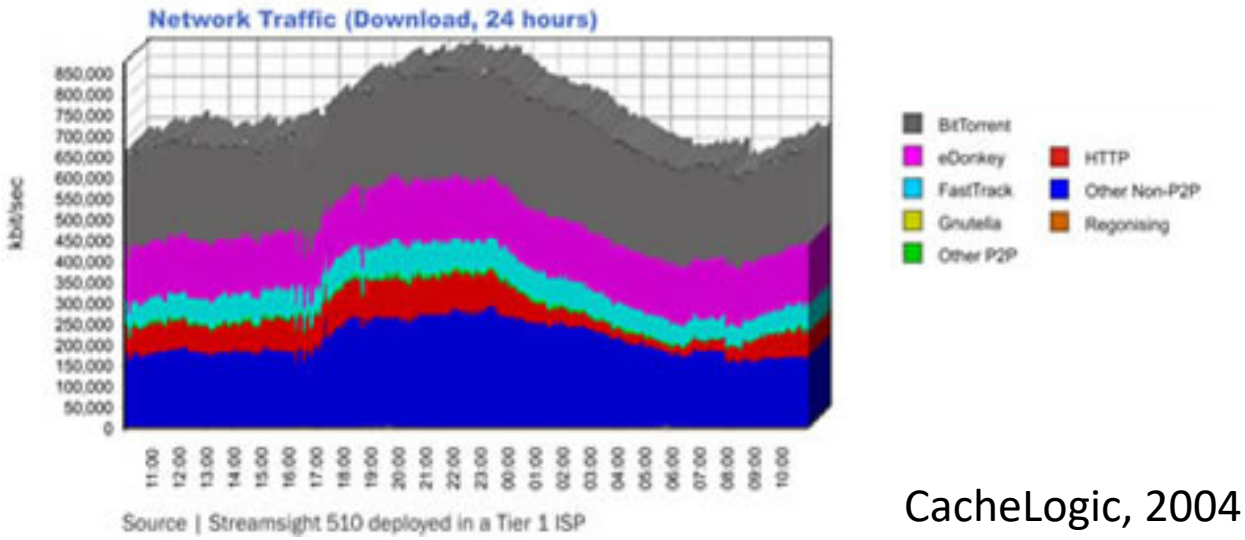
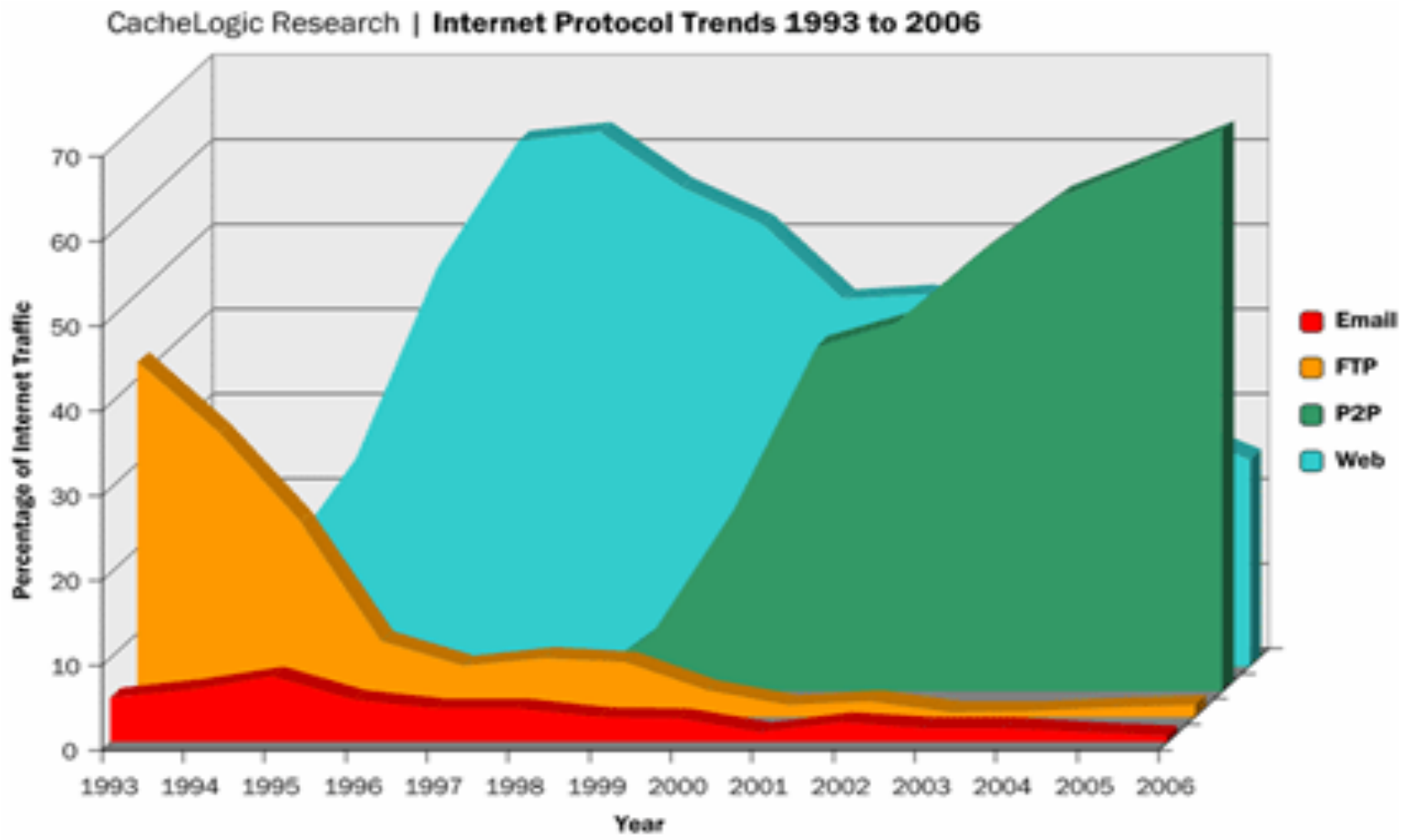


CacheLogic, 2004

PEER-TO-PEER APPLICATIONS

More data
(Source: CacheLogic)



Spotify - Large Scale, Low Latency, P2P Music-on-Demand Streaming

Gunnar Kreitz, Fredrik Niemelä

IEEE P2P'10

Following slides are adapted from authors' slides at P2P in 2010 & 2011.

SPOTIFY

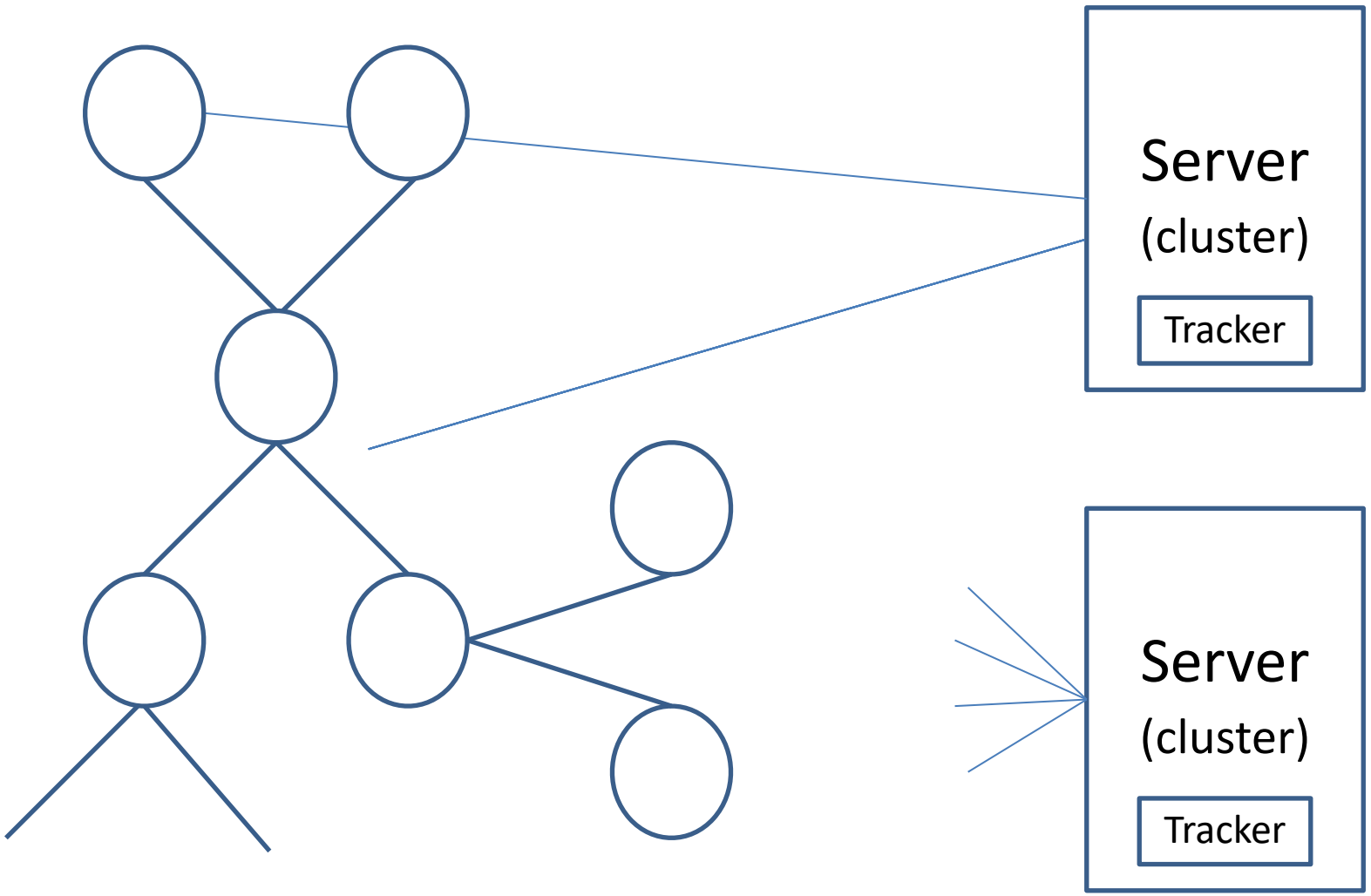
Spotify.com, 2004

- Commercially deployed system, KTH start-up (Sweden)
- Peer-assisted on-demand music streaming
- Legal and licensed content, only
- Large catalogue of music (over 15 million tracks)
- Available in U.S. & 7 European countries, over 10 million users, 1.6 million subscribers (in 2004)
- Fast (median playback latency of 265 ms)
- Proprietary client software for desktop & phone (not p2p)
- Business model: Ad-funded and free & monthly subscription, no ads, premium content, higher quality streaming

Overview of Spotify Protocol

- Proprietary protocol
- Designed for on-demand music streaming
- Only Spotify can add tracks
- 96–320 kbps audio streams (most are Ogg Vorbis q5, 160 kbps)
- Relatively simple and straightforward design
- Phased out in 2014: "*We're now at a stage where we can power music delivery through **our** growing number of **servers** and ensure our users continue to receive **a best-in-class service***."
- Conclusion: *Commercially*, P2P technology is good for **startups** who demand more resources than their servers offer. Avoid "***death by success***".

Spotify architecture: Peer-assisted



Why a Peer-to-peer Protocol?

Why a Peer-to-peer Protocol?

- **Improve scalability** of service

Why a Peer-to-peer Protocol?

- **Improve scalability** of service
- **Decrease load** on servers and network resources

Why a Peer-to-peer Protocol?

- **Improve scalability** of service
- **Decrease load** on servers and network resources

Why a Peer-to-peer Protocol?

- **Improve scalability** of service
- **Decrease load** on servers and network resources

- Explicit design goal

Why a Peer-to-peer Protocol?

- **Improve scalability** of service
- **Decrease load** on servers and network resources

- Explicit design goal
  - Use of peer-to-peer should not decrease overall performance (i.e., playback latency & stutter)

Peer-to-peer Overlay Structure

- Unstructured overlay
- Does not use a DHT
  - **Fast lookup required (Hybrid p2p)**
  - Let us do some rough estimates (ping times)
    - Latency UK – Netherlands ~ 10 ms and up
    - Latency across EU more like ~ 80 ms and up
    - Latency US – Europe ~ 100 ms and up
    - Playback latency ~265 ms
    - <1% Playbacks have stuttering
  - **Simplicity of protocol design & implementation**

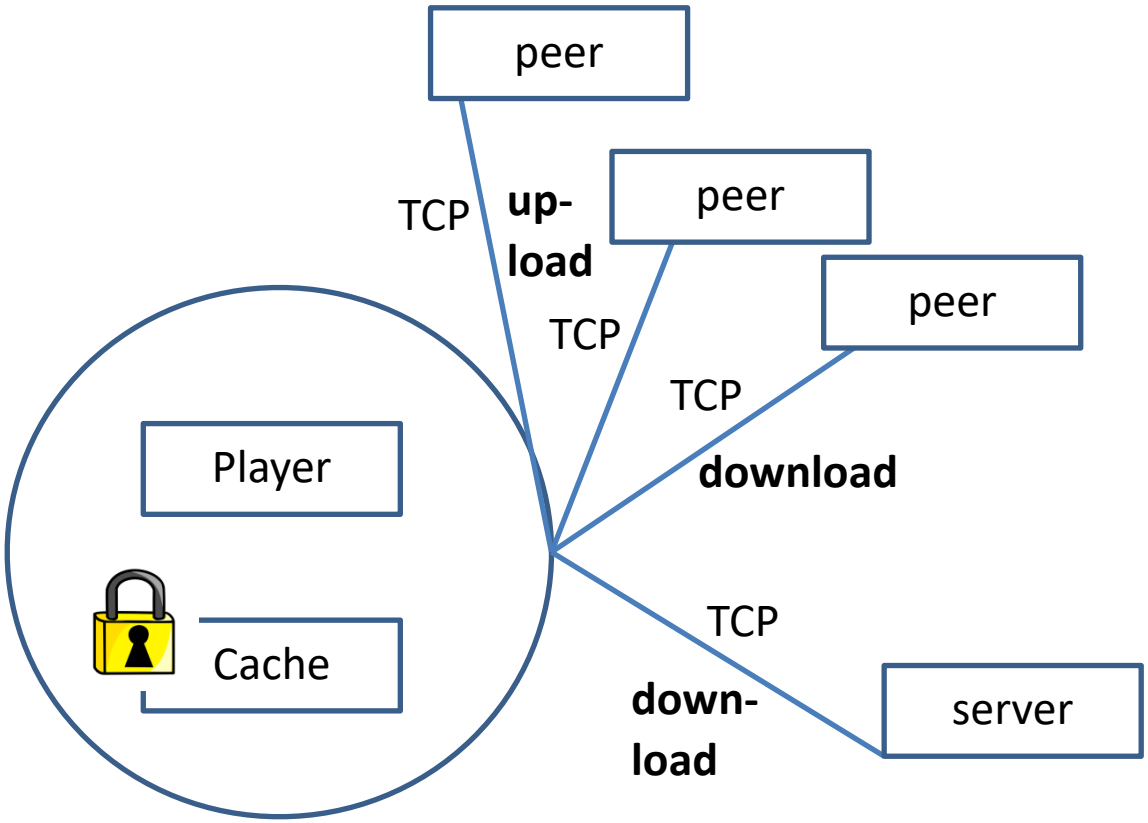
Peer-to-peer Overlay Structure

- Nodes have fixed maximum degree (60)
- Neighbour eviction by heuristic evaluation of utility
- Looks for and connects to new peers when streaming new track
- Overlay becomes (weakly) clustered by interest
- Client only downloads data user needs

Finding Peers

- Server-side tracker (cf. BitTorrent)
  - Only remembers 20 peers per track
  - Returns 10 (online) peers to client on query
- Clients broadcast query in small (2 hops) neighbourhood in overlay (cf. Gnutella)
- Client uses both mechanisms for every track

Peers


peer
TCP
up-
load
peer
TCP
peer
TCP
download
Player
Cache
TCP
down-
load
server

Protocol

- (Almost) everything encrypted
- (Almost) everything over TCP
- Persistent connection to server while logged in
- Multiplex messages over a single TCP connection

Caches

- Client (player) caches tracks it has played
- Default policy is to use 10% of free space (capped at 10 GB)
- Caches are often larger (56% are over 5 GB)
- Least Recently Used policy for cache eviction
- Over 50% of data comes from local cache
- Cached files are served in peer-to-peer overlay (if track completely downloaded)

Streaming a Track

- Tracks are decomposed into 16 kB chunks
- Request first chunk of track from Spotify servers
- Meanwhile, search for peers that cache track
- Download data in-order (chunk by chunk via TCP)
- Towards end of a track, start prefetching next track

Streaming a Track

- If a remote peer is slow, re-request data from new peers
- If local buffer is sufficiently **filled**, only download from peer-to-peer overlay
- If **buffer** is getting **low**, download from central server as well
  - Estimate at what point p2p download could resume
- If **buffer** is **very low**, stop uploading

Security Through Obscurity, ☹

- Music data lies encrypted in caches
- Client must be able to access music data
- Reverse engineers should not be able to access music data
- Details are secret and client code is obfuscated

- **Do not do this "at home"**
  - ***Security through obscurity*** is a bad idea
  - It is a matter of time until someone hacks the Spotify client (cf. the various Skype reverse engineering efforts)

Data sources: 8.8% from servers, 35.8% from p2p network, 55.4 % from caches

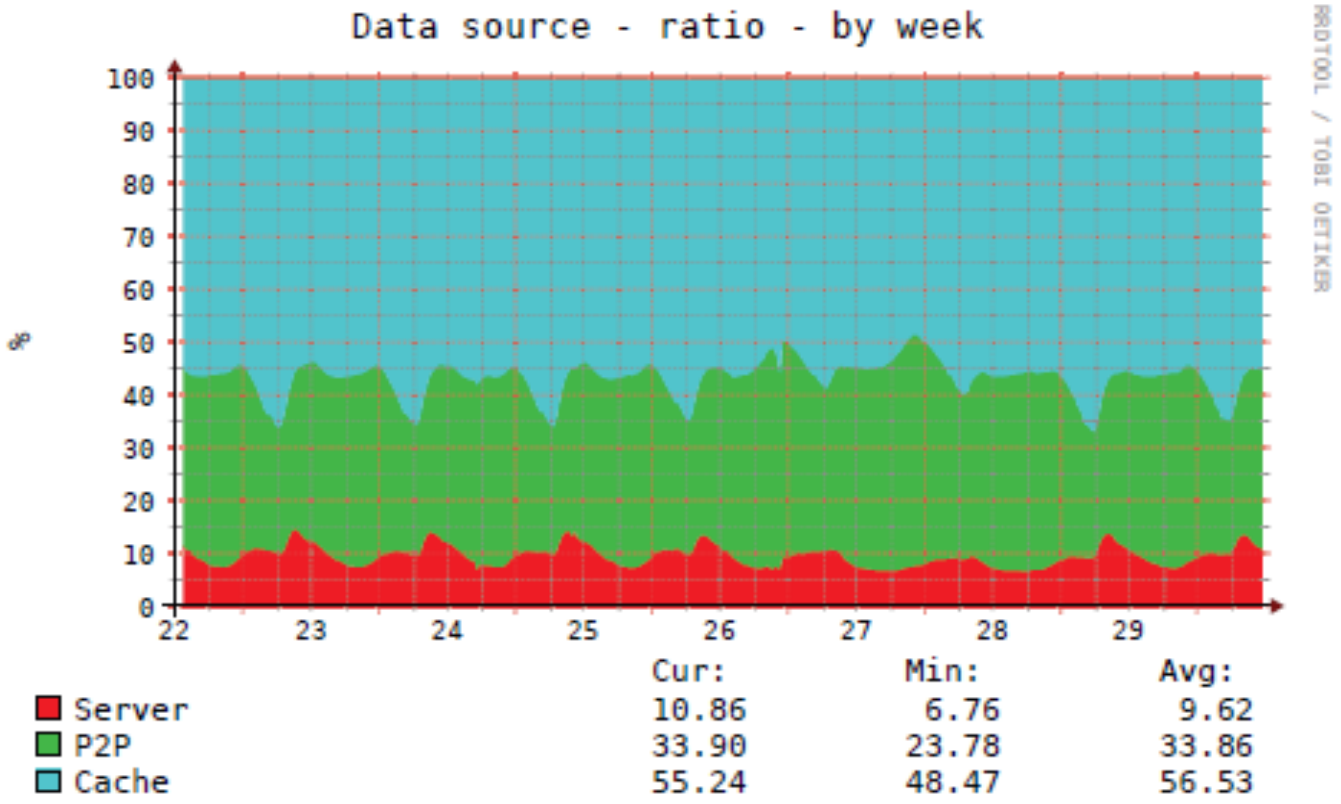
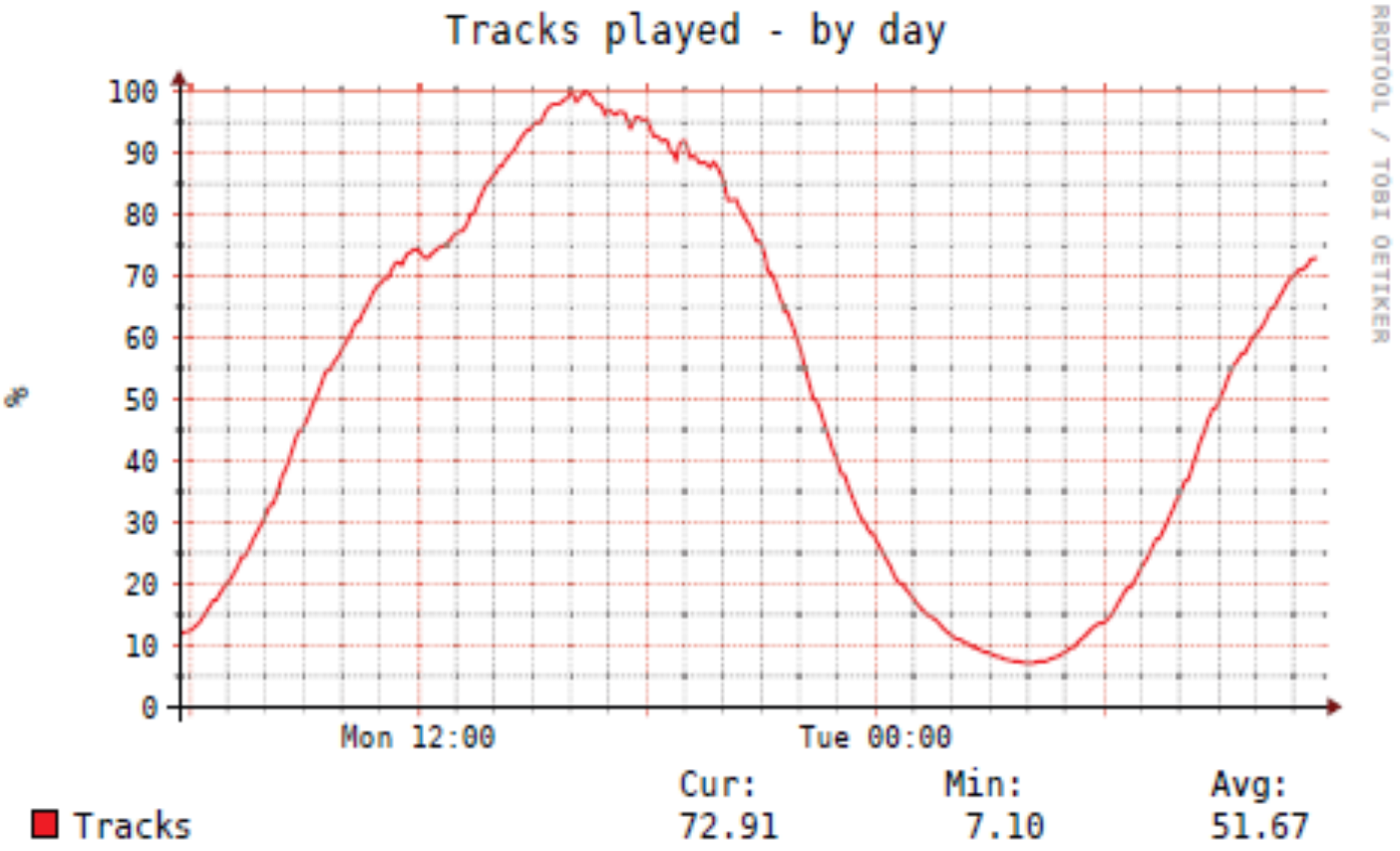


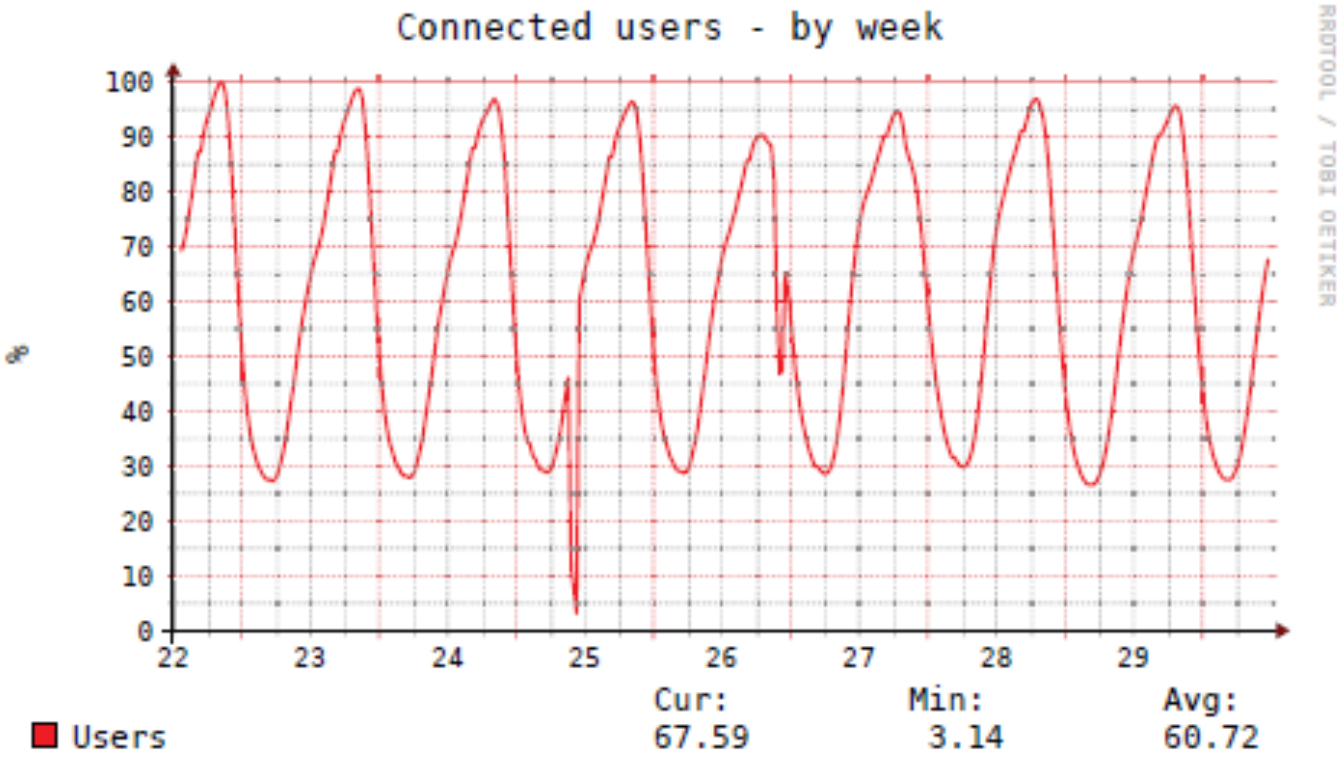
Figure 2. Sources of data used by clients

Tracks played



(a) Tracks played

Users connected



(b) Users connected

Key Points

- Simplicity of architecture, protocol, design
- Peer-assisted, i.e., rely on centralized server
- Use of peer-to-peer techniques for scalability and avoid heavy, over-provisioned infrastructure
- Use of centralized tracker

Incentives build robustness in BitTorrent
by Bram Cohen

BitTorrent Protocol Specification
http://www.bittorrent.org/protocol.html

BIT TORRENT

BitTorrent

- Written by Bram Cohen (in Python) in 2001
- Pull-based, swarming approach (segmented)
  - Each file is split into smaller pieces (& sub-pieces)
  - Peers request desired pieces from neighboring peers
  - Pieces are not downloaded in sequential order
- Encourages contribution by all peers
  - Based on a **tit-for-tat model**

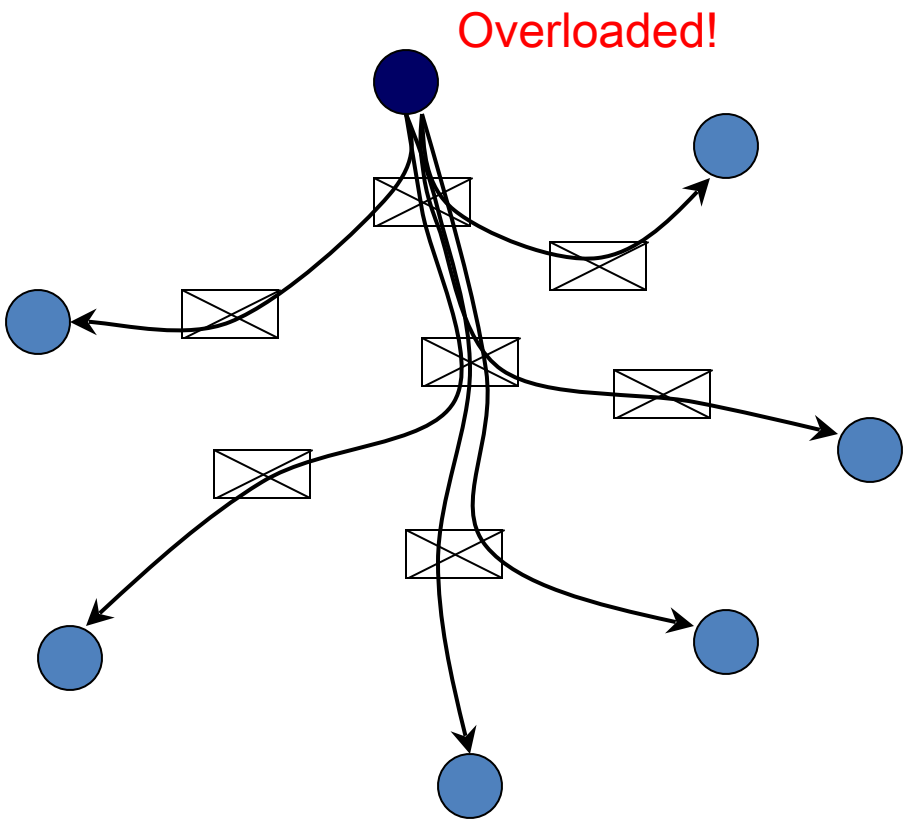
BitTorrent Use cases

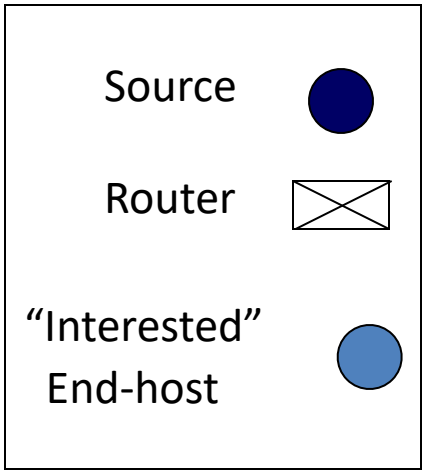
- File-sharing
- *What uses does BitTorrent support?*
  - Downloading (licensed only ☺) movies, music, etc.
  - *And …?*

BitTorrent Use cases

- File-sharing
- *What uses does BitTorrent support?*
  - Downloading (licensed only ☺) movies, music, etc.
  - *And …?*
    - Update distribution among servers at Facebook et al.
    - Distribution of updates and releases (e.g., World of Warcraft -> Blizzard Downloader)
    - …

Client-server





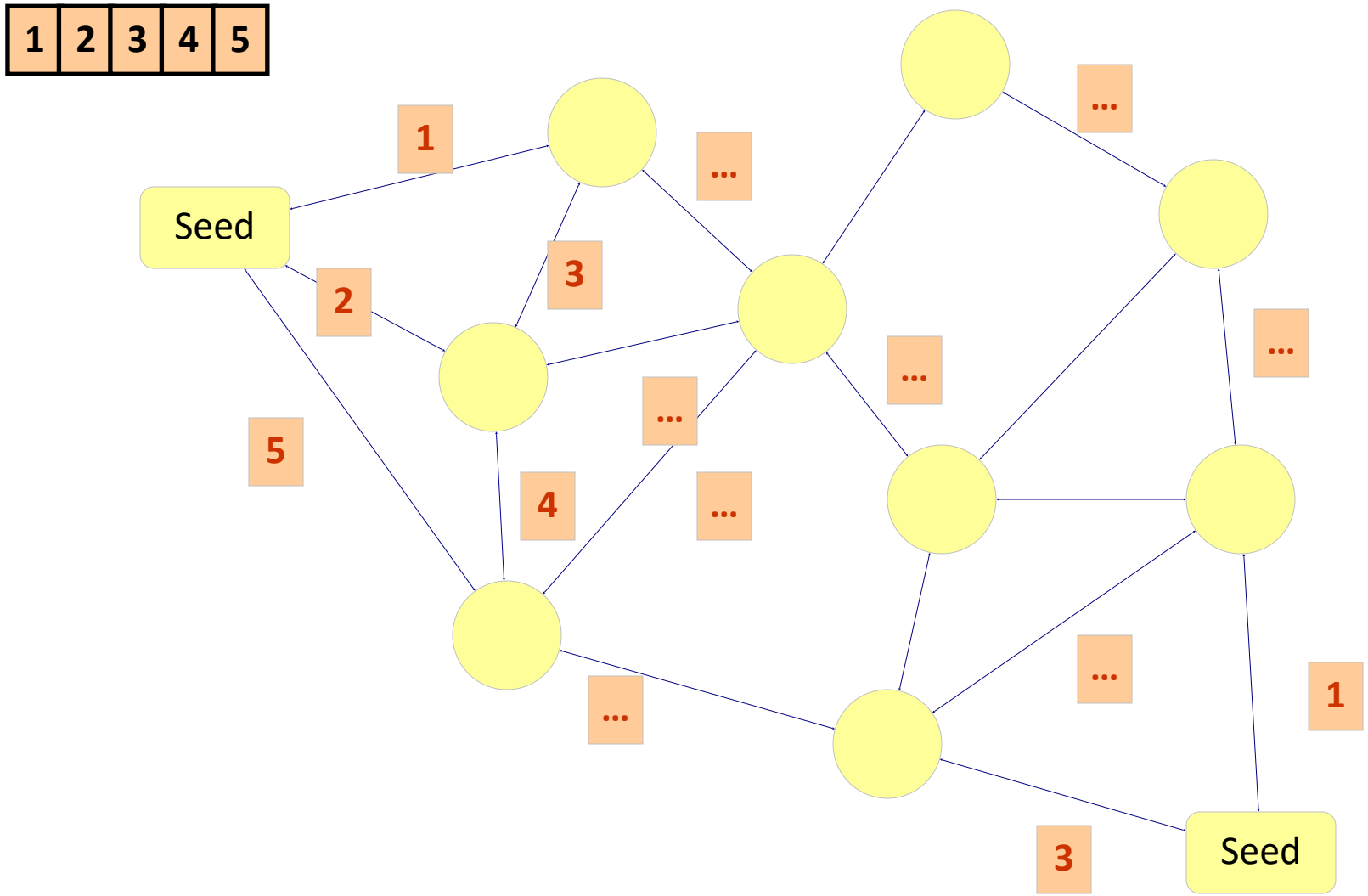
BitTorrent Swarm

- **Swarm**
  - Set of peers downloading the same file
  - Organized as a randomly connected mesh of peers
- Each peer knows list of pieces downloaded by neighboring peers
- Peer requests pieces it does not own from neighbors

BitTorrent Terminology

- **Seed**: Peer with the entire file
  - **Original Seed**: First seed for a file
- **Leech**: Peer downloading the file
  - Leech becomes a seed, once file downloaded, if the peer stays online & continues by protocol
- **Sub-piece**: Further subdivision of a piece
  - “Unit for requests” is a sub-piece (16 kB)
  - Peer uploads piece only after assembling it completely

BitTorrent Swarm



* From Analyzing and Improving BitTorrent by Ashwin R. Bharambe, Cormac Herley and Venkat Padmanabhan

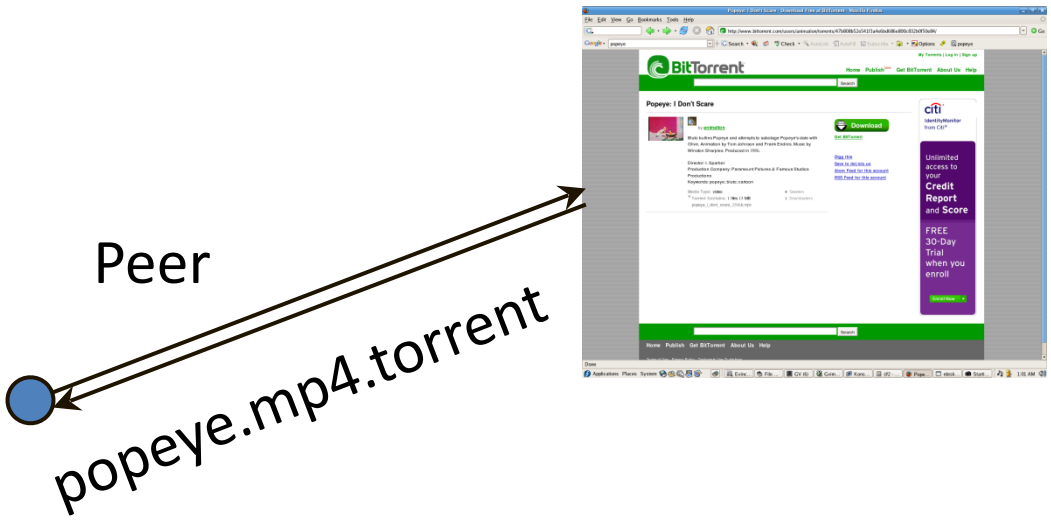
Entering a Swarm

for file “popeye.mp4”

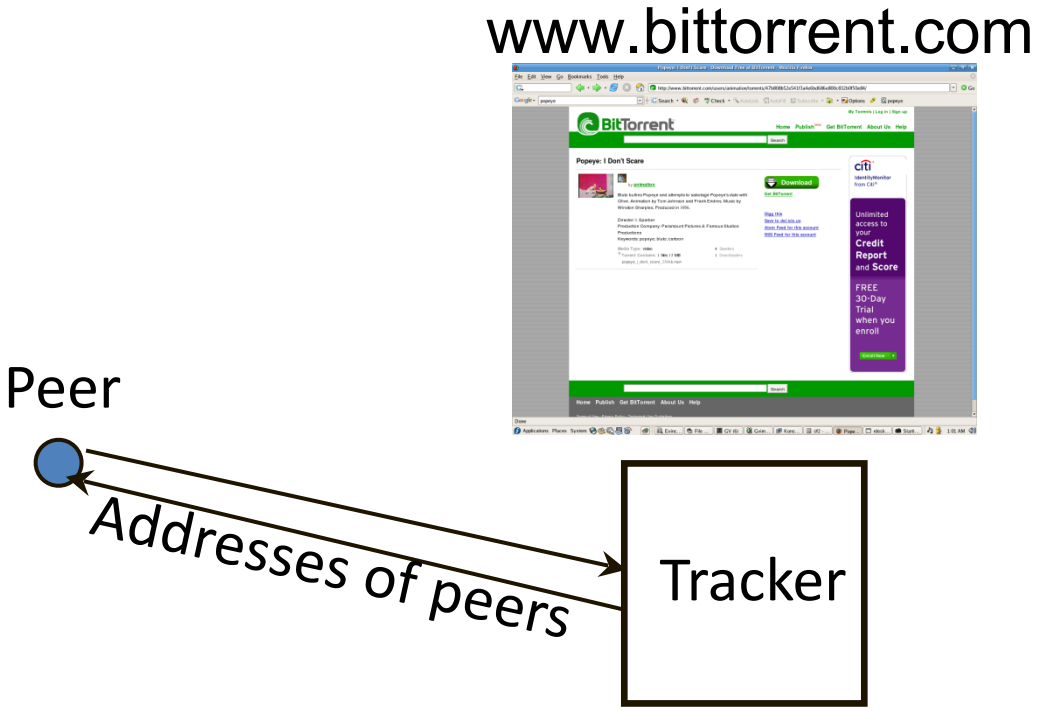
- File **popeye.mp4.torrent** hosted at a (well-known) web server
- The **.torrent** has address of **tracker** for file
- The tracker, which runs on a web server as well, keeps track of all peers downloading file

Find the Torrent for Desired Content

www.bittorrent.com or anywhere

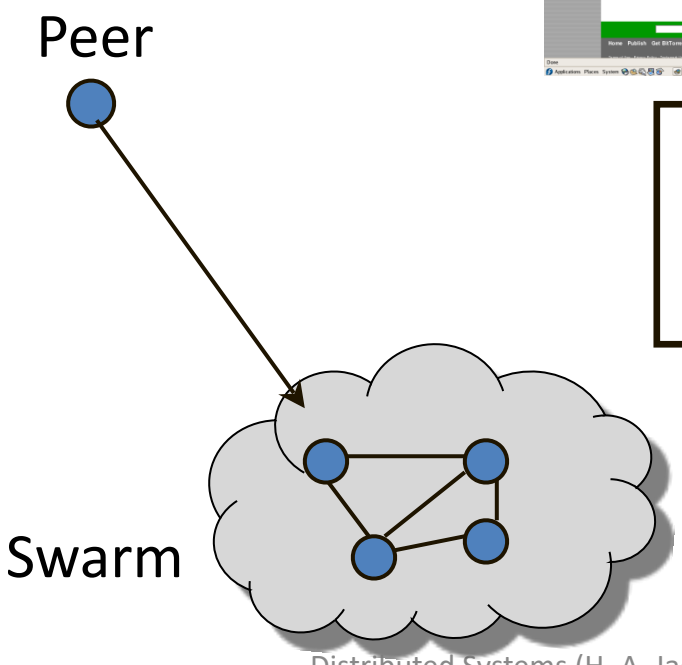


Contact the Tracker of Retrieved Torrent



Connect to Available Peers

www.bittorrent.com



Contents of .torrent File

- URL of tracker
- Piece length – usually 256 KB
- SHA-1 hashes of each piece in file - ***Why?***

Download Phases

- **Beginning**: First piece
- **Middle**: Piece 2 to n-x
- **End-game**: Pieces n-x to n

Middle: Choosing Pieces to Request

- *What is a good strategy?*
  - Most frequent first vs. rarest first?

Middle: Choosing Pieces to Request

- *What is a good strategy?*
  - Most frequent first vs. rarest first?
- **Rarest-first**: Look at all pieces at all neighboring peers and request a piece that's owned by the fewest peers – ***Why?***

Middle: Choosing Pieces to Request

- *What is a good strategy?*
  - Most frequent first vs. rarest first?
- **Rarest-first**: Look at all pieces at all neighboring peers and request a piece that's owned by the fewest peers – ***Why?***
  - **Increases diversity** in pieces downloaded
    - Avoids case where a node and each of its peers have exactly the same pieces
    - Increases system-wide throughput
  - Increases **likelihood all pieces still available** even if original seed leaves before any one node has downloaded entire file

Choosing Pieces to Request: Initial Piece

- First, pick a random piece (**random first policy**)
  - When peer starts to download, request random piece from a peer
  - When first complete piece assembled, **switch to rarest-first**

Choosing Pieces to Request: Initial Piece

- First, pick a random piece (**random first policy**)
  - When peer starts to download, request random piece from a peer
  - When first complete piece assembled, **switch to rarest-first**
  - Get first piece to quickly participate in swarm with upload
  - Rare pieces are only available at few peers (slows download down)
- **Strict priority policy**
  - Always **complete download of piece** (sub-pieces) before starting a new piece
  - Getting a complete piece as quickly as possible

Choosing Pieces to Request: Final Pieces

End-game mode

- Coupon's collector problem
- Send requests for the final piece to **all peers**
- Upon download of entire piece, cancel request for downloading that piece from other peers
- Speeds up completion of download, otherwise last piece could delay completion of download

- ***Why isn't this done all along?***

Why BitTorrent took off

- Working implementation (Bram Cohen) with simple well-defined interfaces for publishing content
- Open specification
- Many competitors got sued & shut down (Napster, KaZaA)
- Simple approach
  - Doesn't do "search" per se
  - Users use well-known, trusted sources to locate content

Pros & Cons of BitTorrent

- 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 & **Cons** of BitTorrent

- Assumes all interested peers active at same time
- Performance deteriorates if swarm “cools off”
- Too much overhead to disseminate small files

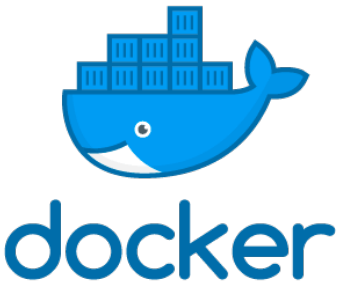
Pros & Cons of BitTorrent

- Dependence on centralized trackers
  - Single point of failure
    - New nodes can't enter swarm if tracker goes down
    - Simple to design, implement, deploy
  - Today, BitTorrent supports trackerless downloads

- Lack of a search feature
  - Users need to resort to out-of-band search: Well known torrent-hosting sites, plain old web-search

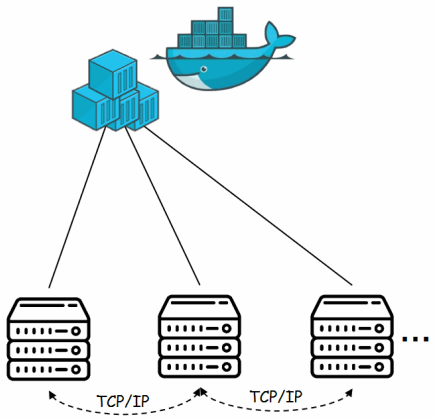
Docker Image Distribution via BitTorrent

- Docker images are containers that encapsulate applications and systems with full run-time stacks
- Image comprised of minimal file system composed of layers
- When launching a system based on containers, its images must be present on every node.
- Layers are downloaded from **central registry**
- Layers can be downloaded **in parallel**



Traditional Approach

- Every node downloads image layers from registry



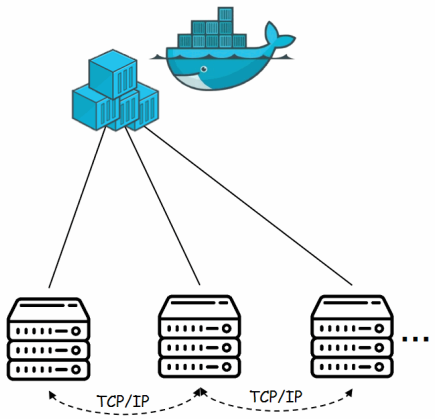
Fact:

- 76% of the time spent on container deployment is spent on downloading layers

- Problem
  - High contention at registry
  - Images are pulled entirely from registry to every node
  - Massive network bandwidth overhead and performance bottleneck at the central registry!

Traditional Approach

- Every node downloads image layers from registry



Fact:

- 76% of the time spent on container deployment is spent on downloading layers

- Problem
  - High contention at registry
  - Images are pulled entirely from registry to every node
  - Massive network bandwidth overhead and performance bottleneck at the central registry!

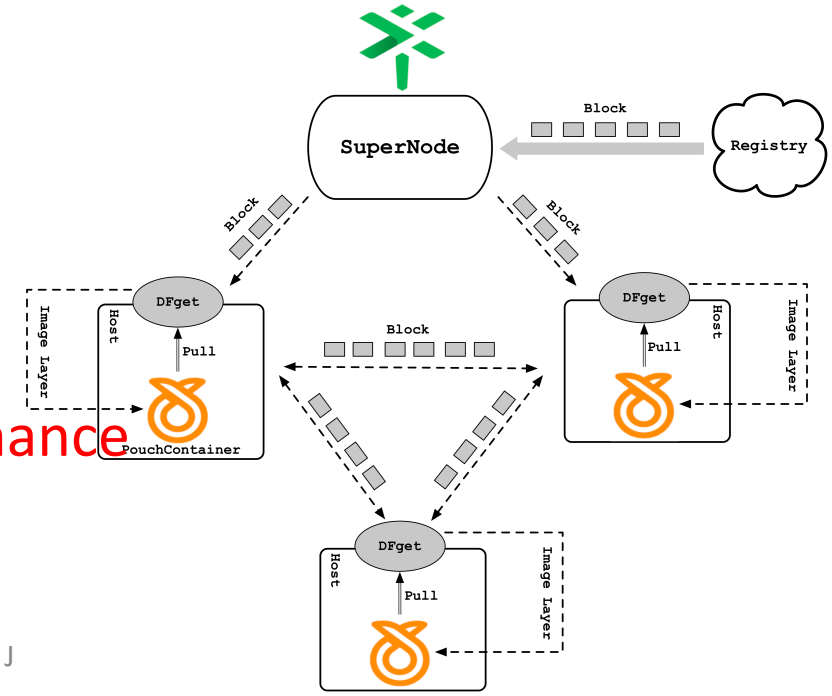
New Approach

P2P via BitTorrent (organization-internal)

- Nodes exchange available layers instead of inundating registry
- **Idea**: Chunk layers into small-sized blocks and distribute via BitTorrent (layer*XYZ*.torrent)
- Widely adopted in industry:
  - E.g., Alibaba Dragenfly, Uber Kraken, etc.

https://github.com/dragonflyoss/Dragonfly

https://github.com/uber/kraken



- Drastically improves performance (almost 100x speedup!)

Spotify vs. BitTorrent

- Live listening of streamed track
- One peer-to-peer overlay for all tracks
- Does not inform peers about downloaded blocks
- Downloads blocks in order
- Does not enforce fairness
- Informs peers about urgency of request
- Supports search

- Batch download of large files
- Essentially, a swarm (overlay) per torrent
- Exchange of downloaded blocks with peers
- Random download order
- Tit-for-tat (game theoretic roots)
- No notion of urgency
- No search feature