Peer-to-Peer Networking

Network Programming

Interesting Approaches

- High-level overview
  - Only scratch the surface!
- Too many to cover any in depth
  - High level overview
- I encourage everyone to dig more deeply if anything intrigues you

Motivation

- Routing resilience
  - If a node breaks, should be able to route around the damage
- Communication still feasible
- Formerly, prioritize connectivity
- Now prioritize content from endpoints!

Problems

- What if an endpoint goes down?
  - Gmail? Facebook? Twitter?
- What if a state-level agency decides Facebook is bad?
  - Maybe Bhutan bans depression sites?
- What about natural disasters?
  - I need to find my family/friends

Peer-to-Peer

- All peers are equal
  - Content providers
  - Routing partners
- Pay to play
  - Each host generates workload
  - Each host also contributes resources
- Conceptually, scales well

Examples

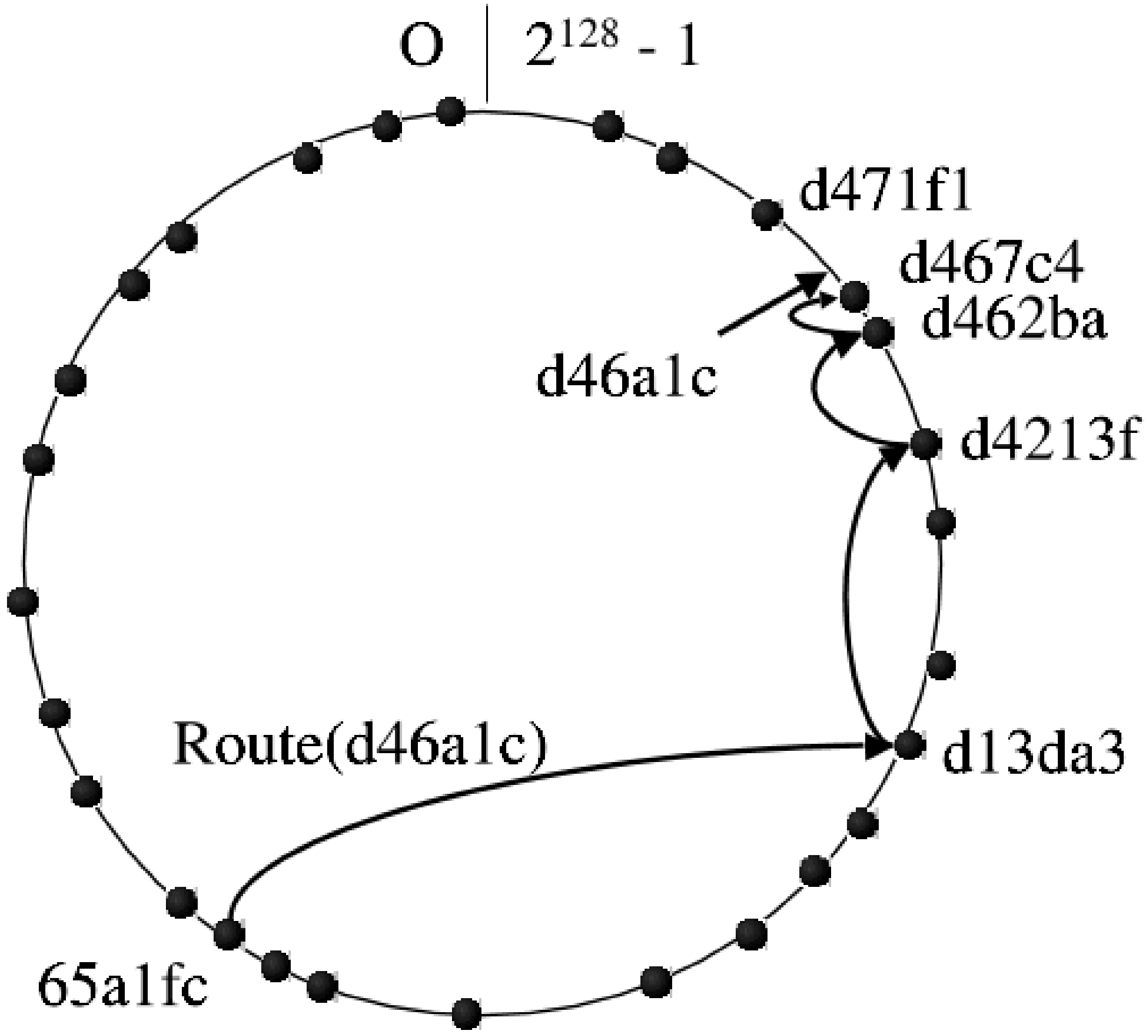
- Napster
  - P2P w/ a centralized server
  - Single point of failure!
- Gnutella
  - Creates an overlay network
  - Floods content requests
  - No single point of failure, more resilient than Napster but scalability issues

- Both have limitations!

Distributed Hash Tables

- Properties:
  - Decentralized
  - Fault Tolerant
  - Scalable
- Generally use keyspace partitioning scheme
- Each node maintains a (partial) routing table for the overlay network

Routing in Overlay Network



Chord

- ids are length m (so entire space is 0… 2^m-1)
  - Common value is 16-bits (64k keys/nodes)
- Node a wants to try to find key k
- Who is successor(k), the node responsible for k?
  - If a node j has ID k, then successor(k) = j
  - o.w. node j' (the first node going clockwise from where k would be, i.e. closest node with ID>k) is the successor
- If successor(k) ≠ a and ≠ a+1, ask a+1 to find the next node.
- O(N) search, not great!

Chord – Finger Tables

- Every node keeps a “finger table” with up to m entries
- Entry i for node n is
  - successor($(n + 2^{i-1}) \bmod 2^m$)
  - Now we can “jump” further along the circle as needed, instead of having to ask n+1, n+2, n+3, … successor(k)
  - O(log N) search time, much better!
  - Should feel a lot like a binary search
- See Wikipedia for more details

Gnutella

- Nodes are called servants
- Connect, then send descriptors (Gnutella protocol concept, not C file descriptor)
  - Ping to discover
  - Reply with one or more pongs to describe availability
  - Query to ask for data
  - QueryHit to reply that you have the data

Gnutella Routing

- Pongs can only travel back along the path the Ping came from
- QueryHit similarly can only follow Query's path
- Forward Ping/Query to all direct connections except the one you received it on
- Decrement TTL field and increment Hops before forwarding. Do not forward if TTL = 0
- Don't forward if payload descriptor and descriptor ID already seen - prevents replication of messages!

Gnutella Downloads

- Remember that Gnutella is an overlay
- For downloads we do a direct client-to-client connection
  - Don't route through other servants in the overlay network
- This is still P2P, no dedicated server
  - But if we could stop the initial distributor before copies were shared, we could prevent a file from spreading
  - Easier said than done!

BitTorrent

- Break files up into *pieces*
- Upon downloading a piece, able to then serve it to another consumer
  - Popularity increases number of hosts for that piece / content
- Cryptographic hashing used to ensure no changes to piece made

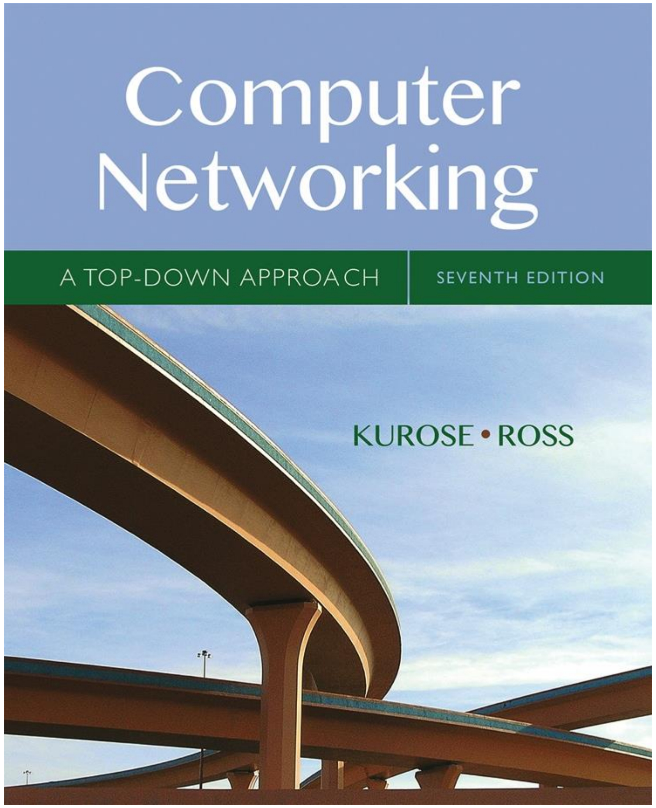
Chapter 2
Application Layer

A note on the use of these Powerpoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR



Computer Networking: A Top Down Approach

7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks

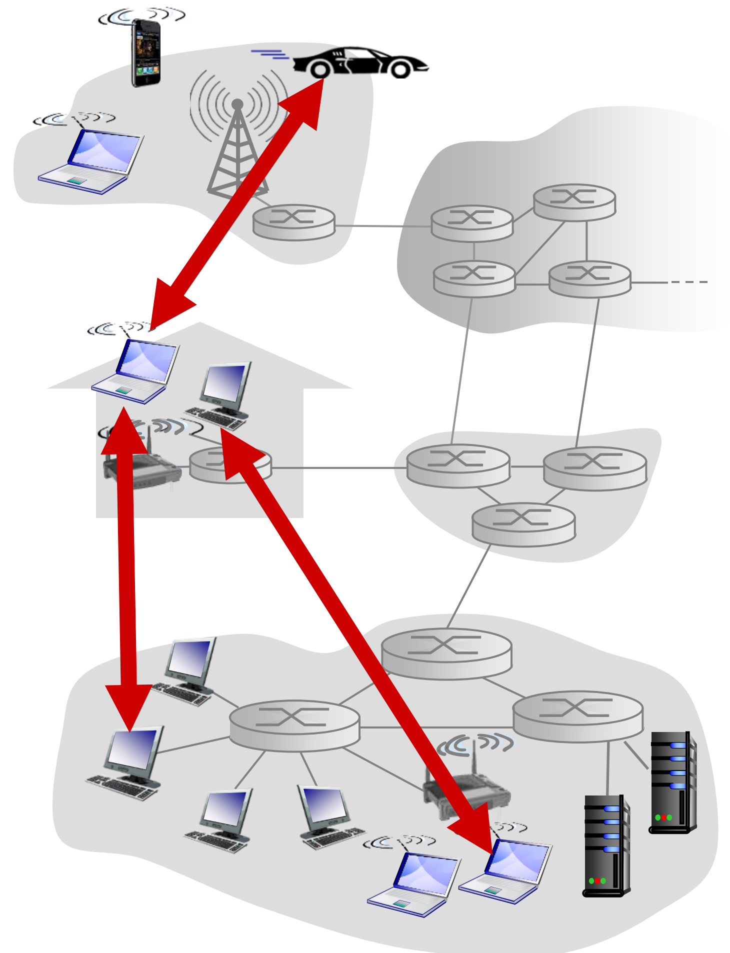
2.7 socket programming with UDP and TCP

Pure P2P architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

examples:

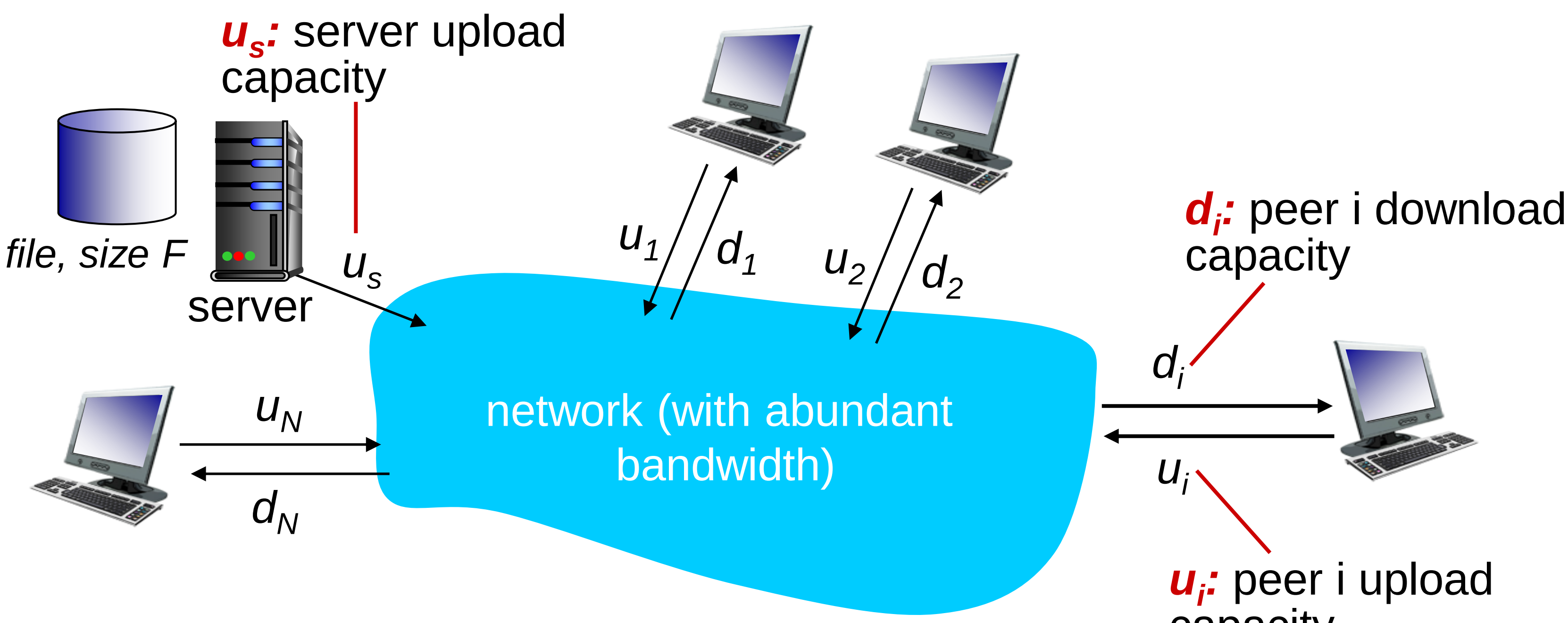
- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)



File distribution: client-server vs P2P

Question: how much time to distribute file (size *F*) from one server to *N* *peers*?

- peer upload/download capacity is limited resource

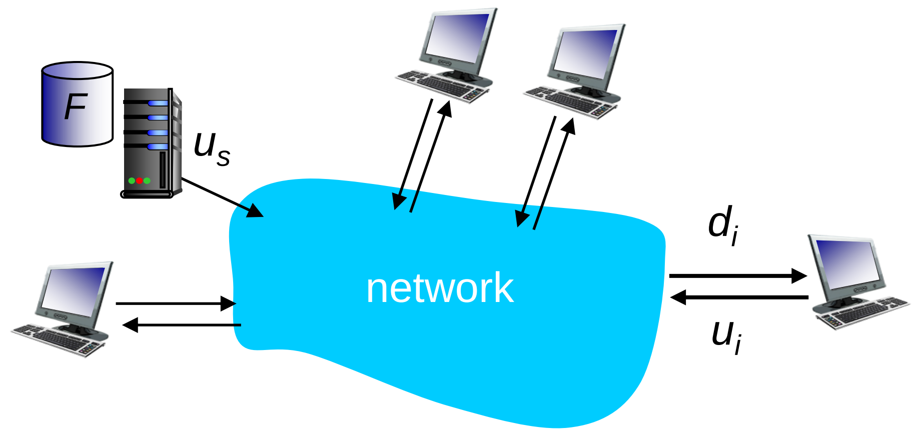


Side Question

- If a tracker is required, is the tracker legally responsible?
- What if we killed/blocked the tracker?
  - What effect does killing the tracker have?
  - Does it kill the P2P network?
  - Can we somehow still access content?

File distribution time: client-server

- *server transmission:* must sequentially send (upload) N file copies:
  - time to send one copy: F/u_s
  - time to send N copies: NF/u_s



- *client:* each client must download file copy
  - d_{min} = min client download rate
  - min client download time: F/d_{min}

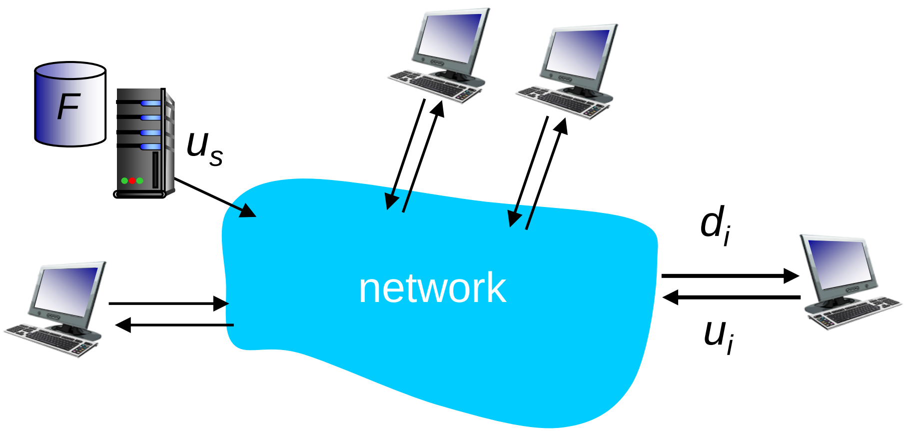
time to distribute F to N clients using client-server approach

$$D_{c\text{-}s} \geq max\{NF/u_s, F/d_{min}\}$$

increases linearly in N

File distribution time: P2P

- *server transmission:* must upload at least one copy
  - time to send one copy: F/u_s



- *client:* each client must download file copy
  - min client download time: F/d_{min}
- *clients:* as aggregate must download *NF* bits
  - max upload rate (limiting max download rate) is $u_s + \Sigma u_i$

time to distribute F to N clients using P2P approach

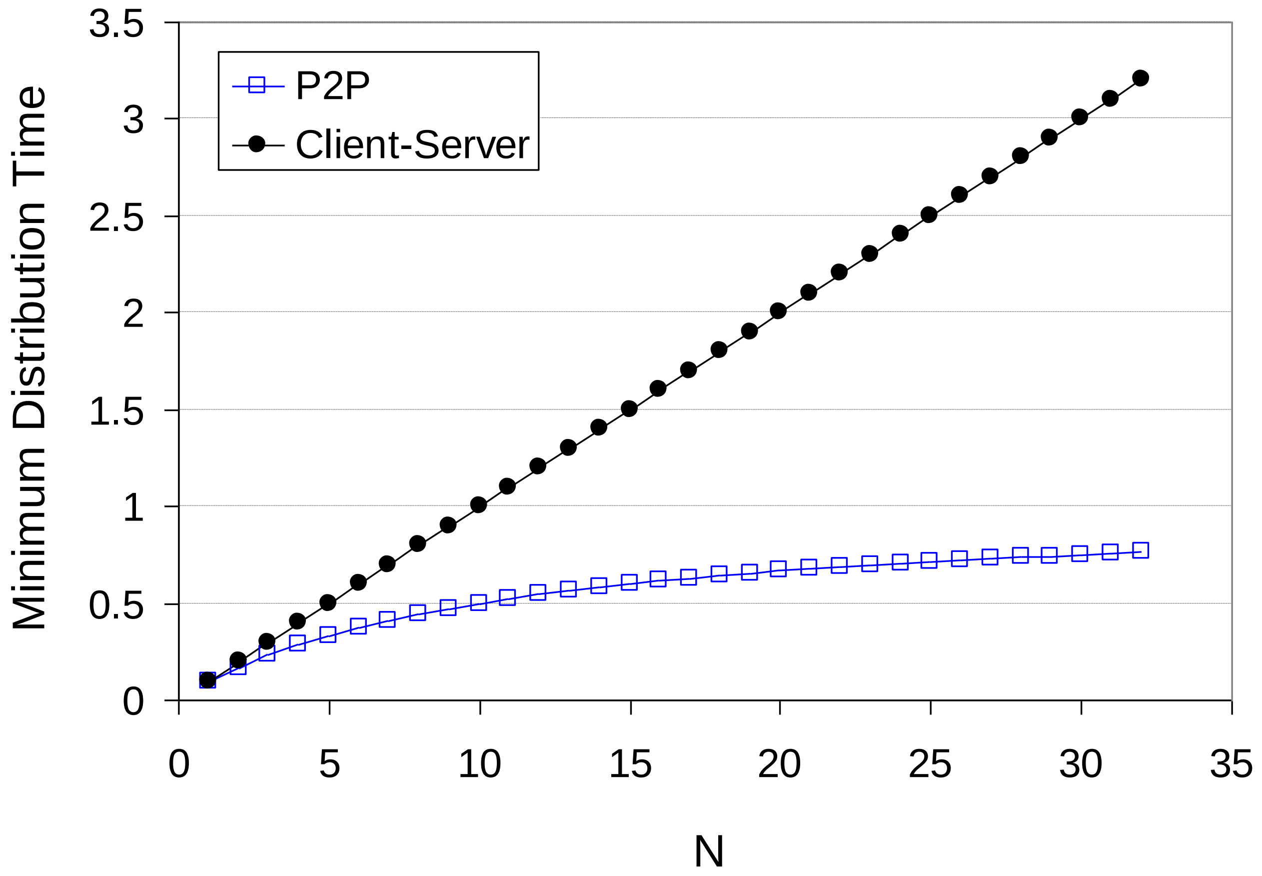
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in *N* …

… but so does this, as each peer brings service capacity

Client-server vs. P2P: example

client upload rate = u, F/u = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$

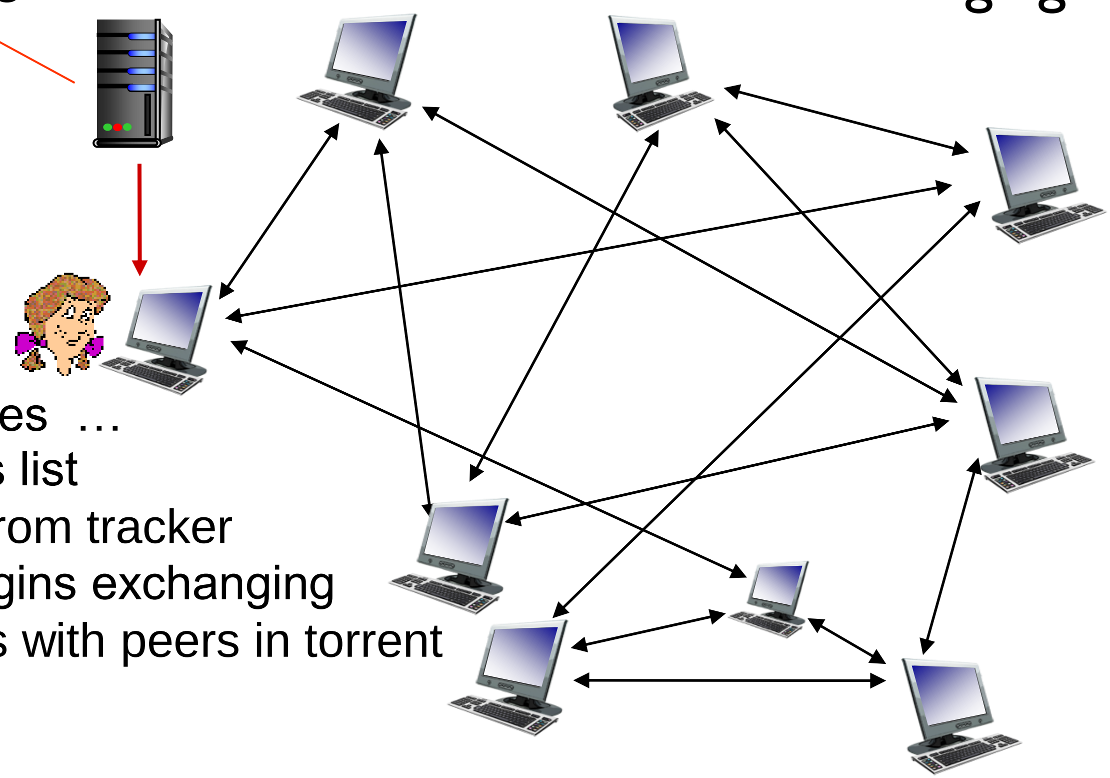


P2P file distribution: BitTorrent

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

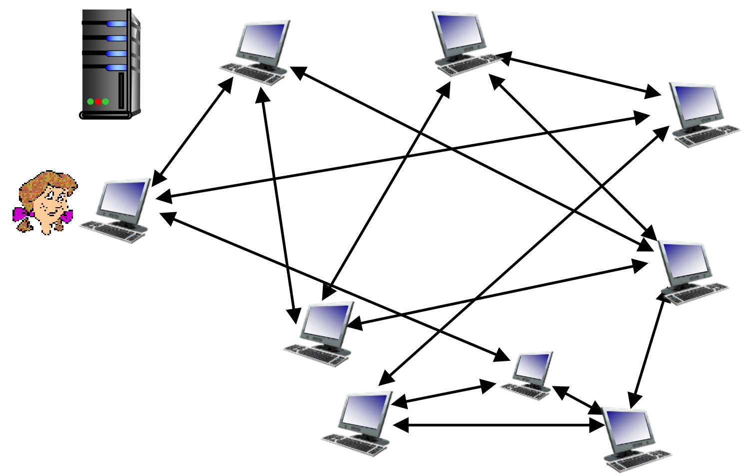
tracker: tracks peers participating in torrent

torrent: group of peers exchanging chunks of a file



Alice arrives …
… obtains list
of peers from tracker
… and begins exchanging
file chunks with peers in torrent

P2P file distribution: BitTorrent



- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")
- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn:* peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

BitTorrent: requesting, sending file chunks

requesting chunks:

- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

sending chunks: tit-for-tat

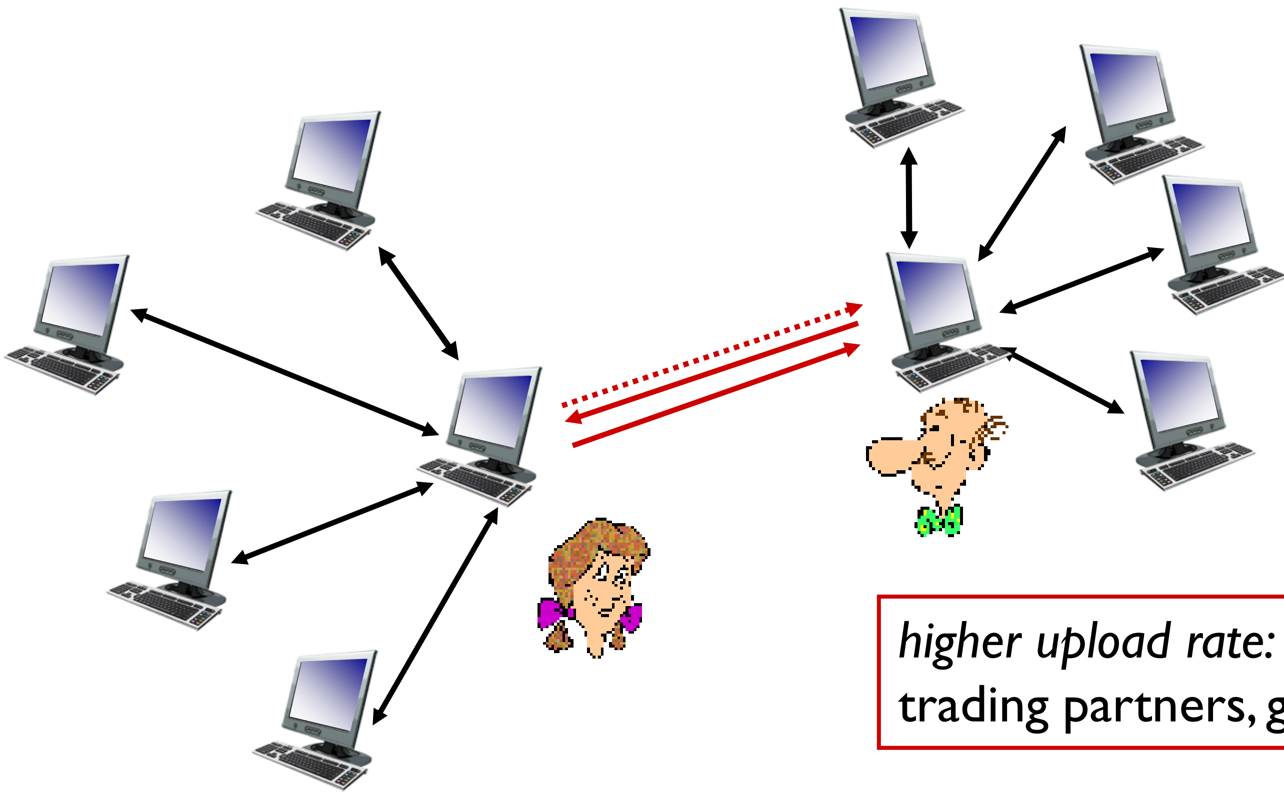
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4

BitTorrent: tit-for-tat

(1) Alice "optimistically unchokes" Bob

(2) Alice becomes one of Bob's top-four providers; Bob reciprocates

(3) Bob becomes one of Alice's top-four providers



higher upload rate: find better trading partners, get file faster !

Blockchain

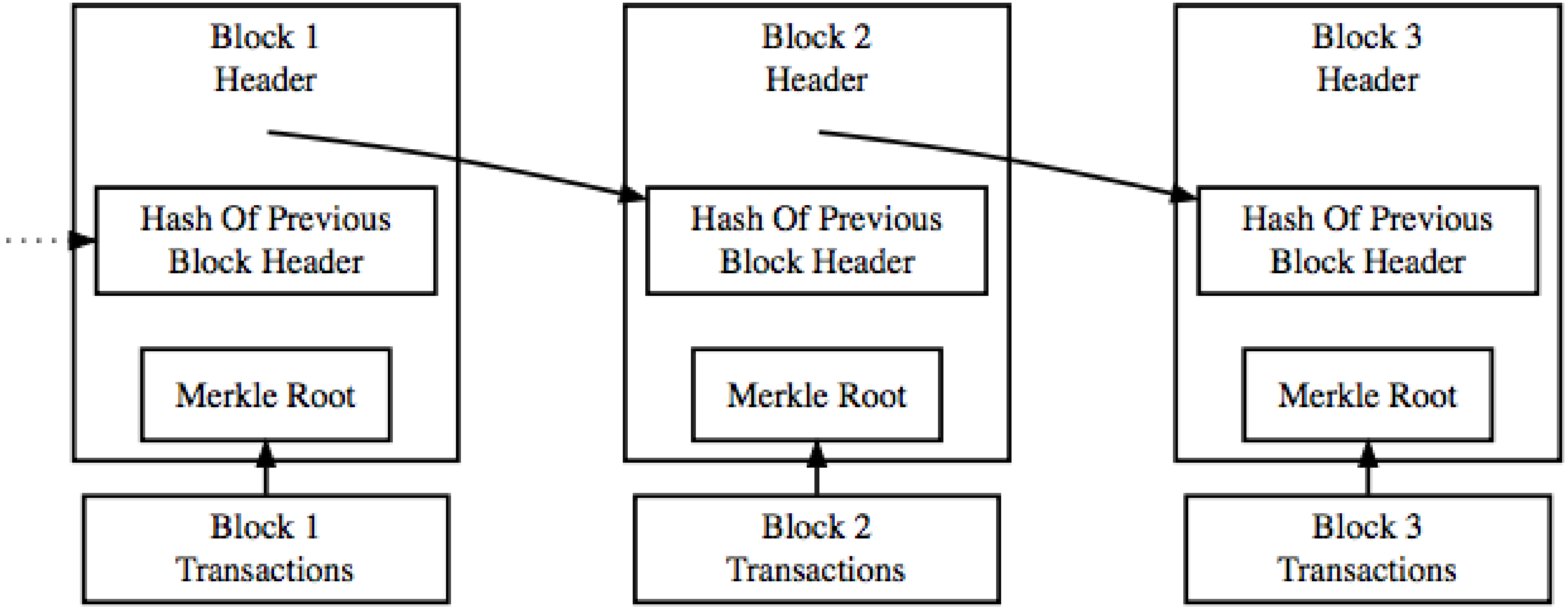
- String of records, or blocks
- Each block contains a hash of the previous block in the chain
- Distributed across many nodes, fairly difficult to modify
  - Nodes are incentivized to reach consensus

Blockchain continued

- Inherently decentralized
  - All nodes have a copy of the blockchain
  - Makes tampering more difficult, must use distributed consensus

- Proof of Work
  - Mining! Finding blocks is hard, verification is easy!
  - Difficulty in mining increases over time

Blockchain illustrated



Simplified Bitcoin Block Chain

Bitcoin

- First widely-adopted cryptocurrency
  - See https://bitcoin.org/en/developer-guide
- Built on blockchain transaction ledger
  - Contents within distributed database
  - Bitcoins are awarded for updating/verifying ledger
- Bitcoin are mined
  - Essentially, rewards for perpetuating the network

Kademlia

Network Programming

DHT?

- Why is Kademlia called a Distributed Hash Table?
  - Partition the key space by using node IDs
  - Each ID stores a subset of the table (hence distributed)
  - How do we partition? We use XOR of ID and key as a hash
- So do nodes store <key,value> pairs as a hash table?
  - They can, but don't need to. Often use linked lists.

Understanding Kademlia

- For HW3 we're not making an exact implementation but…
  - [details in next week's HW3 handout]
  - Will use Python / gRPC to do our RPCs
- Real Kad
  - UDP
  - Viewed as RPCs

Understanding Kademlia

- For HW3 we're not making an exact implementation but...
- Every node generates an ID of length N
  - Fun fact - the behavior for ID collisions is undefined!
- To compute **distance** between IDs, simply use the bitwise XOR.
  - In Python/C/C++ that's operator^
- Keys are also length N. So we can compute node ID vs key **distance** the same way.

Node ID Generation

- In real Kad, just pick from 2^160 randomly
  - If there is a collision, nodes are probably too far to see each other (unless k is large)

OpenSSL SHA-1 Code (C, 1/2)

```
#include <openssl/evp.h>
EVP_MD_CTX *mdctx;
const EVP_MD *md;
char input[] = "12345";
unsigned char id[EVP_MAX_MD_SIZE];
int md_len;

//Create a context
mdctx = EVP_MD_CTX_new();
```

OpenSSL SHA-1 Code (C 2/2)

```
/*Initialize our digest to use the md TYPE object
given by EVP_sha1 */
/*Need to do this every time we want to hash
something */
EVP_DigestInit_ex(mdctx, EVP_sha1(), NULL);
//Add to the hash
EVP_DigestUpdate(mdctx, input, strlen(input));
//Finalize the hash, store in md_value
EVP_DigestFinal_ex(mdctx, id, &md_len);

//Free the context
EVP_MD_CTX_free(mdctx);
```

OpenSSL SHA-1 Code (Python 3)

```
#!/usr/bin/env python3
import hashlib
m = hashlib.sha1()
m.update(b"This is a str")
m.update(b"ring in two parts")
m.digest() #Gives the actual hash
```

k and k-buckets

- Routing is split up into k-buckets. There are **N** of them.
- Each bucket $0 \le i < N$ holds nodes with $2^i \le \textit{distance} < 2^{i+1}$
- If a node is in bucket i, then bit i is the first bit that was different (lowest bit on the left)
- For small i, there are very few possible values, probably an empty bucket.
- For large i, there are many possible values. Don't want to store them all.
- Only store up to k values

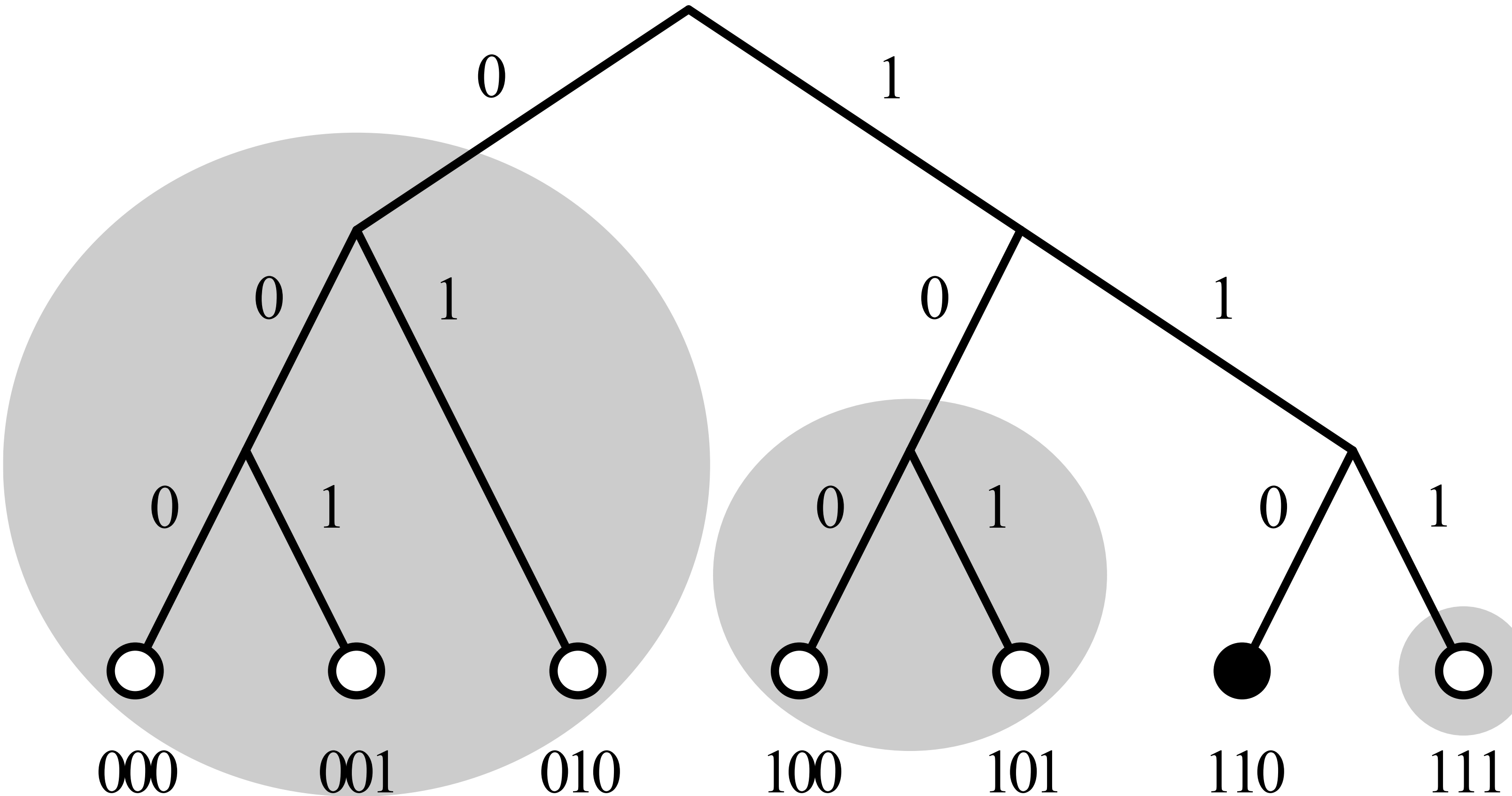
k and k-buckets

- How do we pick which ones to kick out?
- Least Recently Used (LRU) list
  - If we follow the Kad paper the most recently used is the tail and the least recently used is the head
  - Head or tail is fine, as long as it’s an LRU list
  - If you receive a message from a node, send it to the "most recent" end of the list
  - We don't want to update LRU when we send to a node, because it might turn out that node is dead. UDP means we won't get a "connection refused"

k and k-buckets

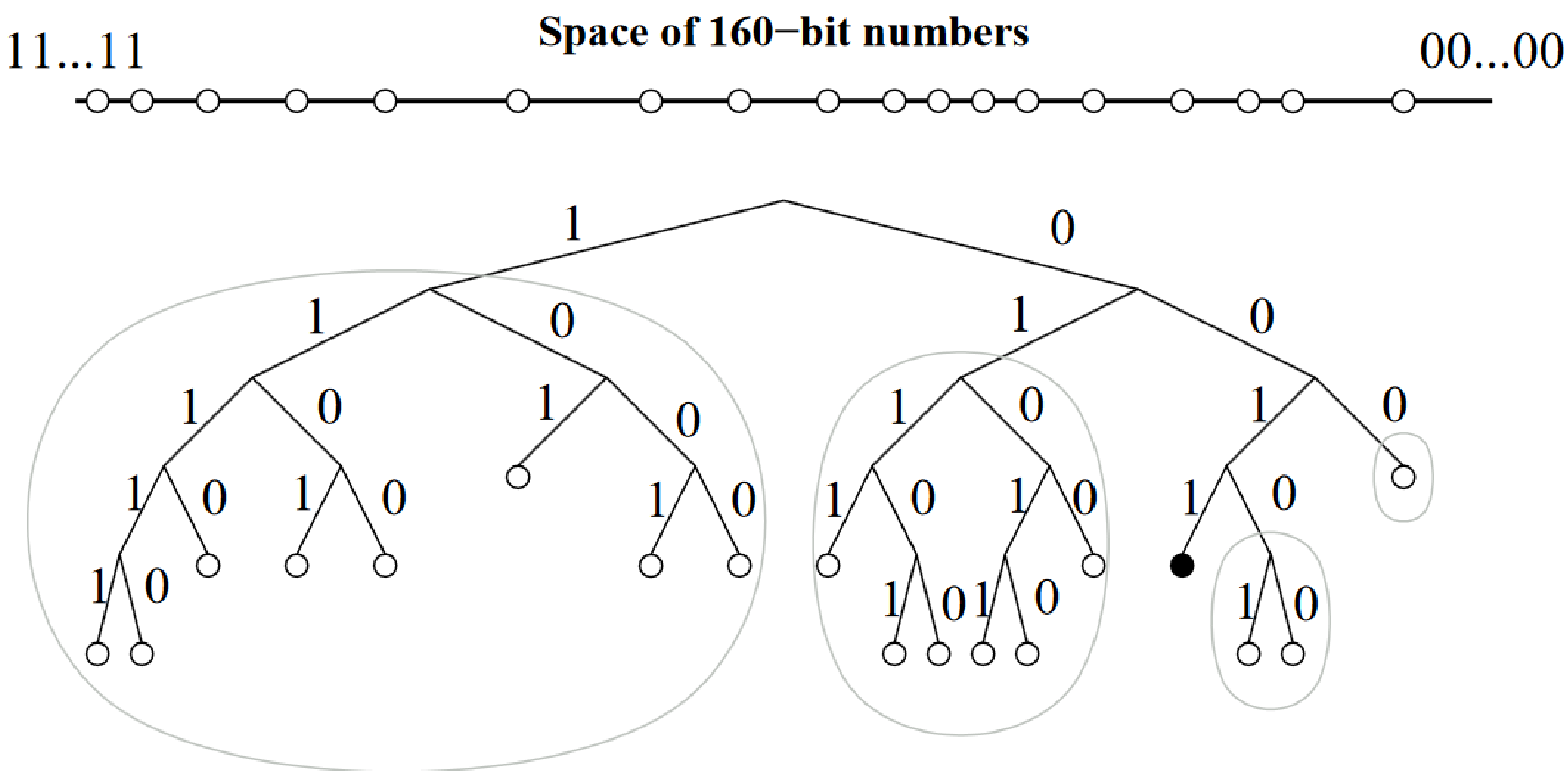
- Replacement Cache
  - We won't do this in HW3
  - In real Kad, can keep a number of nodes as "backup" - if a spot opens up in k-bucket, use a backup node
  - Instead, only kick out if node fails to respond to a PING
  - We won't even implement PING, real systems do though it may be very infrequent, e.g. once per 24 hours

Visualizing k-buckets



Source: https://commons.wikimedia.org/wiki/File:Dht_example_SVG.svg

Visualizing k-buckets



Source: https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

Kademlia Protocol

- Four RPCs
  - FIND_NODE(id)
    - Returns (IP, UDP Port, Node ID) for k closest nodes to id that are known by the remote node
  - FIND_VALUE(key)
    - Same thing as FIND_NODE but if the node is storing a value for that key already, then it just returns the value
  - PING()
    - Check if a node is responding
  - STORE(key,value)
    - Store the key,value pair. Initiator asks k closest nodes to key.

Doing Lookups

- "Concurrency" parameter α
- Pick up to α nodes from closest non-empty k-bucket (we may only know fewer than α)
- In parallel, make the same RPC (FIND_NODE or FIND_VALUE) to each of those nodes
- RPCs will return nodes, update our buckets and repeat the search with another α nodes (but do not ask nodes we already asked)
- Stop when all k closest nodes have been asked

Kademlia Performance

- Hopefully we can do a search in O(log N) time, proving this is a little tricky
- α=1 behaves similar to Chord for searching performance
- If node IDs are very far away, number of nodes is low, and N and k are not chosen well, it is possible not all nodes can reach each other.

Question

- For N=5, k=8, and a network with 32 nodes connected, which node IDs could be in bucket i=3 for the node with ID = 10110?

  You can put your answers in binary, hex, or decimal. (Hint: there will be 8 IDs)