Peer to peer systems (P2P)

- Napster Structure
- client machines ("peers")
- servers - store the file name and the info about the file

Client
- connects to a Napster server
- search
  - sends message to the server
    - returns a list of host
  - client pings each host in the list to find the transfer rates
  - client fetches file from best host
- all communication uses TCP (Transmission Control Protocol)

Joining a P2P system

Problems
- centralized server a source of congestion
- centralized server single point of failure
- no security
- indirect infringement

Gnutella
- eliminate the servers
- client machines search and retrieve amongst themselves
- peers store their own files
- peers connected in an overlay graph
- routes messages within the overlay graph (types of payloads)
  - query (search)
  - queryHit (response to query)
  - ping (to probe network for other peers)
  - pong (reply to ping, contains address of another peer)
  - push (used to initiate file transfer)
- how to search for a file
  - descriptor id bytes - 0 to 15
    - id of this search transaction
  - payload descriptor - 15 to 16
    - type of payload
  - TTL (time to live) - 17
    - decremented at each hop, message is dropped when TTL = 0.
  - Hops - 18
    - incremented every time a message is forwarded
  - Payload length - 22
    - number of bytes of message following this header
- Query
  - contains the minimum speed
  - contains the search criteria

- QueryHit: successful result to a query
  - number of hits
  - port
    - information about the sender
  - ip_address
    - information about the sender
  - speed
    - information about the sender
  - (fileindex,filename,fsize)
    - information about the sender
    - results
  - servent_id
    - unique identifier of responder; a function of its IP address
- Avoiding excessive traffic
  - to avoid duplicate transmissions, each peer maintains a list of recently received messages
  - query forwarded to all neighbors except peer from which received
  - each query forwarded only once
  - QueryHit routed back only to peer from which Query received with same DescriptorID
  - for flooded messages, duplicates with same descriptorID and payload descriptor are dropped
  - QueryHit with DescriptorID for which Query not seen is dropped
- after receiving query hit messages
  - requestor choose best query hit responder
    - initiates HTTP request directly to responder's ip+port
  - responder then replies with file packet after this message
- HTTP file transfer protocol
  - because it's standard, well debugged, and widely used
- Why the range field in the GET request
  - to support partial file transfers
- what if responder is behind a firewall?
  - prevents messages from coming in
  - dealing with firewalls
    - requester sends push to responder asking for file transfer
    - push message
      - servent_id
        - same as in received query hit
      - fileindex
        - same as in received query hit
      - ip_address
        - address at which requestor can accept incoming connections
        - same as in received query hit
      - port
        - address at which requestor can accept incoming connections
    - responder establishes a TCP connection at ip_address, port specified
    - requestor then sends GET to responder (as before) and file is transferred as explained earlier
  - Ping-Pong
    - ping
      - no payload

    - pong
      - port, ip_address, num files shared, number of KB shared
    - peers initiate Ping's periodically
    - Ping's flooded out like Query's, Pong's routed along reverse path like QueryHit's
    - Pong replies used to update set of neighboring peers
      - to keep neighbor lists fresh in spite of peers joining, leaving and falling
  - summary
    - peers/ servants maintain "neighbors", this forms an overlay graph
    - queries flooded out, ttl restricted
    - QueryHit (replies) reverse path routed
    - supports file transfer through firewalls
    - periodic ping-pong to continuously refresh neighbor lists
      - list size specified by user at peer: heterogeneity means some peers may have more neighbors
      - Gnutella found to follow power law distribution
        - $P(\#links = L) \sim L^{-k}$ (k is constant)
  - problems
    - ping/pong constituted 50% traffic
      - solution: multiplex, cache and reduce frequency of pings/ pongs
        - multiplex make one pong message out of multiple songs or pings
    - repeated searches with same keywords
      - solution: cache query, query hit messages
    - modem-connected hosts do not have enough bandwidth for passing Gnutella traffic
      - solution: use a central server to act as proxy for such peers
      - another solution:
        - fast track system

  - freeloaders
    - user who only downloads never uploads
  - Flooding causes excessive traffic
    - is there some way of maintaining meta-information about peers that leads to more intelligent routing?
      - structured peer to peer systems

————-

FastTrack and Bit torrent

FastTrack
- hybrid between Gnutella and Napster
- some peers designated as super nodes
- supernode
  - stores a directory listing a subset of nearby (<filename>, <peer pointer>), similar to Napster servers

- any peer can become a super node, provided it has earned enough reputation

BitTorrent
- seed has full file
- leecher has some blocks of the file
- file split into blocks

- download local rarest first block policy: prefer early download of blocks that are least replicated among neighbors
- tit for tat bandwidth usage: Provide blocks to neighbors that provided it the best download rates
- choking: limit number of neighbors to which concurrent uploads <= a number i.e., the "best" neighbors
  - every else chocked
  - periodically re-evaluate this set
  - optimistic unchoke: periodically, unchoke a random neighbor helps keep unchocked set fresh

Chord

Distributed Hash Table
- allows you to insert, lookup, and delete object keys
- a distributed hash table allows you to do the same in a distributed setting
- performance concerns
  - load balancing
  - fault tolerance
  - efficiency of lookups and inserts
  - locality (messages that are transmitted are transmitted relatively close)
- Napster, Gnutella, FastTrack are all DHT's so is chord

Chord
- intelligent choice of neighbors to reduce latency and message
- uses consistent hashing
  - SHA-1(ip_address.port) -> 160 bit string
  - Truncated to m bits
  - called peer id(number between 0 and 2^m - 1)
  - Not unique but id conflicts very unlikely
  - Can then map peers to one of 2^m logical points on a circle

Ring or Peers
- 2^m logical points
- has nodes onto the peer id on the circle
- e.x., node16 -> node32
- m = 7 so 0 to 127
- every node knows it immediate clockwise successor (ring) N112 -> N16

Finger Tables
- m = 7
- 7 finger tables
- ith entry at peer with id n is first peer with id >= n+(2^i)(mod * 2^m)
- warps around using modulus

What about files
- filenames also mapped using same consistent hash function
  - SHA-1(filename) -> 160 bit string (key)
  - File is stored at first peer with id greater than or equal to its key (mod 2 ^m)

- File cnn.com/index.html that maps to key K42 is stored at first peer with id greater than 42
  - Note that we are considering a different file-sharing application here: cooperative web caching

Search
- at node n, send query for key k to largest successor/finger entry <= k if none exist, send query to successor(n)
- takes O(log(N)) time
  - intuition: at each step, distance between query and peer-with-file reduces by a factor of 2
  - intuition: after log(N) forwarding, distance to key is at most 2^m / 2^log(n) = 2^m / N
  - Number of node identifiers in a range of 2^m / N is O(log(N)) with high probability
  - so using successors in that range will be ok, using another O(log(N)) hops

Failures In Chord
search under peer failures
- one solution: maintain r multiple successor entries
- in case of failures, use successor entries
- choosing r = 2log(n) suffices to maintain lookup correctness
- if node fails that contains the file
- solution store the file at one successor and one predecessor

Need to deal with dynamic changes
- peers fails
- new peers join
- peers leave
  - P2P systems have a high rate of churn (node join, leave and failure)
    - lower in managed clusters
    - common feature in all distributed systems, including wide-area, clusters, clouds, etc
- So, all the time, need to
  - need to update successors and fingers, and copy keys

New Peers joining
- server gives ip address of some peer in the system
- routes message to self using hard routing protocol
  - makes it way to successor
- updates to successors and predecessors
- copies peers from successor as the peers in the system
- copies its successor finger table it uses it as it's own finger table
- stabilization protocol runs in the background (followed by all nodes)
  - asks immediate neighbors predecessors and successors for their finger tables
- may need to copy some files/keys from the clockwise node to itself

Stabilization protocol
- concurrent peer joins, leaves, failures might cause loopiness of pointers, and failure of lookups
  - chord peers periodically run a stabilization algorithm that checks and updates pointers and keys
  - ensures non-loopiness of fingers, eventual success of lookups and O(log(n)) lookups
  - each stabilization round at a peer involves a constant number of messages

    - strong stability takes O(N^2) stabilization rounds

Churn
- when nodes are constantly joining, leaving, failing
    - significant effect to consider: traces from the Overnet system show hourly peer turnover rates (churn) could be 25-100% of total number of nodes in system
    - leads to excessive (unnecessary) key copying (remember that keys are replicated)
    - stabilization algorithm may need to consume more bandwidth to keep up
    - main issue is that files are replicated, while it might be sufficient to replicate only meta information about files
    - alternatives
        - introduce a level of indirection
        - replicate metadata more

Virtual Nodes
- Hash can get non-uniform -> Bad load balancing
    - treat each node as multiple virtual nodes behaving independently
    - each joins the system
    - reduces variance and reduces load imbalance

Pastry
- assign ids to nodes
- leaf set - each node knows it successor(s) and predecessor(s)
- routing tables (based on prefix matching) - think of hypercube
- routing is based on prefix matching is therefore log(n)
    - and hops are short

Pastry routing
- id 01110100101
- It maintains a neighbor peer with an id matching each of the following prefixes
- when it needs to route to a peer 01110111001 it starts by forwarding to a neighbor with the largest matching prefix
- for each prefix, among all potential neighbors with a matching prefix, the neighbor with the shortest round trip time is selected
- since shorter prefixes have many more candidates, the neighbors for shorter prefixes are likely to be closer than the neighbors for longer prefixes
- Thus, in the prefix routing, early hops are short and later hops are longer

Summary of Chord and Pastry
- Chord and Pastry protocols
    - more structured than Gnutella
    - Black box lookups algorithms
    - Churn handling can get complex
    - O(log(N)) lookup hops may be high
    - can we reduce the number of hops

Kelips - A 1 Hop lookup DHT

- k affinity groups
  - k ~ sqrt(N)
- each node based to a group
- Nodes neighbors
  - almost all other nodes in its own affinity group
  - one contact node per foreign affinity group

- peer knows about all the other peers in its group
  - also knows about one contact member from another group

Kelips files and metadata

- file can be stored at any node
- decouple file replication/location from file querying
- each filename hashed to a group
  - all nodes in the group replicate pointer information and the file (meta information)
  - affinity group does not store files
- lookup
  - find file affinity group
  - go to your contact for the file affinity group
  - failing that try another of your neighbors to find a contact
- lookup = 1 hop (or a few)
  - memory cost O(sqrtN)
  - 1.93 mb for 100k nodes. 10m files
  - fits in ram of most workstations/laptops

Kelips Soft state

- membership lists
  - gossip based membership
  - within each affinity group
  - and also across affinity groups
  - O(log(n)) dissemination time
- file meta data
  - needs to be periodically refreshed from source code
  - times out (deletes the information about the file)
- range of tradeoffs available
  - memory vs lookup cost vs background bandwidth