ECS656U/ECS796P Distributed Systems

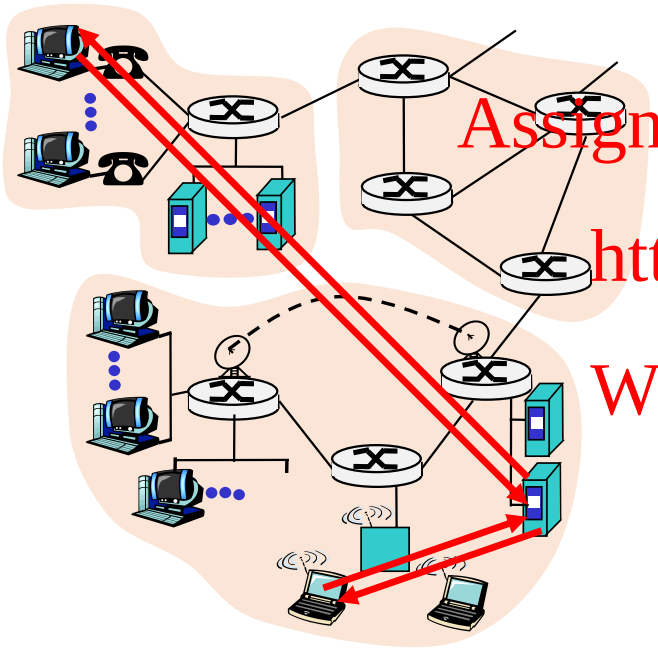
What this lecture is about

- Peer to Peer
- Distributed Hash Tables

Peer to Peer

Thanks to Prof. Marco Chiesa

Introduction: client-server architecture

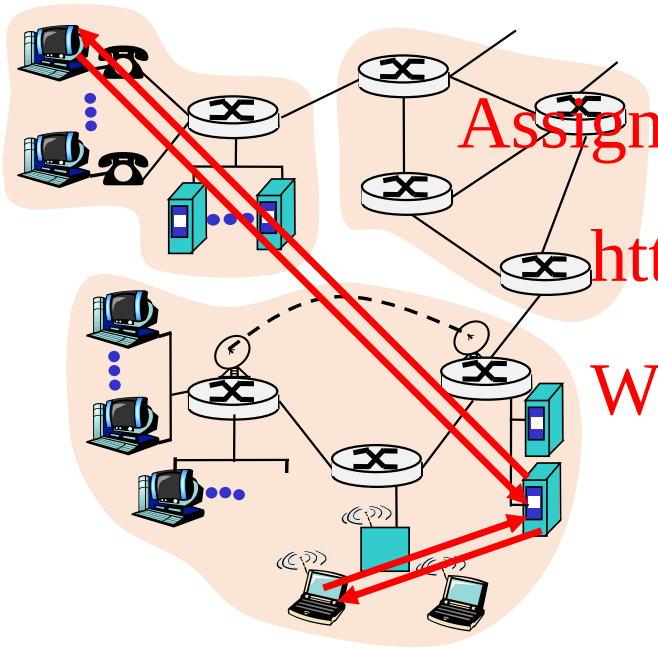


© J. Kurose and K. Ross, 1996-2006

<u>Server:</u>

- Always-on host
- Permanent IP address
- Server farms for scaling

Introduction: client-server architecture



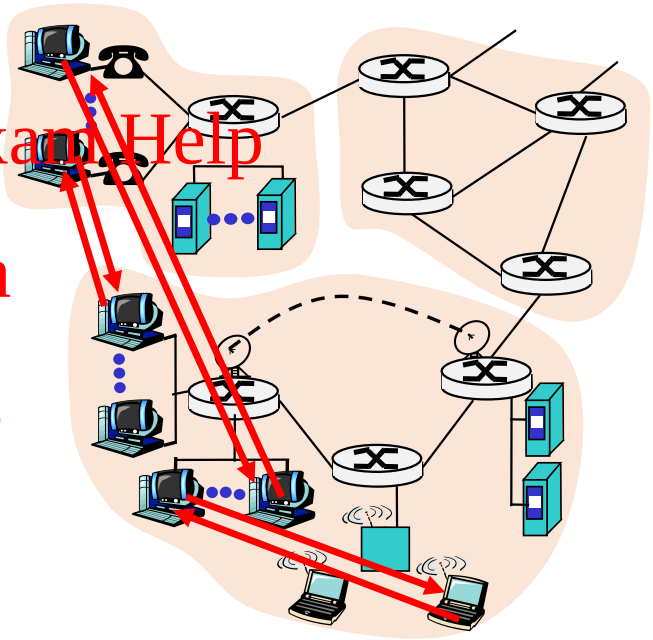
© J. Kurose and K. Ross, 1996-2006

Clients:

- May be intermittently connected
- May have dynamic IP addresses
- Communicate with server
- Do not communicate directly with each other

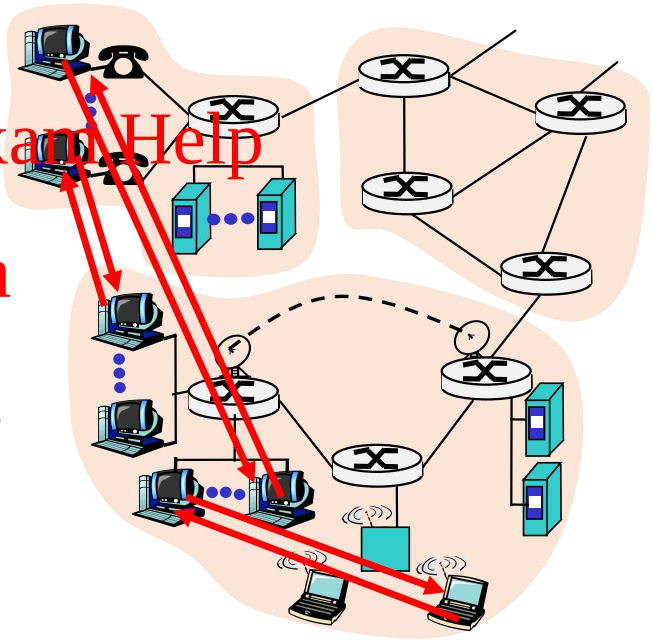
Introduction: peer-to-peer architecture

- No always-on server ("pure" P2P)
- Arbitrary end-systems communicate directly
- Peers are intermittently connected and change IP addresses – like clients



Introduction: peer-to-peer architecture

- P2P Examples:
  - **File sharing:** Napster, Gnutella, Kazaa, BitTorrent
  - **Streaming:** KanKan, Tribler
  - **VoIP:** Skype



Why Peer-to-peer (P2P)?

- Scaling: system scales with number of clients, by definition
- Eliminate centralization:
  - No single point of failure
  - No single point of control
- Self managing

P2P example

- Alice runs P2P client application on her notebook computer
- Intermittently connects to Internet; gets new IP address for each connection
- Asks for MP3 file “Hey Jude”
- Application displays other peers that have a copy of “Hey Jude”
- Alice chooses one of the peers, say Bob

P2P example

- File is copied from Bob's PC to Alice's notebook, e.g., by HTTP
- While Alice does this, other users download *from* Alice
  - E.g., some other MP3 file
- Alice's peer is both a Web client and a transient Web server

P2P for the win!

✓More users = more servers

✓Scalability!

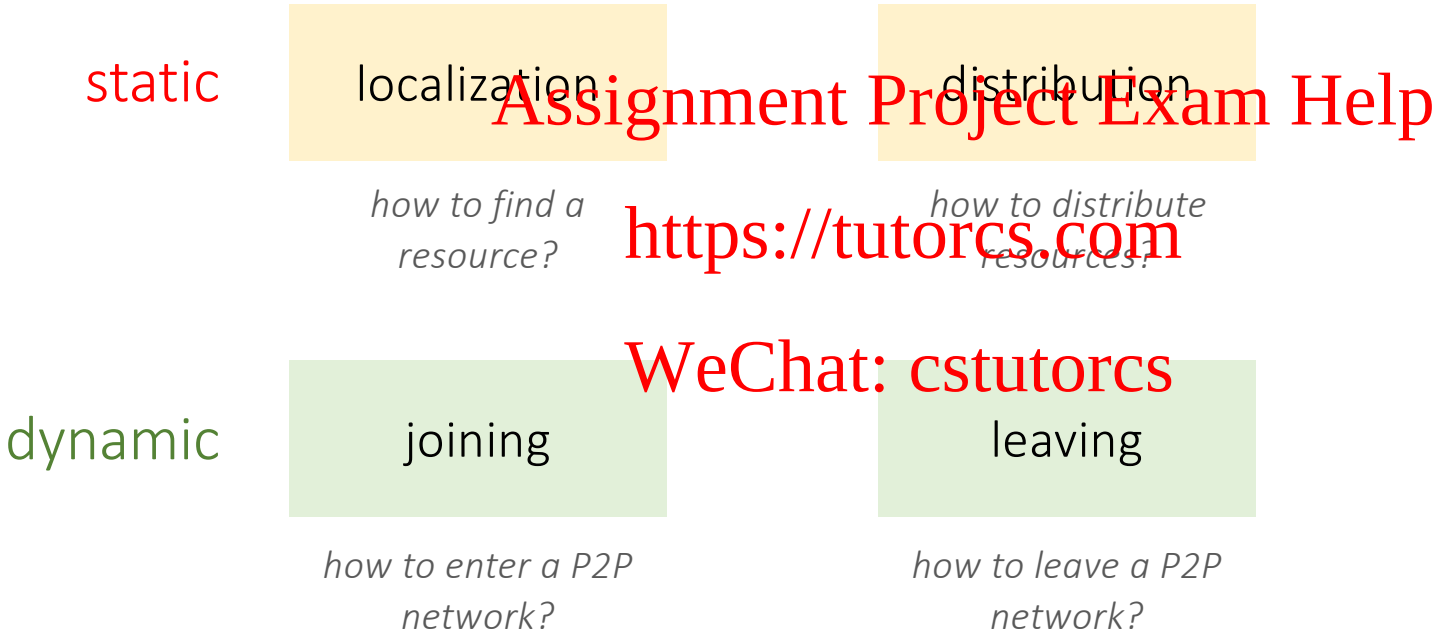
In short:

- **Client/Server:** Adding a user mean less resources per user
- **P2P:** Adding a user adds that peer's resources to the system

So, what's P2P have in common with this course?

- Typically, each member stores/provides access to content
- Basically, a **replication system for files**
  - P2P allows files to be anywhere -> searching is the challenge
  - Dynamic member list makes it more difficult

P2P fundamentals



Mapping Keys (files) and Resources to IP addresses

dynamic

joining

leaving

how to enter a P2P network?

how to leave a P2P network?

Locating content

- Locating content is often a two-step procedure
  - **What** content is available? I.e., can MP3 ”Hey Jude” be found at all?
  - **Who** (what peers) can provide the content?

- We need a directory with the content available!

Distributing content

- Central directory
  - Easy to manage ❤️
  - Bottleneck and single point of failure 💔
  - If content directory is overloaded or down, whole system suffers 💔

- Distributed directory
  - No bottlenecks, no single point of failures or oveloading ❤️
  - Hard to manage 💔
  - Harder the more peers and/or content you have in system 💔

Locating and distributing content

- Three basic architectures:
  - Centralized directory (Napster, early BitTorrent)
  - Query flooding (Gnutella)
  - Hierarchical and non-hierarchical overlay designs (Kazaa, BT DHT)

Locating and distributing content

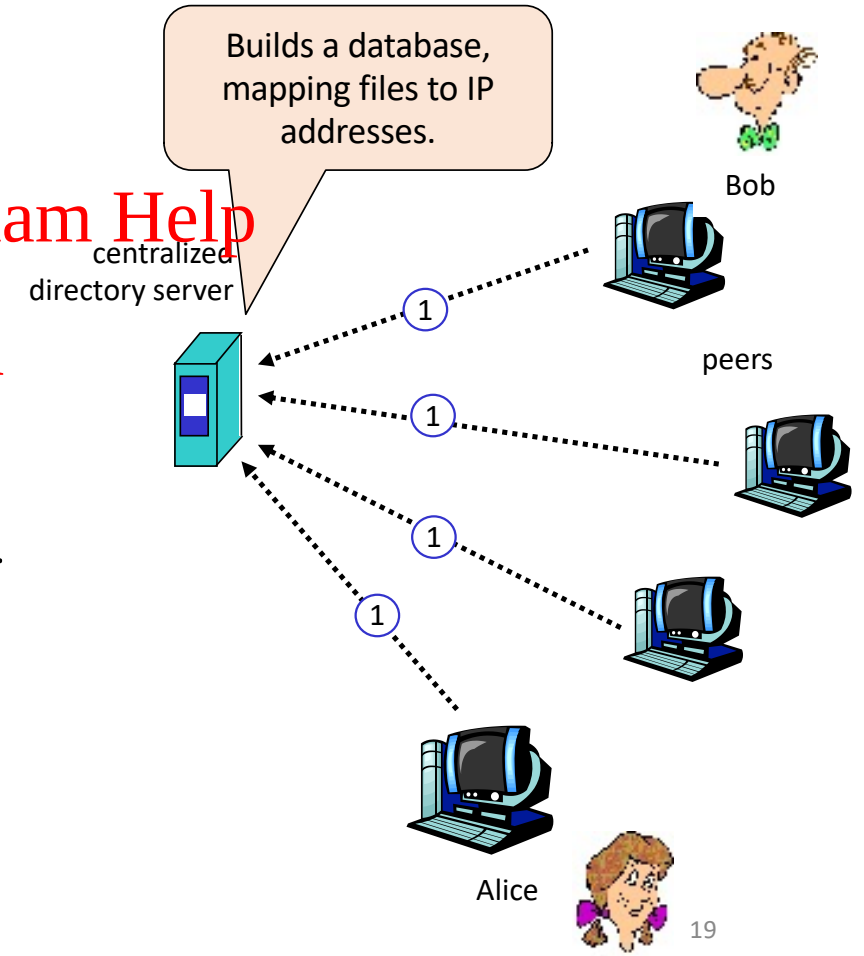
- Three basic architectures:
  - Centralized directory (Napster, early BitTorrent)
  - Query flooding (Gnutella)
  - Hierarchical and non-hierarchical overlay designs (Kazaa, BT DHT)

Napster

- Original “Napster” design
- Step 1: connection

1. Client upload a list of files that want to share
2. Server maintains a global list: <filename,host>

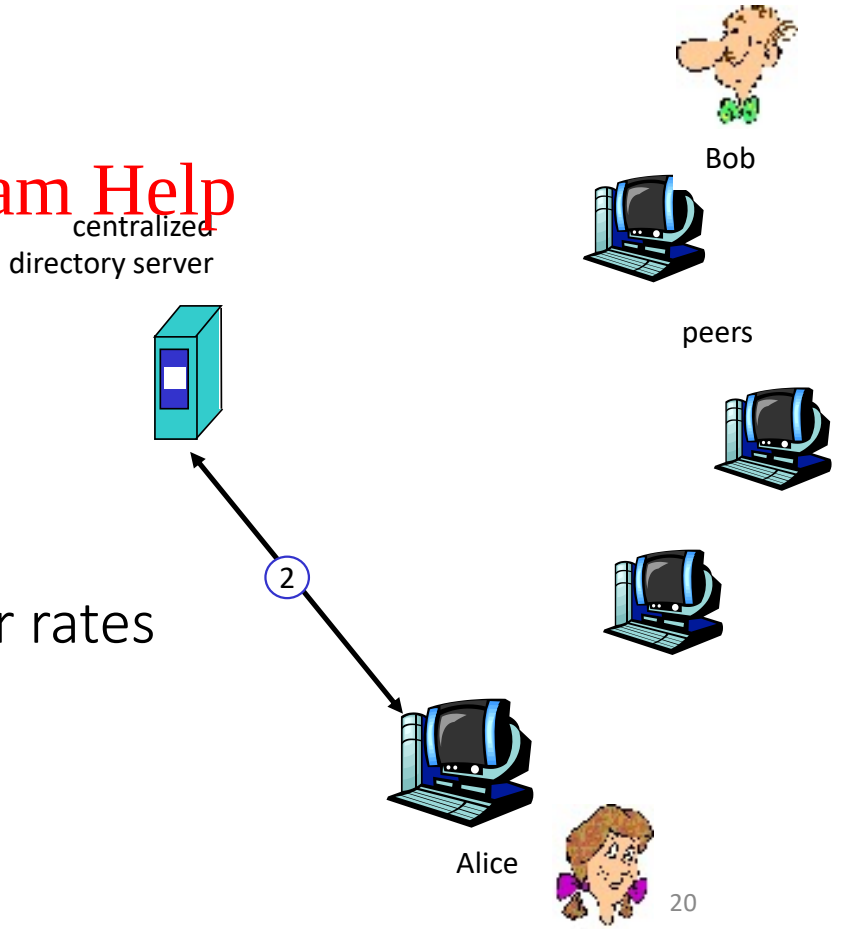
Note: Server does not store any file



Napster

- Original “Napster” design
- Step 1: connection
- Step 2: search

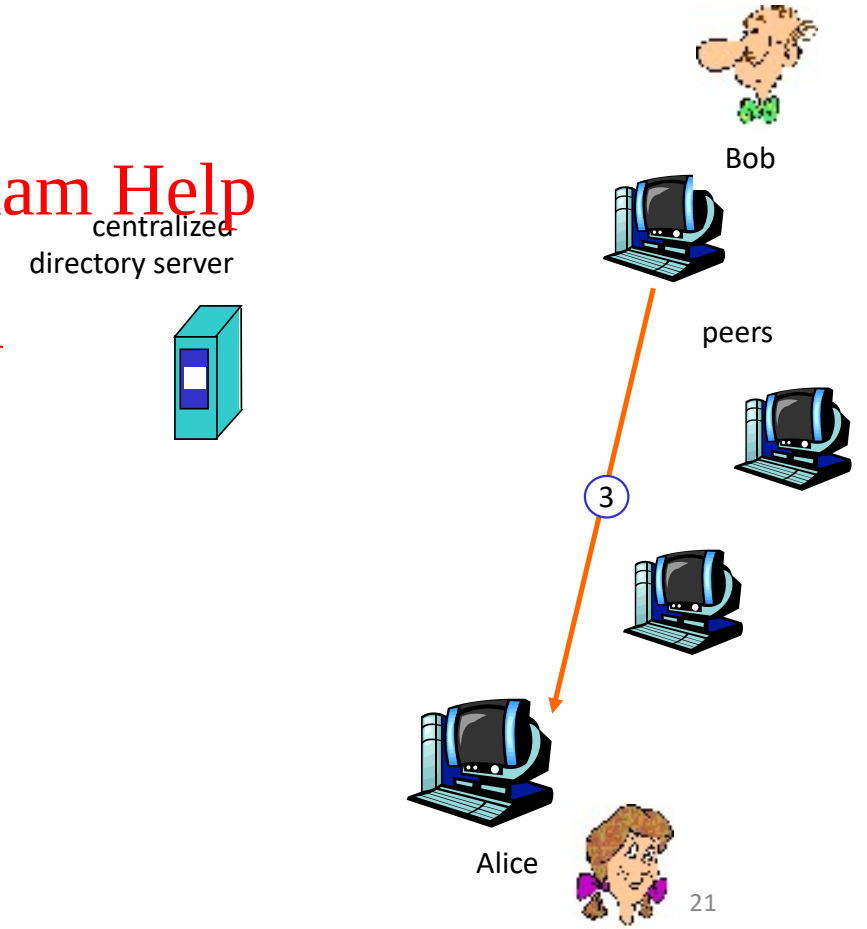
1. Client sends to server keywords to search with
2. Servers returns a list of hosts
3. Client pings each host in the list to find transfer rates
4. Client fetches file from best host



Napster

- Original “Napster” design
- Step 1: connection
- Step 2: search
- Step 3: download

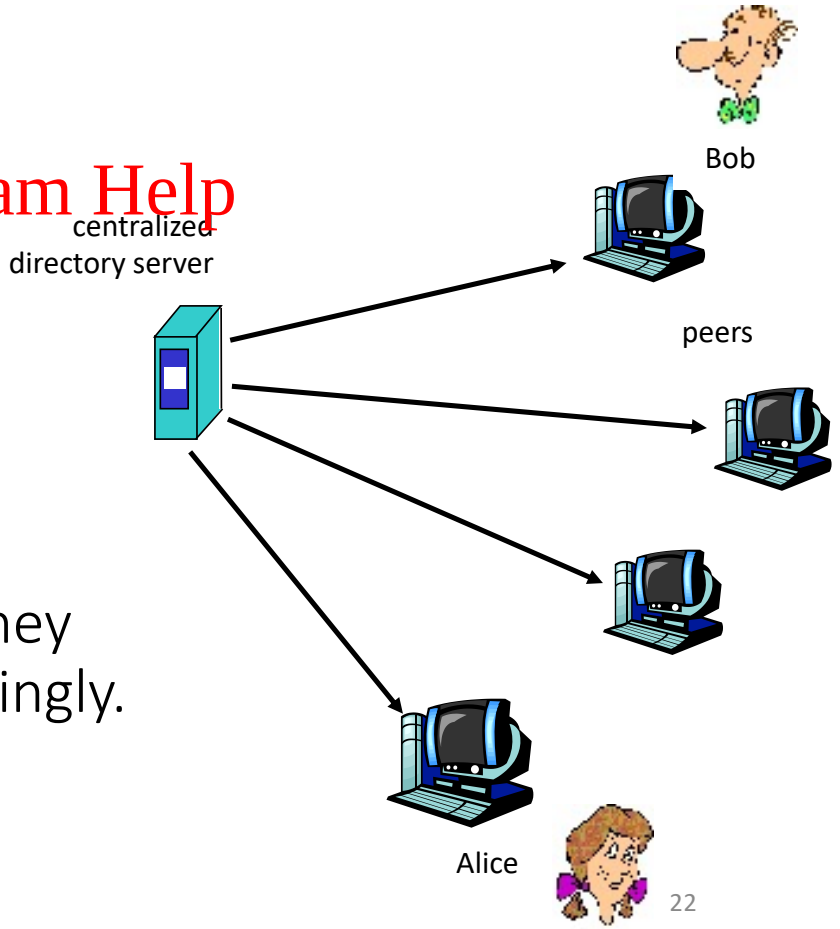
All communication uses TCP protocol



Napster

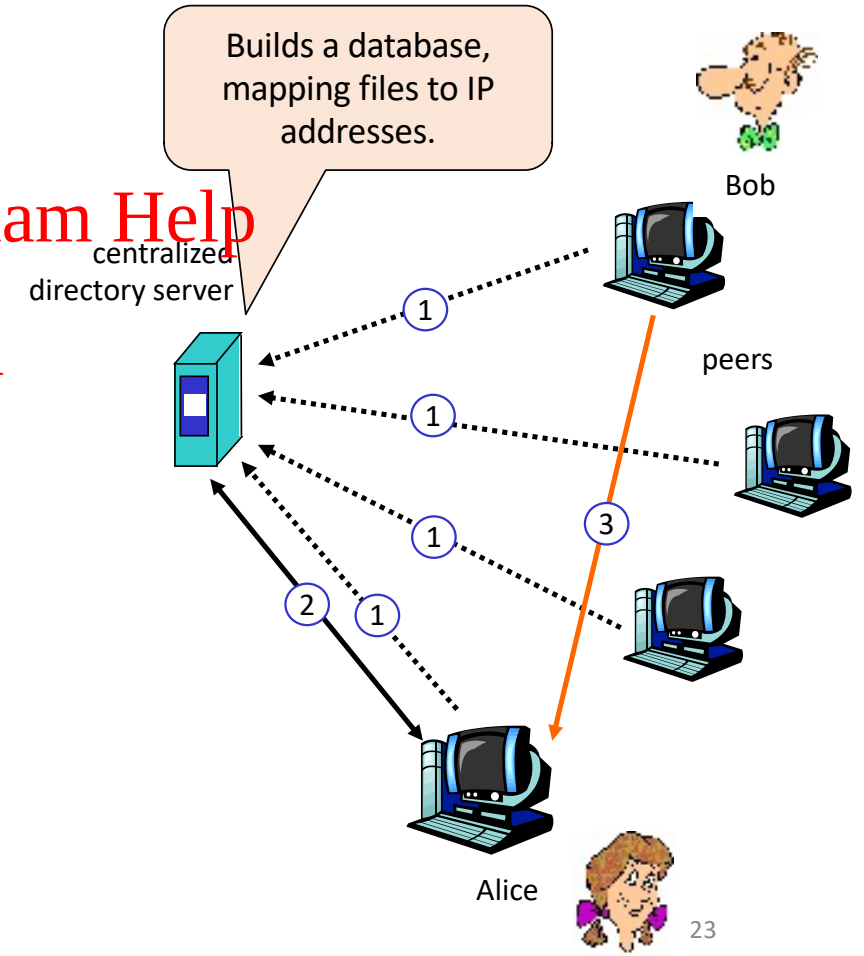
- Original “Napster” design
- Step 1: connection
- Step 2: search
- Step 3: download

Server can regularly probe peers to determine if they are still connected and update its database accordingly.



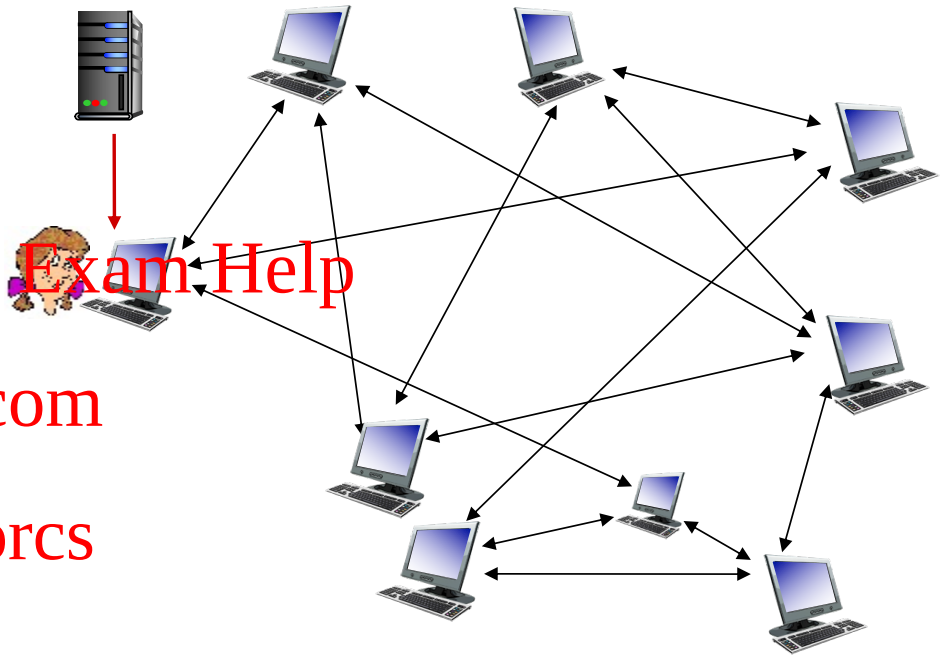
Napster

Can be seen as a hybrid between P2P and client-server



BitTorrent: overwiev

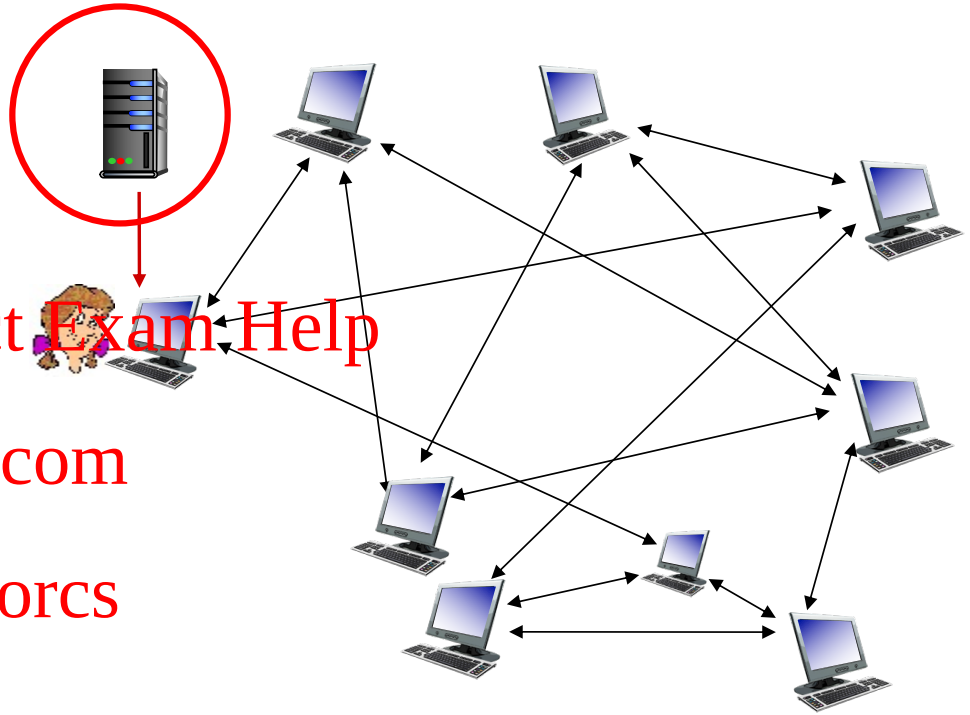
- Contact a centralized tracker that have a list of peers
- The torrent file contains the list of trackers associated to that specific file
- If Alice wants to join the torrent:
  - Register with the tracker
  - Tracker randomly selects peers and send to Alice the IP addresses



BitTorrent: the players

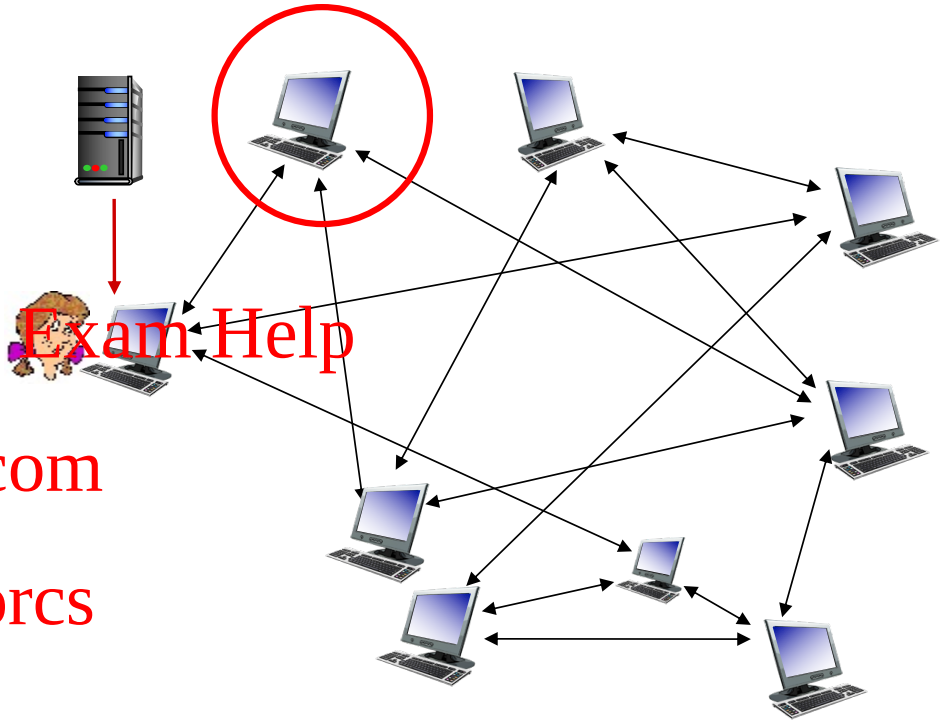
- Tracker: a special type of server.

It keeps track of where file copies reside on peer machines, which ones are available at time of the client request (it does so by receiving heartbeats from peers) and helps coordinate efficient transmission and reassembly of the copied file.



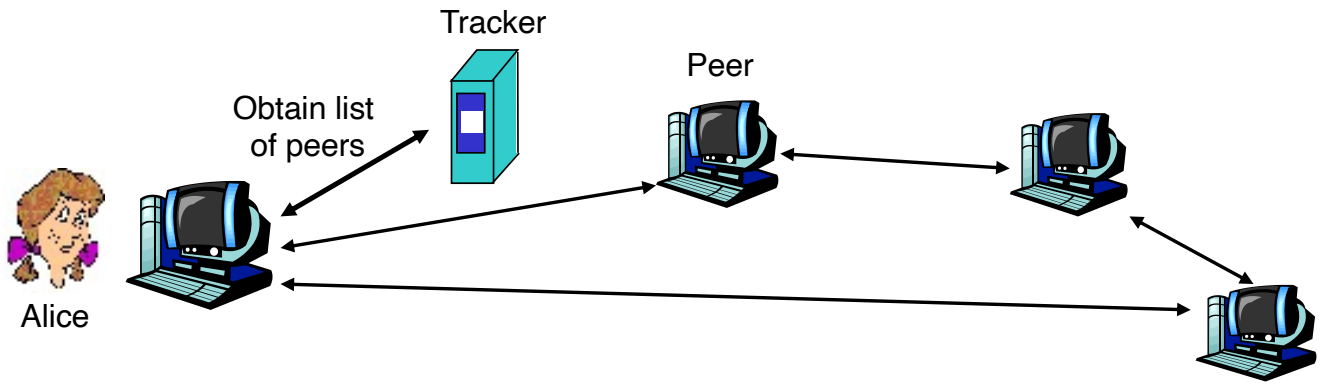
BitTorrent: the players

- Two kinds of peers
- Seed: has the full file. It continues to be part of the system and help the other peers to download the file
- Leecher: has some blocks from the file and it is still looking to download the other part of the file



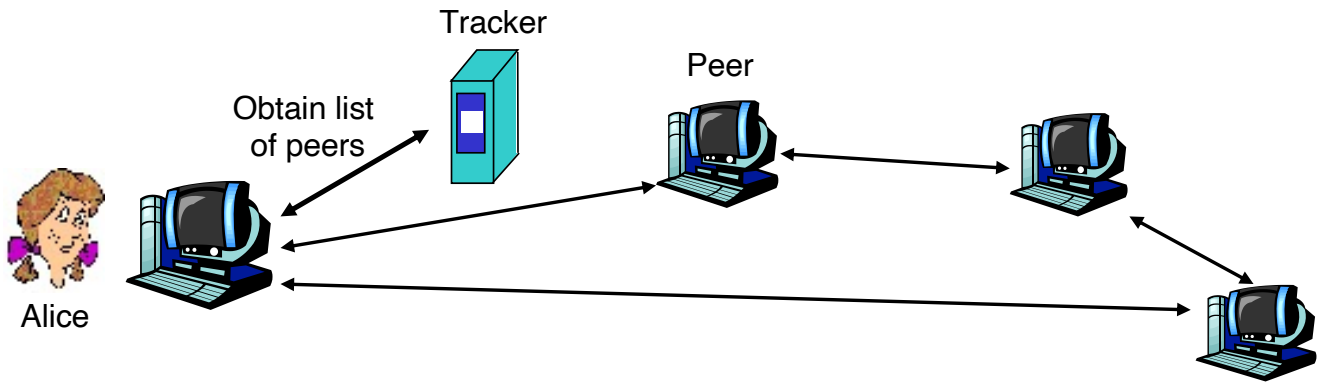
BitTorrent: operation

- Alice receives list of peer IP addresses from the Tracker
- Attempts to establish TCP connections to these peers
  - TCP-connected peers are “neighbouring peers”
- Neighbouring peers will fluctuate over time



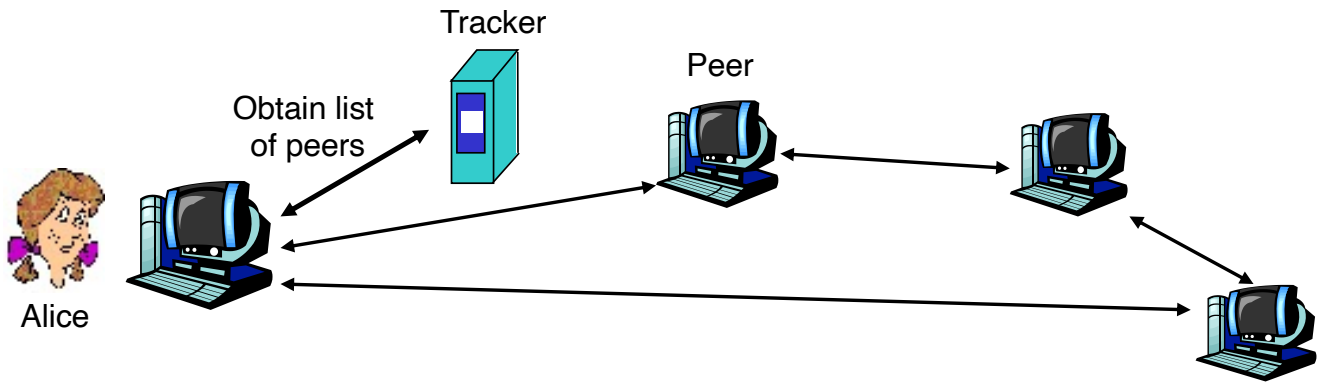
BitTorrent: operation

- File split into chunks (32KB – 256KB)
- Alice periodically asks neighboring peers for lists of chunks
- Alice will issue requests for chunks she is currently missing
- Alice will receive requests for chunks she has



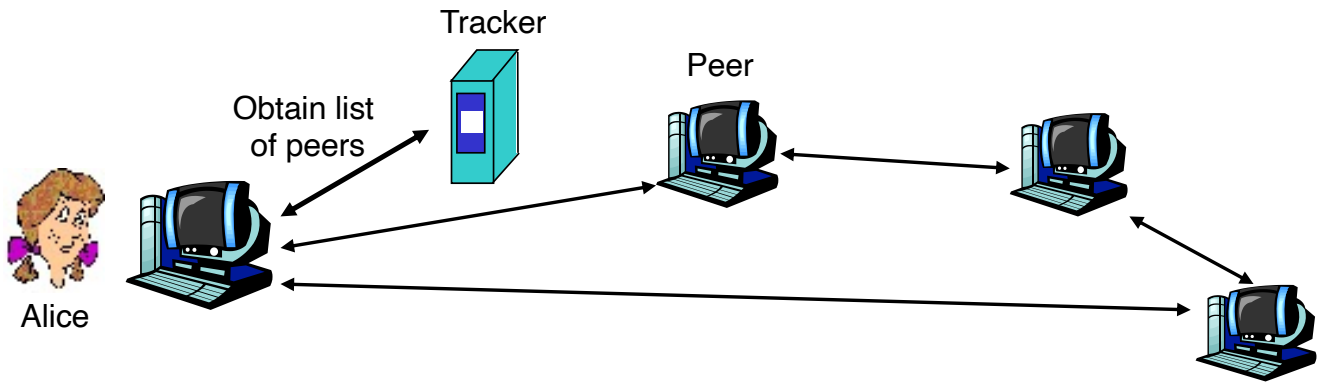
BitTorrent: operation

- At any given time, Alice has
  - A subset of chunks
  - Knowledge about chunks her neighboring peers have
- Two important decisions to make:
  - Which chunks should be requested next?
  - To which neighbor should she send requested chunks?



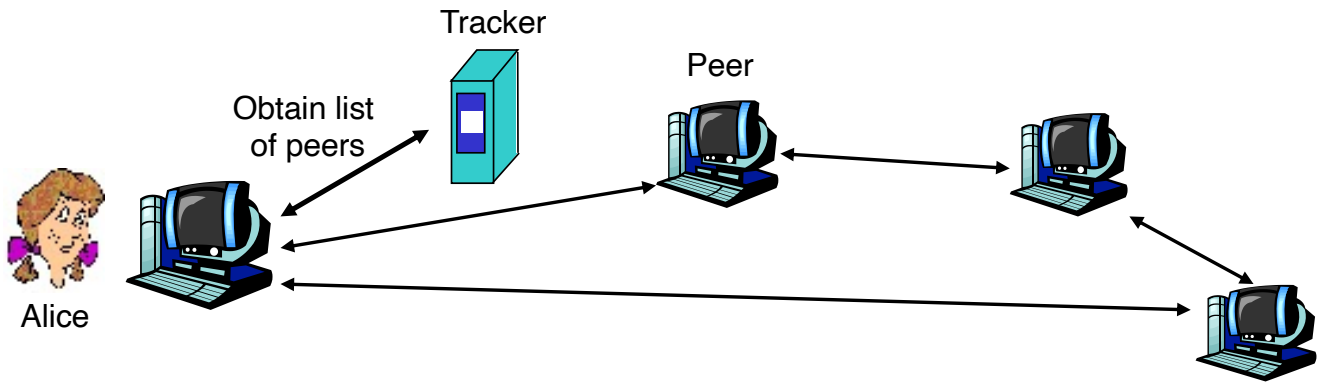
BitTorrent: operation

- Rarest chunks first
  - Determine chunks that are rarest among her neighbors
  - Request those rarest chunks first
  - Rarest chunks will then get more quickly redistributed
    - Equalizing the number of copies of each chunk in the torrent



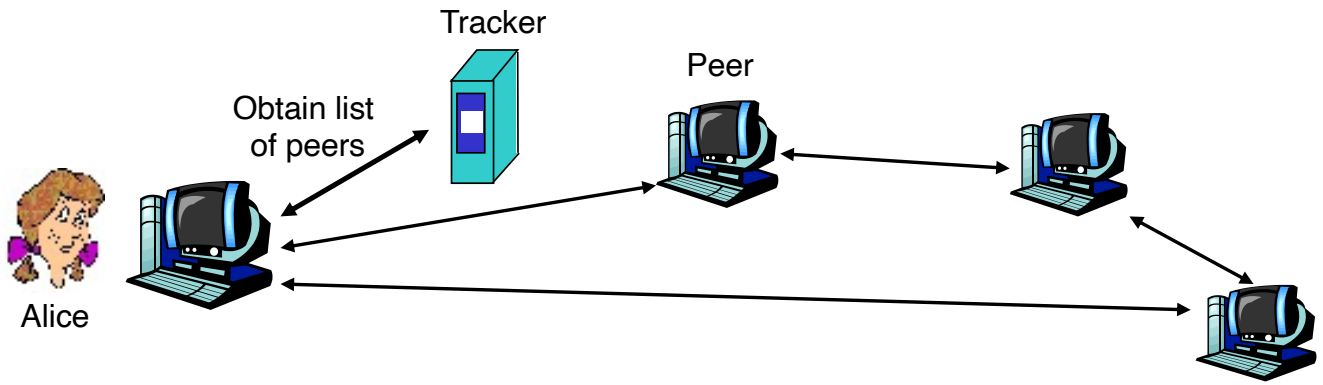
BitTorrent: operation

- At any given time, Alice has
  - A subset of chunks
  - Knowledge about chunks her neighboring peers have
- Two important decisions to make:
  - Which chunks should be requested next?
  - To which neighbor should she send requested chunks?



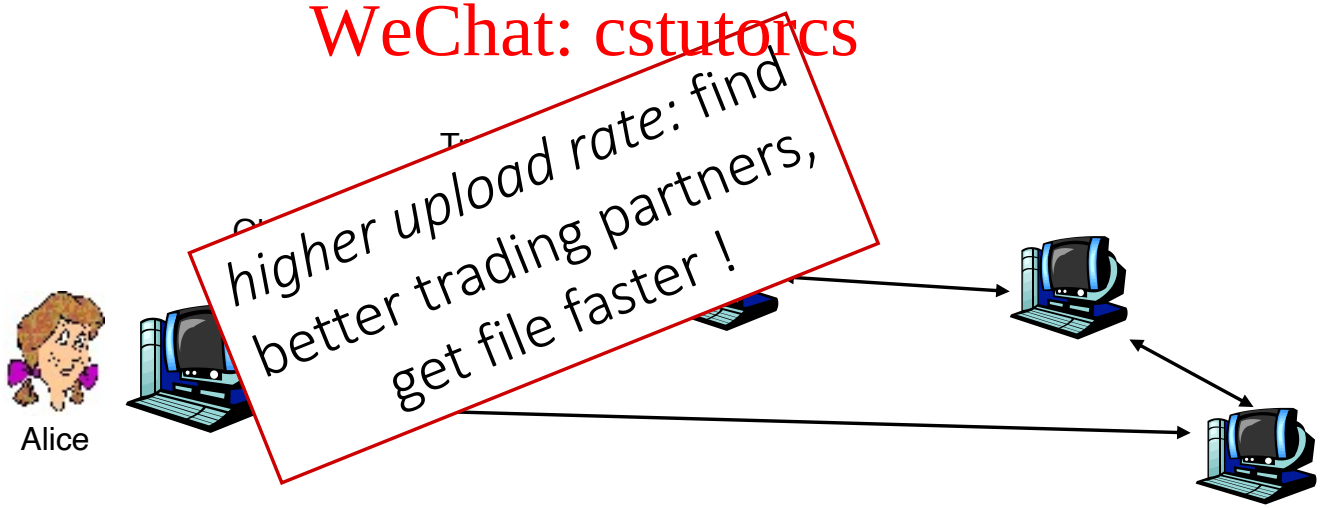
BitTorrent: operation

- Give priority to neighbors currently supplying her data *at the highest rate*
  - For each of her neighbors, Alice measures her receiving rate
  - Determine the four neighbors feeding her at highest rate
  - Return favor by sending chunks to the top four neighbors



BitTorrent: operation (tit-for-tat)

- Give priority to neighbors currently supplying her data *at the highest rate*
  - For each of her neighbors, Alice measures her receiving rate
  - Determine the four neighbors feeding her at highest rate
  - Return favor by sending chunks to the top four neighbors



BitTorrent: operation

- Every 10 seconds, recalculate the rates
  - Possibly modifying the set of top four peers
- Every 30 seconds, be optimistic!
  - Pick a new neighbor, Bob, at random and send it chunks
  - Alice may then become one of Bob's top four uploaders
  - Bob would then start sending to Alice and maybe become one of her top four uploaders

BitTorrent: operation

- Random neighbor selection →
  - Peers capable of uploading at compatible rates tend to find each other
  - New peers will get chunks so that they can start trading

BitTorrent compared to Napster

- Chunk based downloading

- Anti-freeloading mechanisms

Centralized directory: limitations

- Single point of failure
  - If server crashes, the entire P2P application crashes
- Performance bottleneck
  - Huge amount of users → huge database and many queries/second
- Directory size
  - When peers connect, they upload their content manifest (list of files)
  - As the number of peers increases, the resulting database does too

Centralized directory: limitations

- Single point of failure
  - If server crashe
- Performance bottl
  - Huge amount c s/second
- Directory size
  - When peers connect, they upload their content manifest (list of files)
  - As the number of peers increases, the resulting database does too

file transfer is **decentralized**, but locating content is highly **centralized**

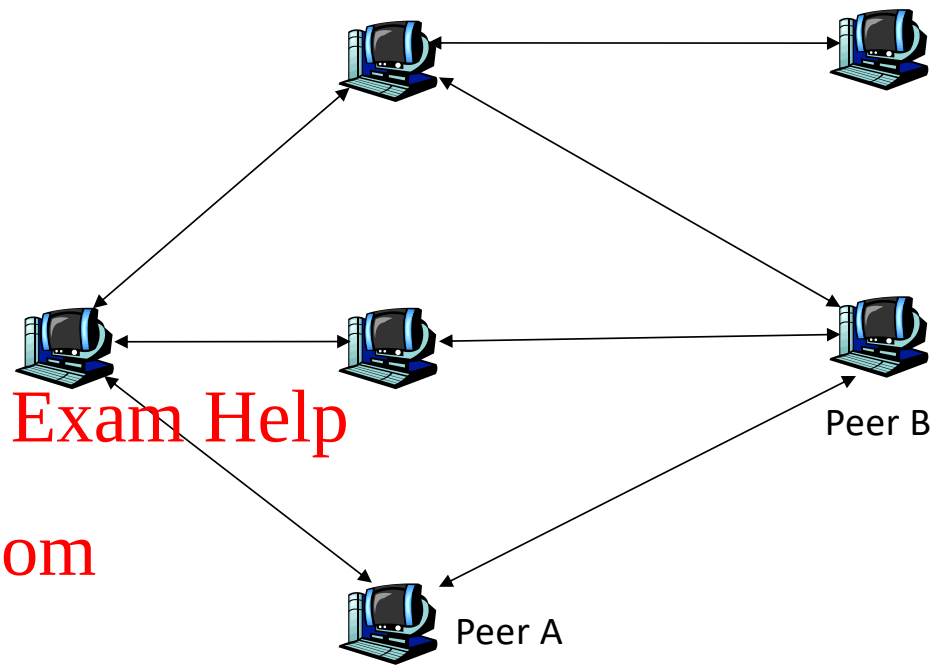
Locating and distributing content

- Three basic architectures:
  - Centralized directory (Napster, early BitTorrent)
  - Query flooding (Gnutella)
  - Hierarchical and non-hierarchical overlay designs (Kazaa, BT DHT)

Query flooding

- No central directory of either content or peers
- Search by sending query to all directly connected peers
- Peers forward query to their directly connected peers, etc. i.e., flooding network with queries.
- Some rules to limit reach, prevent loops etc. Still very costly.

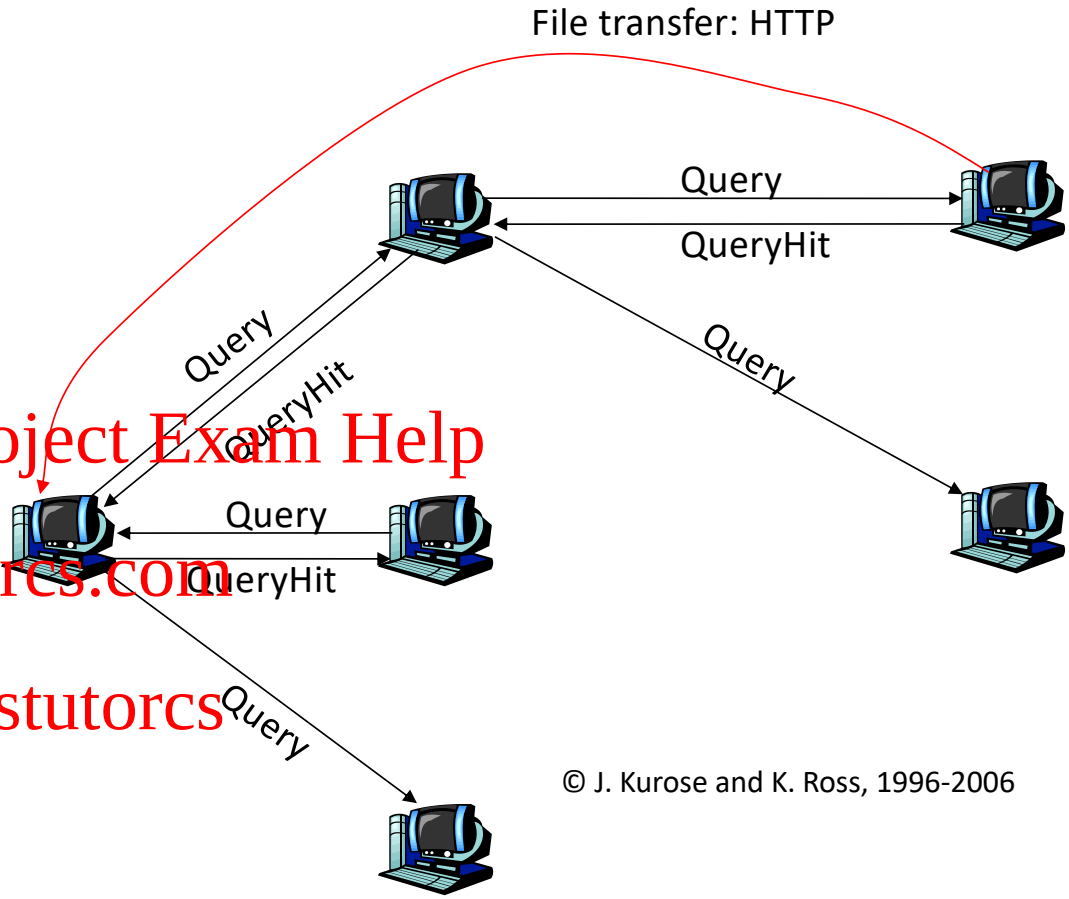
Gnutella



- Each Peer stores:
  - Their own files
  - Peer pointers (to the neighbour)
- Peers connected in a overlay network
- Each link is an implicit Internet path
- Peer A and Peer B are neighbour if they know their IP addresses and can send/receive messages

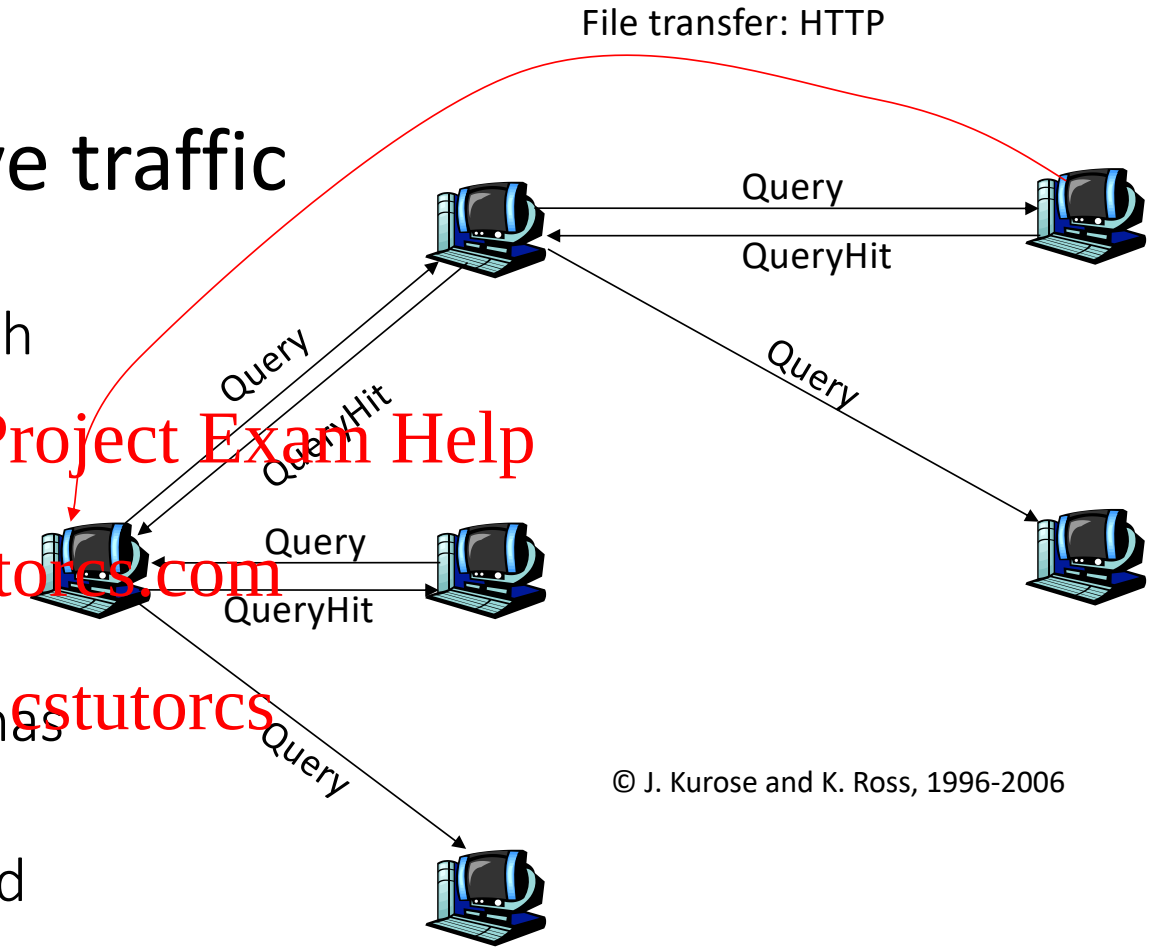
Gnutella

- How to find a file:
  - Send request to all neighbors (Query)
  - Neighbours recursively forward the request
  - Eventually a machine that has the file receives the request, and it sends back the answer (QueryHit)
  - Transfers are done with HTTP between peers. If several QueryHits, select one!



Gnutella: avoiding excessive traffic

- To avoid duplicate transmissions: each peer to maintain a list of recently received messages
- Query forwarded to all neighbours except from which received
- Each Query forwarded only once (it has an ID associated)
- TTL associated to each Query to avoid messages running forever in the network



Gnutella: how to join the network

- Joining peer X must find some other peer in the Gnutella network: use list of candidate peers that are often up (e.g., at a Gnutella site)
- X sequentially attempts to establish TCP connections with peers on list until connection setup with peer Y
- X sends Ping message to Y; Y forwards Ping message (peer-count field used - TTL)
- All peers receiving Ping message respond with Pong message back through the overlay

Gnutella: problems

- Ping-Pong constituted 50% traffic
- Popular files are requested many times: lots of Query
- Some peers might not have enough traffic for all Query/Pings/Pongs
- 70% of users in 2000 were freeloaders: just downloading never uploaded any file
- Flooding causes excessive traffic

Locating and distributing content

- Three basic architectures:
  - Centralized directory (Napster, early BitTorrent)
  - Query flooding (Gnutella)
  - Hierarchical and non-hierarchical overlay designs (Kazaa, BT DHT)

KaZaA

- **Basic idea:** leverage heterogeneity of peers-> assign more responsibility to peers with better resources-> impose a hierarchy

- Kazaa borrows ideas from Napster and Gnutella

  - Like in Gnutella, there is no dedicated server (or server farm) for tracking and locating content

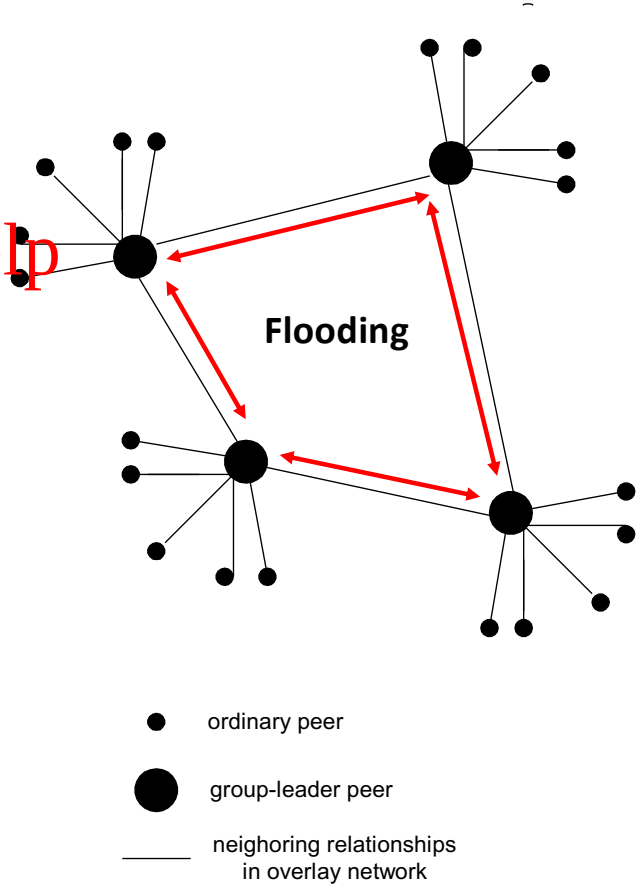
  - Unlike Gnutella, all peers are not equal in Kazaa—group leaders (or super peers) exist

KaZaA

- Group leaders maintain a database of content and IP addresses in a Napster-like fashion (group leader is a Napster-like hub)
  - In contrast to Napster, a group leader is not a dedicated server

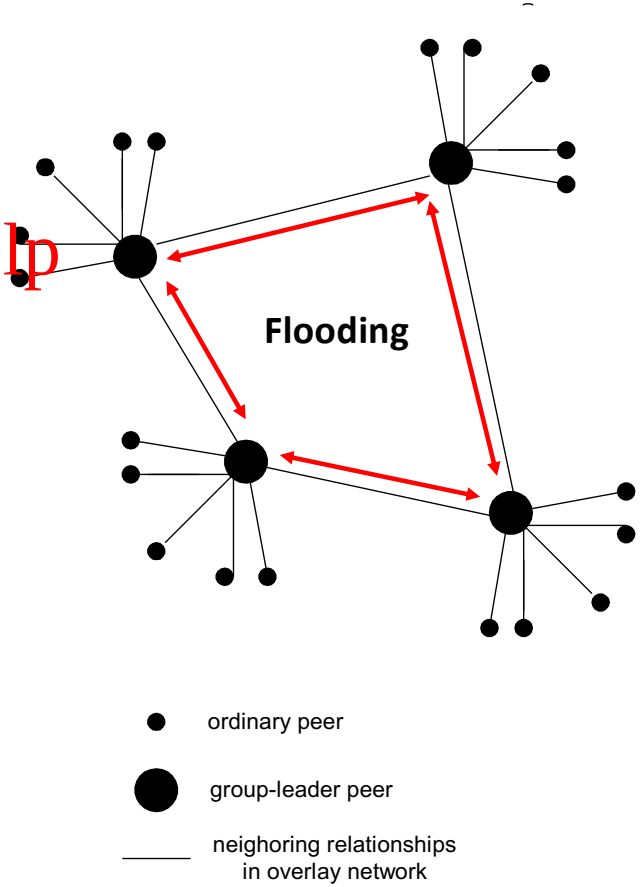
KaZaA: hierarchical overlay network

- Each peer is either a group leader or assigned to a group leader
- Group leader tracks the content in all its children
- Group leaders have high bandwidth connections and high Internet connectivity
- A hierarchical overlay network
  - Flooding limited to overlay of super peers



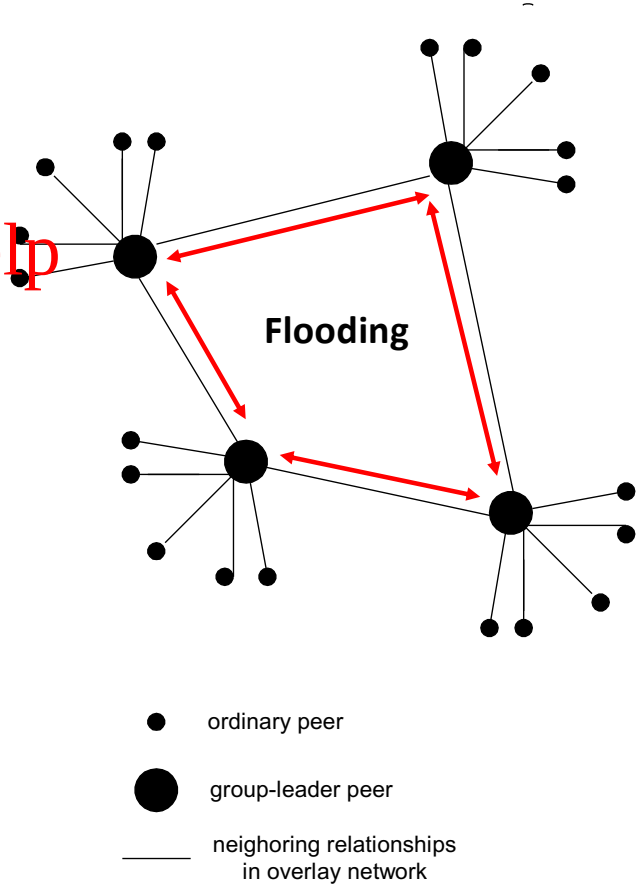
KaZaA: operation

- Publish:
  - Node inform group-leader of the list of files it wants to share
- Search:
  - Node queries group-leader. If file not found, group-leader flood the query to a subset of group-leaders
  - Group-leader responds to node
- Fetch:
  - Node get file directly from peers

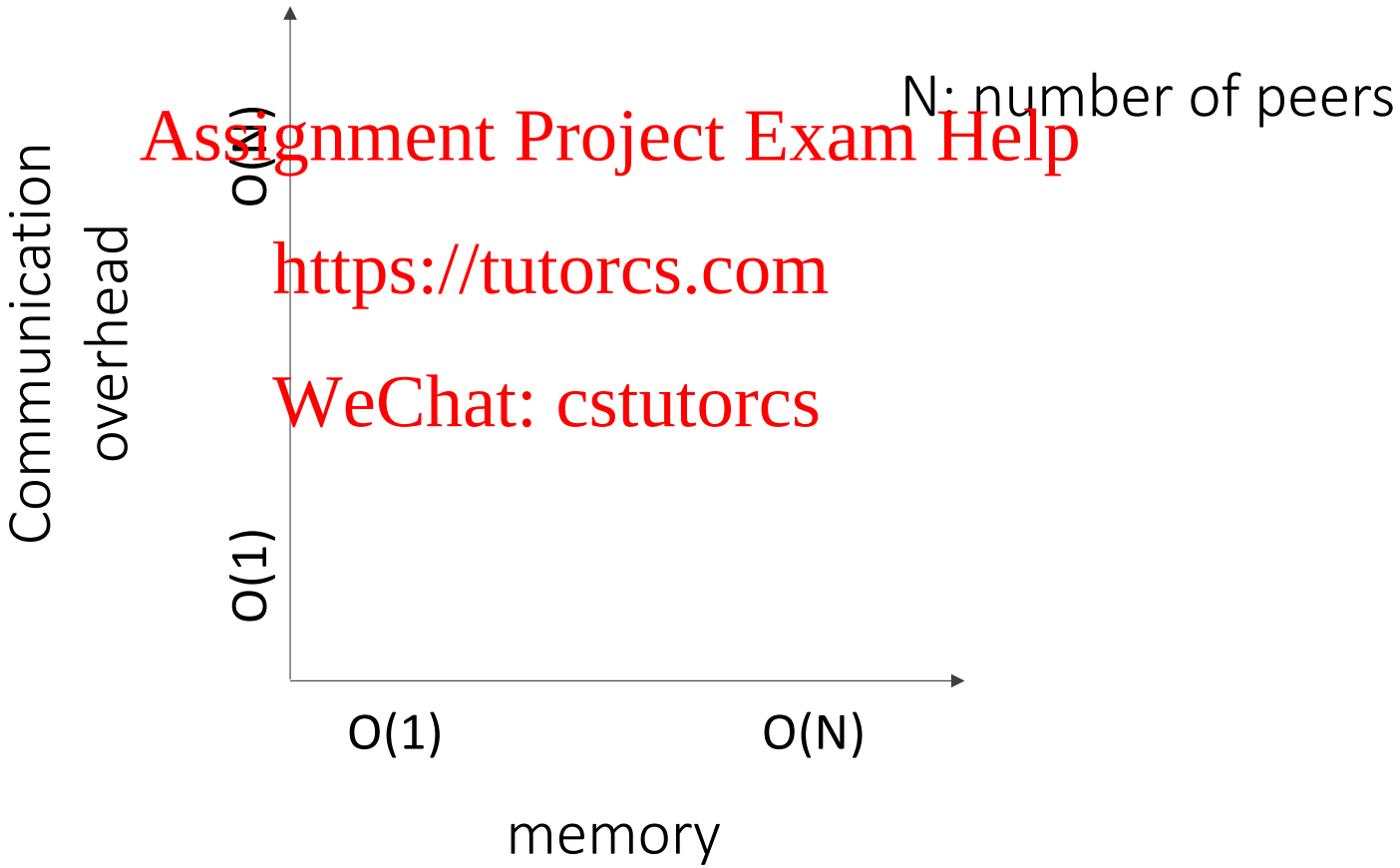


KaZaA: pros and cons

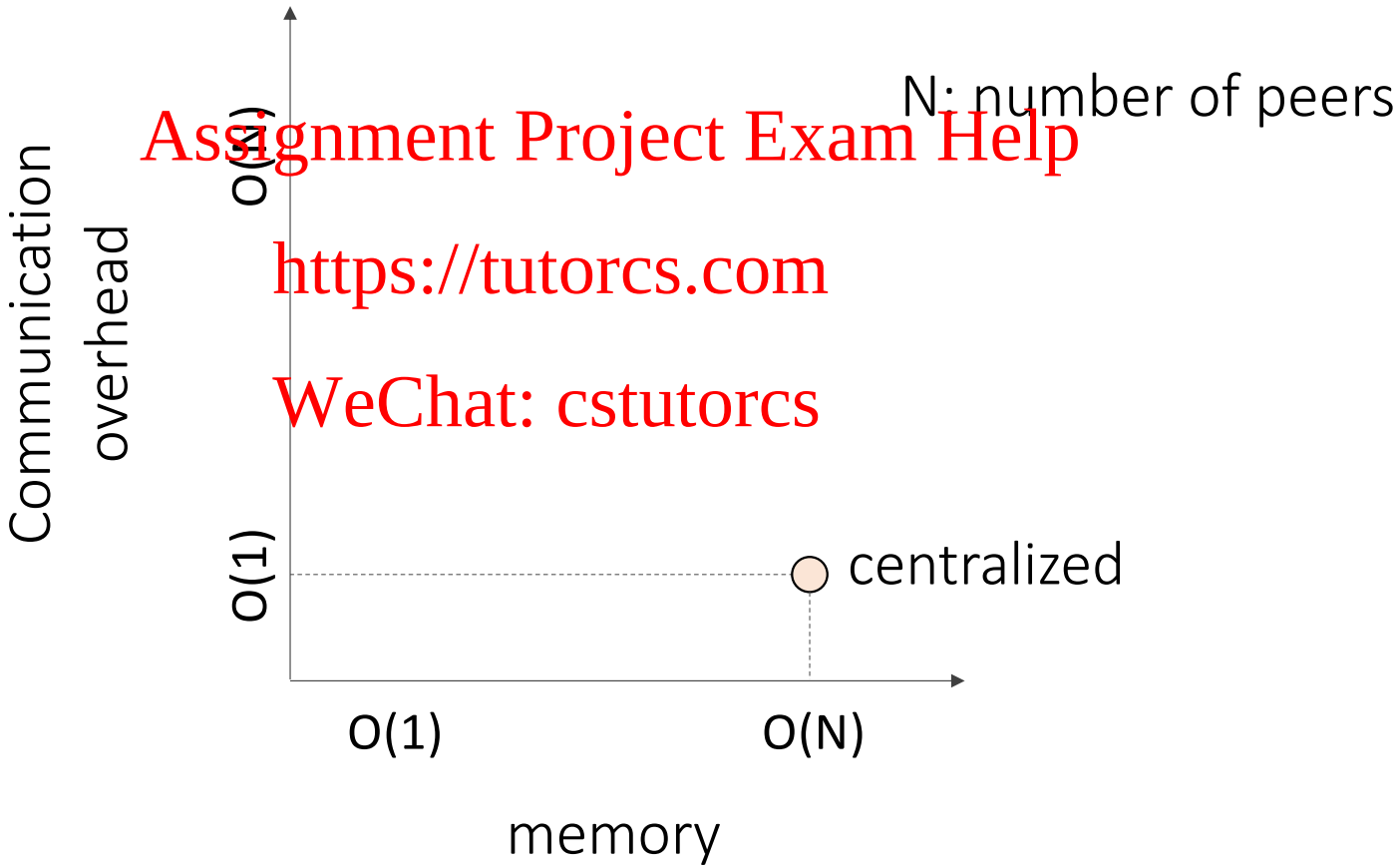
- Pros
  - Improves scalability and search efficiency by exploiting node heterogeneity
- Cons
  - No guarantees on search time
  - No mechanism to tackle freeloading



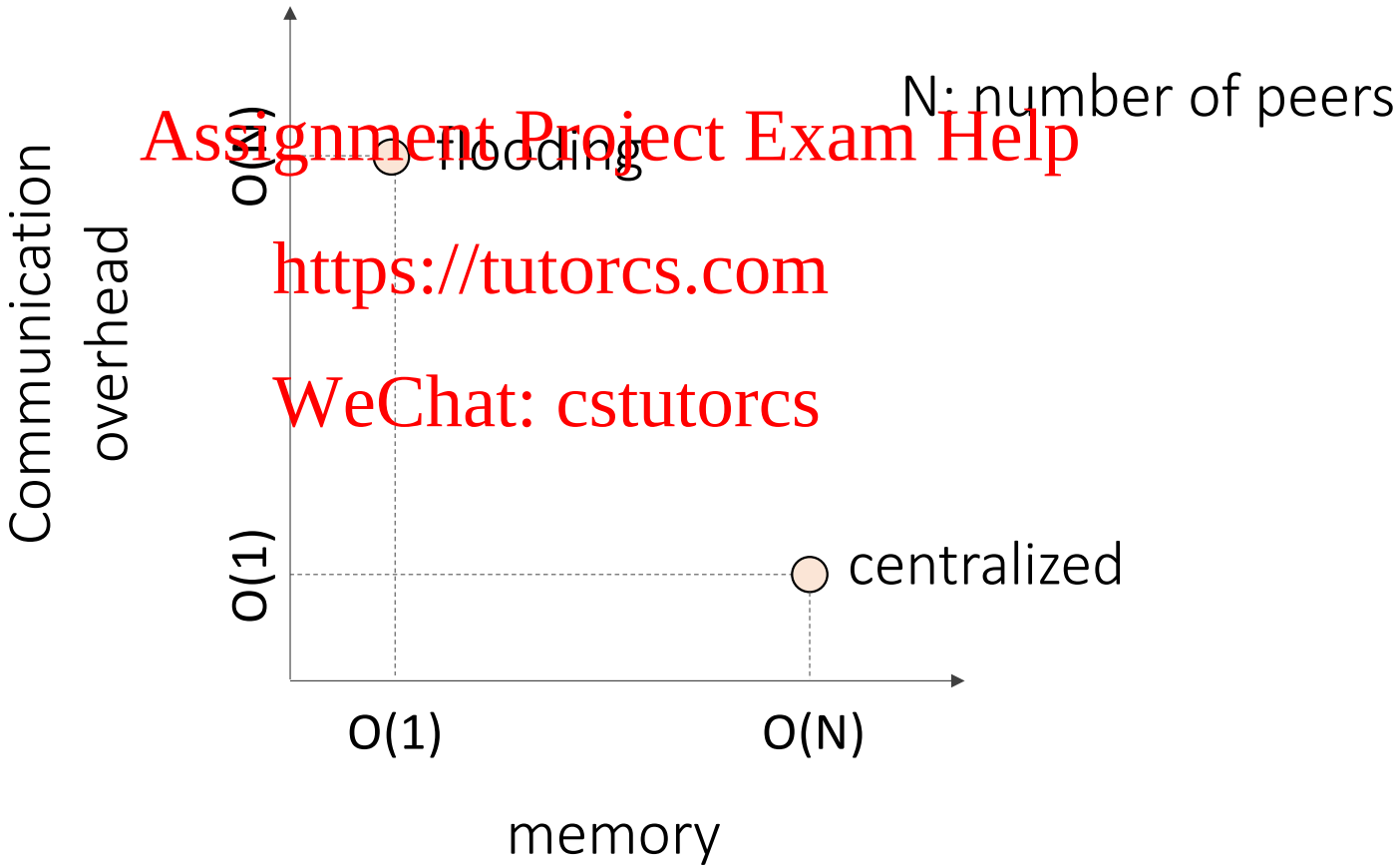
The lookup complexity graph



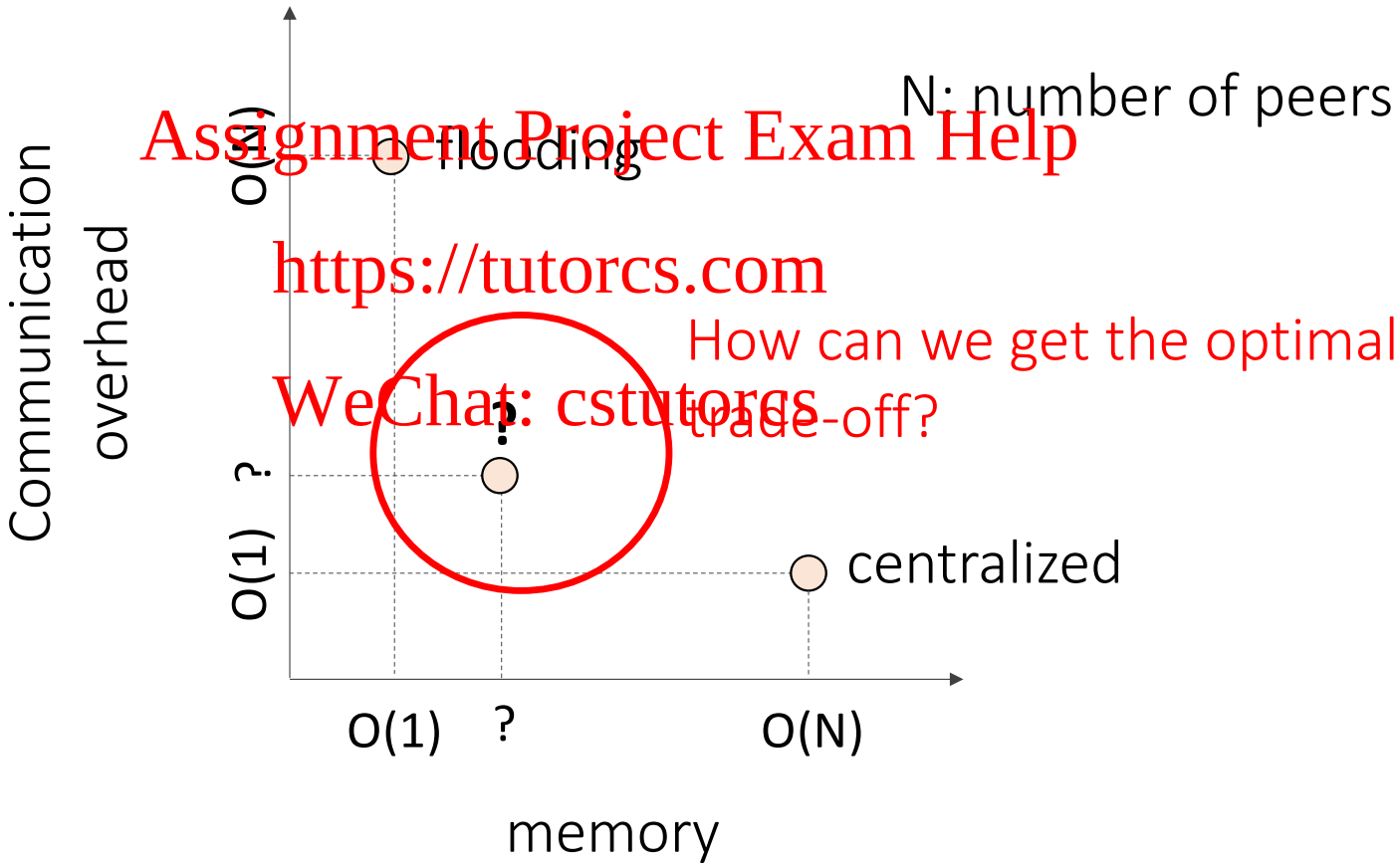
The lookup complexity graph



The lookup complexity graph

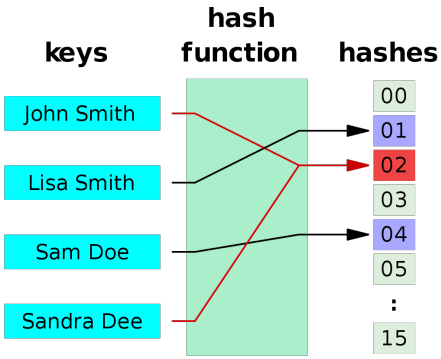


The lookup complexity graph



How can we get the optimal trade-off?

Distributed Hash Table (DHT)



Hash function

- Turns data into a small number (a "fingerprint")
- Input → Hash function → Hash (or hash sum, or hash value)
- Choosing a good hash function can be tricky

- Hash table
  - a structure that can map keys to values. It uses a hash function to compute an index into an array, from which the desired value can be found

Distributed Hash Table (DHT)

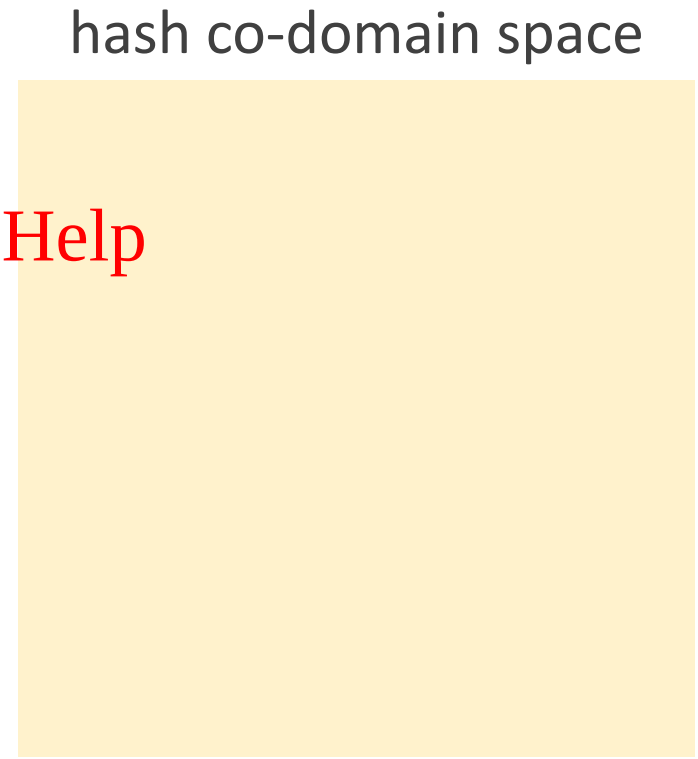
- Widely adopted in P2P systems
  - Creates a fully decentralized index that maps file IDs to locations
  - Allows a user to determine (basically) all the locations of a file without generating an excessive amount of search traffic

DHT for locating and distributing content

Can we cleverly distribute the mapping so that finding the resource is easy?

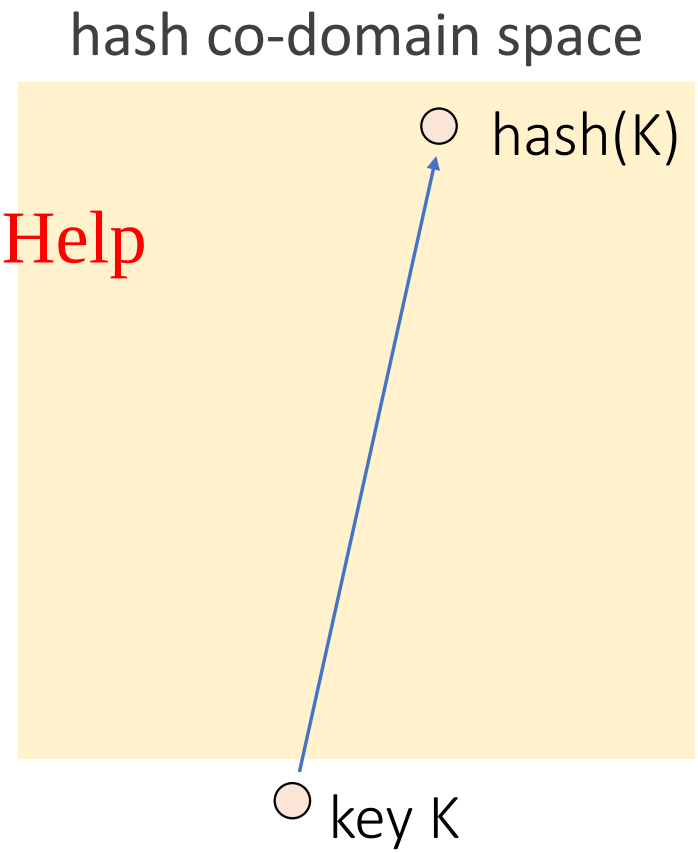
DHT basics

The hash function maps a resource-ID to a point in the hash output space



DHT basics

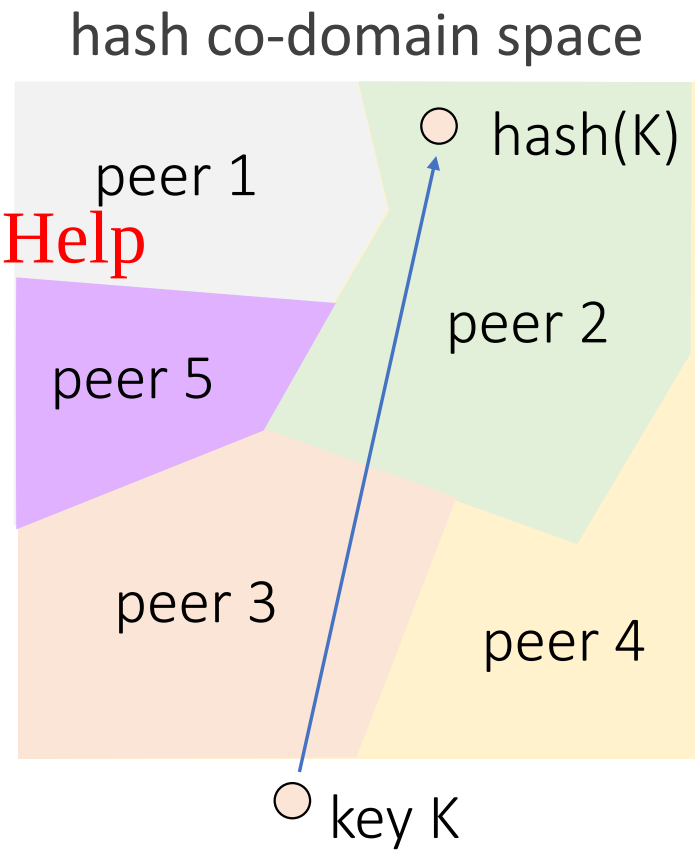
The hash function maps a resource-ID to a point in the hash output space



DHT basics

The codomain of the hash function is split among the peers

- each peer is responsible to map a hashed ID to the IP address that can provide the content

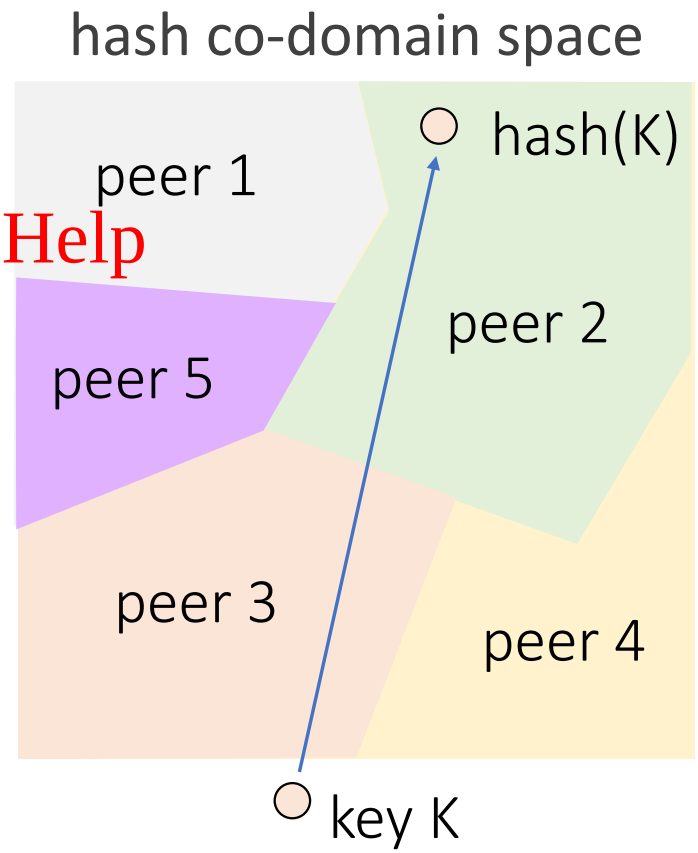


the peer does not store the resource, just the mapping!

DHT basics

The codomain of the hash function is split among the peers

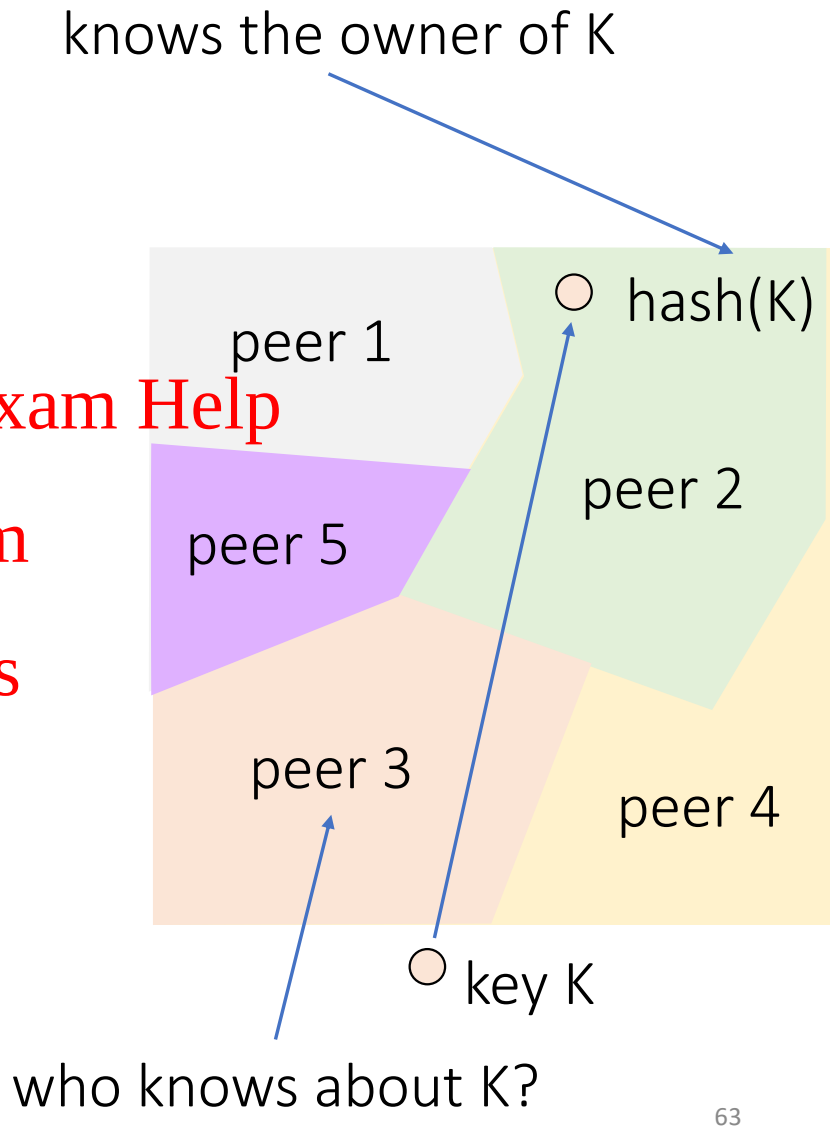
- to fetch a resource, the client must contact the correct peer
- the hash table maps keys to the peer that is responsible for that key



DHT basics

This is challenging!!!!

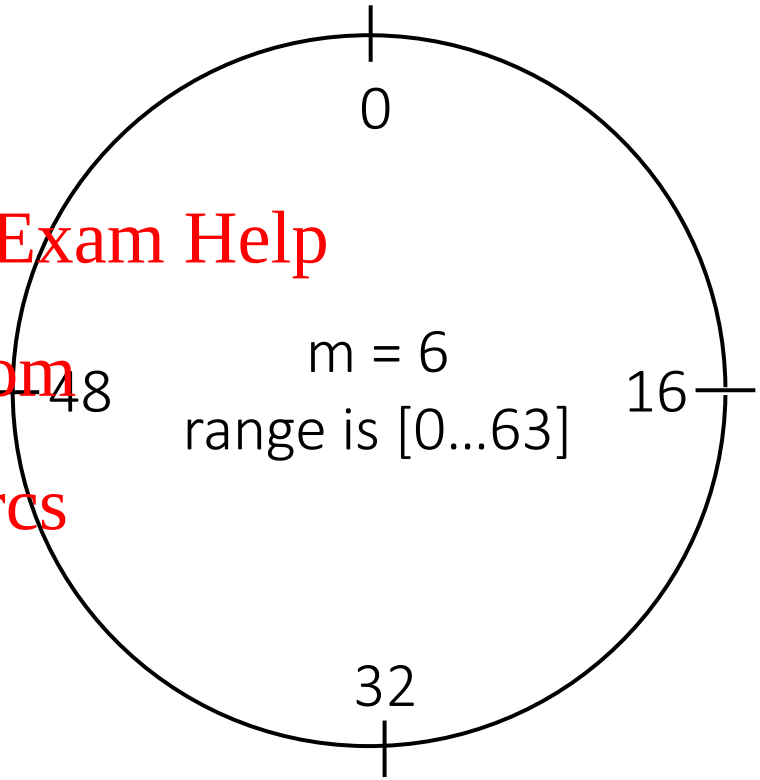
- How to keep uniformly distributed load?
- How to handle peers that join and leave?
- **How does a peer find who knows about a resource with key K?**



The Chord example

- Developed at MIT

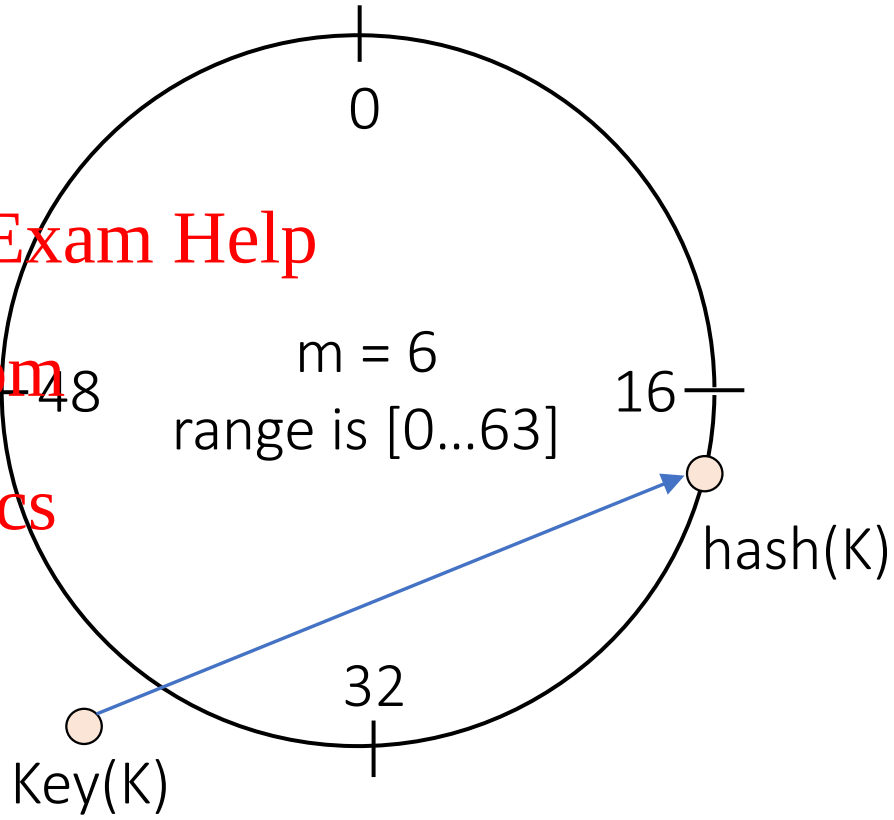
- Basic idea: assign to each node *and* key an m-bit *identifier* using a base hash function such as SHA-1
  - ID(node) = hash(IP, Port)
  - ID(key) = hash(key)



The Chord example

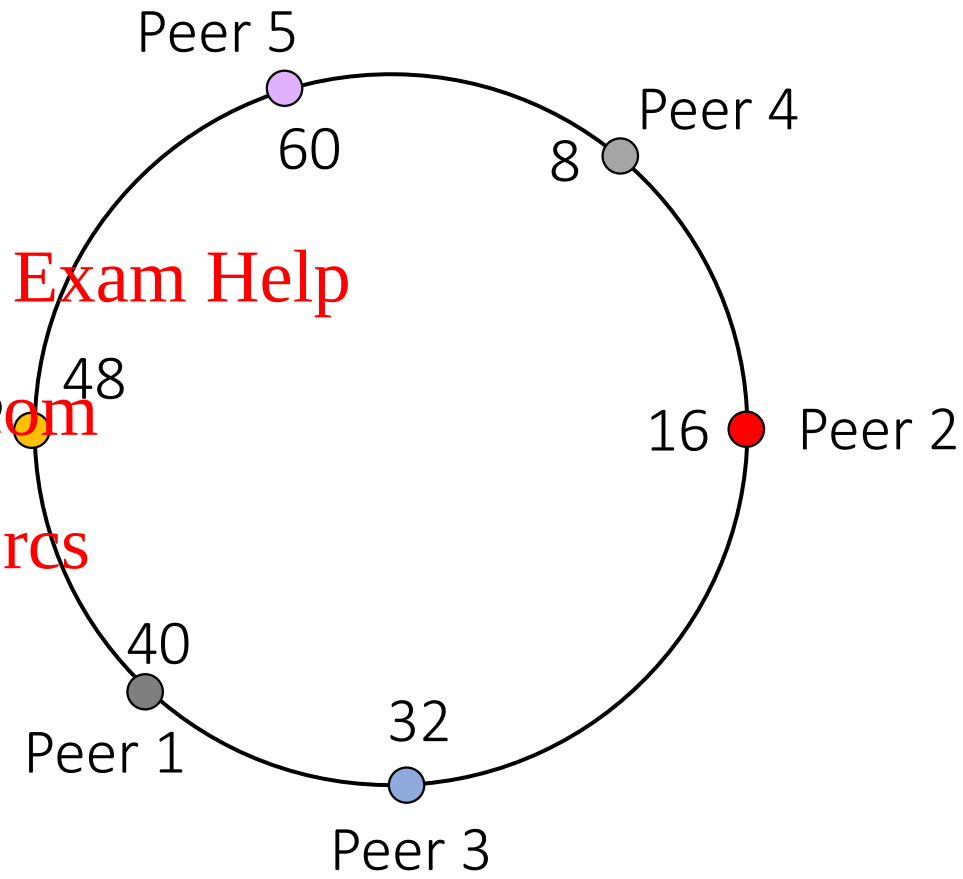
- Developed at MIT

- The key id space
  - interval between 0 and $2^m - 1$
  - chord uses m = 160
  - one can assume $m = O(\log N)$



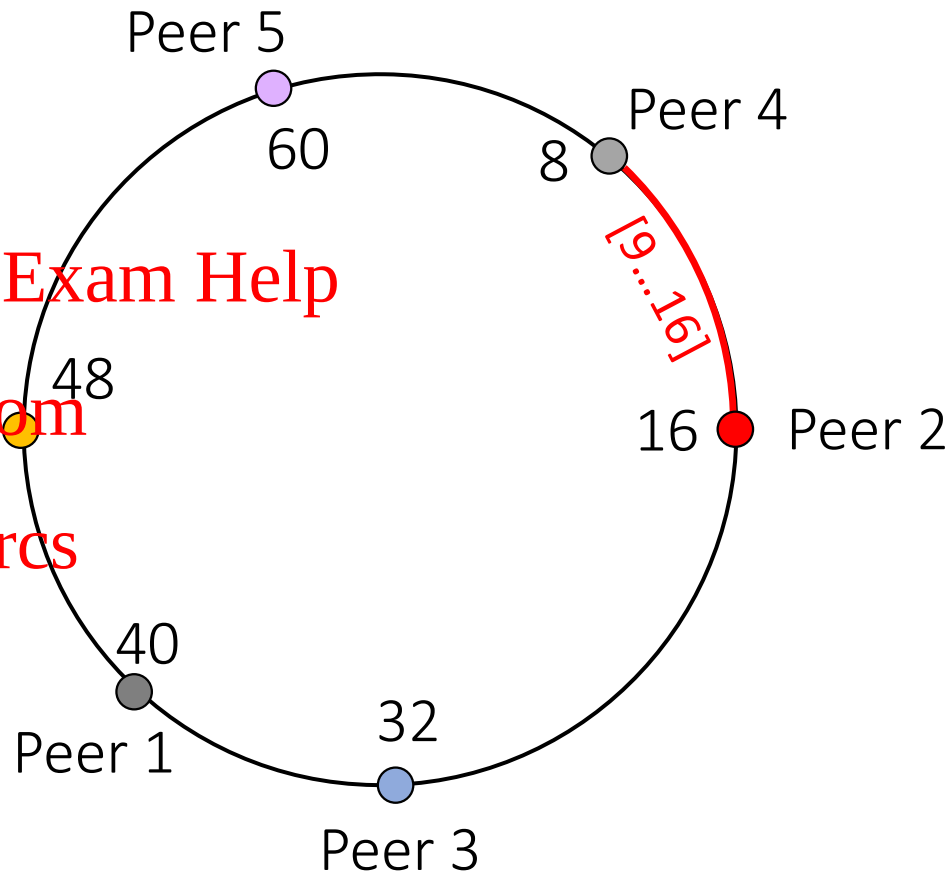
The Chord example

- The key id space
  - interval between 0 and $2^m - 1$
  - chord uses m = 160
  - one can assume $m = O(\log N)$
  - we assume m = 6



The Chord example

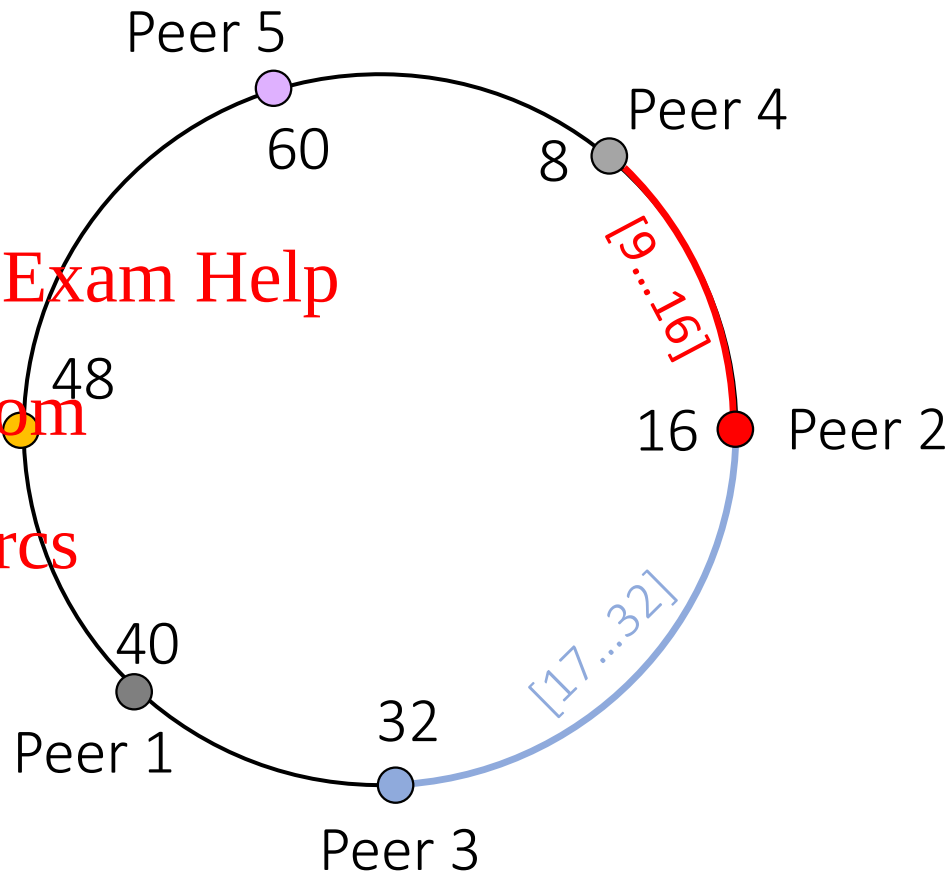
- The key id space
  - interval between 0 and $2^m - 1$
  - chord uses m = 160
  - one can assume $m = O(\log N)$
  - we assume m = 6



Each peer has **authority** over its **id** and all the **smaller** ones until its predecessor peer

The Chord example

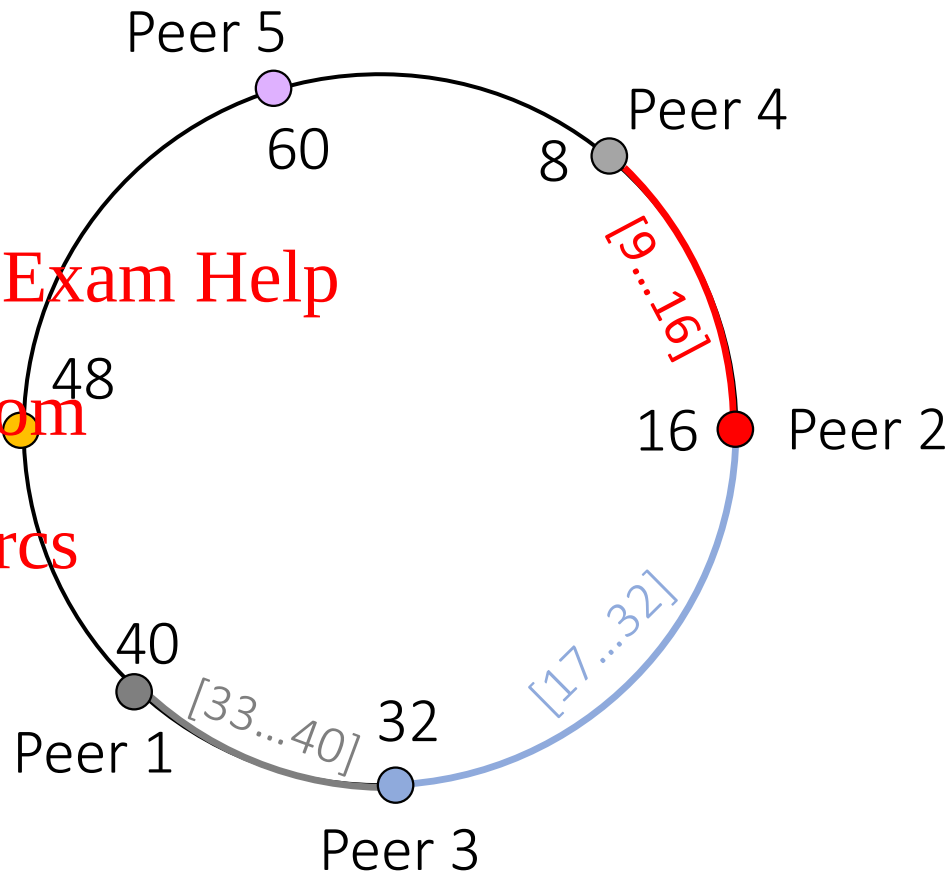
- The key id space
  - interval between 0 and $2^m - 1$
  - chord uses m = 160
  - one can assume $m = O(\log N)$
  - we assume m = 6



Each peer has **authority** over its **id** and all the **smaller** ones until its predecessor peer

The Chord example

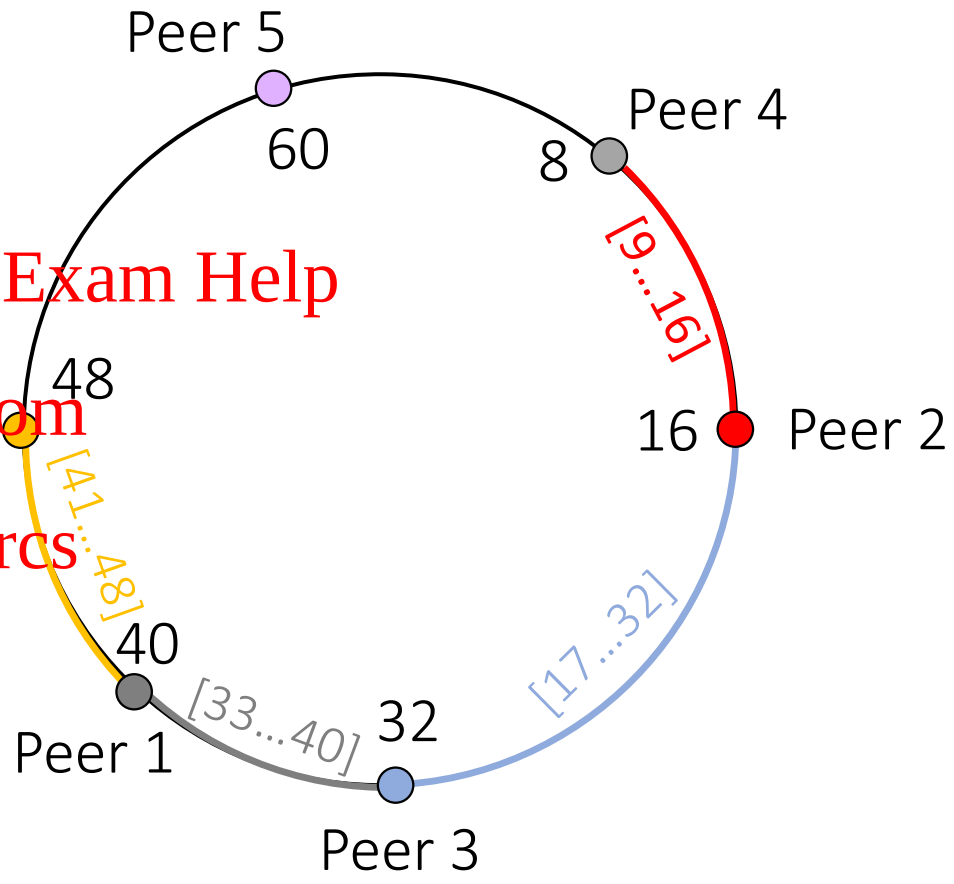
- The key id space
  - interval between 0 and $2^m - 1$
  - chord uses m = 160
  - one can assume $m = O(\log N)$
  - we assume m = 6



Each peer has **authority** over its **id** and all the **smaller** ones until its predecessor peer

The Chord example

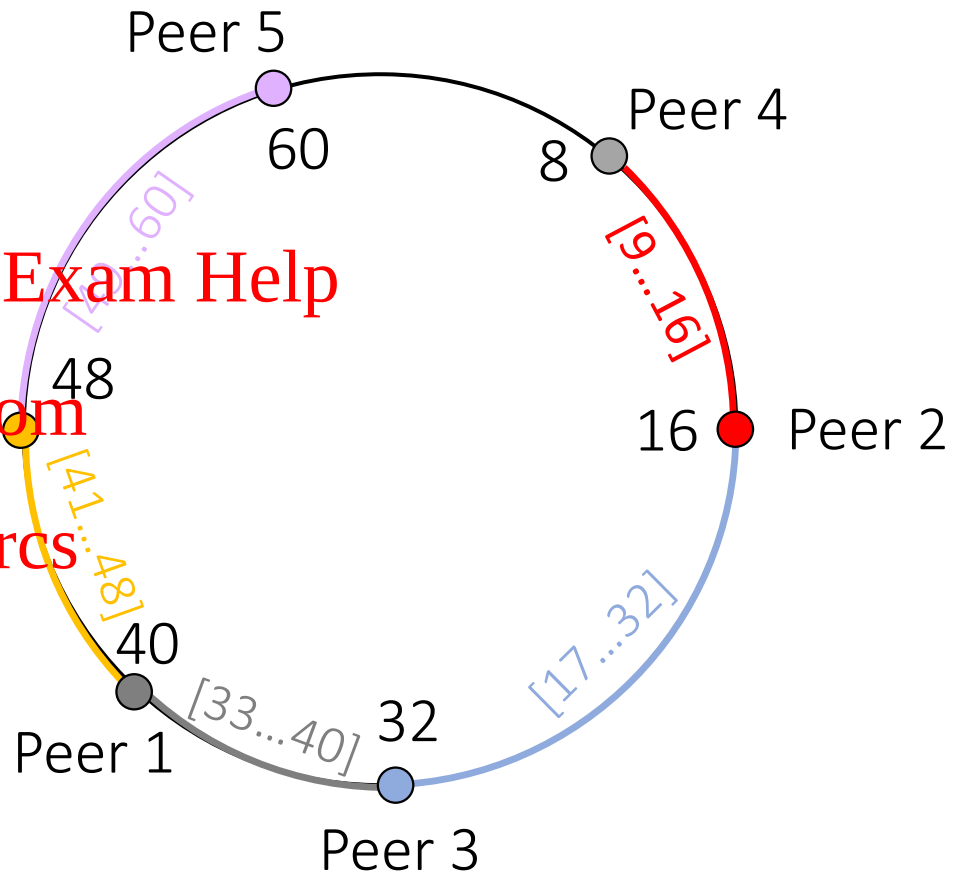
- The key id space
  - interval between 0 and $2^m - 1$
  - chord uses m = 160
  - one can assume $m = O(\log N)$
  - we assume m = 6



Each peer has **authority** over its **id** and all the **smaller** ones until its predecessor peer

The Chord example

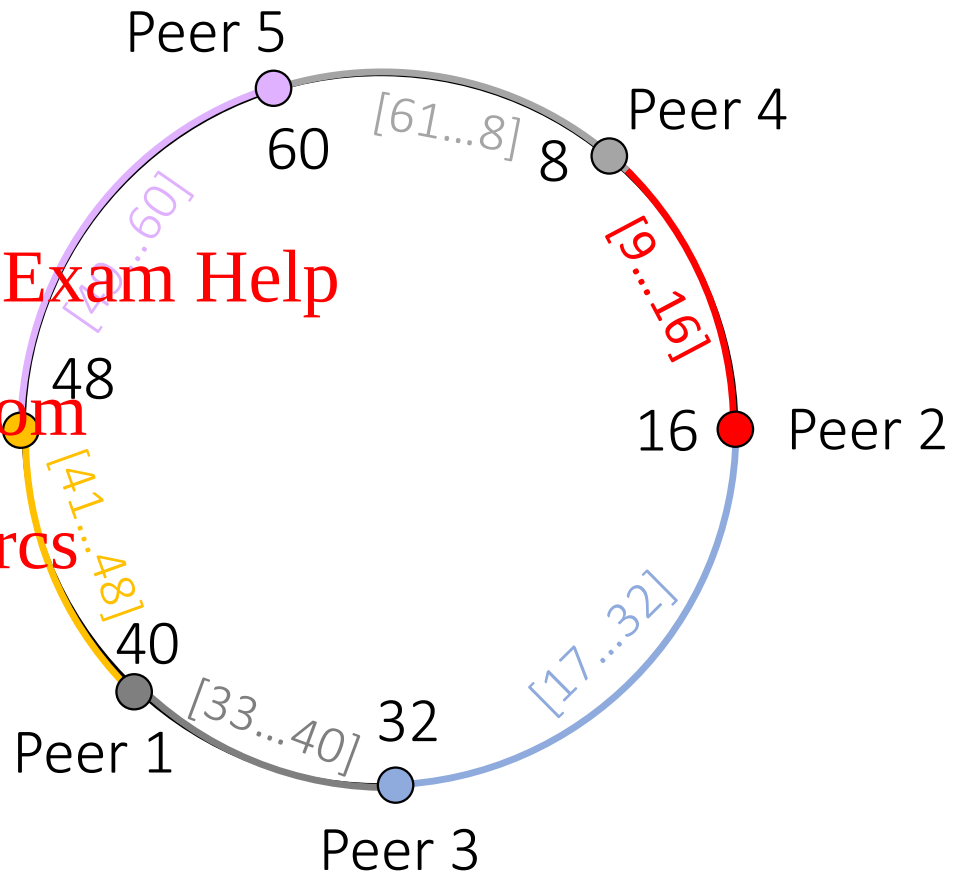
- The key id space
  - interval between 0 and $2^m - 1$
  - chord uses m = 160
  - one can assume $m = O(\log N)$
  - we assume m = 6



Each peer has **authority** over its **id** and all the **smaller** ones until its predecessor peer

The Chord example

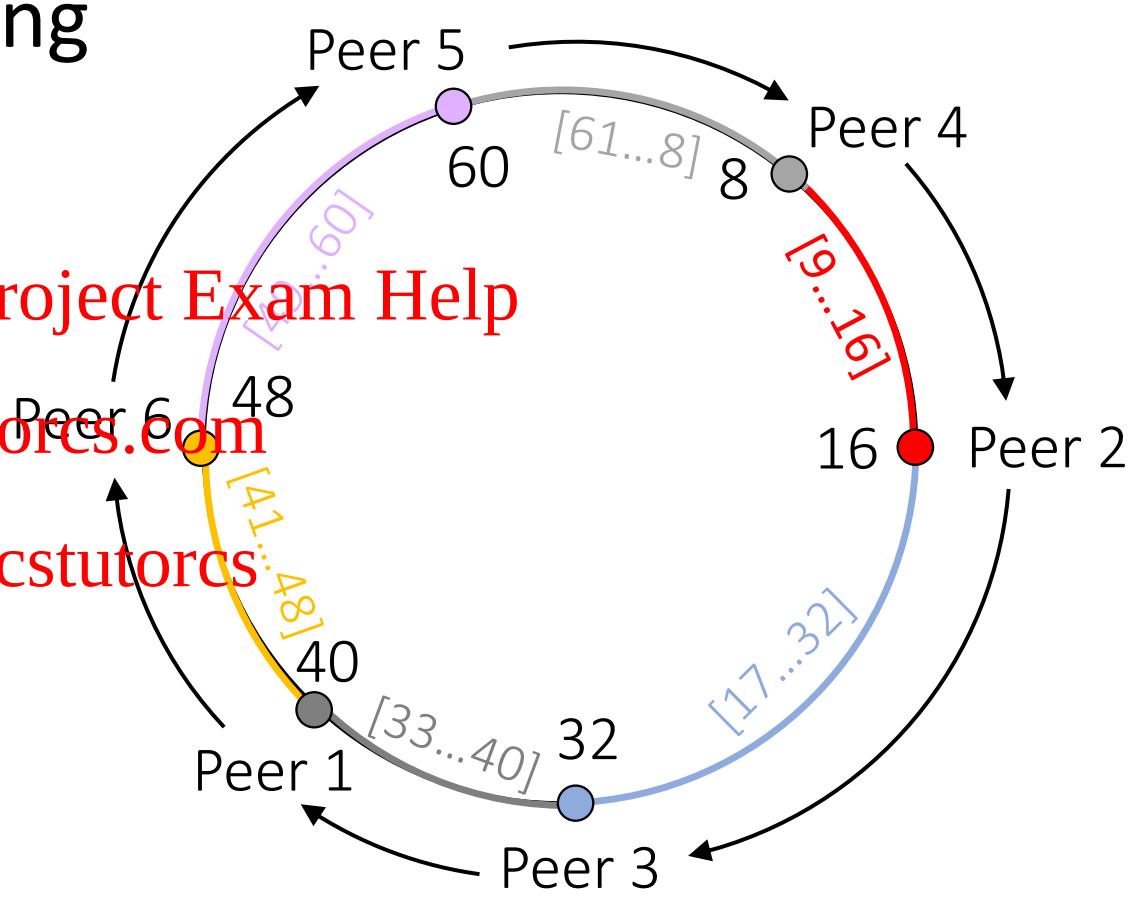
- The key id space
  - interval between 0 and $2^m - 1$
  - chord uses m = 160
  - one can assume $m = O(\log N)$
  - we assume m = 6



Each peer has **authority** over its **id** and all the **smaller** ones until its predecessor peer

Chord with simple routing

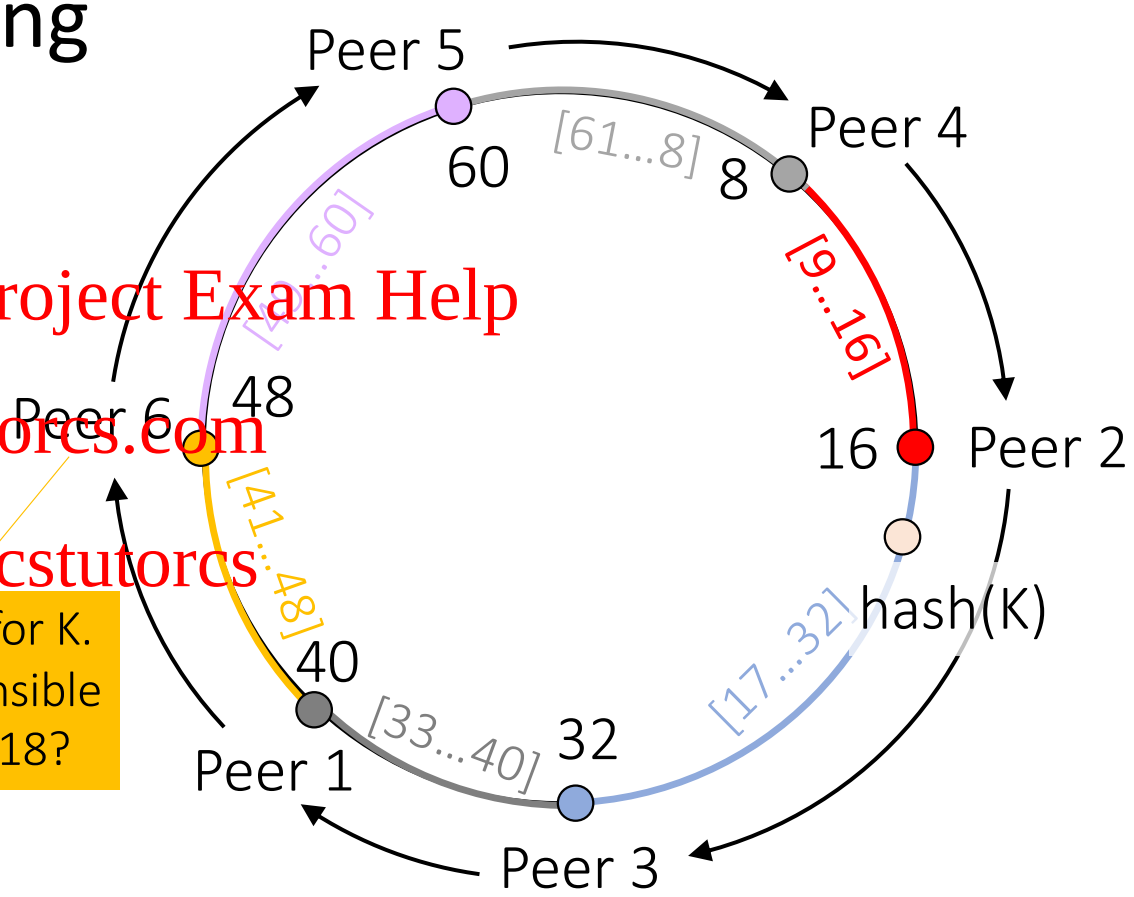
Each peer keeps a pointer (IP address and port) to its successor



Chord with simple routing

Each peer keeps a pointer (IP address and port) to its successor

I am looking for K. Who is responsible for hash(K)=18?

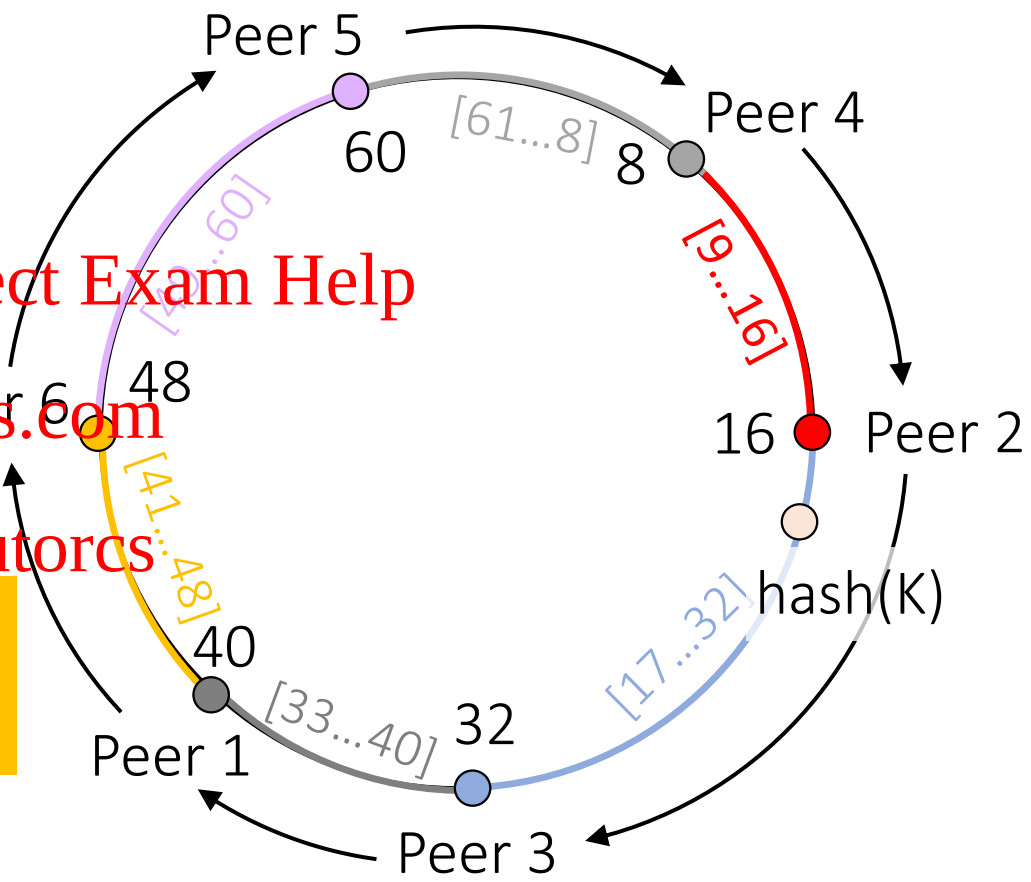


Chord with simple routing

Each peer keeps a pointer (IP address and port) to its successor

Simple Routing: Forward a lookup request to the successor until the correct peer is found

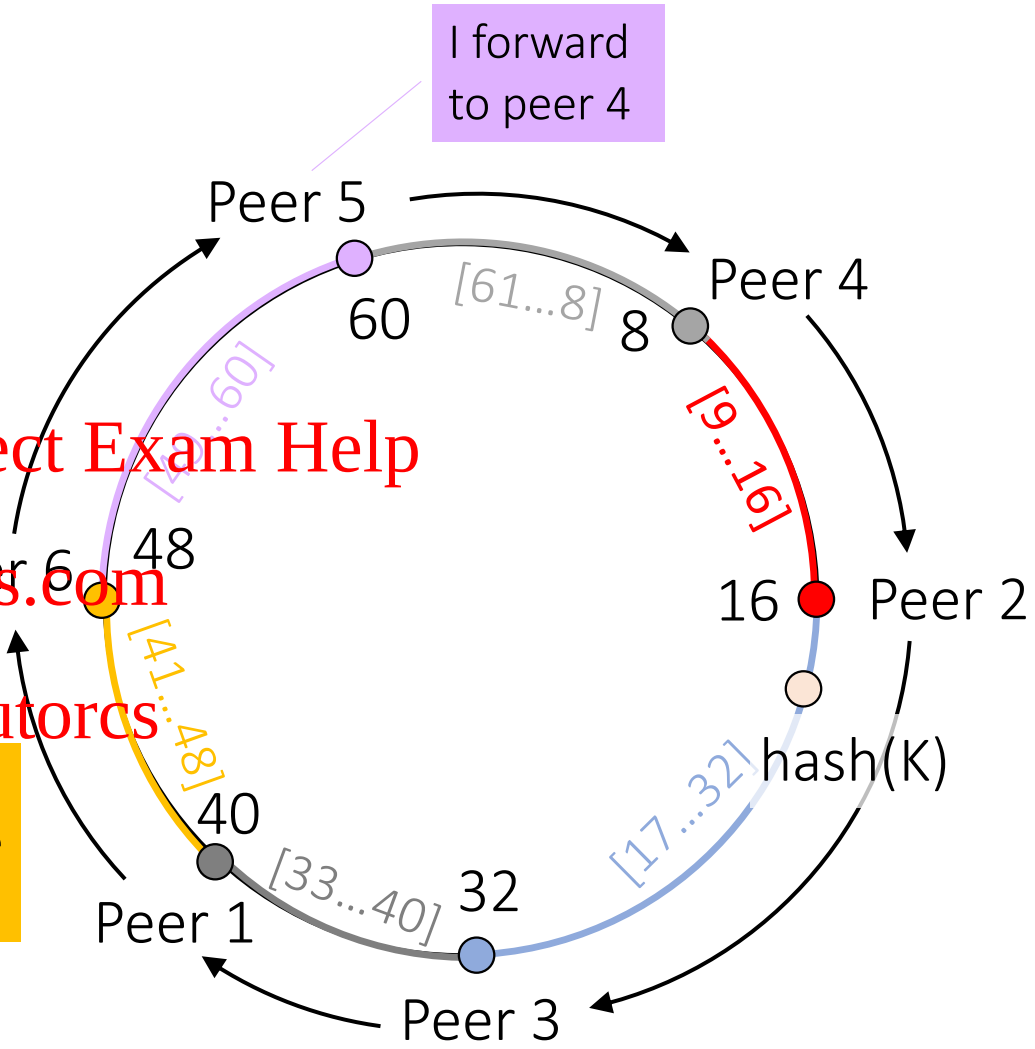
I am looking for K. Who is responsible for hash(K)=18?



Chord with simple routing

Each peer keeps a pointer (IP address and port) to its successor

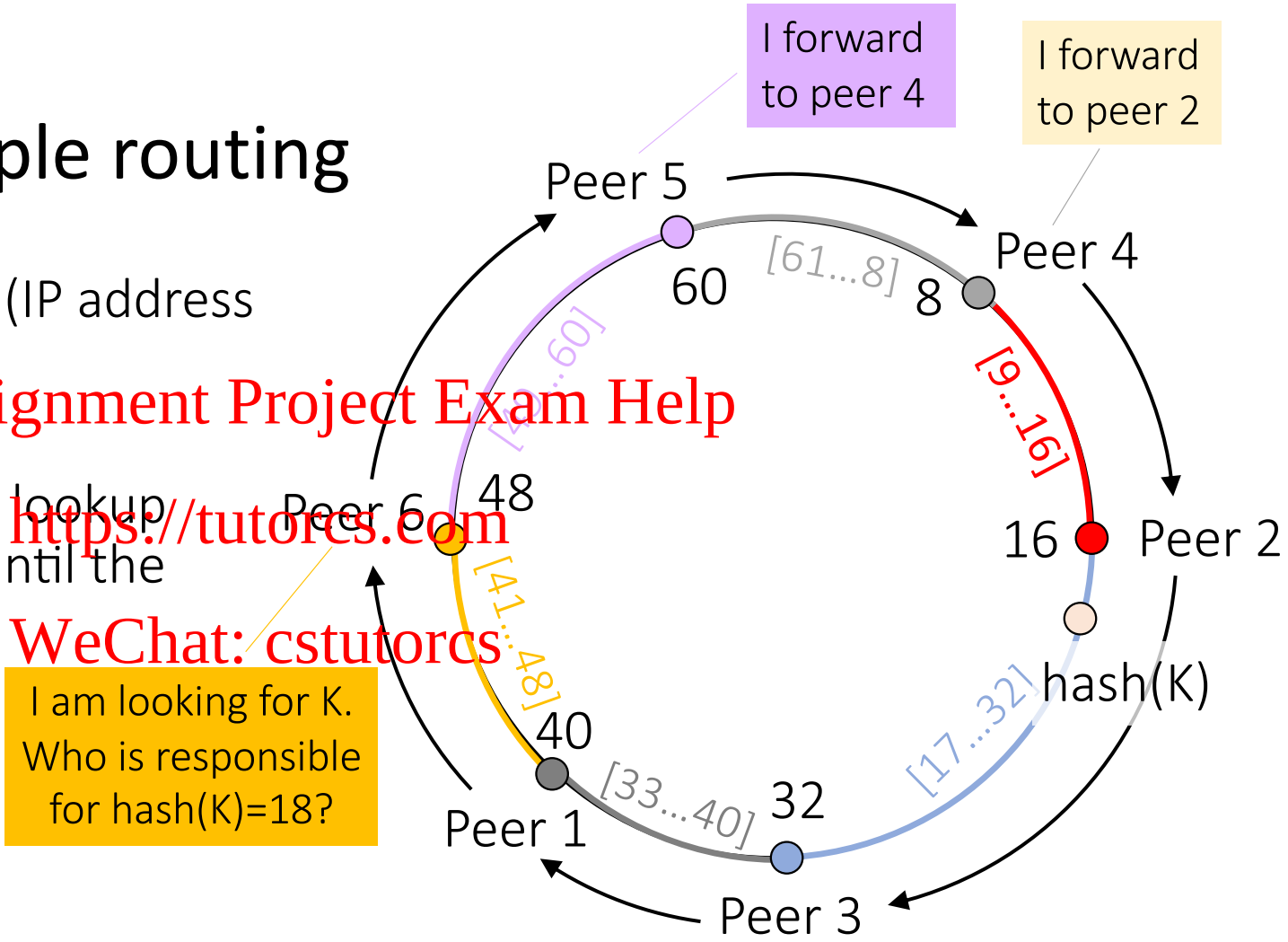
Simple Routing: Forward a lookup request to the successor until the correct peer is found



Chord with simple routing

Each peer keeps a pointer (IP address and port) to its successor

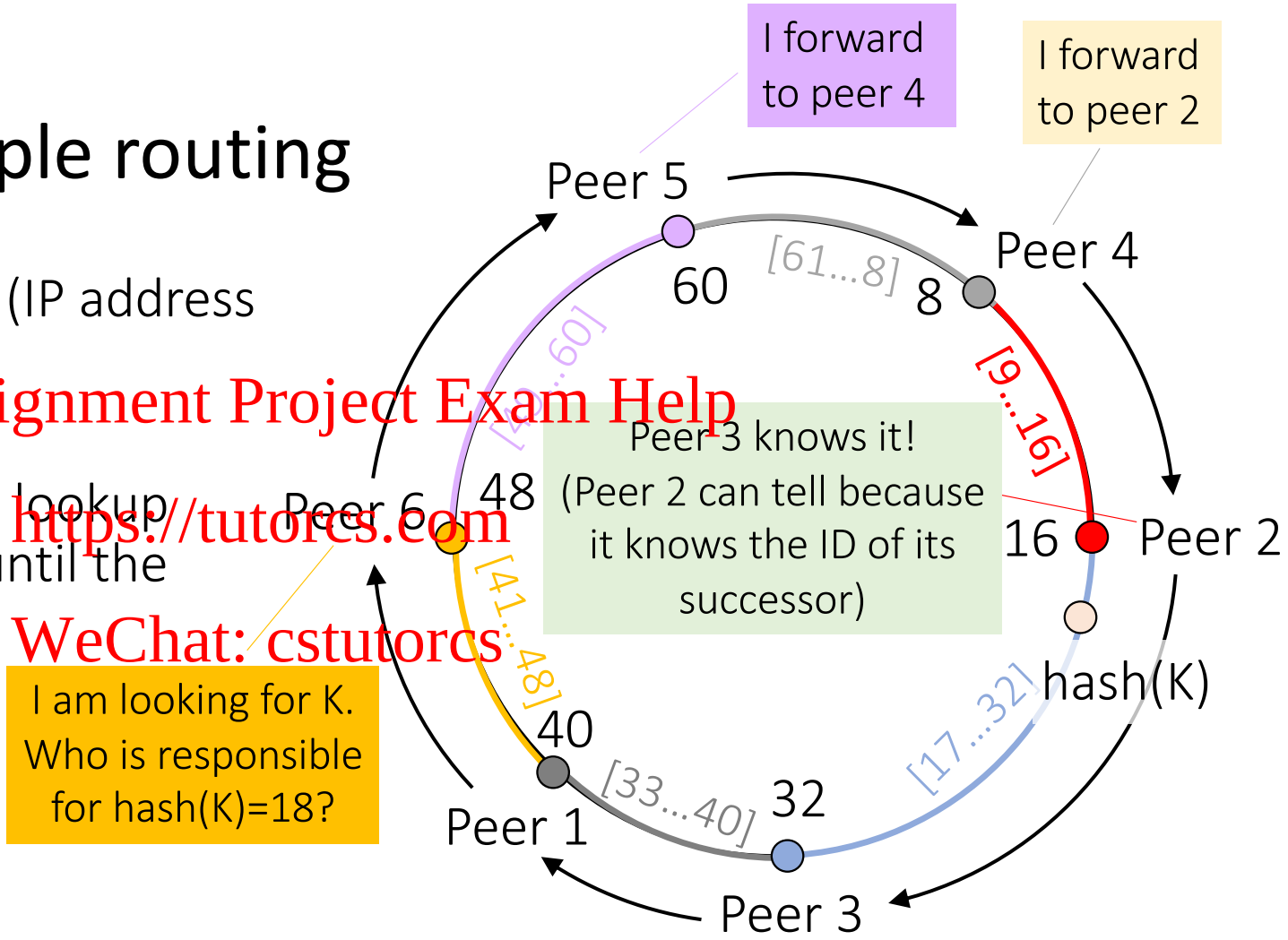
Simple Routing: Forward a lookup request to the successor until the correct peer is found



Chord with simple routing

Each peer keeps a pointer (IP address and port) to its successor

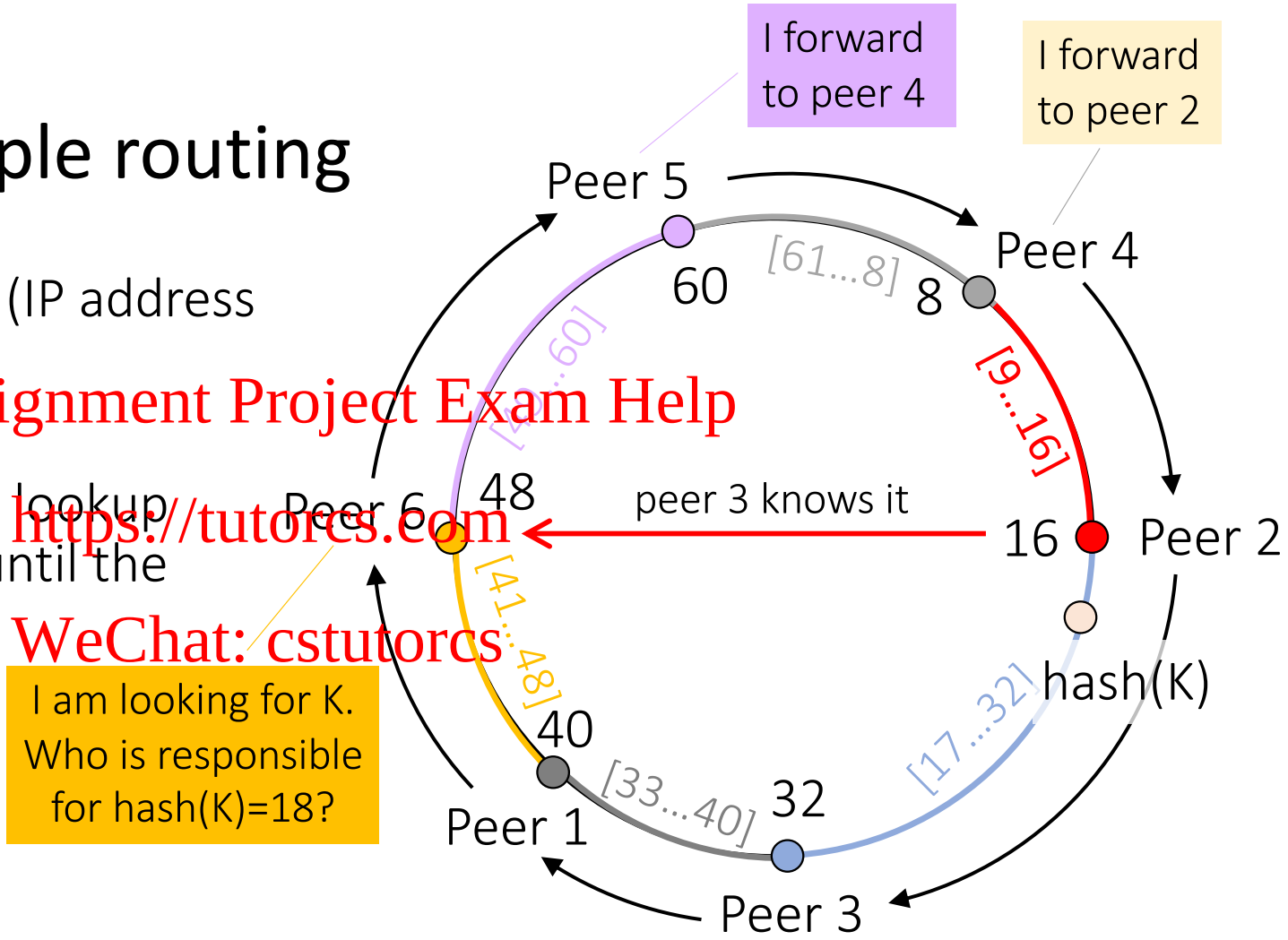
Simple Routing: Forward a lookup request to the successor until the correct peer is found



Chord with simple routing

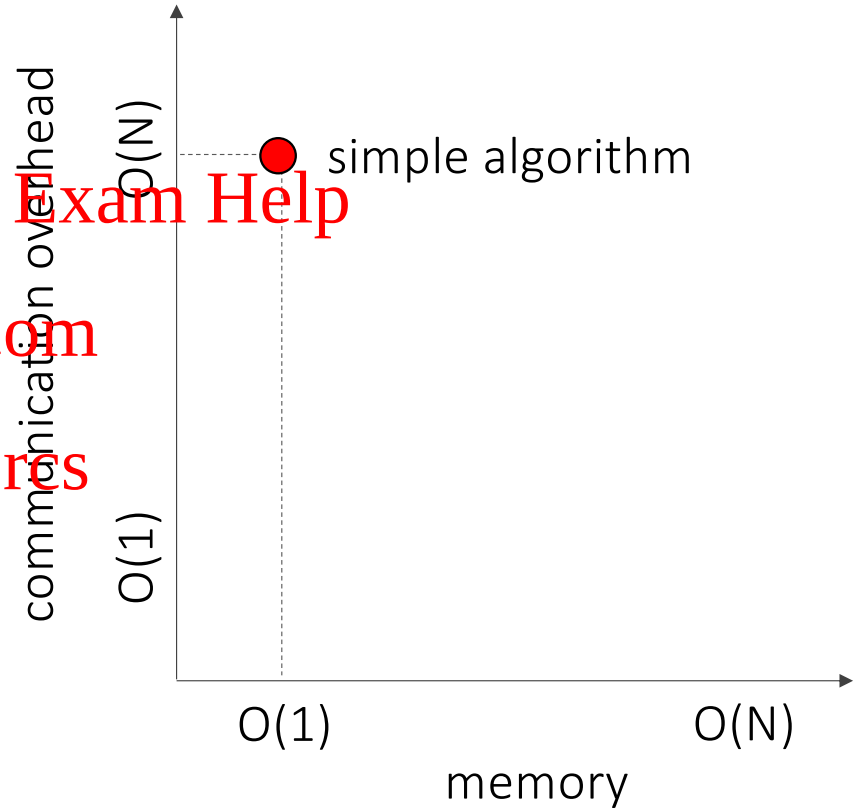
Each peer keeps a pointer (IP address and port) to its successor

Simple Routing: Forward a lookup request to the successor until the correct peer is found



Simple routing

- Each node stores just its successor: O(1) memory
- A request traverses n/2 hops on average: O(N) communication
- Not robust: if one node fails, communication is interrupted until the mechanism reconverge

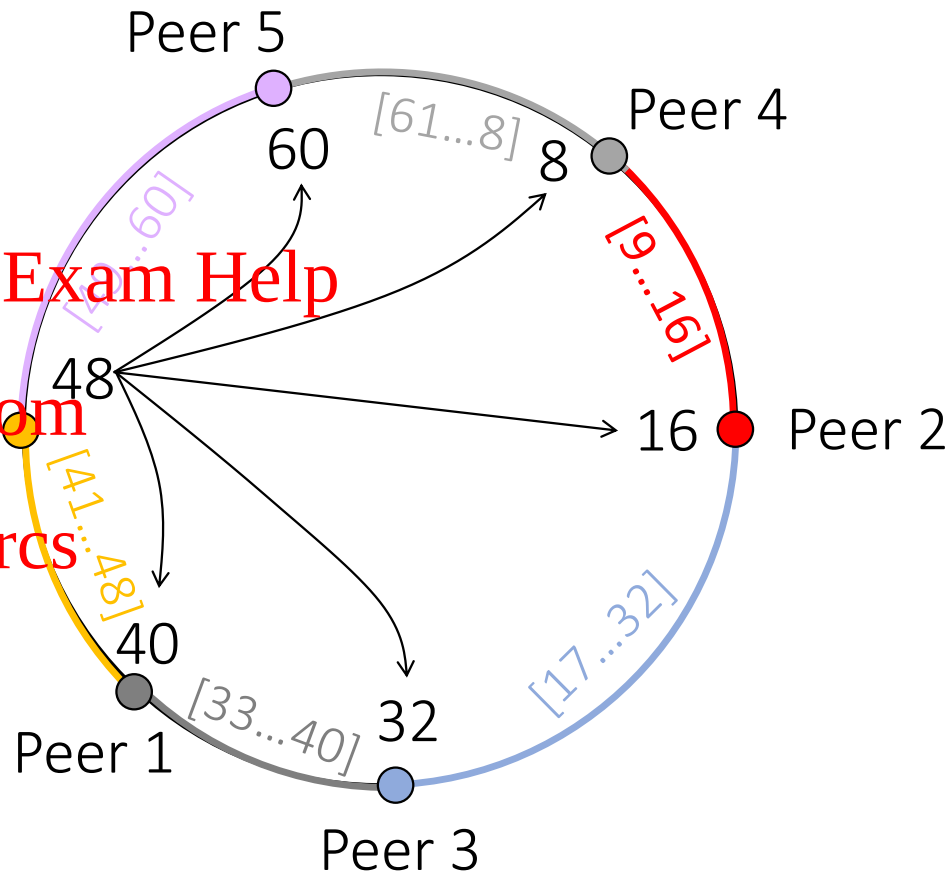


Chord with simple routing++

Each peer keeps a list of **all the other peers**

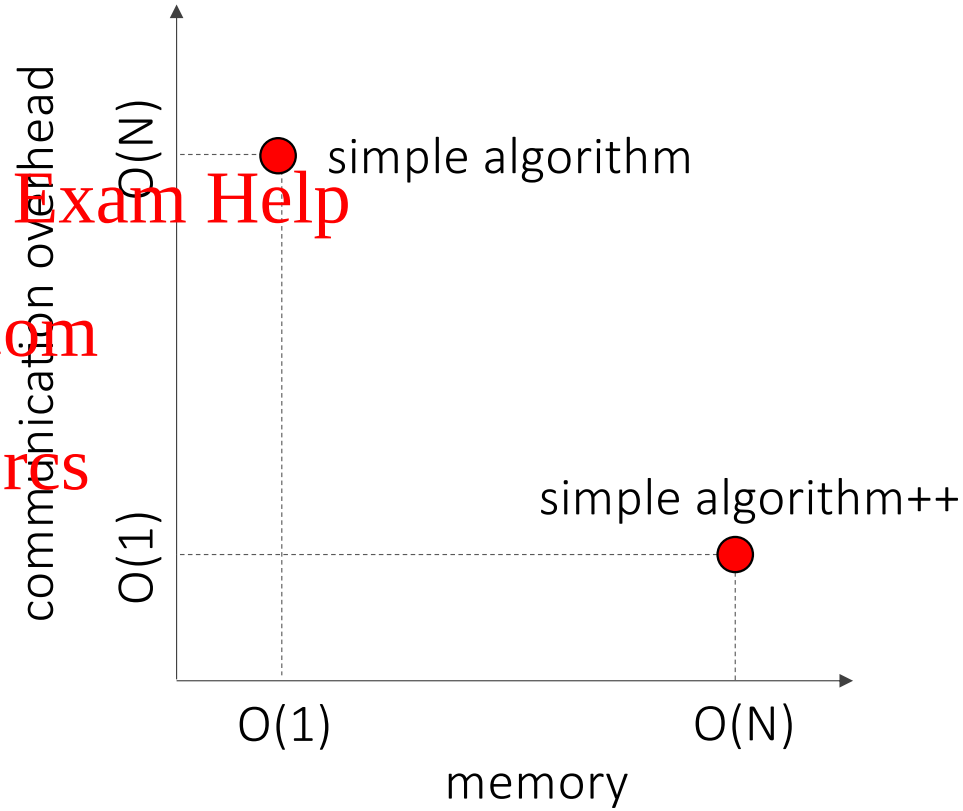
No routing is needed

- a peer already knows who is responsible for what key IDs



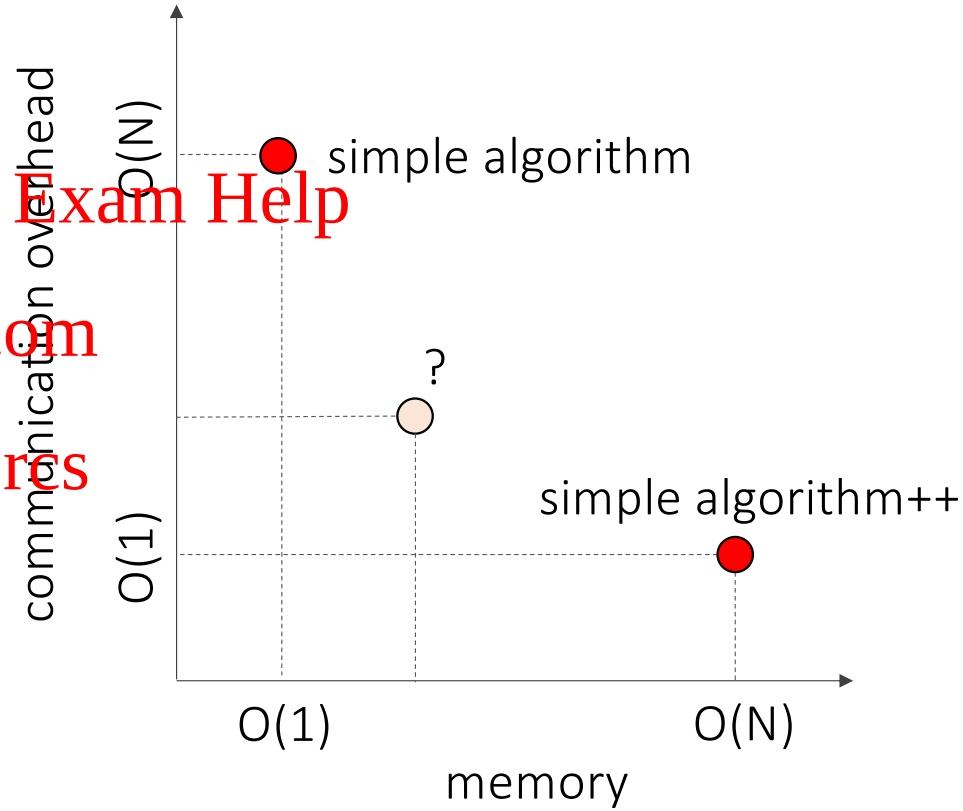
Simple routing++

- Each node stores the address of all the others: O(N) memory
- A request traverses 1 hop: O(1) communication
- Can we do better?



Simple routing++

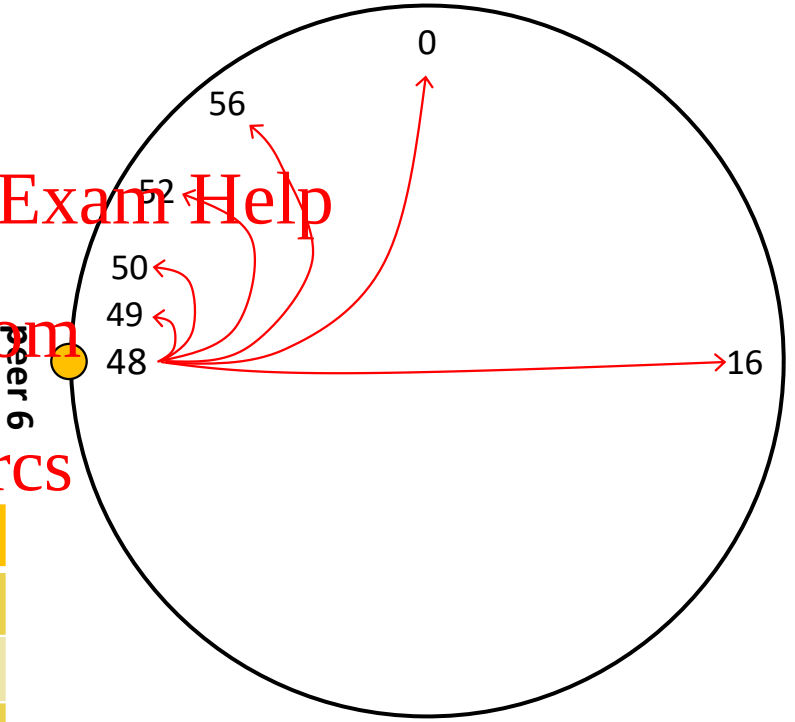
- Each node stores the address of all the others: O(N) memory
- A request traverses 1 hop: O(1) communication
- Can we do better?



Finger tables

Peer *k* stores information about peer authority at increasing exponential distance

- distances are power of 2
- $k + 2^i$ mod (2^m), for each *i=0,…,m-1*



Finger table at peer 6

i	key id	successor
0	$48 + 2^0$ mod 64 = 49	
1	$48 + 2^1$ mod 64 = 50	
2	$48 + 2^2$ mod 64 = 52	
3	$48 + 2^3$ mod 64 = 56	
4	$48 + 2^4$ mod 64 = 0	
5	$48 + 2^5$ mod 64 = 16	

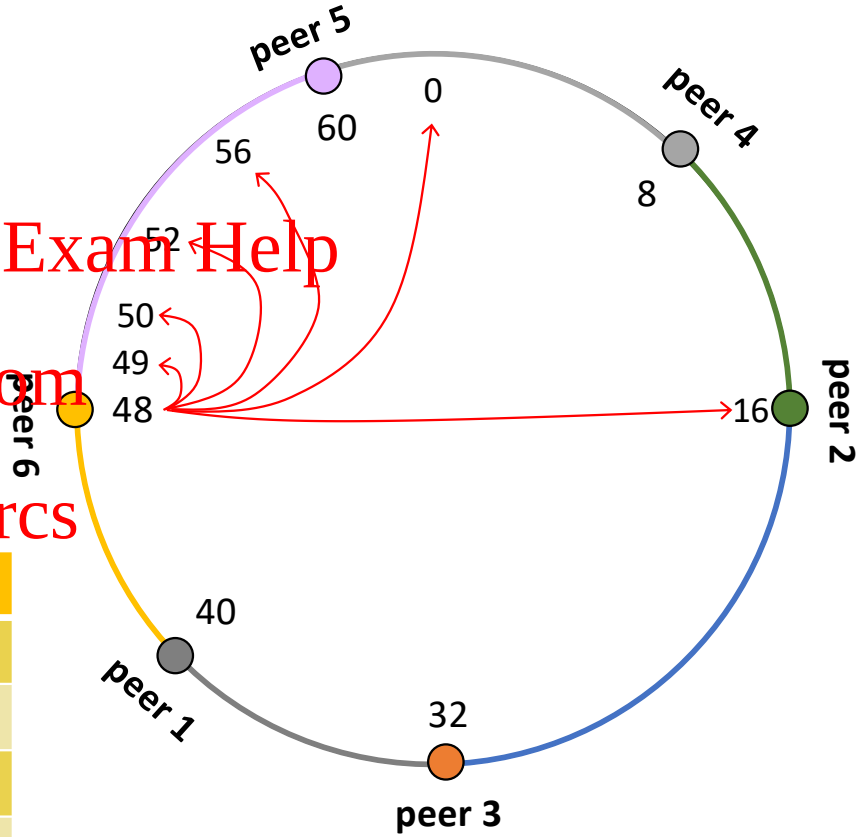
Routing with finger tables

Peer *k* stores information about peer authority at increasing exponential distance

- distances are power of 2
- $k + 2^i$ mod (2^m), for each *i=0,…,m-1*

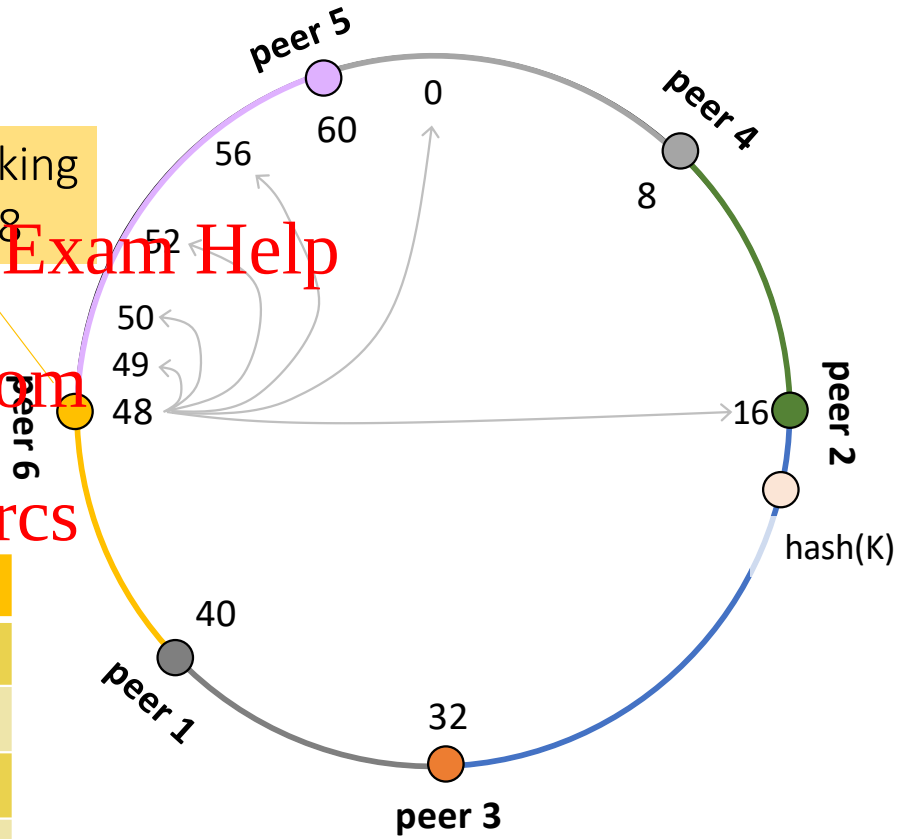
Finger table at peer 6

i	key id	successor
0	$48 + 2^0$ mod 64 = 49	Peer 5
1	$48 + 2^1$ mod 64 = 50	Peer 5
2	$48 + 2^2$ mod 64 = 52	Peer 5
3	$48 + 2^3$ mod 64 = 56	Peer 5
4	$48 + 2^4$ mod 64 = 0	Peer 4
5	$48 + 2^5$ mod 64 = 16	Peer 2



Routing with finger tables

Each peer sends a request to the closest successor peer that precedes or equal the key ID

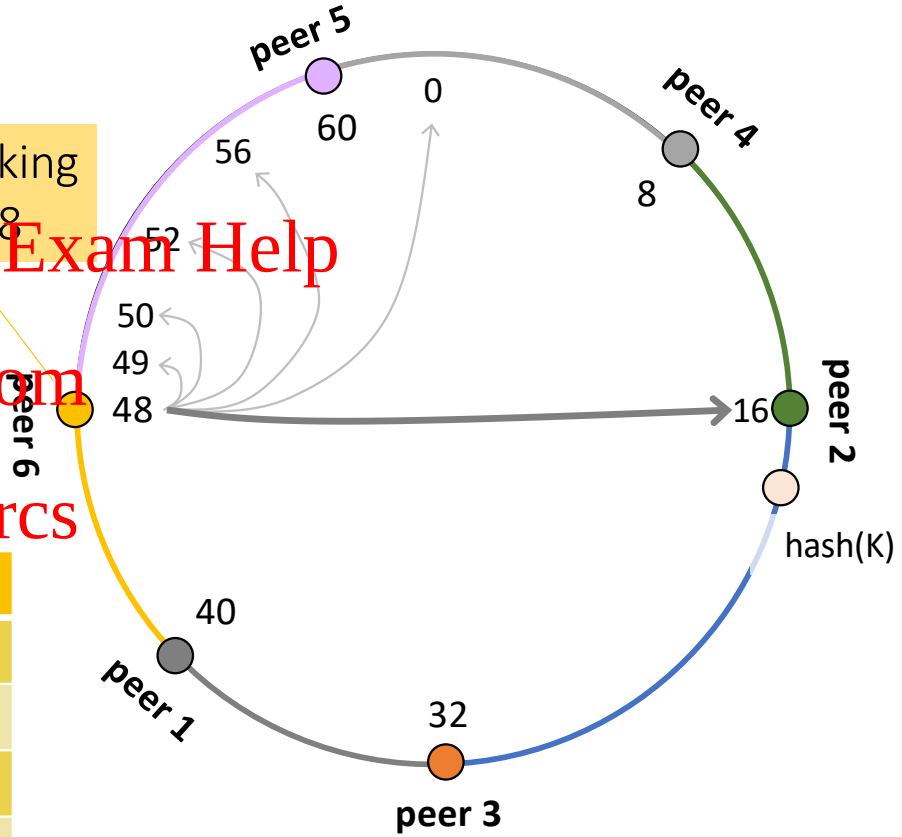


Finger table at peer 6

i	key id	successor
0	$48 + 2^0 \bmod 64 = 49$	Peer 5
1	$48 + 2^1 \bmod 64 = 50$	Peer 5
2	$48 + 2^2 \bmod 64 = 52$	Peer 5
3	$48 + 2^3 \bmod 64 = 56$	Peer 5
4	$48 + 2^4 \bmod 64 = 0$	Peer 4
5	$48 + 2^5 \bmod 64 = 16$	Peer 2

Routing with finger tables

Each peer sends a request to the closest successor peer that precedes or equal the key ID

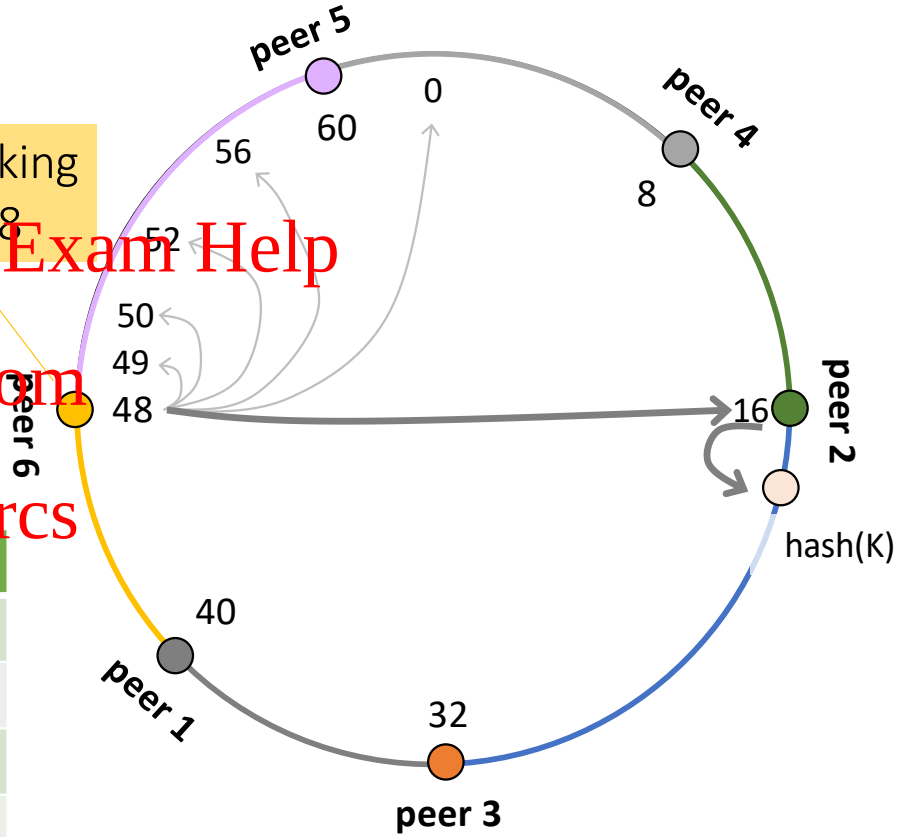


Finger table at peer 6

i	key id	successor
0	$48 + 2^0 \bmod 64 = 49$	Peer 5
1	$48 + 2^1 \bmod 64 = 50$	Peer 5
2	$48 + 2^2 \bmod 64 = 52$	Peer 5
3	$48 + 2^3 \bmod 64 = 56$	Peer 5
4	$48 + 2^4 \bmod 64 = 0$	Peer 4
5	$48 + 2^5 \bmod 64 = \mathbf{16}$	**Peer 2**

Routing with finger tables

Each peer sends a request to the closest successor peer that precedes or equal the key ID

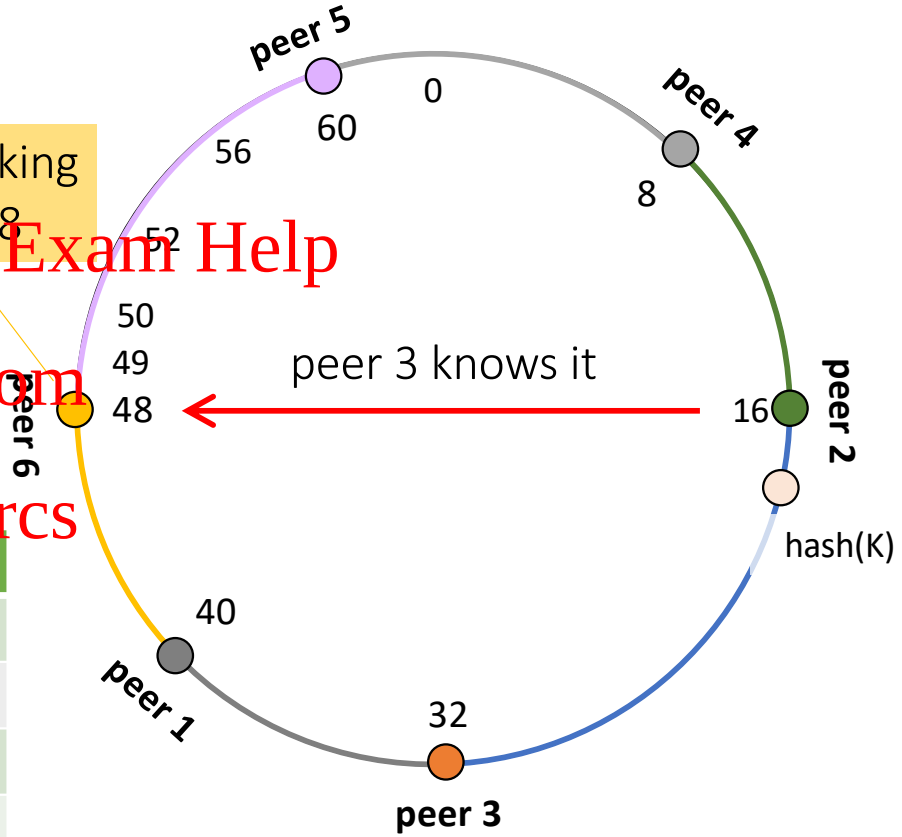


Finger table at peer 2

i	key id	successor
0	$16 + 2^0 \bmod 64 = 17$	peer 3
1	**$16 + 2^1 \bmod 64 = 18$**	**peer 3**
2	$16 + 2^2 \bmod 64 = 20$	peer 3
3	$16 + 2^3 \bmod 64 = 24$	peer 3
4	$16 + 2^4 \bmod 64 = 32$	peer 3
5	$16 + 2^5 \bmod 64 = 48$	peer 6

Routing with finger tables

Each peer sends a request to the closest successor peer that precedes or equal the key ID



Finger table at peer 2

i	key id	successor
0	$16 + 2^0 \bmod 64 = 17$	peer 3
1	**$16 + 2^1 \bmod 64 = 18$**	**peer 3**
2	$16 + 2^2 \bmod 64 = 20$	peer 3
3	$16 + 2^3 \bmod 64 = 24$	peer 3
4	$16 + 2^4 \bmod 64 = 32$	peer 3
5	$16 + 2^5 \bmod 64 = 48$	peer 6

Chord (finger tables) analysis

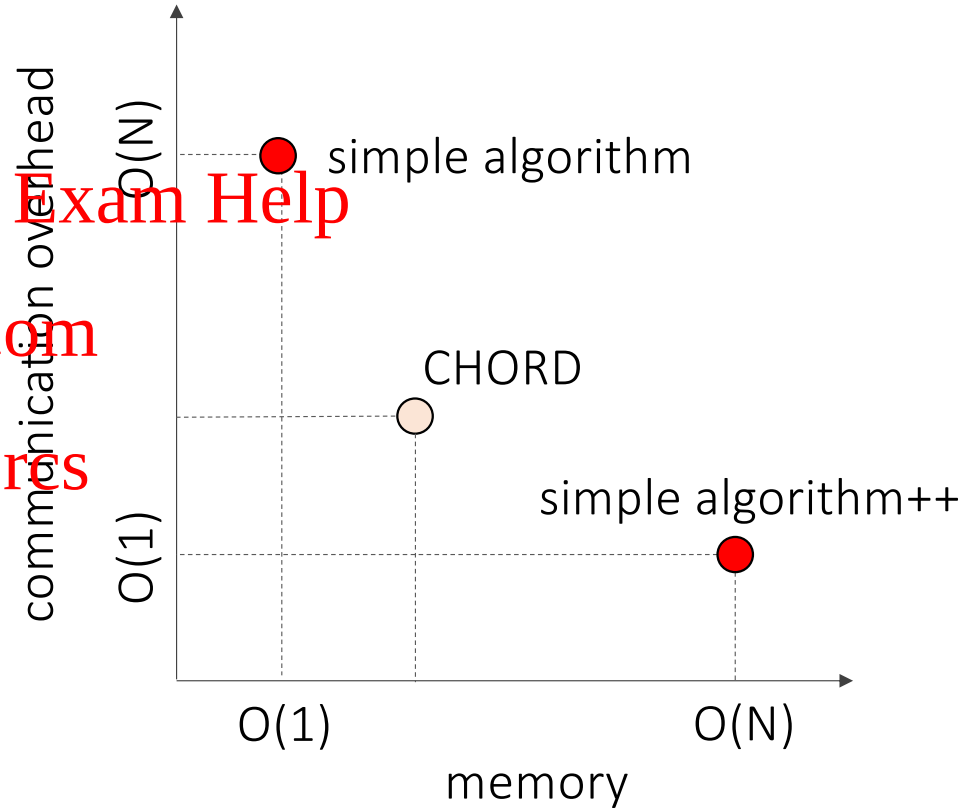
Each node stores a subset of successors:

- O(log N) memory

The search space is halved at each hop:

- O(log N) communication

More robust: unless the authority peer of the key ID fails, lookup operations work correctly



What we have seen in one slide summary

- Many different types of P2P networks: centralized, flooding, hierarchical
- Issues:
  - Failure mode: single point of failure?
  - Flooding is onerous
  - Network topology different than overlay topology
  - Nodes are not all the same
  - Search can be hard