UNIVERSITÀ DI PARMA

Popular Peer-to-Peer Overlay Schemes

Tecnologie Internet
a.a. 2022/2023

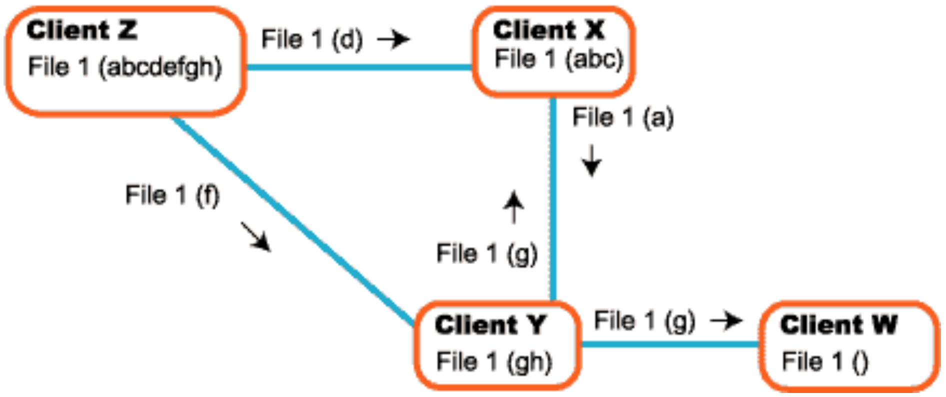
P2P applications

- ❑ Content sharing
- ❑ Distributed storage
- ❑ Parallel and distributed computing
- ❑ VoIP and multimedia streaming
- ❑ Gaming
- ❑ Education and academia
- ❑ Internet of Things
- ❑ Blockchain

Content sharing

In P2P content sharing, users can share their content by contributing their own resources to one another. However, since there is no incentive for contributing contents or resources to others, users may attempt to obtain content without any contribution.

Content download:
- single-source
- multi-source (example: MFTP)



In the following we describe the most important content sharing protocols, with particular attention to those that motivate users to contribute their resources.

Soulseek

Soulseek is a file-sharing network and application based on the HM scheme.

It is used mostly to exchange music, although users are able to share a variety of files.

The central server coordinates searches and hosts chat rooms, but it does not actually plays a part in the transfer of files, which takes place directly between the concerned users.

Being one of the first HM protocols, it has many limitations with respect to others. In particular, it does not allow multi-source download of files.

http://www.slsknet.org

Napster

Also based on the HM scheme, the Napster protocol gained success because of several free implementations, both open source (gnapster, Knapster, etc.) and closed source (the official Napster client), and also several related utilities. The official client went live in september 1999, and by mid-2001 it had over 25 million users.

The District Court ordered Napster to monitor the activities of its network and to block access to infringing material when notified of that material's location. Napster was unable to do this, and so shut down its service in July 2001. Napster finally declared itself bankrupt in 2002 and sold its assets.

In 2003 Napster 2.0 has been released by a division of Roxio Inc. and its servers now offer online music store services, having extensive content agreements with the five major record labels, as well as hundreds of independents.

eDonkey

The eDonkey protocol (also known as eDonkey2000 or eD2k), HM-based, allows to create file sharing networks for the exchange of multimedia content.

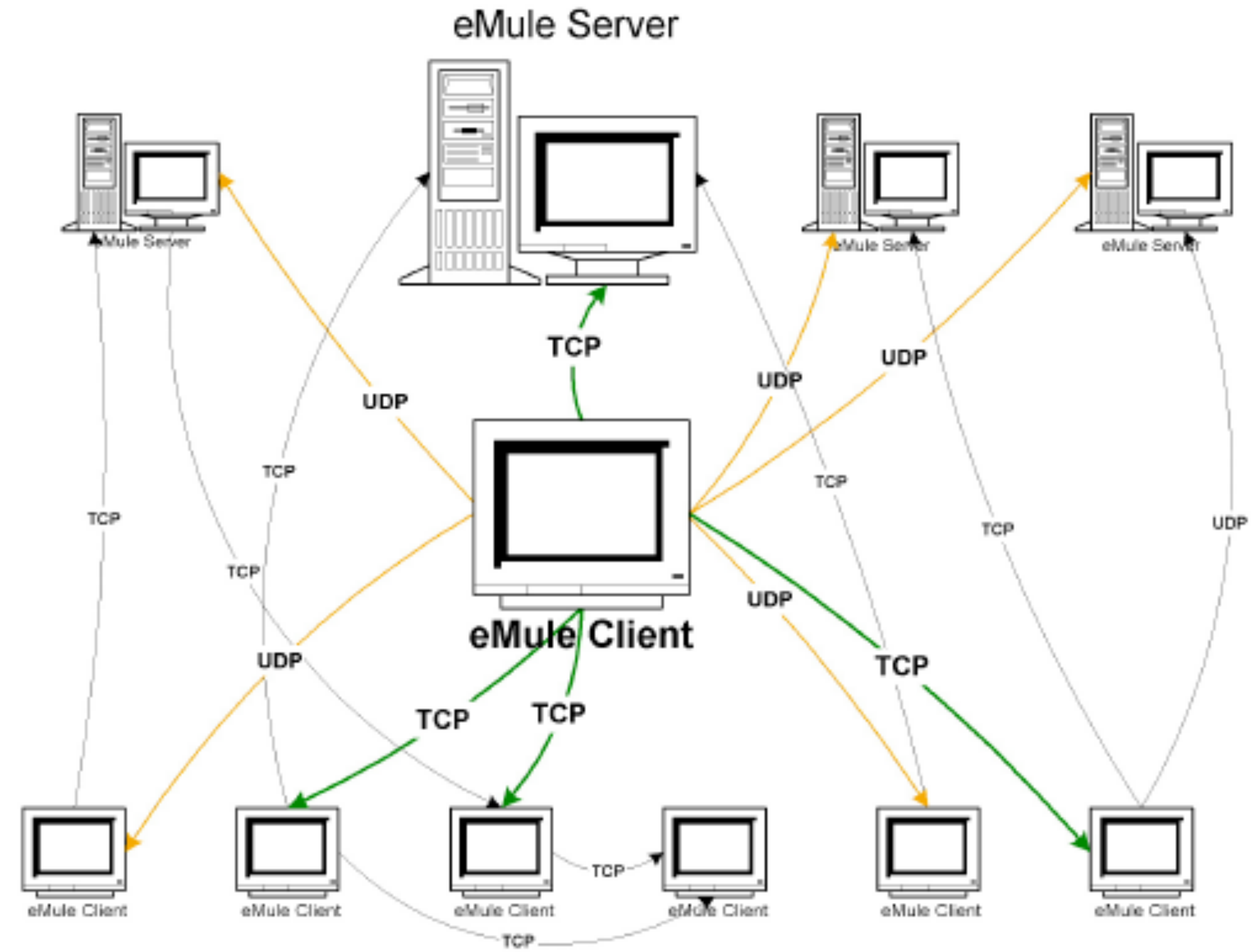
Client programs connect to the network to share files, and publish to servers that can be set up by anyone.

Servers act as communication hubs for the clients and allow users to locate files within the network.

Most used implementation: **eMule**

http://www.emule-project.net

eMule overview



eMule overview

The servers are performing centralized indexing services (like in Napster) and **do not communicate with other servers**.

A client uses a single TCP connection to an eMule server for logging into the network, getting information about desired files and available clients.

The eMule client also uses several hundreds of TCP connections to other clients which are used to upload and download files.

Each eMule client maintains a **download queue** for each of his shared files. Downloading clients join the queue at its bottom and advance gradually until they reach the top of the queue and begin downloading his file.

A client may download the same file from several other eMule clients, getting different fragments from each on.

eMule client-to-server connection

Upon startup the client connects using TCP to a single eMule server. The server provides the client with a client ID which is valid only through the client-server connection's life time.

Them the client send the server its **list of shared files**.

The eMule client also sends his **download list** which contains the files that it wishes to download.

The eMule server sends the client a list of other clients that posses files which the connecting client wishes to download (these clients are called **sources**).

The client/server TCP connection is kept open during all the client's session.

eMule client-to-client connection

An eMule client connects to another eMule client (a source) in order to download a file.

Each client has a **download queue** which holds a list of clients that are waiting to download files. An eMule client may be on the download queue of several other clients, registered to download the same file parts in each one.

When a downloading client reaches the head the download queue, the uploading client initiates a connection in order to send him his needed file parts.

In the first 15 minutes, a downloading client may be **preempted** by a waiting client with a higher **queue ranking**.

There is no attempt to serve more than a few clients in a given moment providing a minimum bandwidth of 2.4 kbytes/sec for each.

eMule client ID

The client ID is an a **4 byte identifier** provided by the server at their connection handshake.

Client IDs are divided to low IDs and high IDs. The eMule server will typically assigns a client with a **low ID** when the client can't accept incoming connections, or when the client is connected through a NAT or proxy servers.

A low ID is always lower than 16777216 (0x1000000).

A **high ID** is given to clients that allow other clients to freely connect to eMule's TCP port on their host machine (the default port number is 4662).

High IDs are calculated in the following way: assuming the host IP is X.Y.Z.W the ID will be $X+2^{8}Y+2^{16}Z+2^{24}W$ ('big endian representation').

eMule user ID

The user ID is a 128 bit (16 byte) GUID created by concatenating random numbers.

While the client ID is valid only through a client's session with a specific server, **the user ID is unique and is used to identify a client across sessions** (the user ID identifies the workstation).

eMule supports an **cryptographic scheme** which is designed to prevent fraud and user impersonation. The implementation is a simple challenge-response exchange which relies on RSA public/private key encryption.

eMule file ID

File IDs are used both to **uniquely identify files in the network** and for file **corruption detection and recovery**.

Note that eMule doesn't rely on the file's name in order to uniquely identify and catalog it, a file is identified by a globally unique ID computed by hashing the file's content.

eMule credit system

When a client uploads files to his peer, the downloading client updates his credit according to the amount of data transferred.

Note that **the credit system is not global** - being a sort of trust, credit is subjective.

$$credit = \min\{2U_{tot}/D_{tot}, sqrt(U_{tot} + 2)\}$$

where $2U_{tot}/D_{tot} = 10$ if $D_{tot} = 0$

BitTorrent

The specificity of BitTorrent is the notion of **swarm**, which defines group of peers sharing a **torrent**, which is a set of replicas of the same file.

A peer that wants to share a content (file or directory), creates a **.torrent static metainfo file** and publishes it to one of the torrent Web servers. Each server is responsible for linking the .torrent file with a suitable **tracker**, which keeps a global registry of all providers of the corresponding file.

Trackers are responsible for helping downloaders find each other, to let them form groups of peers which are interested in the same files. Since trackers do not participate in the swarm, the model is HM.

http://www.bittorrent.org/beps/bep_0003.html
http://www.bittorrent.com

BitTorrent swarms

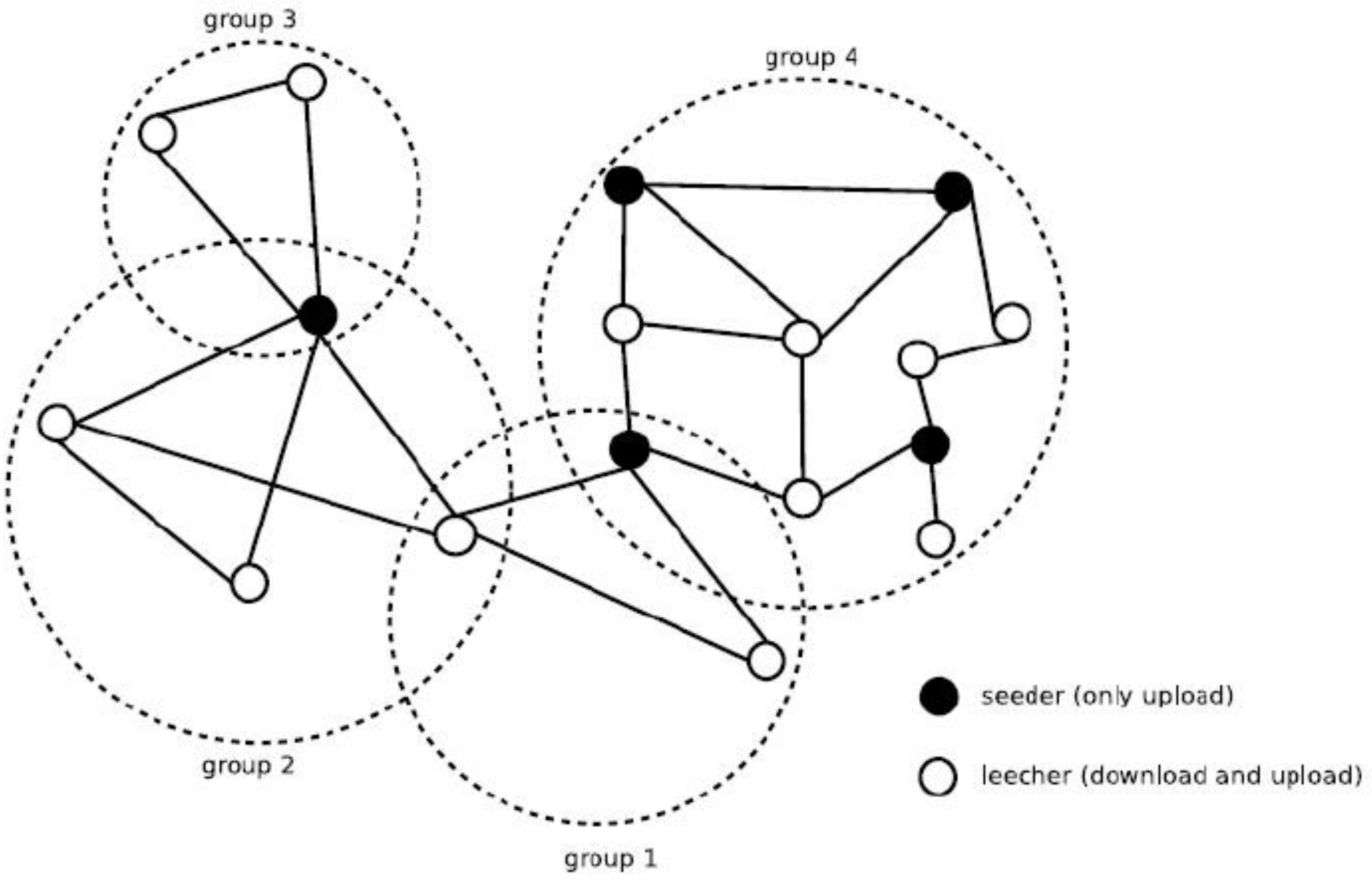
In each swarm, peers which have the complete file are called **seeder**s, while peers which only have parts of the file and are trying to download the other parts are called **leechers**. The initial seeder is the peer that is first source of the content.

When joining a swarm, the peer asks to the tracker a list of IP address of peers to build its **initial peer set**. The initial peer set will be augmented by peers connecting directly to this new peer.

Each peer **reports its state** to the tracker every 30 minutes in steady-state regime, or when disconnecting from the torrent, indicating each time the number of bytes it has uploaded and downloaded since it joined the swarm.

BitTorrent swarms

Each peer maintains a list of other peers it knows about. We call this list **peer set** (also known as neighbor set). A peer can only send data to a subset of its peer set. We call this subset **active peer set**.



BitTorrent pieces and blocks

Files transferred using BitTorrent are split into **pieces of 256KB**, and each piece is split into **blocks of 16KB**.

Partially received pieces cannot be served by a peer, only complete pieces can.

BitTorrent encrypts both the header and the payload, for each transferred piece. Using only 60-80 bits for the cipher, the aim is not to protect the data but instead to simply **obfuscate the stream** enough so that it is not detectable without incurring much of a performance hit.

BitTorrent pieces and blocks

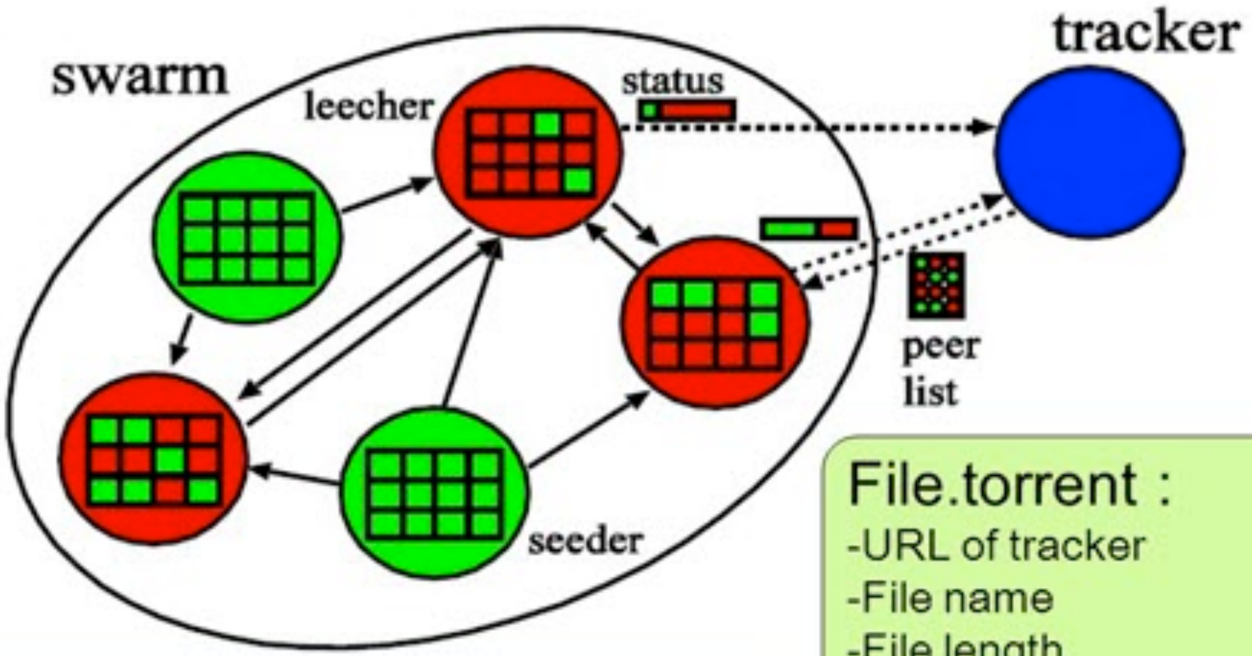
Each downloader reports to all its peers what pieces it has, thus each peer knows the distribution of the pieces for each peer in its peer set. We say that **peer B is interested in peer A** when peer A has pieces that peer B does not have.

We say that **peer A chokes peer B** when peer A decides not to send data to peer B.

Conversely, **peer A unchokes peer B** when peer A decides to send data to peer B.

Blocks can be downloaded from different peers.

BitTorrent pieces and blocks



Seeder – peer having entire file
Leecher – peer downloading file

File.torrent :
-URL of tracker
-File name
-File length
-Chunk length
-Checksum for each chunk (SHA1 hash)

BitTorrent - Rarest First Algorithm (RFA)

RFA is the piece selection strategy used in BitTorrent.

The **rarest pieces** are the pieces that have the least number of copies in the peer set.

Each peer maintains a list of the number of copies of each piece in its peer set. It uses this information to define a **rarest pieces set** - formed by all pieces with *m* copies, being *m* the number of copies of the least replicated piece.

Each peer selects the next piece to download at random in its rarest pieces set. When the piece has been downloaded, the peer sends a HAVE message to the peers in its active peer set.

BitTorrent - Choke Algorithm (CA)

CA is the **peer selection strategy** used in BitTorrent, introduced to guarantee a reasonable level of upload and download reciprocation.

For the sake of clarity, **interested** always means interested in the local peer, and **choked** always means choked by the local peer.

At most 4 peers can be interested in a peer at the same time.

At most 4 leechers can be unchoked by a peer at the same time.

Leechers and seeders adopt different rules for unchoking interested leechers.

BitTorrent - Choke Algorithm (CA)

Leechers:

1. Every 10 seconds, the interested remote peers are ordered according to their download rate to the local peer and the 3 fastest peers are unchoked (we call them **regular unchoked (RU) peers**).

2. Every 30 seconds, one additional interested remote peer is unchoked at random. We call this random unchoke the **optimistic unchoke (OU)**. The OU has two purposes. It allows to evaluate the download capacity of new peers in the peer set, and it allows to bootstrap new peers that do not have any piece to share by giving them their first piece.

BitTorrent - Choke Algorithm (CA)

Seeders:

1. Every 10 seconds, the unchoked and interested remote peers are ordered according to the time they were last unchoked, most recently unchoked peers first.

2. For two consecutive periods of 10 seconds, the 3 first peers are kept unchoked and an additional 4th peer that is choked and interested is selected at random and unchoked.

3. For the third period of 10 seconds, the 4 first peers are kept unchoked.

Gnutella

The Gnutella DUM-based protocol was published in late 1999 by Nullsoft (inventor of WinAmp media player).
Current version: **0.6**
Popular client: **http://gtk-gnutella.sourceforge.net**

A Gnutella node (*servent*) connects itself to the network by establishing a connection with another node, typically one of the several known hosts that are almost always available. In general, the acquisition of another peer's address is not part of the protocol definition.

Gnutella-specific messages:
- Group Membership (**Ping** and **Pong**, for peer discovery)
- Search (**Query** and **QueryHit**, for file discovery)
- File Transfer (**Push**, a mechanism that allows a firewalled servent to contribute file-based data to the network)

Gnutella

Once a servent receives a QueryHit message, it may initiate the direct download of one of the files described by the message's Result Set.

Files are downloaded out-of-network, i.e., a direct connection between the source and target servent is established in order to perform the data transfer. File data is never transferred over the Gnutella network.

The file download protocol is HTTP. It is recommended to use HTTP 1.1 (RFC 2616), but HTTP 1.0 (RFC 1945) can be used instead.

http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html

Gnutella

Every Ping or Query message received by a node is forwarded to all the neighbors of the node (flooding). The specification makes no recommendations as to the frequency at which a peer should send Ping descriptors, although peer developers should make every attempt to minimize Ping traffic on the network.

To avoid network congestion, the Ping and Query messages are always associated to a **Time To Live (TTL)**, which is the max number of times the descriptor can be forwarded before it is removed from the network.

$$TTL(0) = TTL(i) + Hops(i)$$

Pong and QueryHit descriptors may only be sent along the same path that carried the incoming Ping and Query messages.

Gnutella

Moreover, various studies show that the distribution of Gnutella queries is similar to the distribution of HTTP requests in the Internet (they both follow the Zipf's law).

$$q_j \propto 1/j^\tau \quad \text{with } \tau \text{ close to } 1$$

Therefore, the proxy cache mechanism used in the Web context might have useful applications in the Gnutella P2P context.

The difference between Gnutella caching and Web caching is in content location. In traditional Web caching, the content is provided by proxies which are well-defined Web servers. On the contrary, in Gnutella, the content is the sum of the responses to a query and so it is provided by several nodes.

Mute

The Mute network (DUM-based) uses **Utility Counters (UCs)** instead of TTLs.

Search queries are forwarded with a broadcast scheme just like they are in a TTL network, and the UC on a query is modified before forwarding in two different ways:

1. if a node generates results for a given query, the node adds the number of results generated to the UC before forwarding the query
2. the number of neighbors that a node plans to forward the query to is added to the UC

Nodes in a UC-based network each enforce a UC limit: if a search query's UC hits that limit, it is dropped without being forwarded to neighbors.

Freenet

The Freenet DUM-based P2P system was conceptualized by Clarke in 1999, and public development of the open source reference implementation began in early 2000.

In designing Freenet, the authors focused on:

- privacy for information producers, consumers and holders
- resistance to information censorship
- high availability and reliability through decentralization
- efficient, scalable, and adaptive storage routing

The Freenet Project
http://freenetproject.org/

https://ieeexplore.ieee.org/document/978368

Freenet

A location-independent Content-Hash Key (CHK) is assigned to each file to be shared. The CHK is calculated by hashing (with SHA-1) the contents of the file to be stored.

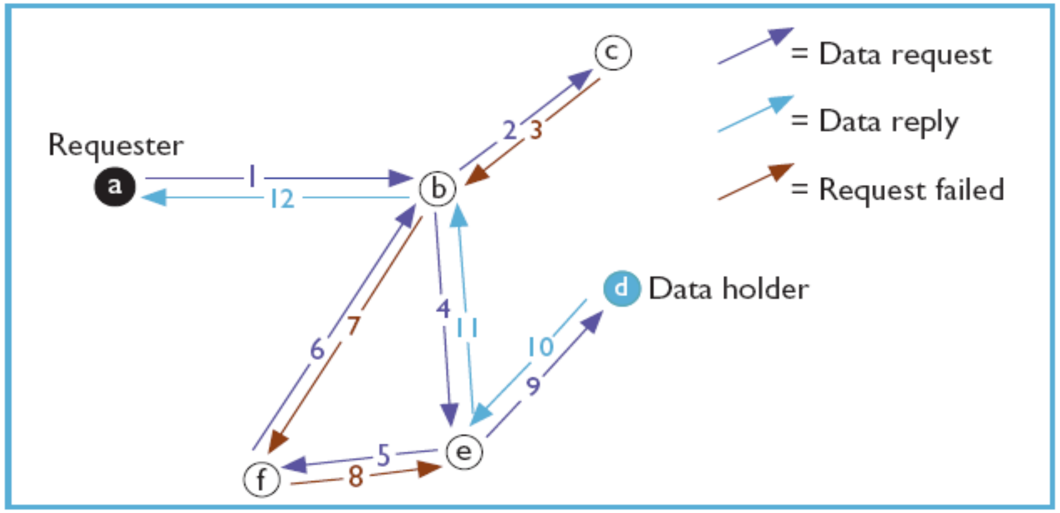
A human-readable short text description is hashed as well, becoming a Signed-Subspace Key (SSK).

Example: basketball/us/nba/

To add a new file, a node sends in the network **an insert message containing the file and its assigned CHK**, which causes the file to be stored on some set of nodes. Files with pointers to the CHK may also be published, using SSKs.

During a file’s lifetime, it might migrate to or be replicated on other nodes.

Freenet



Every node maintains a **routing table** that lists the addresses of other nodes and the CHKs it thinks they hold.
Insert and request messages are forwarded to the node associated to the closest key to the one to be inserted or requested.

To search for a file, SSK requests are sent. These may allow the node to find suitable CHKs. Then, CHK requests are sent.

When a CHK request reaches a node where the file is stored, the file is passed back along the path of the CHK request.

DKS(N,k,f)

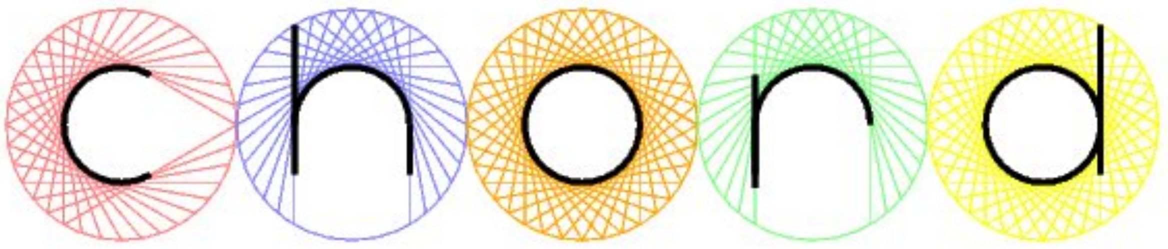
Distributed K-ary Search (DKS) is probably the most known DSM-based protocol, including Chord and Pastry. Each instance of DKS is a fully decentralized overlay network characterized by three parameters:

- ***N*** the maximum number of nodes that can be in the network
- ***k*** the search arity within the network (the key space size is a power of *k*)
- ***f*** the degree of fault tolerance

Once these parameters are instantiated, the resulting network has several desirable properties.
For example, each lookup request is resolved in at most **$\log_k N$ overlay hops** and each node has to maintain only $(k-1)\log_k N + 1$ addresses of other nodes for routing purposes.

https://ieeexplore.ieee.org/document/1199386

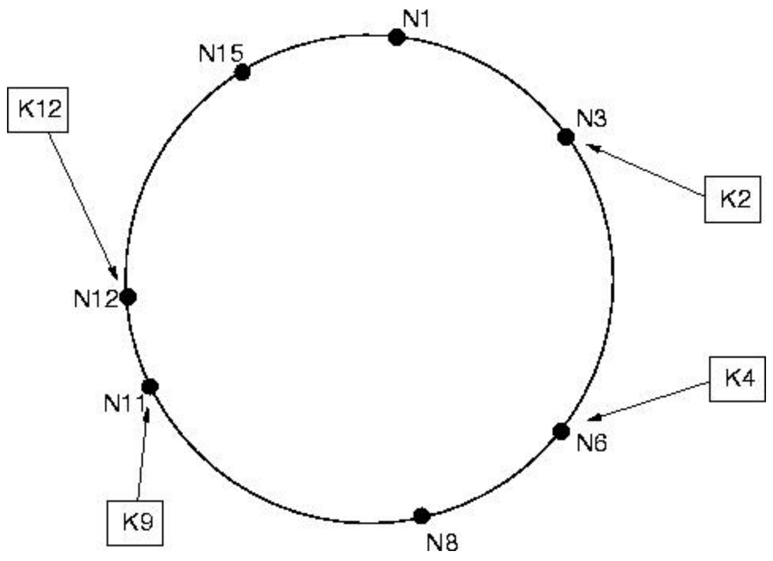
Chord



Chord = DKS(N,2,f)

A hash function (like SHA-1) assigns to each node a *m*-bit identifier. Each resource has an *m*-bit identifier, too.
Thus, the key space size is 2^m.

Node identifiers are ordered in a circle.



Resource key *K* is assigned to the first node clockwise whose identifier is ≥ *K*.

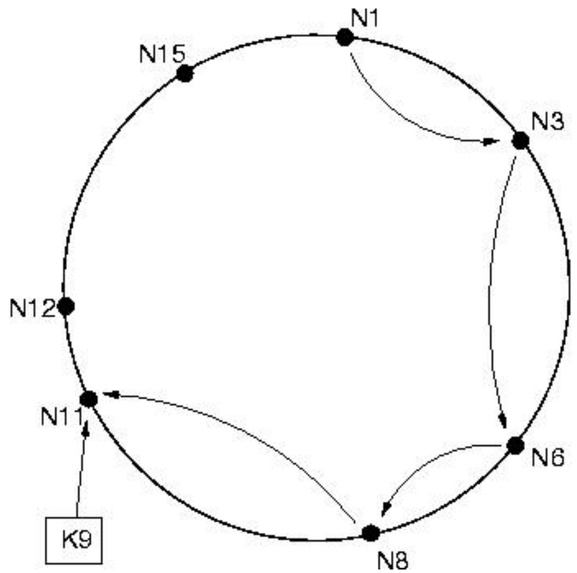
Such node is called *successor* of key *K*.

https://dl.acm.org/doi/10.1145/964723.383071

Chord

Basic search algorithm

Each node knows its successor (= the node with identifier that immediately follows the identifier of the node). A query for a specific key *K* is forwarded clockwise until it reaches the first node whose identifier is ≥ *K*.



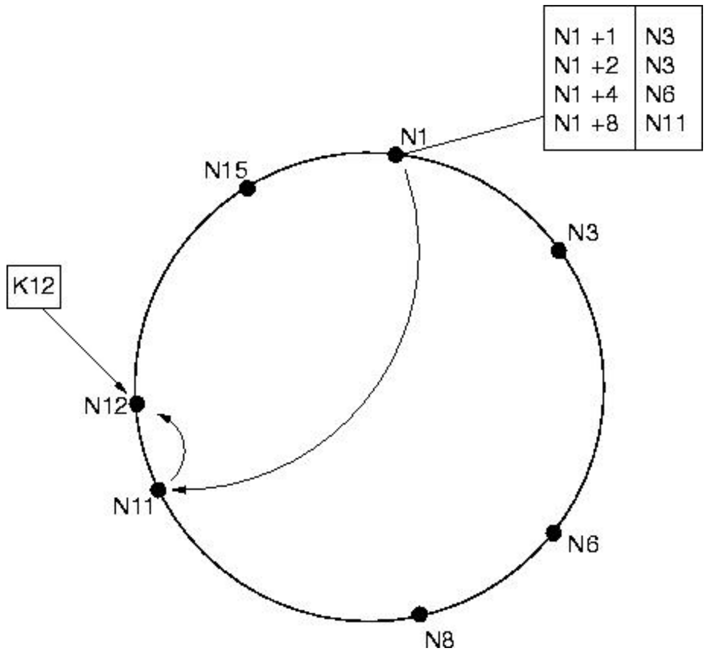
The answer to the query follows the inverse path, until the query originator.

With this approach, the number of nodes to traverse is *O*(*N*).

Chord

Enhanced search algorithm

Each node maintains a routing table with no more than *m* entries, called *finger table*.
The *i*-th entry is: $finger[i] = successor(node\_id + 2^{i-1})$ con $1 \leq i \leq m$



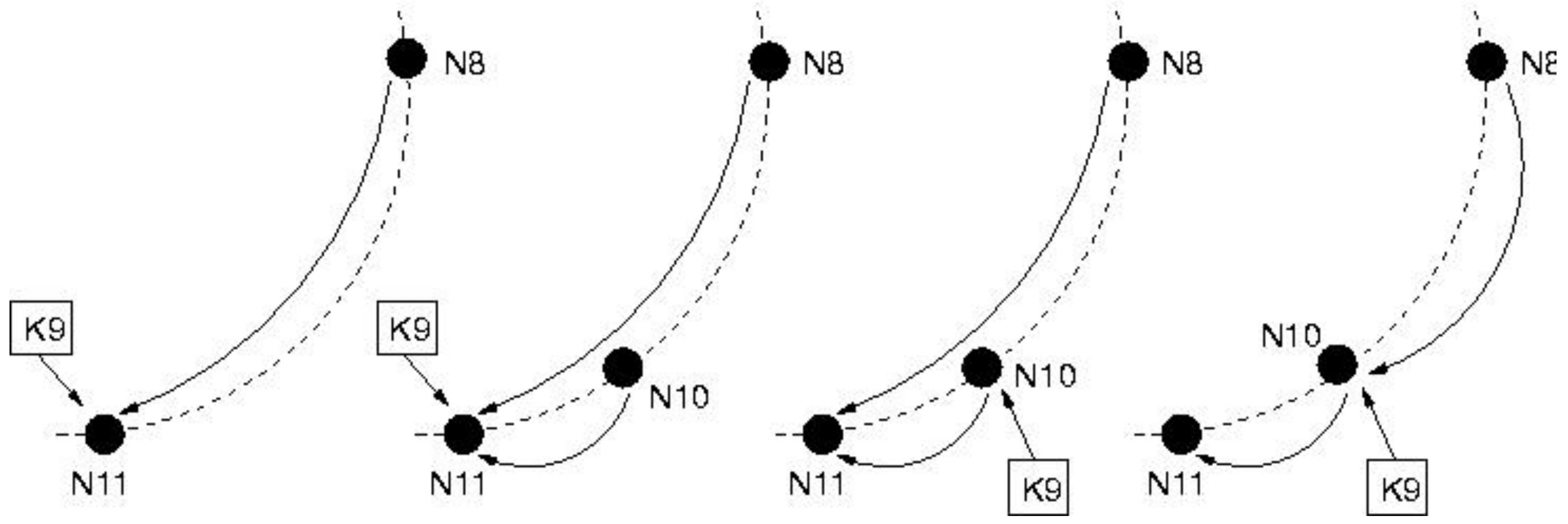
Using the finger table to find the node whose identifier precedes the one of the successor node of the searched key, the number of nodes to traverse is

$$O(\log_2 N)$$

with high probability.

Chord

To guarantee a correct execution of the search algorithm also when new nodes come and old ones go, Chord has a stabilization protocol that each node should periodically execute in background, to update the finger table.



Kademlia

Kademlia is a P2P protocol designed by Petar Maymounkov and David Mazieres at New York University in 2002.

As a **DSM-based protocol**, Kademlia specifies the structure of the network, the communication rules among nodes and the way information has to be exchanged.

Kademlia is based on the **Distributed Hash Table (DHT)** technology, where each published resource is associated to a **<key,value> pair**. The value may be the resource itself or just a resource descriptor. The key is usually a hash of the resource descriptor.

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

Kademlia distance between identifiers

Kademlia uses **160-bit** keys for identifying resources and nodes.

In the publication phase, the <key,value> pair associated to the resource is assigned to the node whose identifier is the nearest one, with respect to the resource key.

The adopted metrics is based on the **XOR** logical operation.

Let X and Y be two identifiers.

$d(X,Y) = X \oplus Y$

<ins>Ex.</ins> the distance between 010101 and 110001 is 100100, namely 36.

Kademlia distance between identifiers

Characteristics of the XOR metrics:

$d(X,X) = 0$

$d(X,Y) > 0$ if $X \neq Y$

$d(X,Y) = d(Y,X)$ for all X,Y

$d(X,Y) + d(Y,Z) \geq d(X,Z)$

The metrics is unidirectional, i.e., for all X and distance $\Delta > 0$ there is a unique Y such that $d(X,Y) = \Delta$

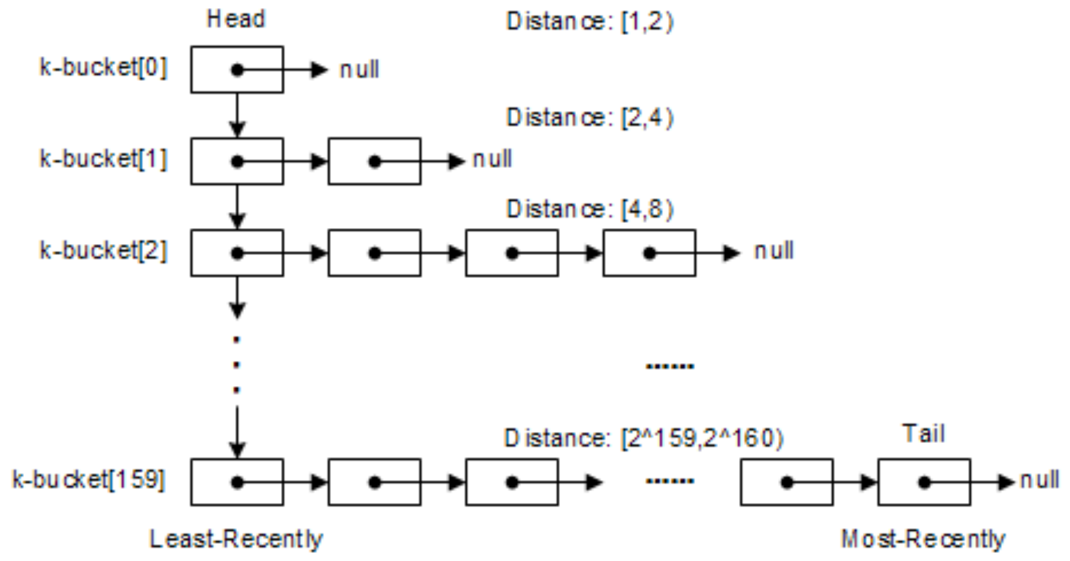
UNIVERSITÀ DI PARMA

Kademlia node data structures

Every node has **160 k-buckets**.

A k-bucket is a list of (max k) <IP address, UDP port, Node ID> tuples related to nodes whose distance from the current node is included between 2^i and 2^{i+1} (with i between 0 and 159).

Every k-bucket is ordered as follows: to the head, the least recently contacted node; to the tail, the most recently contacted node.



Kademlia distance between identifiers

When a node receives a message from another node, it checks the k-bucket that should contain the descriptor of the sender. If such a descriptor is found, the node moves it to the tail of the bucket; otherwise, the node adds it to the tail of the bucket.

If the bucket is full, the node sends a PING message to the node whose descriptor is located to the head of the bucket (least-recently seen node). If no response is obtained, the descriptor of the contacted node is removed from the bucket and the one of the new node is added to the tail of the bucket. Otherwise, this descriptor of the new node is discarded.

Why to favor long-time available nodes with respect to newcomers?

Remember Gnutella and Shifted Pareto node lifetimes, which are characterized by E[R] > E[L] ..

Kademlia RPCs

Kademlia protocol consists of 4 RPCs:

- PING
- STORE
- FIND_NODE
- FIND_VALUE

Kademlia RPCs

PING

Node n_1 invokes PING over node n_2 to check if n_2 is online.

STORE

Node n_1 invokes STORE(<key,value>) over node n_2 to make it store the <key,value> pair passed as an argument.

Kademlia RPCs

FIND_NODE

Node n_1 invokes FIND_NODE(<key>) over node n_2 to get the <IP address, UDP port, Node ID> descriptors of the **k** nodes in n_2's k-buckets that are the closest ones to the requested key. Such descriptors may come from the same k-bucket or from different k-buckets, if the first one being considered is not complete.

FIND_VALUE

Like FIND_NODE, with the following exception: if n_2 has been previously requested for executing a STORE with the same <key>, then n_2.FIND_VALUE(<key>) returns the value associated to that <key>.

Kademlia recursive node search

Node n selects α nodes from the k-bucket that is nearest to a specified identifier X. If such a k-bucket does not contain α nodes, node n gets the α nodes whose ID is nearest to X, among those that n knows.

Then, n invokes FIND_NODE(X) over the α selected nodes, in parallel.

From the obtained results, node n takes the k nodes that are nearest to the one with identifier X. Among these nodes, node n takes α nodes that have not been asked yet and invokes FIND_NODE(X) over them, in parallel.

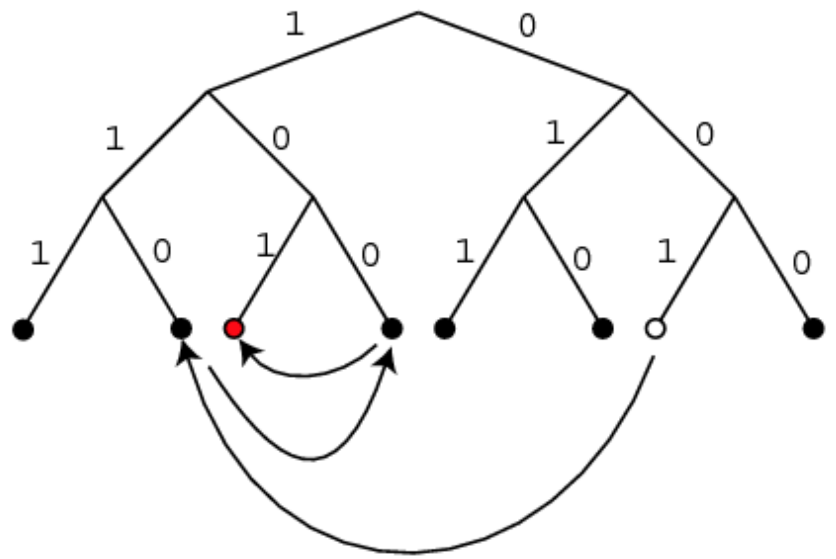
Such a process is repeated recursively. If a round of FIND_NODEs does not return any nearest node with respect to the target, node n invokes FIND_NODE(X) over all the other k-α nodes (from the most recently obtained k-set of nearest nodes). The process stops when node n has (successfully) asked all the k nearest nodes with respect to the target.

Kademlia main principle

Geometric intuition: the distance between nodes of a same subtree is smaller than the distance between nodes of two distinct subtrees.

Kademlia leverages on this principle!

At every new step towards the target, the number of possible candidates gets halved (in theory).



Kademlia resource publication and lookup

In Kademlia, the node lookup illustrated above is the basis for resource publication and lookup mechanisms.

To **publish a resource descriptor** (<key,value> pair, where the value is the descriptor and the key is a hash of the descriptor), node n looks for the k nearest nodes with respect to the key of the resource.
Then, n invokes STORE(<key,value>) over them.

The **lookup for a <key,value> pair** works in the same way: node n looks for the k nearest nodes with respect to the key of the resource.
Then, n invokes FIND_VALUE(<key>) over them.

Both publication and lookup are characterized by running time $O(\log_2 N)$, where N is the number of nodes in the network.

$\log_2 N$ is the height of the binary tree illustrated above!

Kademlia connectivity

Any node n_1 that joins the network, does it through a known node n_2.

Node n_1 insert n_2 in the most suitable k-bucket, then starts a node lookup with its own key as a parameter.

In this way, n_1 populates its own k-buckets and notifies its presence to all the contacted nodes.

In steady state:

- the routing table of each node contains the k nearest nodes
- a k-bucket is empty iff, in the network, there are no nodes with ID that falls within that k-bucket
- the probability that a node is contacted by a node at a distance in between 2^i and 2^{i+1} is constant and independent from the value of i (which falls between 0 and 159)