Cristina Nita-Rotaru

CS4700/5700: Network fundamentals

P2P Systems.

Traditional Internet Services Model

- Client-server
  - Many clients, 1 (or more) server(s)
  - Web servers, DNS, file downloads, video streaming
- Problems
  - Scalability: how many users can a server support?
    - What happens when user traffic overload servers?
    - Limited resources (bandwidth, CPU, storage)
  - Reliability: if # of servers is small, what happens when they break, fail, get disconnected, are mismanaged by humans?
  - Efficiency: if your users are spread across the entire globe, how do you make sure you answer their requests quickly?

The Alternative: Peer-to-Peer

- A simple idea
  - Users bring their own resources to the table
  - A cooperative model: clients = peers = servers
- The benefits
  - Scalability: # of “servers” grows with users
    - BYOR: bring your own resources (storage, CPU, B/W)
  - Reliability: load spread across many peers
    - Probability of them all failing is **very** low…
  - Efficiency: peers are distributed
    - Peers can try and get service from nearby peers

Peer-to-Peer Systems Challenges

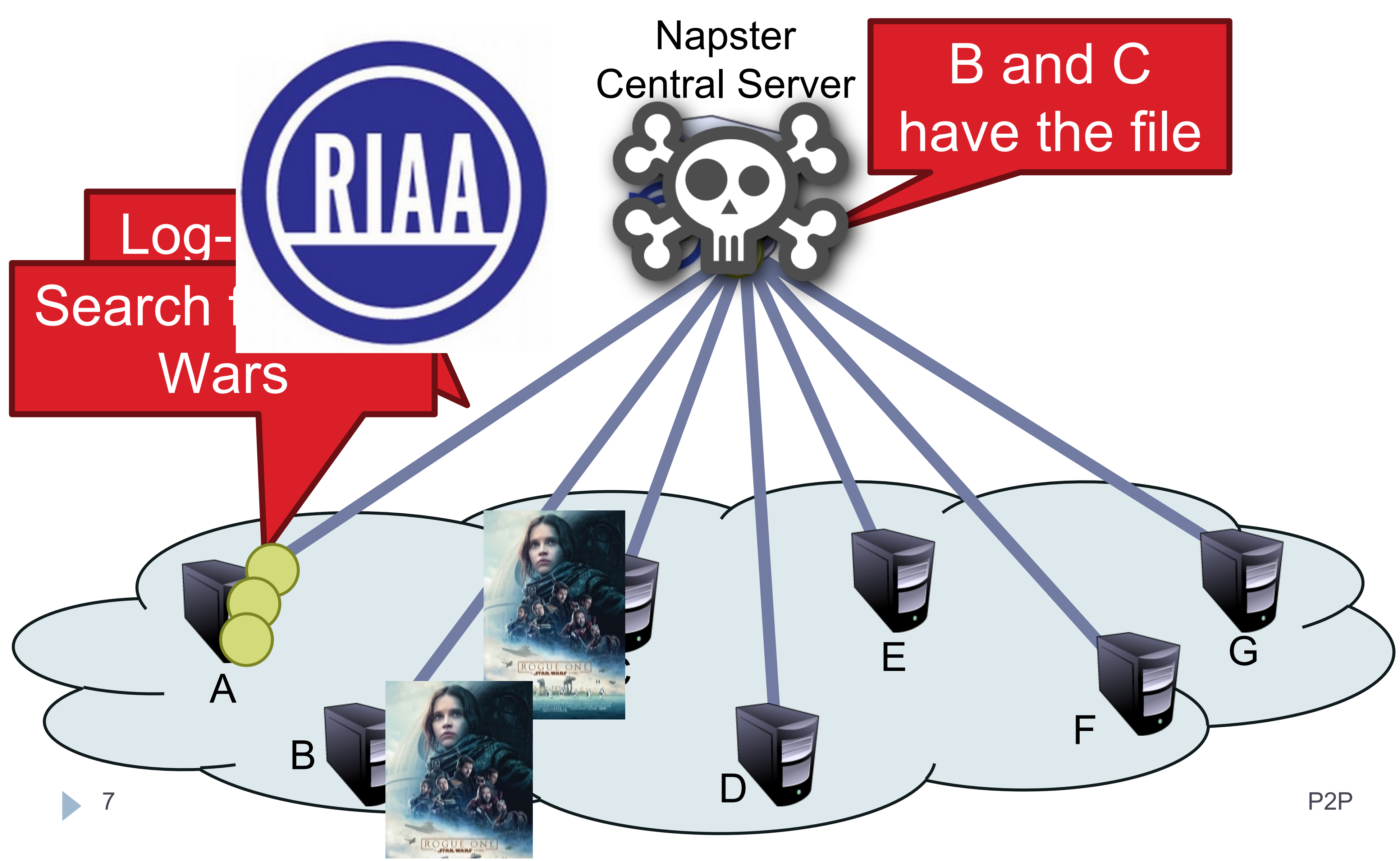
- What are the key components for leveraging P2P?
  - Communication: how do peers talk to each other
  - Service/data location: how do peers know who to talk to
- New reliability challenges
  - Network reachability, i.e. dealing with NATs
  - Dealing with churn, i.e. short peer uptimes
- What about security?
  - Malicious peers and cheating
  - The Sybil attack

1: Napster. Gnutella. Kazaa.

Centralized Approach

- The original: Napster
  - 1999-2001
  - Shawn Fanning, Sean Parker
  - Invented at NEU
  - Specialized in MP3s (but not for long)
- Centralized index server(s)
  - Supported all queries
- What caused its downfall?
  - Not scalable
  - Centralization of liability

Napster Architecture



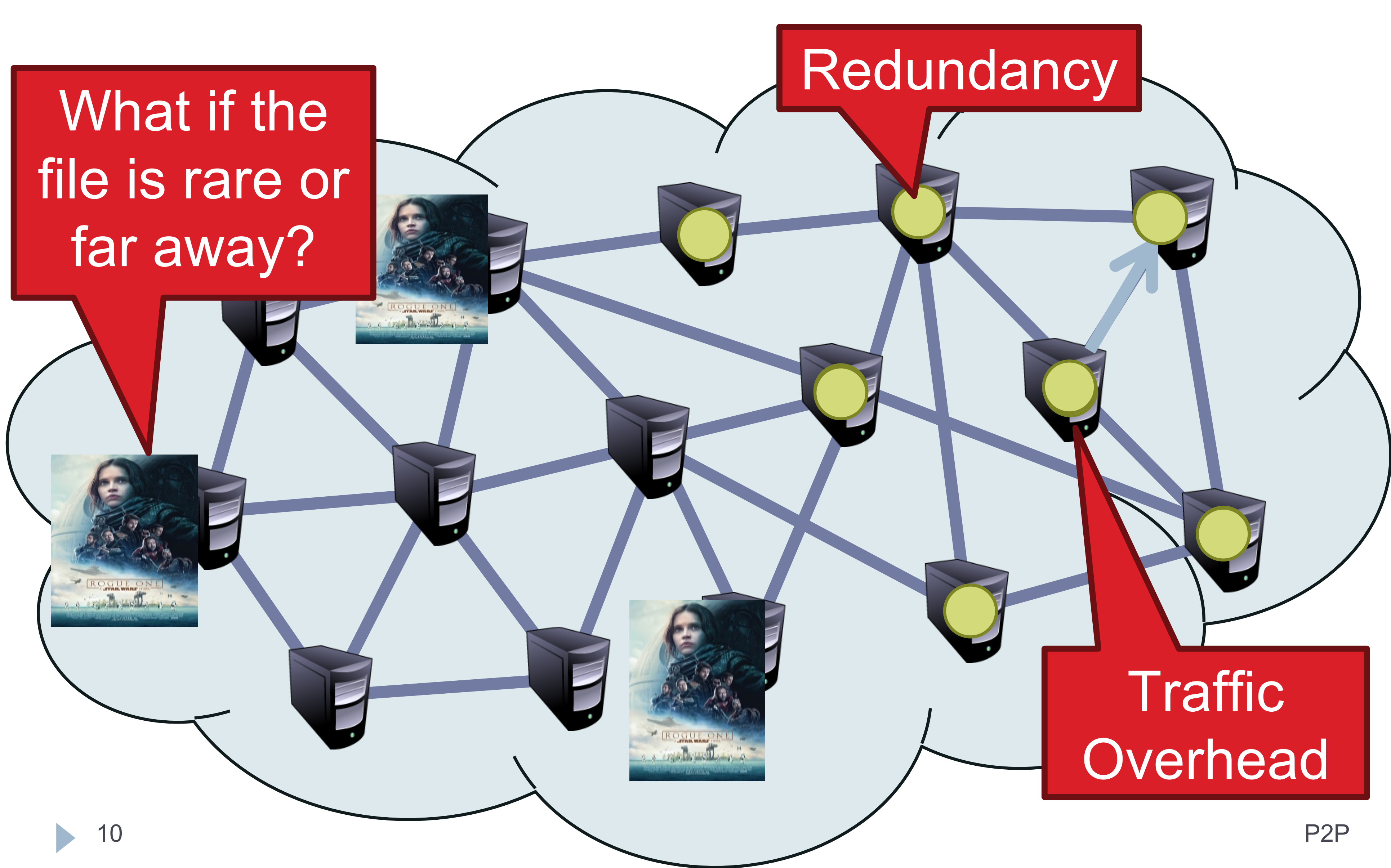
Unstructured P2P Applications

- Centralized systems have single points of failure
- Response: fully unstructured P2P
  - No central server, peers only connect to each other
  - Queries sent as controlled flood
  - Later systems are hierarchical for performance reasons
- Limitations
  - Bootstrapping: how to join without central knowledge?
  - Floods of traffic = high network overhead
  - Probabilistic: can only search a small portion of the system
  - Uncommon files are easily lost

Gnutella

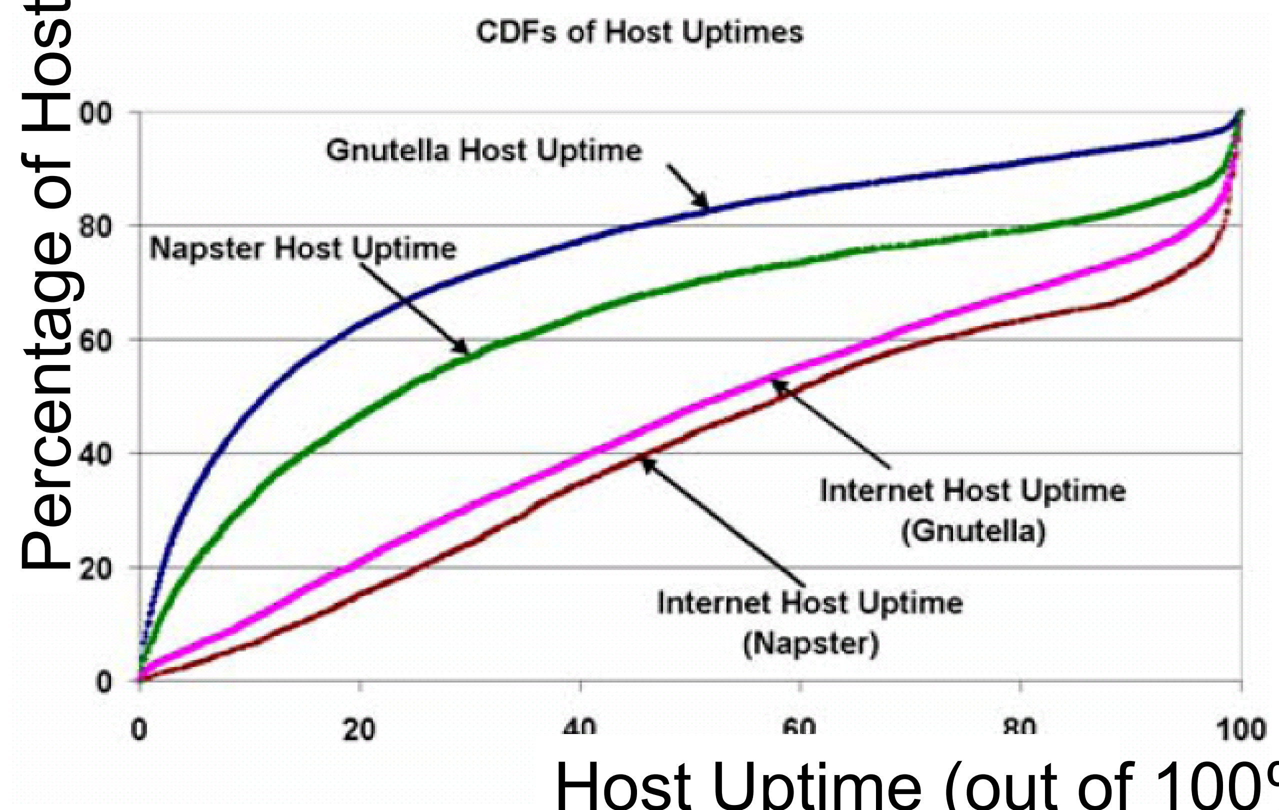
- First massively popular unstructured P2P application
  - Justin Frankel, Nullsoft, 2000
  - AOL was not happy at all
- Original design: flat network
  - Join via bootstrap node
  - Connect to random set of existing hosts
  - Resolve queries by localized flooding
    - Time to live fields limit hops
- Recent incarnations use hierarchical structure
- Problems
  - High bandwidth costs in control messages
  - Flood of queries took up all avail b/w for dialup users

File Search via Flooding in Gnutella



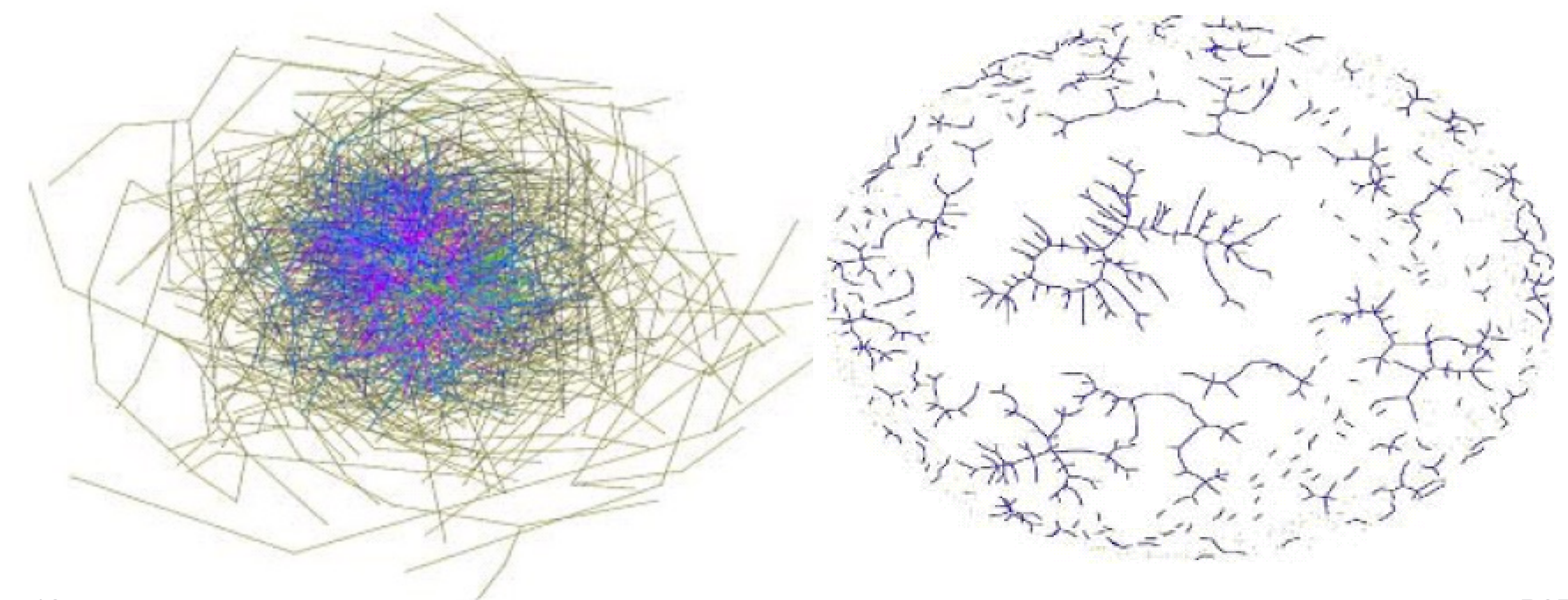
Peer Lifetimes

- Study of host uptime and application uptime (MMCN 2002)
  - Sessions are short (median is 60 minutes)
  - Hosts are frequently offline



Resilience to Failures and Attacks

- Previous studies (Barabasi) show interesting dichotomy of resilience for "scale-free networks"
  - Resilient to random failures, but not attacks
- Here's what it looks like for Gnutella



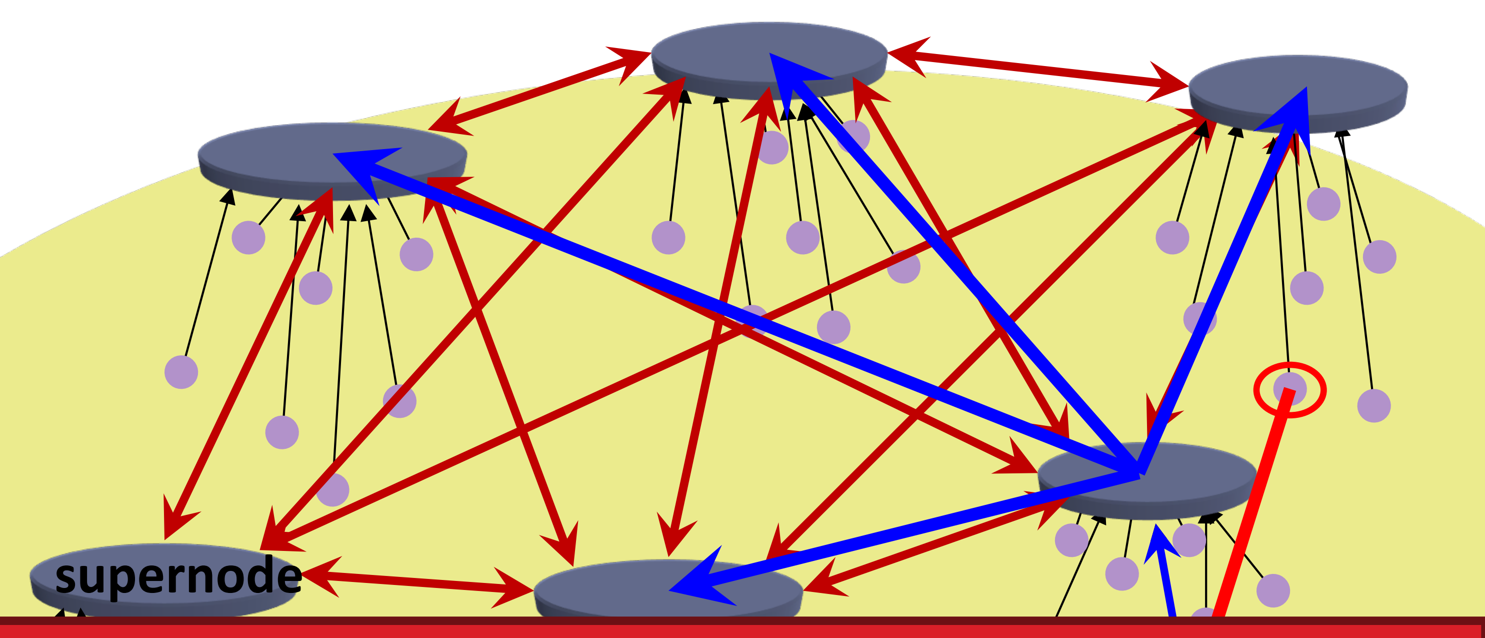
1771 Peers in Feb, 2001

After top 4% of peers are remov

After random 30% of peers rem

Hierarchical P2P Networks

- FastTrack network (Kazaa, Grokster, Morpheus, Gnutella++)



- Improves scalability
- Limits flooding
- Still no guarantees of performance
- What if a supernode leaves the network?

Kazaa

- Very popular from its inception
  - Hierarchical flooding helps improve scale
  - Large shift to broadband helped quite a bit as well
  - Based in Europe, more relaxed copyright laws
- New problem: poison attacks
  - Mainly used by RIAA-like organizations
  - Create many Sybils that distribute "popular content"
    - Files are corrupted, truncated, scrambled
    - In some cases, audio/video about copyright infringement
  - Quite effective in dissuading downloaders

Skype: P2P VoIP

- P2P client supporting VoIP, video, and text based conversation, buddy lists, etc.
  - Based on Kazaa network (FastTrack)
  - Overlay P2P network consisting of ordinary and Super Nodes (SN)
  - Ordinary node connects to network through a Super Node
- Each user registers with a central server
  - User information propagated in a decentralized fashion
- Uses a variant of STUN to identify the type of NAT and firewall

What's Different About Skype

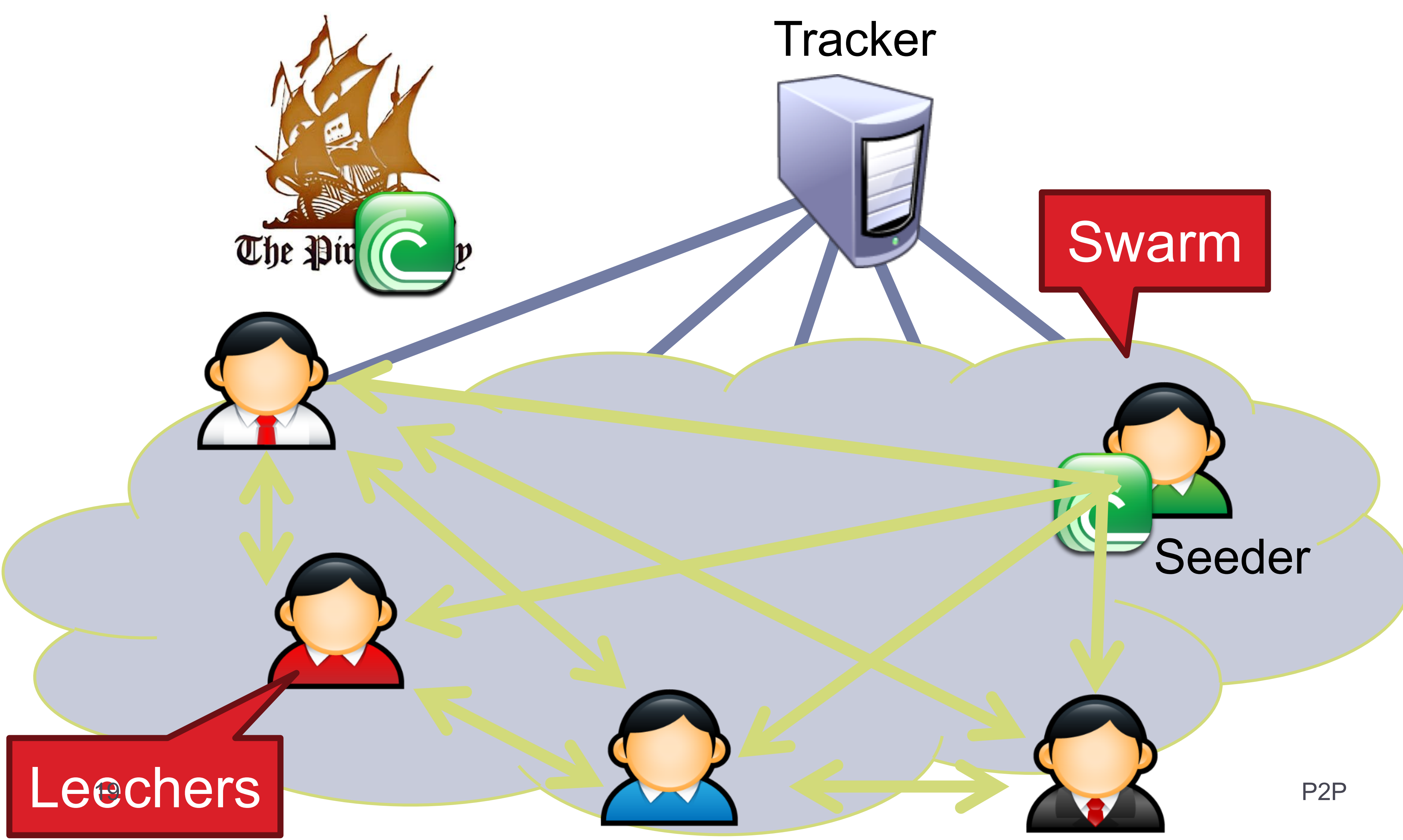
- MSN, Yahoo, GoogleTalk all provide similar functionality
  - But generally rely on centralized servers
- So why peer-to-peer for Skype?
  - One reason: cost
    - If redirect VoIP through peers, can leverage geographic distribution
    - i.e. traffic to a phone in Berlin goes to peer in Berlin, thus becomes a local call
  - Another reason: NAT traversal
    - Choose peers to do P2P rendezvous of NAT'ed clients
- Increasingly, MS is using infrastructure instead of P2P

2: BitTorrent

What is BitTorrent

- Designed for fast, efficient content distribution
  - Ideal for large files, e.g. movies, DVDs, ISOs, etc.
  - Uses P2P file swarming
- Not a full fledged P2P system
  - Does not support searching for files
  - File swarms must be located out-of-band
  - Trackers acts a centralized swarm coordinators
    - Fully P2P, trackerless torrents are now possible
- At times, insanely popular
  - 35-70% of all Internet traffic (in 2007ish)

BitTorrent Overview



.torrent File

- Contains all meta-data related to a torrent
  - File name(s), sizes
  - Torrent hash: hash of the whole file
  - URL of tracker(s)
- BitTorrent breaks files into pieces
  - 64 KB – 1 MB per piece
  - .torrent contains the size and SHA-1 hash of each piece
- Basically, a .torrent tells you
  - Everything about a given file
  - Where to go to start downloading

Torrent Sites

- Just standard web servers
  - Allow users to upload .torrent files
  - Search, ratings, comments, etc.
- Some also host trackers
- Many famous ones
  - Mostly because they host illegal content
- Legitimate .torrents
  - Linux distros
  - World of Warcraft patches

Torrent Trackers

- Really, just a highly specialized webserver
  - BitTorrent protocol is built on top of HTTP
- Keeps a database of swarms
  - Swarms identified by torrent hash
  - State of each peer in each swarm
    - IP address, port, peer ID, TTL
    - Status: leeching or seeding
    - Optional: upload/download stats (to track fairness)
  - Returns a random list of peers to new leechers

Peer Selection

- Tracker provides each client with a list of peers
  - Which peers are best?
    - Truthful (not cheating)
    - Fastest bandwidth
- Option 1: learn dynamically
  - Try downloading from many peers
  - Keep only the best peers
  - Strategy used by BitTorrent
- Option 2: use external information
  - E.g. Some torrent clients prefer peers in the same ISP

Sharing Pieces

Initial Seeder

1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8

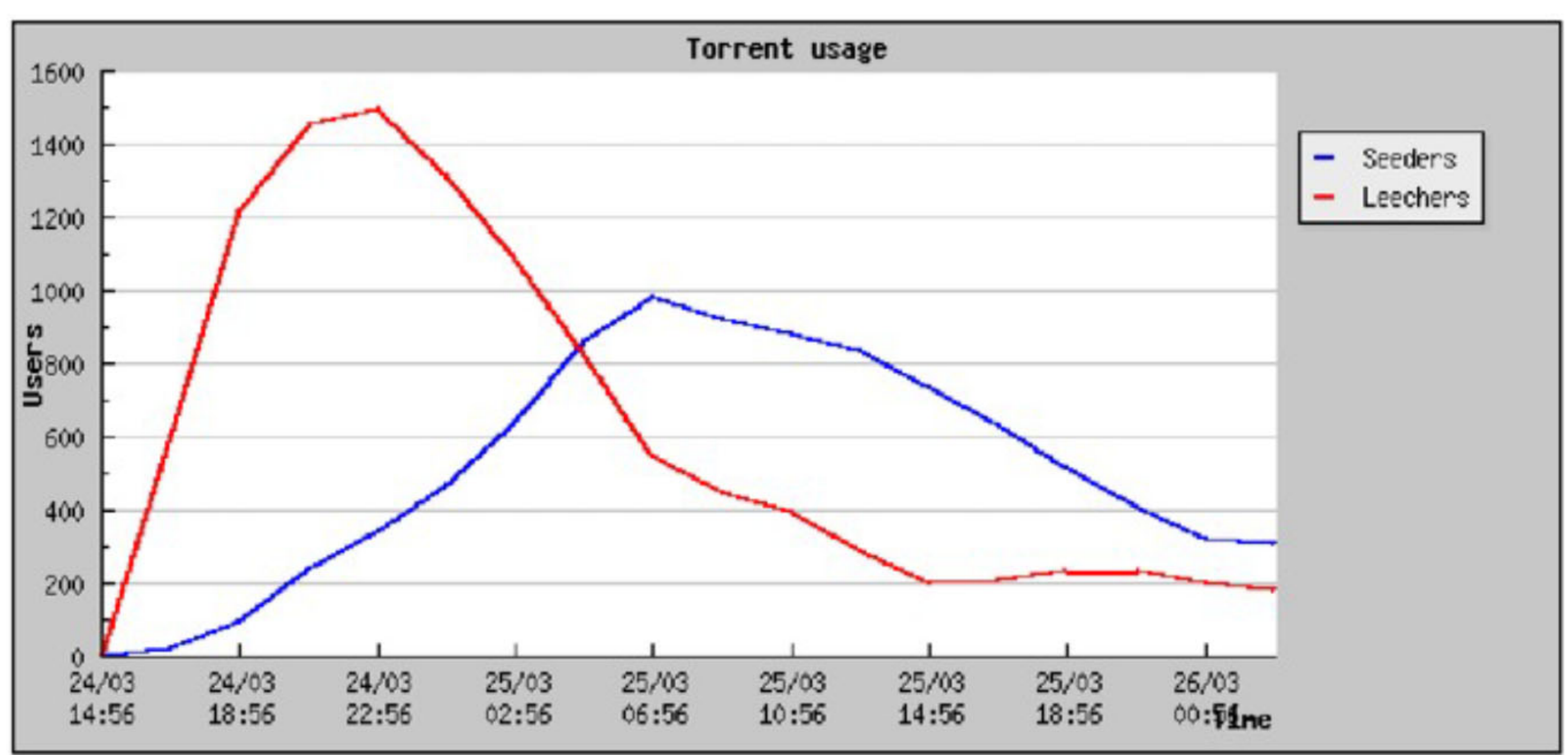
Leecher / Seeder

Leecher / Seeder

The Beauty of BitTorrent

- More leechers = more replicas of pieces
- More replicas = faster downloads
  - Multiple, redundant sources for each piece
- Even while downloading, leechers take load off the seed(s)
  - Great for content distribution
  - Cost is shared among the swarm

Typical Swarm Behavior



Sub-Pieces and Pipelining

- Each piece is broken into sub-pieces
  - ~16 KB in size
- TCP Pipelining
  - For performance, you want long lived TCP connections (to get out of slow start)
  - Peers generally request 5 sub-pieces at a time
  - When one finished, immediately request another
  - Don't start a new piece until previous is complete
    - Prioritizes complete pieces
    - Only complete pieces can be shared with other peers

Piece Selection

- Piece download order is critical
  - Worst-case scenario: all leeches have identical pieces
    - Nobody can share anything :(
  - Worst-case scenario: the initial seed disappears
    - If a piece is missing from the swarm, the torrent is broken
- What is the best strategy for selecting pieces?
  - Trick question
  - It depends on how many pieces you already have

Download Phases

0%

% Downloaded

100%

- Bootstrap: random selection
  - Initially, you have no pieces to trade
  - Essentially, beg for free pieces at random
- Steady-state: rarest piece first
  - Ensures that common pieces are saved for last
- Endgame
  - Simultaneously request final pieces from multiple peers
  - Cancel connections to slow peers
  - Ensures that final pieces arrive quickly

Upload and Download Control

- How does each peer decide who to trade with?
- Incentive mechanism
  - Based on tit-for-tat, game theory
  - "If you give a piece to me, I'll give a piece to you"
  - "If you screw me over, you get nothing"
  - Two mechanisms: choking and optimistic unchoke

A Bit of Game Theory

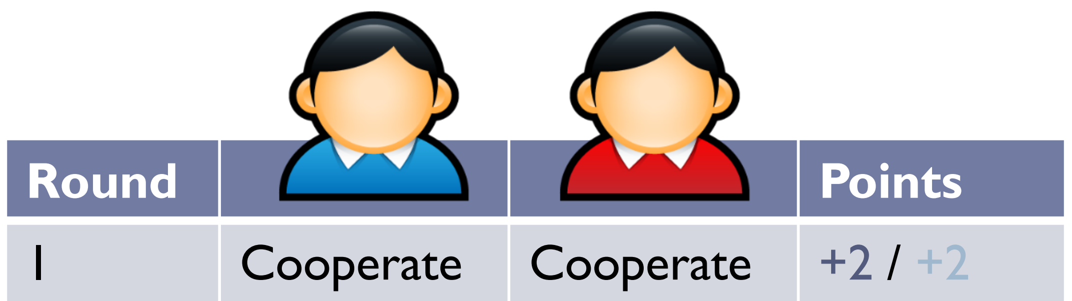
- Iterated prisoner's dilemma
- Very simple game, two players, multiple rounds
  - Both players agree: +2 points each
  - One player defects: +5 for defector, +0 to other
  - Both players defect: +0 for each
- Maps well to trading pieces in BitTorrent
  - Both peers trade, they both get useful data
  - If both peers do nothing, they both get nothing
  - If one peer defects, he gets a free piece, other peer gets nothing
- What is the best strategy for this game?

Tit-for-Tat

- Best general strategy for iterated prisoner's dilemma
- Meaning: "Equivalent Retaliation"

<u>Rules</u>

1. Initially: cooperate
2. If opponent cooperates, cooperate next round
3. If opponent defects, defect next round



Round			Points
1	Cooperate	Cooperate	+2 / +2

Choking

- Choke is a temporary refusal to upload
  - Tit-for-tat: choke free riders
  - Cap the number of simultaneous uploads
    - Too many connections congests your network
  - Periodically unchoke to test the network connection
    - Choked peer might have better bandwidth

Optimistic Unchoke

- Each peer has one optimistic unchoke slot
  - Uploads to one random peer
  - Peer rotates every 30 seconds
- Reasons for optimistic unchoke
  - Help to bootstrap peers without pieces
  - Discover new peers with fast connections

BitTorrent Protocol Fundamentals

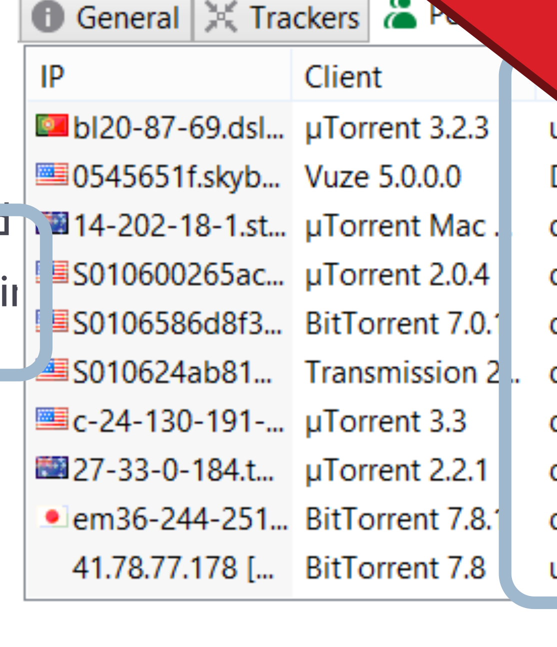
- BitTorrent divides time into rounds
  - Each round, decide who to upload to/download from
  - Rounds are typically 30 seconds
- Each connection to a peer is controlled by four states
  - Interested / uninterested – do I want a piece from you?
  - Choked / unchoked – am I currently downloading from you?
- Connections are bidirectional
  - You decide interest/choking on each peer
  - Each peer decides interest/chocking on you

Connection States

Error states. Connection should be closed.

Most peers are d or D. No need to connect with uninteresting peers.

- ...ed
- ...oked
  - – uninterested and unchoked
  - S – snubbed (no data received i...
  - F – piece(s) failed to hash
- Upload control
  - u – interested and choked
  - U – interested and unchoked
  - O – optimistic unchoke
  - ? – uninterested and unchoked
- Connection information
  - I – incoming connection
  - E/e – Using protocol encryption



More on this later…

- h – used UDP hole pu...ng
- P – connection uses µTP

- How was this peer located?
  - H – DHT (distributed hash table)
  - L – local peer discovery (multicast)
  - X – peer exchange

Upload-Only Mode

- Once a peer completes a torrent, it becomes a seed
  - No downloads, no tit-for-tat
  - Who to upload to first?
- BitTorrent policy
  - Upload to the fastest known peer
  - Why?
  - Faster uploads = more available pieces
  - More available pieces helps the swarm

Trackerless Torrents

- New versions of BitTorrent have the ability to locate swarms without a tracker
  - Based on a P2P overlay
  - Distributed hash table (DHT)
- Recall: peers located via DHT are given "H" state
- More on this next week

2: DHTs. Chord

Distributed Hash Tables

- Decentralized distributed systems that partition a set of keys among participating nodes
- Goal is to efficiently route messages to the unique owner of any given key
- Typically designed to scale to large numbers of nodes and to handle continual node arrivals and failures
- Examples: Chord, CAN, Pastry, Tapestry

DHT Design Goals

- Decentralized system:
  - One node needs to coordinate with a limited set of participants to find the location of a file; should work well in the presence of dynamic membership
- Scalability:
  - The system should function efficiently even with thousands or millions of nodes
- Fault tolerance:
  - The system should be reliable even with nodes continuously joining, leaving, and failing

DHT: Keys and Overlays

- Key space:
  - Ownership of keys is split among the nodes according to some partitioning scheme that maps nodes to keys
- Overlay network:
  - Nodes self organize in an overlay network; each node maintains a set of links to other nodes (its neighbors or routing table).
  - Overlay and routing information is used to locate an object based on the associated key

DHT: Storing an Object

- Compute key according to the object-key mapping method
- Send message store(k,data) to any node participating in the DHT
- Message is forwarded from node to node through the overlay network until it reaches the node S responsible for key k as specified by the keyspace partitioning method
- Store the pair (k,data) at node S (sometimes the object is stored at several nodes to deal with node failures)

DHT: Retrieving an Object

- Compute key according to the object-key mapping method
- Send a message to any DHT node to find the data associated with k with a message retrieve(k)
- Message is routed through the overlay to the node S responsible for k
- Retrieve object from node S

Key Partitioning

- Key partitioning: defines

what node "owns what keys" <=> "stores what objects"

- Removal or addition of nodes should not result in entire remapping of key space since this will result in a high cost in moving the objects around
- Use consistent hashing to map keys to nodes. A function d(k1,k2) defines the distance between keys k1 to key k2. Each node is assigned an identifier (ID). A node with ID i owns all the keys for which i is the closest ID, measured according to distance function d.
- Consistent hashing has the property that removal or addition of one node changes only the set of keys owned by the nodes with adjacent IDs, and leaves all other nodes unaffected.

Overlay Networks and Routing

- Nodes self-organize in a logical network defined by the set of links to other nodes each node must maintain
- Routing:
  - Greedy algorithm, at each step, forward the message to the neighbor whose ID is closest to k.
  - When there is no such neighbor, then this is the closest node, which must be the owner of key k

CHORD

- Efficient lookup of a node which stores data items for a particular search key.
- Provides only one operation: given a key, it maps the key onto a node.
- Example applications:
  - Co-operative mirroring
  - Time-shared storage
  - Distributed indexes
  - Large-scale combinatorial search

Design Goals

- Load balance: distributed hash function, spreading keys evenly over nodes
- Decentralization: CHORD is fully distributed, nodes have symmetric functionality, improves robustness
- Scalability: logarithmic growth of lookup costs with number of nodes in network
- Availability: CHORD guarantees correctness, it automatically adjusts its internal tables to ensure that the node responsible for a key can always be found

Assumptions

- Communication in underlying network is both symmetric and transitive
- Assigns keys to nodes with consistent hashing
- Hash function balances the load
- Participants are correct, nodes can join and leave at any time
- Nodes can fail

Chord Rings

- Key identifier = SHA-1(key)
- Node identifier = SHA-1(IP address)
- Consistent hashing function assigns each node and key an m-bit identifier using SHA-1
- Mapping key identifiers to node identifiers:
  - Identifiers are ordered on a circle modulo 2^m called a chord ring.
  - **The circle is split into contiguous segments whose endpoints are the node identifiers. If i1 and i2 are two adjacent IDs, then the node with ID greater identifier i2 owns all the keys that fall between i1 and i2.**

Example of Key Partitioning in Chord

m = 6
10 nodes

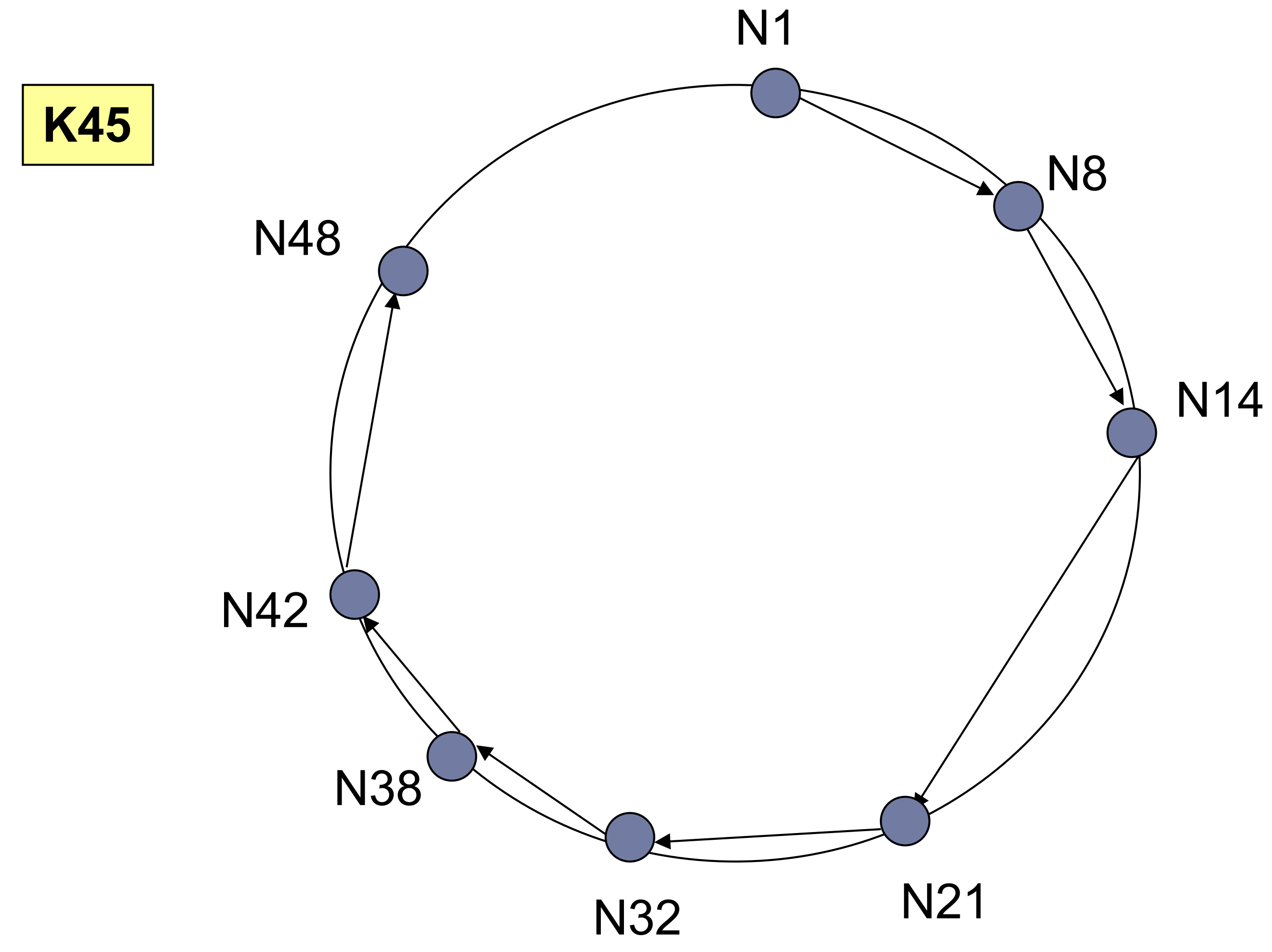
N1
N8
K10
N14
N21
K24
K30
N32
N38
K38
N42
N48
N51
N56
K54

How to Perform Key Lookup

- Assume that each node knows only how to contact its current successor node on the identifier circle, then all node can be visited in linear order.
- When performing a search, the query for a given identifier could be passed around the circle via these successor pointers until they encounter the node that contains the key corresponding to the search.

Example of Key Lookup Scheme

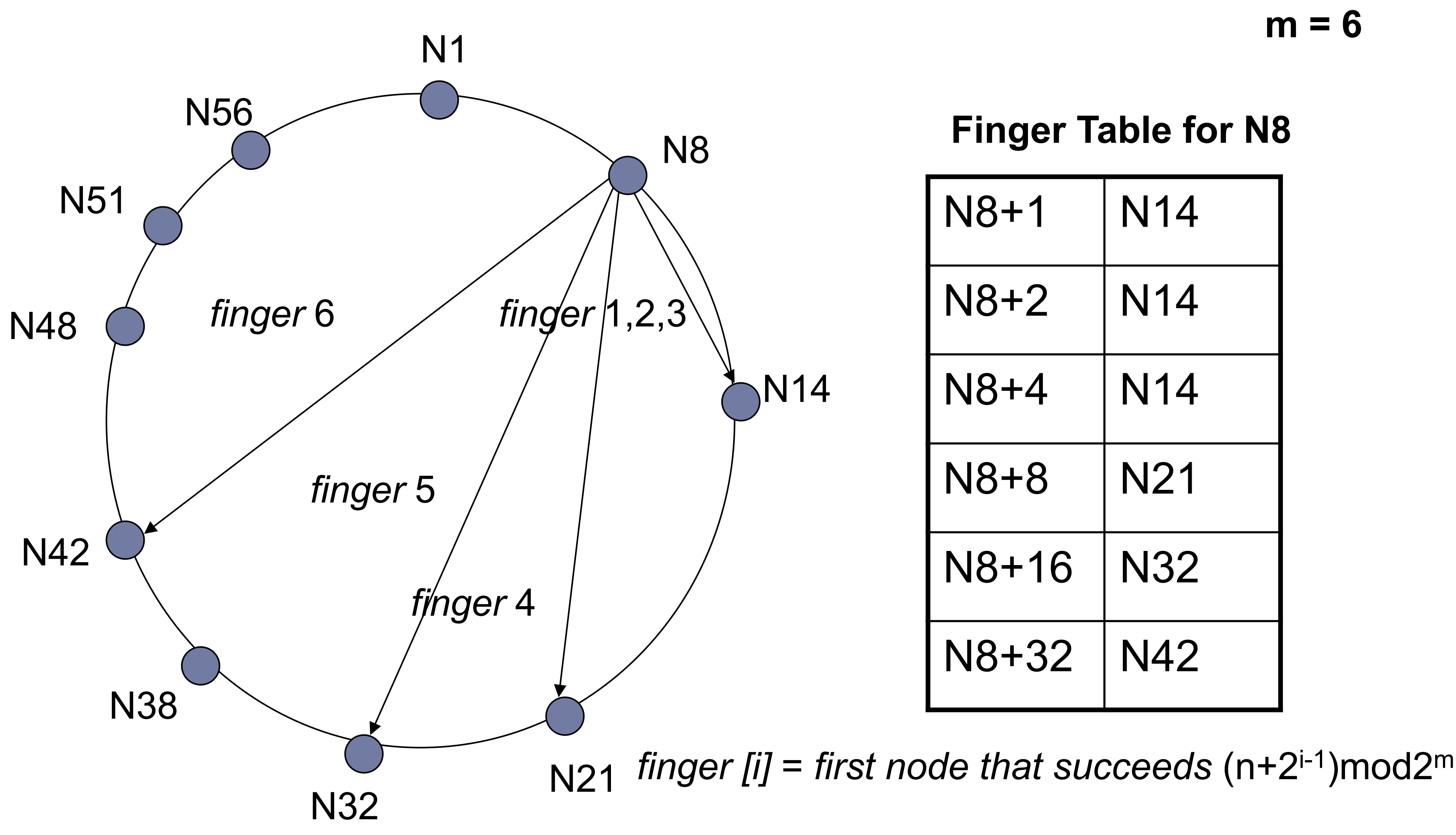
succesor(k) = first node whose ID is >= ID of k in identifier space



Scalable Key Location

- To accelerate lookups, Chord maintains additional routing information (m entries): finger table
- The ith entry in the table at node n contains the identity of the first node s that succeeds n by at least 2^{i-1} on the identifier circle.
- s = successor($n+2^{i-1}$).
- s is called the ith finger of node n

Scalable Lookup Scheme



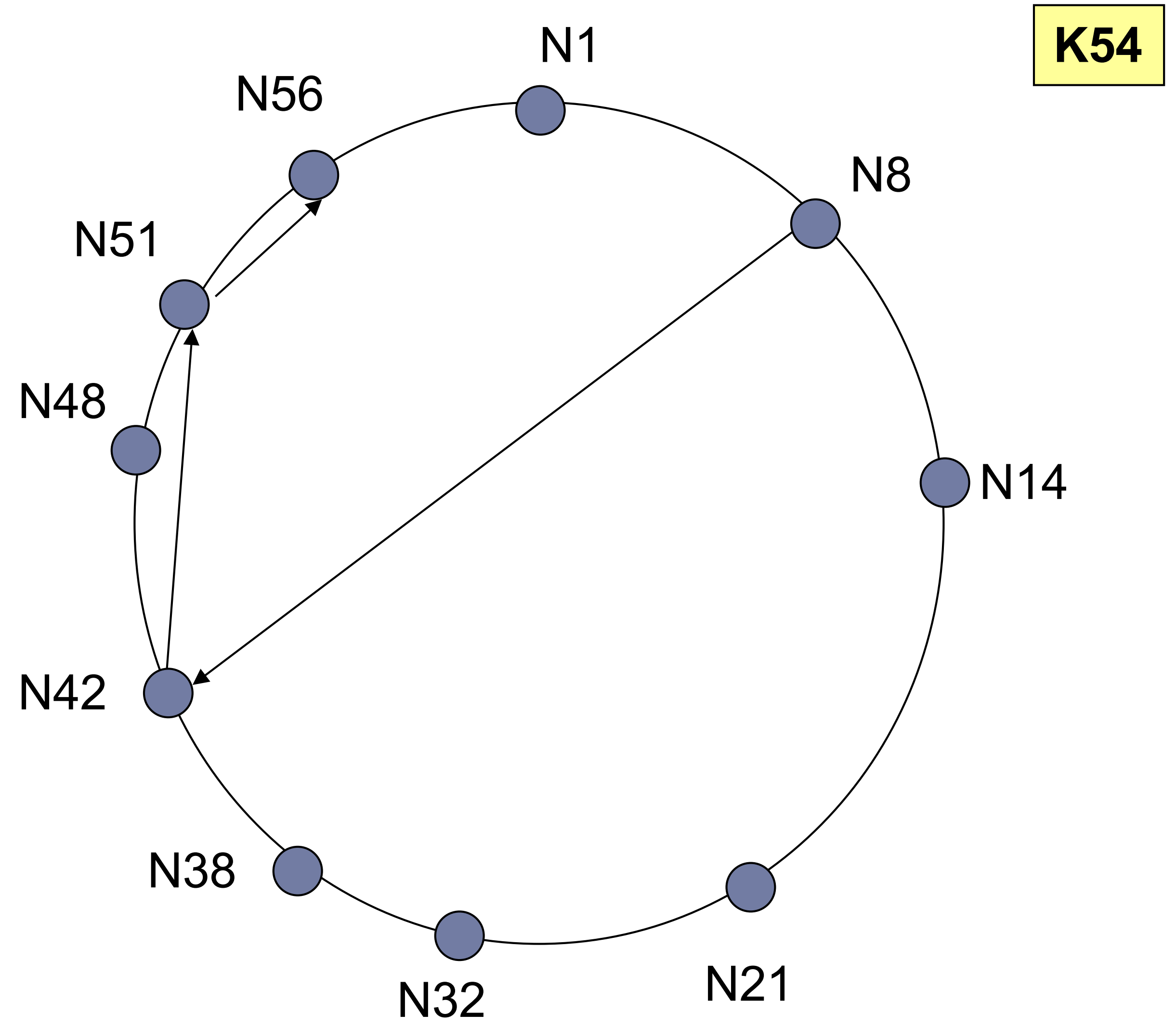
N8+1	N14
N8+2	N14
N8+4	N14
N8+8	N21
N8+16	N32
N8+32	N42

finger [i] = first node that succeeds $(n+2^{i-1}) \bmod 2^m$

Scalable Lookup

- Each node has finger entries at power of two intervals around the identifier circle
- Each node can forward a query at least halfway along the remaining distance between the node and the target identifier.

Lookup Using Finger Table



Node Joins and Failures/Leaves

- When a node N joins the network, some of the keys previously assigned to N's successor should become assigned to N.
- When node N leaves the network, all of its assigned keys should be reassigned to N's successor.
- How to deal with these cases?

Node Joins and Stabilizations

- Everything relies on successor pointer.
- Up to date successor pointer is sufficient to guarantee correctness of lookups
- Idea: run a "stabilization" protocol periodically in the background to update successor pointer and finger table.

Stabilization Protocol

- Guarantees to add nodes in a fashion to preserve reachability
- Does not address the cases when a Chord system thas split into multiple disjoint cycles, or a single cycle that loops multiple times around the identifier space

Stabilization Protocol (cont.)

- Each time node N runs stabilize protocol, it asks its successor for its predecessor p, and decides whether p should be N's successor instead.
- Stabilize protocol notifies node N's successor of N's existence, giving the successor the chance to change its predecessor to N.
- The successor does this only if it knows of no closer predecessor than N.

Impact of Node Joins on Lookups

- If finger table entries are current then lookup finds the correct successor in O(log N) steps
- If successor pointers are correct but finger tables are incorrect, correct lookup but slower
- If incorrect successor pointers, then lookup may fail

Voluntary Node Departures

- Leaving node may transfers all its keys to its successor
- Leaving node may notify its predecessor and successor about each other so that they can update their links

Node Failures

- Stabilize successor lists:
  - Node N reconciles its list with its successor S by copying S's successor list, removing its last entry, and prepending S to it.
  - If node N notices that its successor has failed, it replaces it with the first live entry in its successor list and reconciles its successor list with its new successor.

CHORD Summary

- Efficient location of the node that stores a desired data item is a fundamental problem in P2P networks
- Separates correctness (successor) from performance (finger table)
- Chord protocol solves it in a efficient decentralized manner
  - Routing information: O(log N) nodes
  - Lookup: O(log N) nodes
  - Update: O(log2 N) messages
- It also adapts dynamically to the topology changes introduced during the run