10.Peer to Peer

Different design paradigm to build distributed systems.
Used intensively in file sharing applications.
Also useful for edge computing or fog computing.
It takes advantages of resources at the edges of the network. Opposite of client-server architecture, the distinctions between them are blurry, clients can offer services.

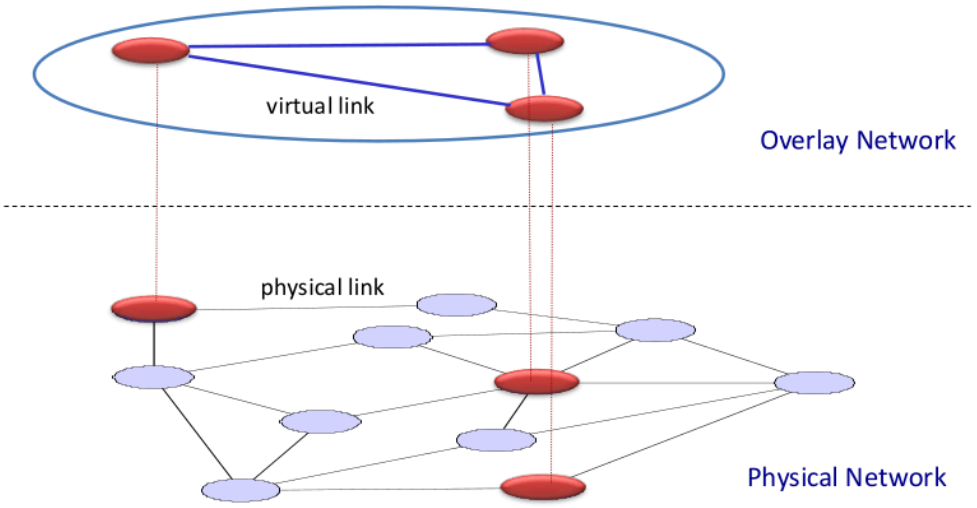
- Client-Server: super simple as you differentiate between who give services and who ask them, easy as you need to deploy a single serve, critical as there is a single point of failure(server) and it is difficult to scale and it can leave some resources unused.
- Peer to Peer: promotes the sharing of resources and services through direct exchange between peers it scales better as when you add more consumers you are adding more producers. Can be applied to different type of computation.

Characteristics of P2P

All nodes are potential users of a service and potential providers of a service

- Nodes act as servers, clients, as well as routers (ad-hoc communication)
  Each node is independent of the other: no central administration is needed.
  Nodes are dynamic: they come and go unpredictably. Capabilities of nodes are highly variable. The scale of the system can be Internet-wide:
- No global view of the system
- Resources are geographically distributed
  There is an heterogeneous of nodes both in type and locality. Consistency can be geographical distributed, can be used to reduce latency instead of centralized systems as you can access to the nearer distributor faster than to hte central storage.

Overlay Network



Peers maps themselves in an overlay network that maps himself on the real network

Retrieve Resources

Retrieving resources is a fundamental issue in P2P systems due to their inherent geographical distribution

- Problem: direct requests towards nodes that can answer them in the most efficient way
  We can distinguish two forms of retrieval operations that can be performed on a data repository

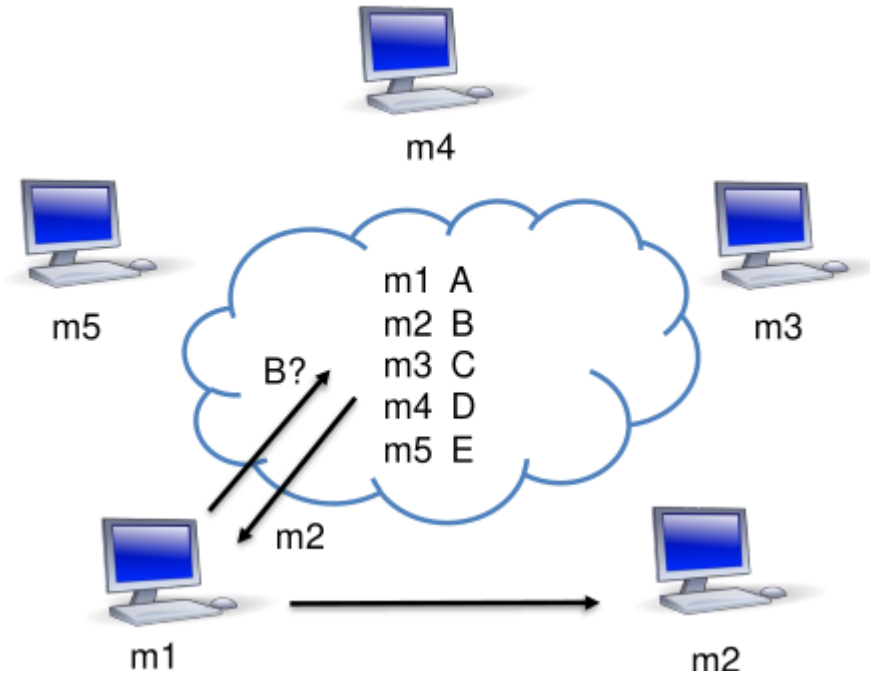
1. Search for something: search something of which we have some characteristics(may be expensive)
   - Locate all documents on “Distributed system”
2. Lookup a specific item
   - Locate a copy of ‘RFC 3268’
   After a look up you can have either the data or a reference to where they are

Centralized Search

Key idea: share the storage and bandwidth of individual (home) users.
Multiple clients connected to a server that offer the service of downloading a file from them. Need a centralized nodes that store what each peer can distribute. To download something you first need to contact the central node to now who has what you are searching.
Operations:

- Join: clients contact a central server

- Publish: submit list of files to central server
- Search: query the server for someone owning the requested file
- Fetch: get the file directly from peer



Many researchers argued that Napster is not a P2P system (or at least not a pure one) since it depends on server availability. Still, Napster allows small computers on edges to contribute:

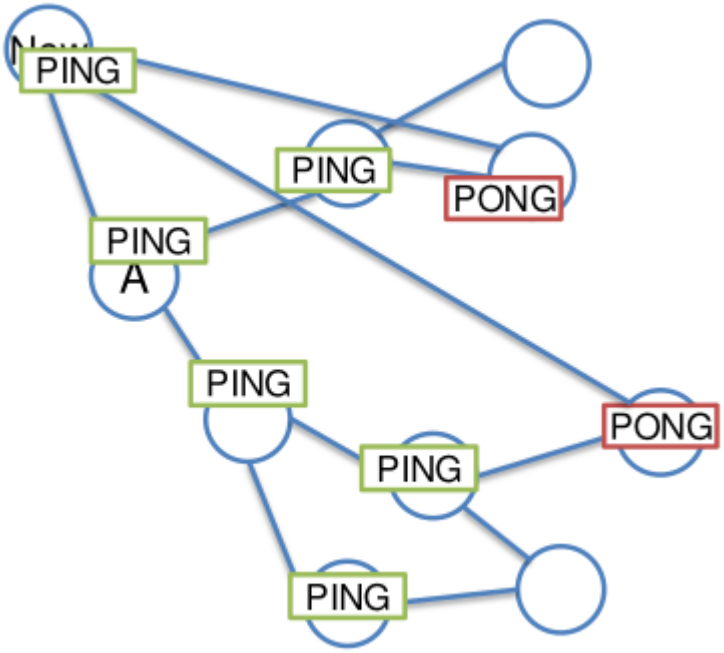
- All peers are active participants as service provider not only as consumer
- Even if they rely on a centralized server for lookup
  PROs:
- Simple
- Search scope is O(1), only connect to the central server
  • CONs:
- Server maintains O(N) State, one for each client
- Server does all search processing
- Single point of failure
- Single point of “control”

Query Flooding

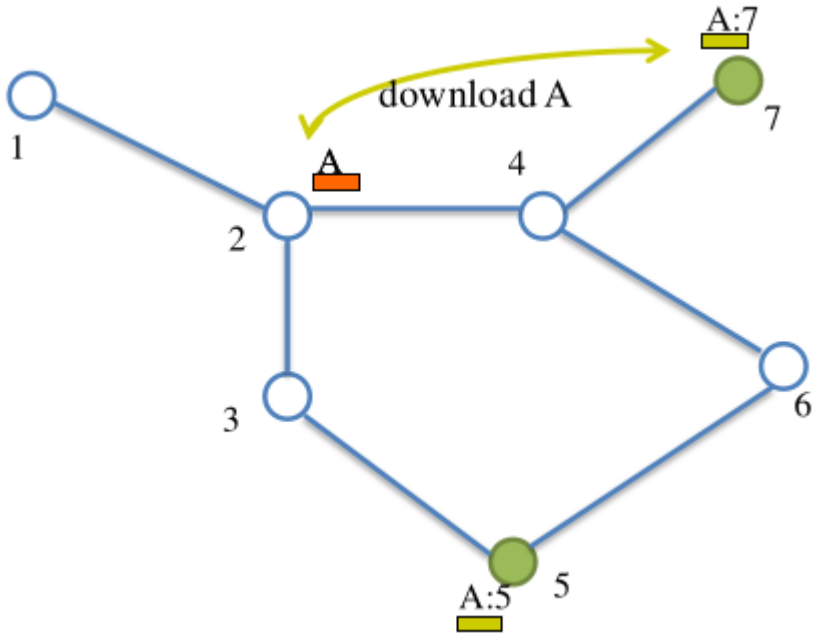
Instead of sending your request to a single node you send them to a lot of nodes. To join the system you need to now the address of at least one node(this nodes are called anchor nodes, not centralized as you can take some of them down and the system work). To join you send a ping that will be send to neighbours and the nodes

will be connected to you in base of a probability that is higher the more the not is not well connected to the overlay network(network self balance, connection are balanced between nodes). Employ the most basic search algorithm: flooding, each query is forwarded to all neighbours. Propagation is limited by a HopsToLive field in the messages.
Operations:

- Join: clients contact a few other nodes
  - They become “neighbours”
- Publish: no need
- Search: ask neighbors, who ask their neighbors, and so on when/if found, reply to sender
- Fetch: get the file directly from peer



The new node connects to a well known “anchor” node. Then sends a PING message to discover other nodes. PONG messages are sent in reply from hosts offering new connections with the new node. Direct connections are then made to the newly discovered nodes.



You search for something you propagate the request to the neighbours until you find what you are searching and stop the research and send the information of the position of the resource to the original sender that can directly download the resource.
PROs:

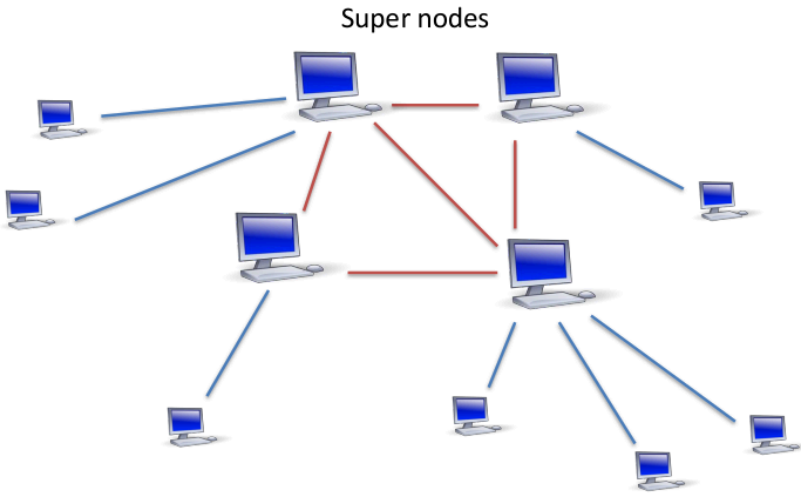
- Fully decentralized (no central coordination required)
- Search cost distributed
- “Search for S” can be done in many ways, e.g., structured database search, simple text matching, “fuzzy” text matching, etc.
  CONs:
- Flood” of Requests. If average number of neighbours is C and average HTL is D, each search can cause C x D request messages
- Search scope is O(N)
- Search time is O(2D)
- Nodes leave often, network unstable
- It is highly intensive on the network. Reduced utilizing HopsToLive that limit the liveness of your message(the higher the probability the lower the HopToLive parameter is) but this not guarantee that you will receive the file as it depends on how you set the parameter

HIERARCHICAL TOPOLOGY

Distinction between normal nodes and “supernodes”:

- Query flooded among supernodes

- Normal nodes contact supernodes to perform search
  Join: clients contact a supernode
- May at some point become supernode themselves
  Publish: send list of files to supernode
  Search: send query to supernode, supernodes flood query amongst themselves
  Fetch: get the file directly from peer(s); can fetch simultaneously from multiple peers



When you have a file you publish it to the nearest supernode to you. If someone ask for a file the first message to the first supernode is a point to point message then start the flooding messages between supernodes until the one with reference on where the file is is founds.
Pros:

- Tries to consider node heterogeneity
  - Bandwidth
  - Host computational resources
  - Host availability
- Kazaa rumored to consider network locality
  Cons:
- Still no real guarantees on search scope or search time

COLLABORATIVE SYSTEMS

Not meant for searching but permits cheaper downloads permitting the download from multiple nodes. It lets clients download files from multiple nodes instead from a single one. A client download multiple chunks from multiple nodes. BitTorrent allows many people to download the same file without slowing down everyone else's download. It does this by having downloaders swap portions of a file with one

another, instead of all downloading from a single server. This way, each new downloader not only uses up bandwidth but also contributes bandwidth back to the swarm. Such contributions are encouraged because every client trying to upload to other clients gets the fastest downloads. Many people are downloading the same files in the same period of time. We split the file in chunks and the nodes distribute the chunks to who request it.
I can dynamically adjust the network bandwidth for each user. Everyone in the network has a portion of the file. I keep monitoring the network. Every 30 second I see which nodes that aren't sharing data with me and drop the connection(avoid free riders and helps to maximise the avoiding of bad connection). It is probabilistic as it need to be started(someone could not have chunks because it just connected to the network). When I upload something I'm uploading something that has an high probability that no one has(tries to not centralize the resource).
I have an higher probability to talk to you if you have something useful for me.
Operations:

- Join: contact centralized “tracker” server, get a list of peers
- Publish: run a tracker server
- Search: out-of-band
  - E.g., use Google to find a tracker for the file you want
- Fetch: download chunks of the file from your peers
  - Upload chunks you have to them

Terminology

Torrent: a meta-data file describing the file(s) to be shared

- Names of the file(s)
- Size(s)
- Checksum of all blocks (file is split in fixed-size blocks)
- Address of the tracker
- Address of peers
  Seed: a peer that has the complete file and still offers it for upload
  Leech: a peer that has incomplete download
  Swarm: all seeders/leeches together make a swarm
  Tracker: a server that keeps track of seeds and peers in the swarm and gathers statistics
- When a new peer enters the network, it queries the tracker to obtain a list of peers

Content Distribution

Breaks the file down into smaller fragments (usually 256KB in size)

- The .torrent holds the SHA1 hash of each fragment to verify data integrity
  Peers contact the tracker to have a list of the peers
  Peers download missing fragments from each other and upload to those who don't have it
  The fragments are not downloaded in sequential order and need to be assembled by the receiving machine
- When a client needs to choose which segment to request first, it usually adopts a "rarest-first" approach, by identifying the fragment held by the fewest of its peers
- This tends to keep the number of sources for each segment as high as possible, spreading load
  Clients start uploading what they already have (small fragments) before the whole download is finished. Once a peer finishes downloading the whole file, it should keep the upload running and become an additional seed in the network. Everyone can eventually get the complete file as long as there is "one distributed copy" of the file in the network, even if there are no seeds.
  Choking is a temporal refusal to upload
- Choking evaluation is performed every 10 seconds
- Each peer un-chokes a fixed number of peers
- The decision on which peers to un/choke is based solely on download rate, which is evaluated on a rolling, 20-second average
  - The more I downloaded from you the higher chances are that I upload to you
  A BitTorrent peer has also a (single) "optimistic un-choke", which is uploaded regardless of the current download rate from it
- The selected peer rotates every 30s
- This allows to discover currently unused connections that are better than the ones being used

Game theory

Employ a "tit-for-tat" sharing strategy

- "I'll share with you if you share with me"
- Be optimistic: occasionally let freeloaders download
  - Otherwise, no one would ever start!

- Also allows you to discover better peers to download from when they reciprocate
  Approximates Pareto efficiency

- If two peers get poor download rates for the uploads they are providing, they can start uploading to each other and get better download rates than before
  Pros:
- Works reasonably well in practice
- Gives peers incentive to share resources; avoids free-riders
  Cons:
- Pareto efficiency is a relatively weak condition
- Central tracker server needed to bootstrap swarm

SECURE STORAGE

We see Freenet as an application:
Freenet (now hyphanet) is a P2P application designed to ensure true freedom of communication over the Internet. It allows anybody to publish and read information with reasonable anonymity
Importance of free flow of information, communication & knowledge

- Democracy assumes a well-informed population
- Censorship restricts freedom
- Anonymity protects freedom of speech
  We want an anonymous web for the publisher and for the consumer as we can still have free speech even if someone tries to silence us.
  The users share bandwidth and storage. The correspondence between ID and content is guaranteed to be unique and we use IDs to route requests, searches and files. Nodes are randomly connected and when you search something you ask your neighbours if they have similar information. Nodes also remember what they have learned from the neighbour and use it to route searches. In this way documents moves from nodes to nodes and you may not have the document you have published anymore but it is on near nodes. When elements aren't relevant anymore they are deleted from the caches. This protocol takes in consideration security as its most priority, performance is taken in consideration secondly(don't want to match the physical network on the overlay network). The idea is that also the participant of the system cannot disrupt the network based on what they contain. They also quantize the packages so you cannot distinguish information also based on message length.

Goals

Allow practical one-to-many publishing of information. Provide reasonable anonymity for producers and consumers of information. Rely on no centralized control or network administration. Be scalable from tens to hundreds of thousands of users. Be robust against node failure or malicious attack.
Operations:
Join: clients contact a few other nodes they know about; get a unique node id
Publish: route file contents toward the node which stores other files whose id is closest to file id
Search: route query for file id using a steepest-ascent hill-climbing search with backtracking
Fetch: when query reaches a node containing file id, it returns the file to the sender

Routing Protocol

Each node in the network stores some information locally. Nodes also have approximate knowledge of what their neighbors store too. Request is forwarded to node's "best guess" neighbor unless it has the information locally. If the information is found within the request's "hops to live", it is passed back through this chain of nodes to the original requestor. The intermediate nodes store the information in their LRU (least recently used) cache as it passes through.

Publishing

Insertion of new data can be handled similarly

- Inserted data is routed in the same way as a request would
  - Search for the id of the data to insert
  - If the id is found, the data is not reinserted (it was already present)
  - Otherwise, the data is sent along the same path as the request
  - This ensures that data is inserted into the network in the same place as requests will look for it
  During searches data is cached along the way
- This guarantees that data migrates towards interested parties
  Each node adds entry to routing table associating the key and the data source (can be random decided)
  The idea is that the things you are sharing is not only bandwidth but also part of your storage: so both files and requests pass through the network. The fact you have some information doesn't give you information on its origin. When a node

reply to a request moves the resources from node to node and augment the probability that the information is stored on other nodes. A high requested information is replicated in a lot of nodes in the network. When I need an information I go to a node that has similar information. To do so I compute the crytograpic hash of the files and search someone that knows a similar hash. The requests/publications follow the route of similar IDs creating geographical regions of data(The fact that are similar IDs doesn't means nothing on the origin).

Routing Properties

“Close” file ids tend to be stored on the same node

- Why? Publications of similar file ids route toward the same place
  Network tend to be a “small world”
- Most nodes have relatively few local connections, but a few nodes have many neighbours
- Well known nodes tend to see more requests and become even better connected
  Consequently, most queries only traverse a small number of hops to find the file

caching properties

Information will tend to migrate towards areas of demand

- Popular information will be more widely cached
- Files prioritized according to popularity
  - Unpopular files deleted when node disk space runs out
  Unrequested information may be lost from the network

anonymity & security

Anonymity

- Messages (content) are forwarded back and forth
- Nodes can’t tell where a message originated
- Security & censorship resistance
  - The ids of two files should not collide
- Otherwise, that could be used to create “denial of service” (censorship)
  - Solution: use robust hashing so that the id of a file is directly related to its

content
Link-level encryption

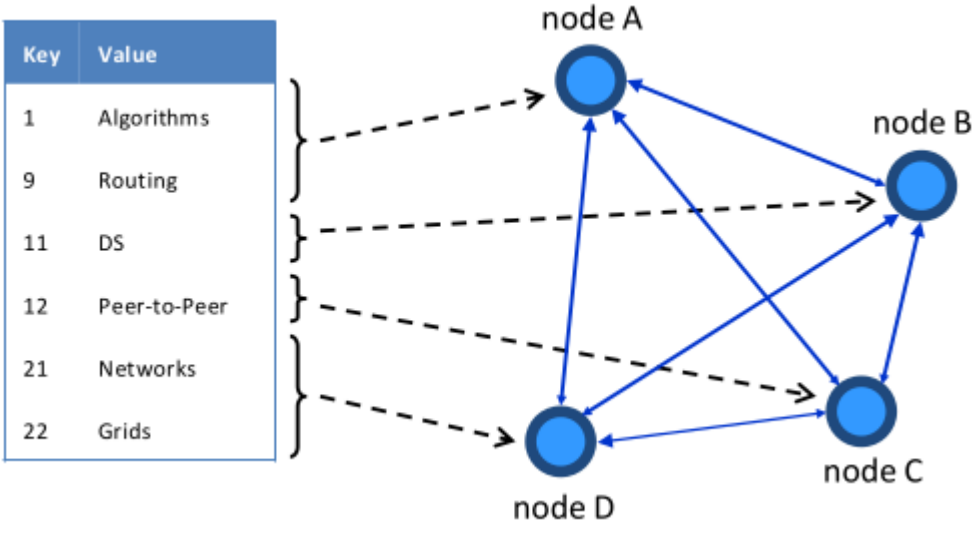
- Prevents snooping of inter-node messages
- Messages are quantized to hinder traffic analysis
- Traffic analysis still possible, but would be a huge task
  Document encryption
- Prevents node operators from knowing what data they are caching
  Document verification
- Allow documents in Freenet to be authenticated
- Facilitate secure date-based publishing
  The sender publishes a new version

Pros:

- Intelligent routing makes queries relatively short
- Search scope small
- Only nodes along search path involved
- No flooding
- Anonymity properties may give you “plausible deniability”
  Cons:
- Still no provable guarantees!
- Anonymity features make it hard to measure, debug

STRUCTURED TOPOLOGY

DHT: Overview



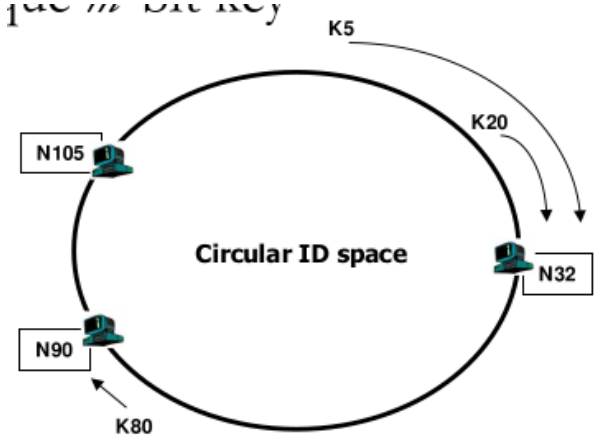
Abstraction: a distributed “hash-table” (DHT) data structure

- put(id, item)
- item = get(id)
  - item can be any resource, such as a reference to a file
  Retrieve a file that is the pointer to the node that contains the content.
  You cannot do search but only lookup. The abstraction is a distributed map, create a structured network.
  Operations:
- Join: clients contact a “bootstrap” node and integrate into the distributed data structure; get a node id
- Publish: route publication for file id toward a close node id along the data structure
- Search: route a query for file id toward a close node id
  - The data structure guarantees that query will meet the publication
- Fetch:
  - Publication contains actual file → fetch from where query stops
  - Publication contains a reference → fetch from the location indicated in the reference

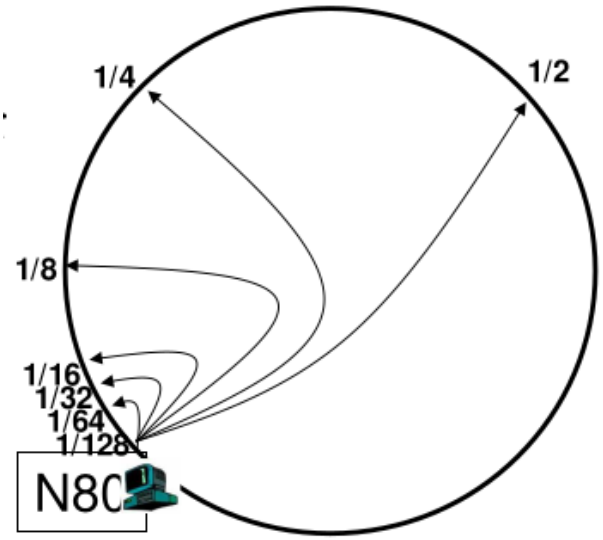
Example: Chord

Nodes are organized in a specific way: nodes have an ID and are connected in a ring with near the node with the next available ID. Key space is much larger than the node space
Nodes and keys are organized in a logical ring:

- Each node is assigned a unique m-bit identifier
  - Usually, the hash of the IP address
- Every item is assigned a unique m-bit key
  - Usually, the hash of the item
- The item with key k is managed (e.g., stored) by the node with the smallest $id \geq k$ (the successor)



We distribute the map key-wise, split based on the IDs.
Each node keeps track of its successor. Search is performed linearly. When you lookup you can trade two things: knowledge of the network by the node and the number of steps a node can do for a lookup. First approach: Searching is super expensive but the routing table to propagate is really simple as you have only one node. Second approach: You have all the nodes in the network in the routing table so the search of the node is expensive but you send a single message. Both are bad so you compromise and so you use a dimension of the routing table logarithmic. Keep the pointer to a node that is 1/2, 1/4, 1/8,... away from you. If we keep a logarithmic number of pointers to nodes the search is less complicated: worst thing is that you don't have a pointer to the part of the network you are searching BUT you go to the nearest node to this partition cutting in half the dimension of the network you are searching in.



Each node maintains a finger table with m entries
Entry i in the finger table of node n is the first node whose id is higher or equal than $n + 2^i$

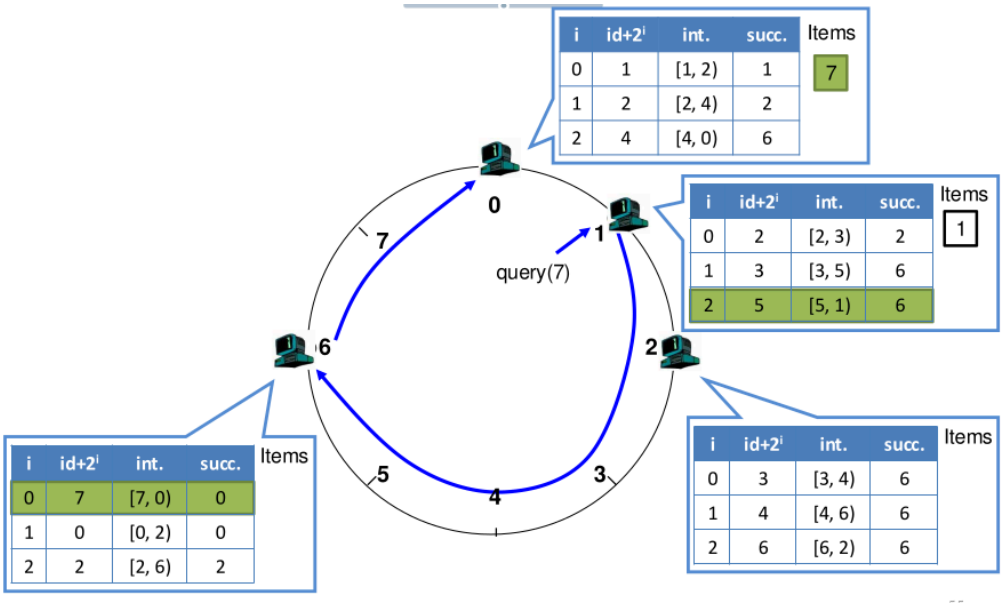
- i = 0 … m-1

- In other words, the ith finger points $1/2^{m-i}$ way around the ring

Routing

Upon receiving a query for an item with key k, a node

- Checks whether it stores the item locally
- If not, forwards the query to the node in its successor table that is responsible for the interval of keys including k



In this example the key-space can contain at max 8 nodes and we have at most 4 connected computer. We can build our routing table(in this case many holes) we point to the next computer in the network.

Joining

Each node also keeps track of its predecessor

- To allow counter-clockwise routing useful to manage join operations
- When a new node n joins, the following actions must be performed

1. Initialize the predecessor and fingers of node n
2. Update the fingers and predecessors of existing nodes to reflect the addition of n

   To initialize its predecessor and fingers, we assume n knows another node n' already into the system

- It uses n' to initialize its fingers

- Finger i = successor of node $n + 2^i$
  To update fingers of other nodes, we observe that node n will become the i-th finger of node p if and only if

1. p precedes n by at least 2^{i-1}
2. The i-th finger of p succeeds n

- The first node p that meets these two conditions is the immediate predecessor of node $n - 2^{i-1}$
- Search for this node and update it
  - We need to do this for each finger i
- log(N) fingers
- For each of them we need to perform a lookup that costs log(N)
- The total cost for joining will be log 2(N)
  The problem is that you need to keep the network structured. Every time a node joins the network you need to update its pointers and the pointers of possibly all the nodes of the networks. When a node joins you know also the immediate predecessor. When you join you immediately learn your position in the network and know all the pointers in your routing table and for each of this pointer you do a logarithmic search for elements for your ID($log(N)^2$) and you need to do the same for each node you have to update to let know that you are there. For the i^{th} line you need to check if backward there is something that is pointing elsewhere but now need to point to you.
  If you have a lot of joining nodes this algorithm only works if there is a single node joining at a time. Problematic but you don't care as you can ask information and so your request don't diverge as other nodes can point you to the node containing the resources and the nodes have a replication mechanism as they save the information on 3-4 nodes in near them.

Stabilization

Periodically we do a search and see if the result is not what we are expecting: if so we rebalance our routing table
The correctness of Chord relies on the correctness of successors pointers

- This may not be preserved in the case of multiple nodes joining and leaving simultaneously
- Chord periodically runs a stabilization procedure to ensure this invariant

    - To stabilize routing information, Chord uses periodic procedures to update successor and finger tables
- Each node n, for each finger i, periodically lookup for the successor of node $n + 2^{i-1}$

failure and replication

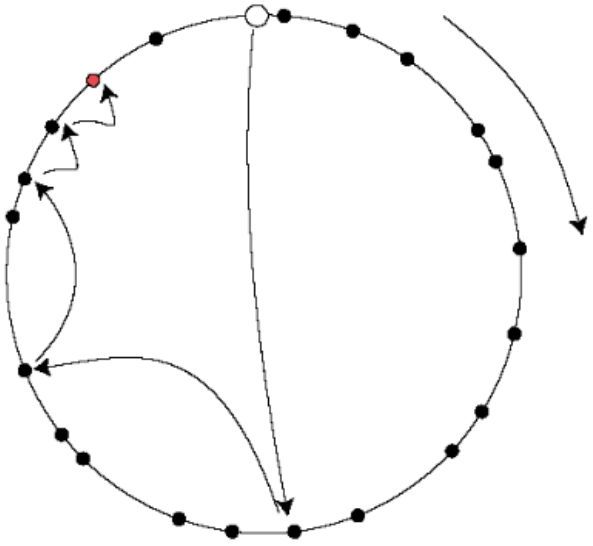
To increase robustness, each Chord node maintains a list of successors of size R

- Contains the node's first R successors
- If the node's immediate successor does not respond, the node contacts the next in the list
- Resilient even if R-1 successors fail simultaneously

summary

Routing table size?

- Log N fingers
- With N= 2m nodes
  Routing time?
- Each hop expects to 1/2 the distance to the desired id => expect O(log N) hops
  Joining time?
- With high probability expect O(log2 N) hops



Pros:

- Guaranteed Lookup

- O(log N) per node state and search scope
  Cons:
- No one uses them? (only one file sharing app)
- It is more fragile than unstructured networks
- Supporting non-exact match search is hard
- It does not consider physical topology
  Chord is just an example, other structures exist
- Different trade-offs between search scope, information to store, cost to maintain the structure when nodes join and leave