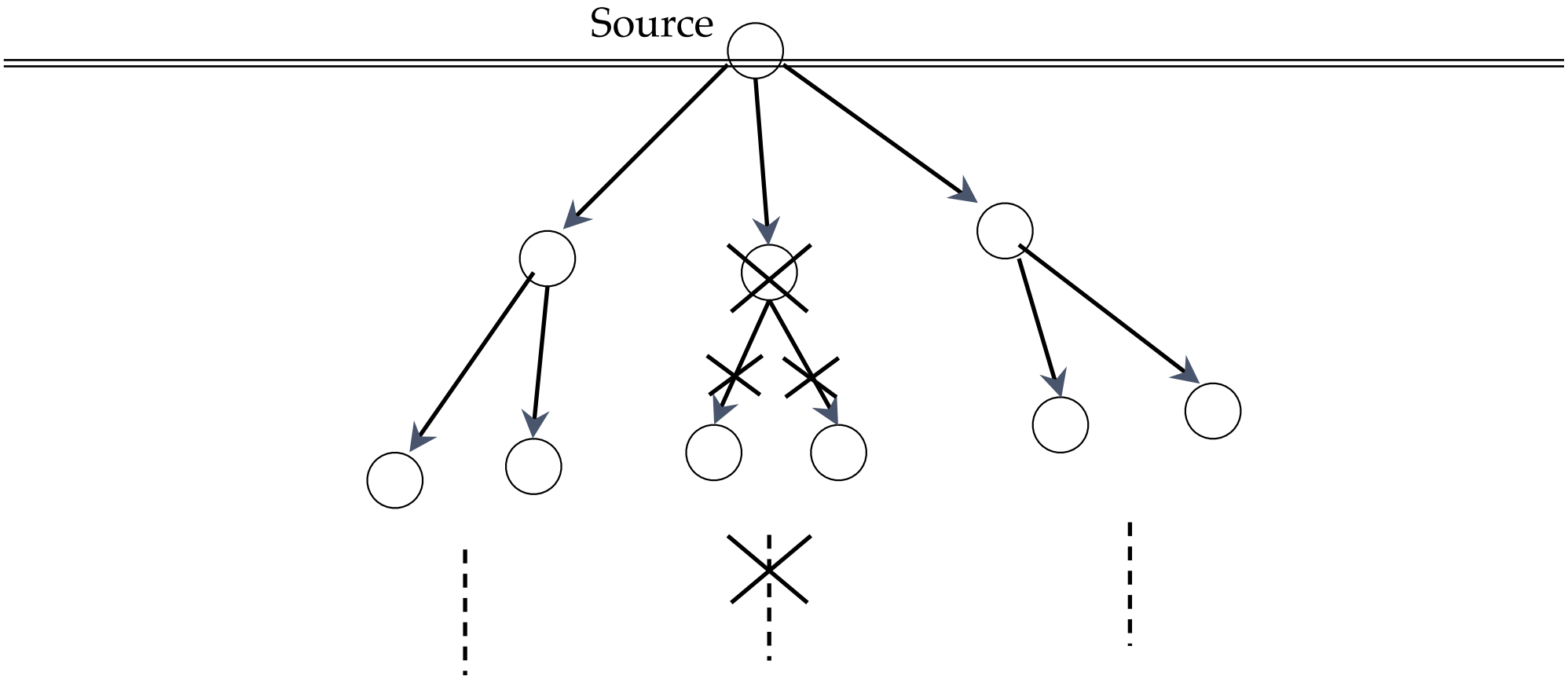
End-host based multicast

- Also called “Application-level Multicast”
- Many protocols proposed early this decade
  - Yoid (2000), Narada (2000), Overcast (2000), ALMI (2001)
    - All use single trees
    - Problem with single trees?

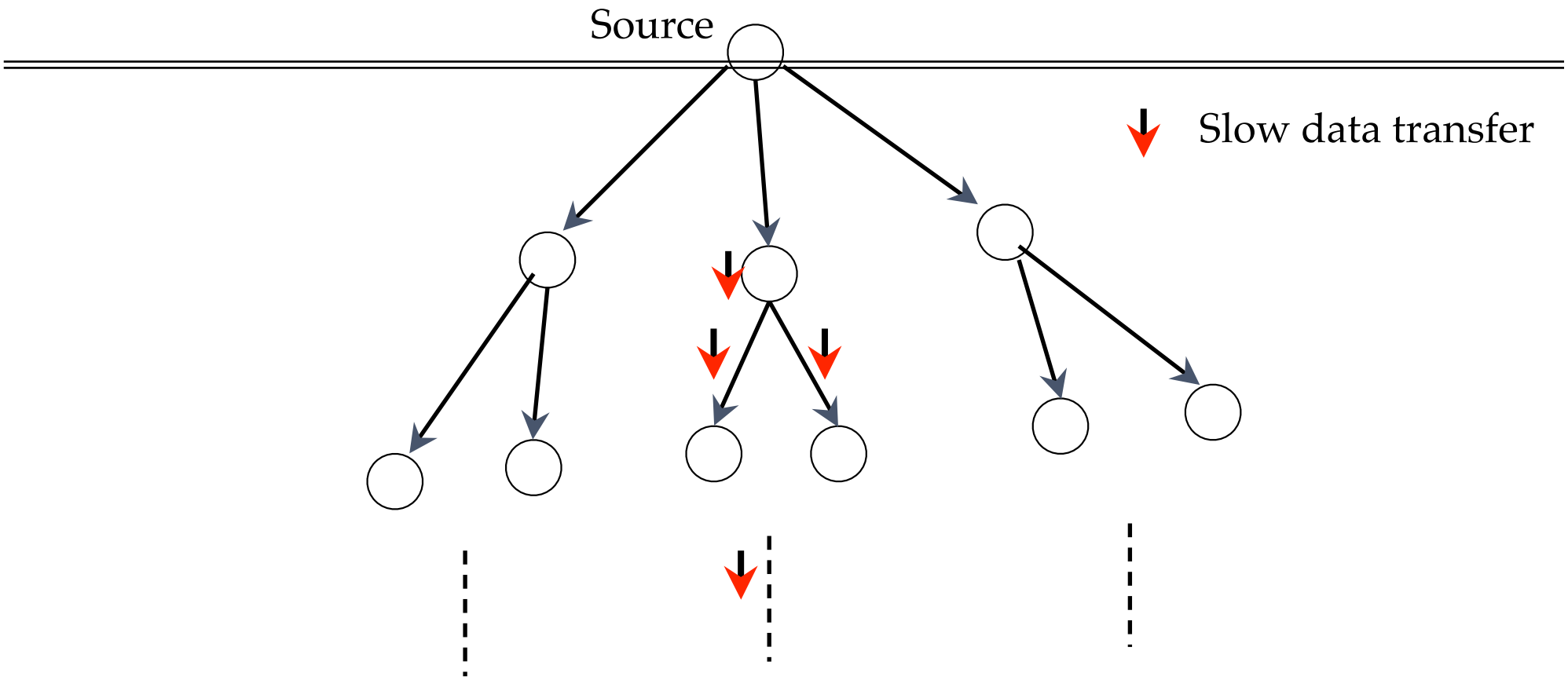
End-host multicast using single tree



End-host multicast using single tree



End-host multicast using single tree



End-host multicast using single tree

- Tree is “push-based” – node receives data, pushes data to children
- Failure of “interior”-node affects downloads in entire subtree rooted at node
- Slow interior node similarly affects entire subtree
- Also, leaf-nodes don’t do any sending!
- Though later multi-tree / multi-path protocols (Chunkyspread (2006), Chainsaw (2005), Bullet (2003)) mitigate some of these issues

BitTorrent

- Written by Bram Cohen (in Python) in 2001
- 27-55% of all Internet traffic (depending on geographical location) as of February 2009
- "Pull-based" "swarming" approach
  - Each file split into smaller pieces
  - Nodes request desired pieces from neighbors
    - As opposed to parents pushing data that they receive
  - Pieces not downloaded in sequential order
  - Previous multicast schemes aimed to support "streaming"; BitTorrent does not
- Encourages contribution by all nodes

BitTorrent

- Identically sized pieces, typically between 64 KB and 4 MB each
- The peer creates a checksum for each piece, using the SHA1 hashing algorithm

BitTorrent Swarm

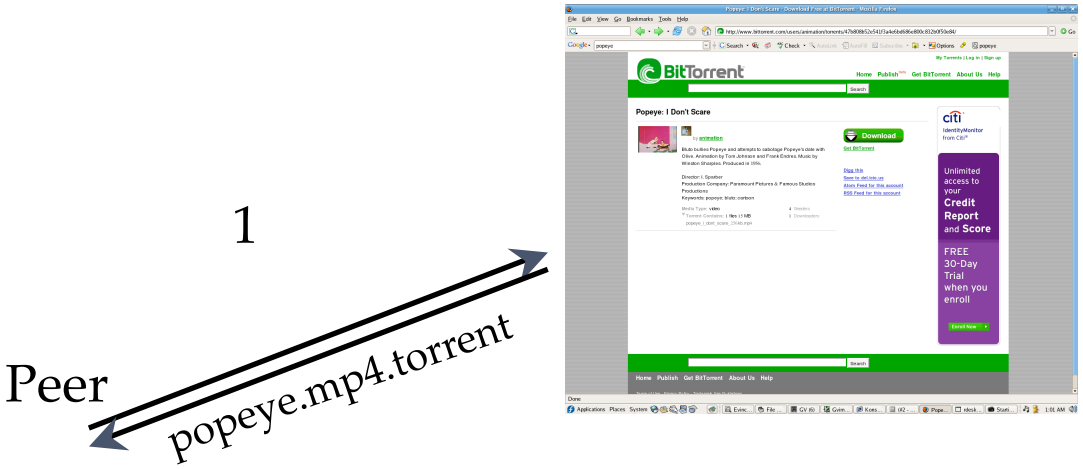
- Swarm
  - Set of peers all downloading the same file
  - Organized as a random mesh
- Each node knows list of pieces downloaded by neighbors
- Node requests pieces it does not own from neighbors

How a node enters a swarm for file "popeye.mp4"

- File popeye.mp4.torrent hosted at a (well-known) webserver
- The .torrent has address of tracker for file
- The tracker, which runs on a webserver as well, keeps track of all peers downloading file

How a node enters a swarm for file "popeye.mp4"

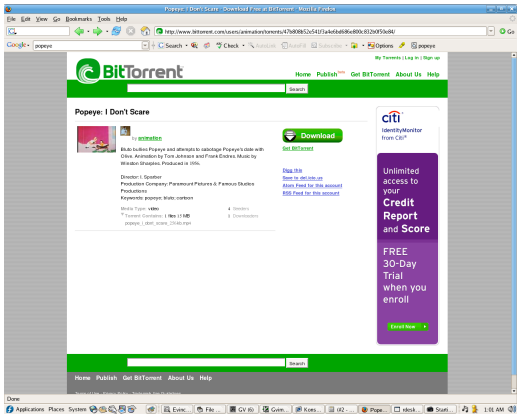
www.bittorrent.com

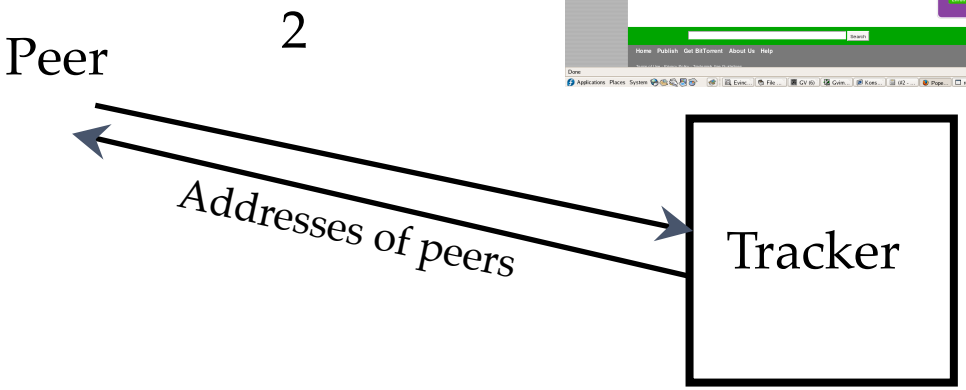


- File popeye.mp4.torrent hosted at a (well-known) webserver
- The .torrent has address of tracker for file
- The tracker, which runs on a webserver as well, keeps track of all peers downloading file

How a node enters a swarm for file "popeye.mp4"

www.bittorrent.com

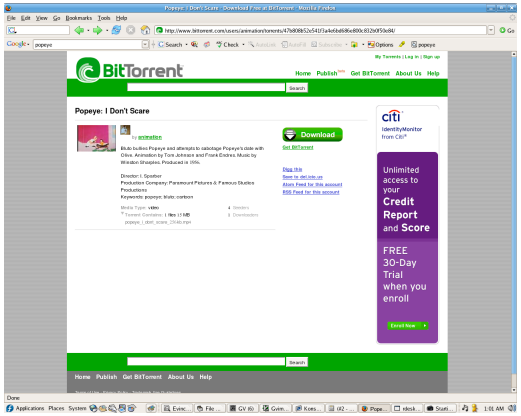


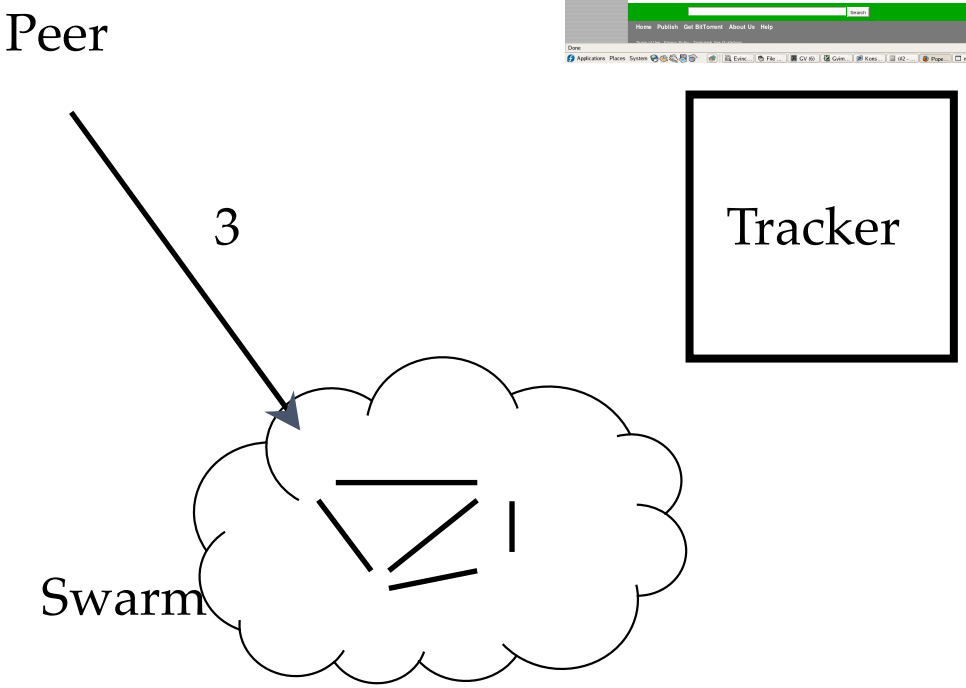


- File popeye.mp4.torrent hosted at a (well-known) webserver
- The .torrent has address of tracker for file
- The tracker, which runs on a webserver as well, keeps track of all peers downloading file

How a node enters a swarm for file "popeye.mp4"

www.bittorrent.com





- File popeye.mp4.torrent hosted at a (well-known) webserver
- The .torrent has address of tracker for file
- The tracker, which runs on a webserver as well, keeps track of all peers downloading file

Contents of .torrent file

- URL of tracker
- Piece length – Usually 256 KB
- SHA-1 hashes of each piece in file
  - For reliability
- “files” – allows download of multiple files

Terminology

- Seed: peer with the entire file
  - Original Seed: The first seed
- Leech: peer that's downloading the file
  - Fairer term might have been "downloader"
- Sub-piece: Further subdivision of a piece
  - The "unit for requests" is a subpiece
  - But a peer uploads only after assembling complete piece

Choosing pieces to request

- Rarest-first: Look at all pieces at all peers, and request piece that's owned by fewest peers
  - Increases diversity in the pieces downloaded
    - avoids case where a node and each of its peers have exactly the same pieces; increases throughput
  - Increases likelihood all pieces still available even if original seed leaves before any one node has downloaded entire file

Choosing pieces to request

- Random First Piece:
  - When peer starts to download, request random piece.
    - So as to assemble first complete piece quickly
    - Then participate in uploads
  - When first complete piece assembled, switch to rarest-first

Tit-for-tat as incentive to upload

- Tit for tat is a highly effective strategy in game theory for the iterated prisoner's dilemma.
- Want to encourage all peers to contribute
  - Unless provoked, the agent will always cooperate
  - If provoked, the agent will retaliate
  - The agent is quick to forgive
  - The agent must have a good chance of competing against the opponent more than once.
- *Optimistic unchoking* corresponds very strongly to always cooperating on the first move in prisoner's dilemma

Tit-for-tat as incentive to upload

- Peer *A* said to choke peer *B* if it (*A*) decides not to upload to *B*
- Each peer (say *A*) unchokes at most 4 *interested* peers at any time
  - The three with the largest upload rates to *A*
    - Where the tit-for-tat comes in
  - Another randomly chosen (Optimistic Unchoke)
    - To periodically look for better choices

Anti-snubbing

- A peer is said to be snubbed if each of its peers chokes it
- To handle this, snubbed peer stops uploading to its peers

➢ Optimistic unchoking done more often

- Hope is that will discover a new peer that will upload to us

Why BitTorrent took off

- Better performance through “pull-based” transfer
  - Slow nodes don’t bog down other nodes
- Allows uploading from hosts that have downloaded parts of a file
  - In common with other end-host based multicast schemes

Why BitTorrent took off

- Practical Reasons (perhaps more important!)
  - Working implementation (Bram Cohen) with simple well-defined interfaces for plugging in new content
  - Many recent competitors got sued / shut down
    - Napster, Kazaa
  - Doesn't do "search" per se. Users use well-known, trusted sources to locate content
    - Avoids the pollution problem, where garbage is passed off as authentic content

Pros and cons of BitTorrent

- Pros
  - Proficient in utilizing partially downloaded files
  - Discourages “freeloading”
    - By rewarding fastest uploaders
  - Encourages diversity through “rarest-first”
    - Extends lifetime of swarm
- Works well for “hot content”

Pros and cons of BitTorrent

- Cons
  - Assumes all interested peers active at same time; performance deteriorates if swarm “cools off”
  - Even worse: no trackers for obscure content

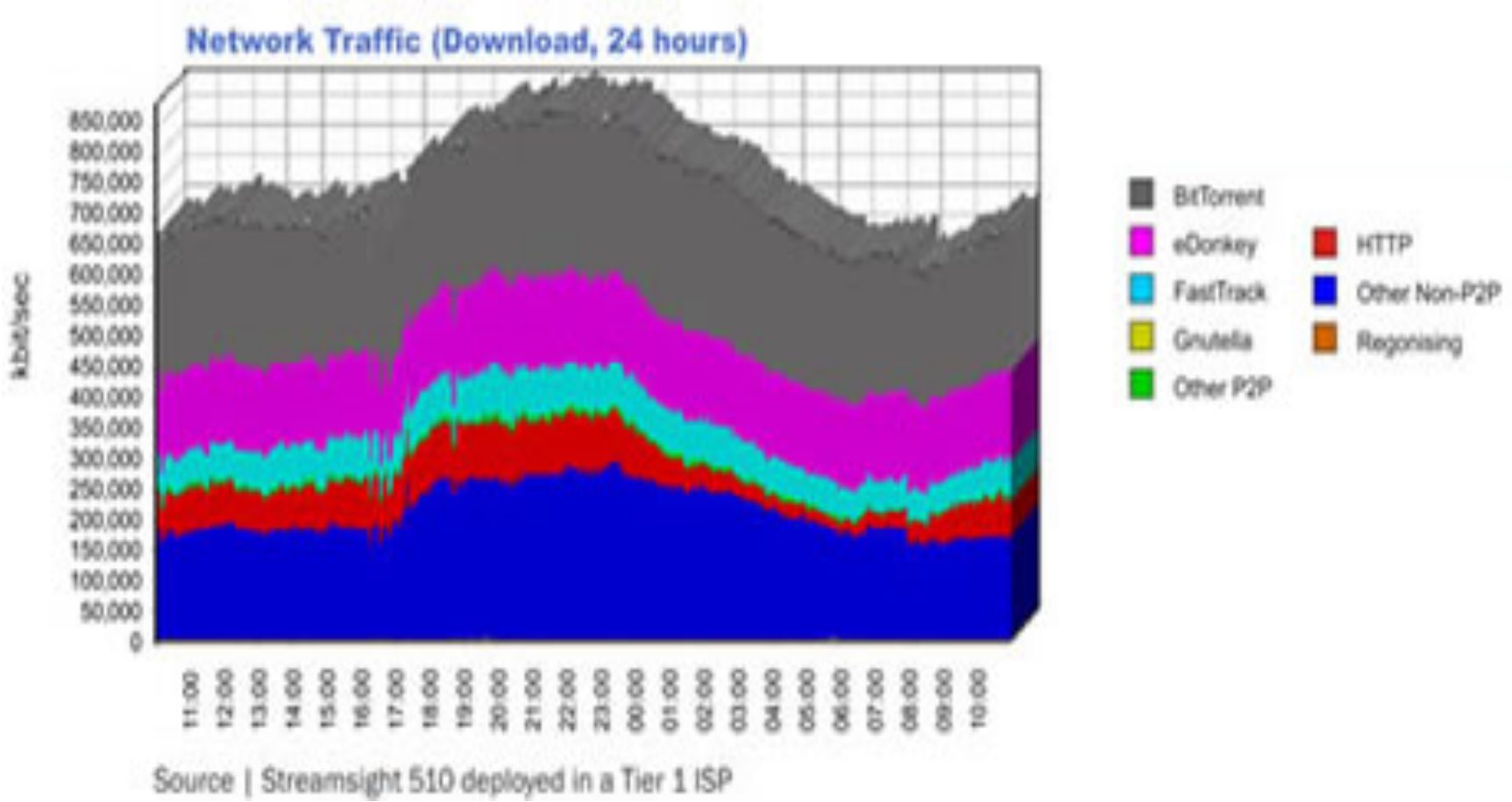
Pros and cons of BitTorrent

- Dependence on centralized tracker: pro/con?
  - ☹ Single point of failure: New nodes can't enter swarm if tracker goes down
  - Lack of a search feature
    - ☺ Prevents pollution attacks
    - ☹ Users need to resort to out-of-band search: well known torrent-hosting sites / plain old web-search

"Trackerless" BitTorrent

- To be more precise, "BitTorrent without a centralized-tracker"
- E.g.: Azureus
- Uses a Distributed Hash Table (Kademlia DHT)
- Tracker run by a normal end-host (not a web-server anymore)
  - The original seeder could itself be the tracker
  - Or have a node in the DHT randomly picked to act as the tracker

Why is (studying) BitTorrent important?



(From CacheLogic, 2004)

Why is (studying) BitTorrent important?

- BitTorrent consumes significant amount of internet traffic today
  - In 2004, BitTorrent accounted for 30% of all internet traffic (Total P2P was 60%), according to CacheLogic
  - Slightly lower share in 2005 (possibly because of legal action), but still significant
  - BT always used for legal software (linux iso) distribution too
  - Recently: legal media downloads (Fox)

Other file-sharing systems

- Prominent earlier: Napster, Kazaa, Gnutella
- Current popular file-sharing client: eMule
  - Connects to the ed2k and Kad networks
  - ed2k has a supernode-ish architecture (distinction between servers and normal clients)
  - Kad based on the Kademlia DHT

File-sharing systems…

- (Anecdotally) Better than BitTorrent in finding obscure items
- Vulnerable to:
  - Pollution attacks: Garbage data inserted with the same file name; hard to distinguish
  - Index-poisoning attacks (sneakier): Insert bogus entries pointing to non-existent files
  - Kazaa reportedly has more than 50% pollution + poisoning

References

- BitTorrent
  - “Incentives build robustness in BitTorrent”, Bram Cohen
  - BitTorrent Protocol Specification: http://www.bittorrent.org/protocol.html
  - http://en.wikipedia.org/wiki/BitTorrent_(protocol)
- Poisoning/Pollution in DHT’s:
  - “Index Poisoning Attack in P2P file sharing systems”
  - “Pollution in P2P File Sharing Systems”

Discussion on Unstructured P2P Systems

- Some scalability limits
- Excessive network bandwidth consumption for system management
- Hard to locate rare and unpopular objects
- More feasible to realize and for mass-usage than structured P2P networks
- Many networks take the form of a Random graph, others that of Scale-free networks (usually the hybrid ones)
- Mixing of structured for rare items and unstructured for popular items?

Algorithm Taxonomy	Unstructured P2P Overlay Network Comparisons				
	Freenet	Gnutella	FastTrack/KaZaA	BitTorrent	Overnet/eDonkey2000
Decentralization	Loosely DHT functionality.	Topology is flat with equal peers.	No explicit central server. Peers are connected to their Super-Peers.	Centralized model with a Tracker keeping track of peers.	Hybrid two-layer network composed of clients and servers.
Architecture	Keywords and descriptive text strings to identify data objects.	Flat and Ad-Hoc network of servents (peers). Flooding request and peers download directly.	Two-level hierarchical network of Super-Peers and peers.	Peers request information from a central Tracker.	Servers provide the locations of files to requesting clients for download directly.
Lookup Protocol	Keys,Descriptive Text String search from peer to peer.	Query flooding.	Super-Peers.	Tracker.	Client-Server peers.
System Parameters	None	None	None	*.torrent* file.	None
Routing Performance	Guarantee to locate data using *Key* search until the requests exceeded the Hops-To-Live (HTL) limits.	No guarantee to locate data; Improvements made in adapting Ultrapeer-client topologies; Good performance for popular content.	Some degree of guarantee to locate data, since queries are routed to the Super-Peers which has a better scaling; Good performance for popular content.	Guarantee to locate data and guarantee performance for popular content.	Guarantee to locate data and guarantee performance for popular content.
Routing State	Constant	Constant	Constant	Constant but choking (temporary refusal to upload) may occur.	Constant
Peers Join/Leave	Constant	Constant	Constant	Constant	Constant with bootstrapping from other peer and connect to server to register files being shared.
Security	Low; Suffers from man-in-middle and Trojan attacks.	Low; Threats: flooding, malicious content, virus spreading, attack on queries, and denial of service attacks.	Low; Threats: flooding, malicious or fake content, viruses, etc. Spywares monitor the activities of peers in the background.	Moderate; Centralized Tracker manage file transfer and allows more control which makes it much harder faking IP addresses, port numbers, etc.	Moderate; Similar threats as the FastTrack and BitTorrent.
Reliability/Fault Resiliency	No hierarchy or central point of failure exists.	Degradation of the performance; Peer receive multiple copies of replies from peers that have the data; Requester peers can retry.	The ordinary peers are reassigned to other Super-Peers.	The Tracker keeps track of the peers and availability of the pieces of files; Avoid Choking by fibrillation by changing the peer that is choked once every	Reconnecting to another server; Will not receive multiple replies from peers with available data.

Wireless sensor networks

- Usually only provide one-to-one communication facilities
- Have to build application level overlay
- Requirements on size of DHT and number of hops may be strict

university of groningen

Thank you.