Description of the YaCy Distributed Web Search Engine

Michael Herrmann*, Kai-Chun Ning†, Claudia Diaz* and Bart Preneel*
*KU Leuven ESAT/COSIC, iMinds, Leuven, Belgium,
Email: firstname.lastname@esat.kuleuven.be
†National Chiao Tung University, Hsinchu, Taiwan
Email: kaichun.ning@gmail.com

***Abstract*—Distributed web search engines have been proposed to mitigate the privacy issues that arise in centralized search systems. These issues include censorship, disclosure of sensitive queries to the search server and to any third parties with whom the search service operator might share data, as well as the lack of transparency of proprietary search algorithms. *YaCy* is a deployed distributed search engine that aims to provide censorship resistance and privacy to its users. Its user base has been steadily increasing and it is currently being used by several hundreds of people every day. Unfortunately, there exists no document that thoroughly describes how YaCy exactly works. We therefore investigated the source code of YaCy and summarize in this document our findings and explanation on YaCy. We confirmed with the YaCy community that our description on YaCy is accurate.**

I. Introduction

Search engine providers build an index of websites and allow users to search through this index with keywords. Therefore, search engine providers maintain huge server farms that analyze the content and the structure of the entire World Wide Web (WWW) [1]. Today, search engines have proven to be key to the functionality of the WWW since most information could not be found without their service.

Maintaining a search engine service is very costly in terms of bandwidth and computation overhead. As a result, most web search engines are maintained by companies with the goal to monetize their web index. This is usually achieved by placing advertisement among the search results.

In the past years significant concerns have emerged with respect to centralized web search engine providers. Privacy concerns are raised because the search engine provider collects all search queries. This information could be used to infer very sensitive information about a user, such as income level, religious beliefs or health conditions [2], [3], and could be used for any kind of discrimination. A user might only be offered a health insurance with higher subscription fees, if she shows signs of health related problems. In the light of recent government surveillance activities[1], users have no means to control what happens with the aggregated information about them. Furthermore, since search engine providers play a key role for society for finding information, concerns have been raised that service providers or users could be intentionally or unintentionally censored by search engine providers. Here unintentionally refers to the *filter bubble* [4] effect: due to personalization, a user's search results match the user's thoughts and beliefs so well that the user is effectively caught in a certain cultural or ideological bubble. And intentionally, refers to a search engine deliberately censoring content for the users, for example the Google censorship in China[2].

Distributed web search [5], [6], [7] aims at making the computation and bandwidth costs of maintaining a search engine manageable by separating the tasks in a peer-to-peer (P2P) network. Hence all tasks that are costly in terms of computation power and bandwidth overhead can be distributed to a large set of P2P nodes. Furthermore, the decentralized P2P structure promises to overcome privacy and censorship concerns, because there is no central instance that is in control of all the data and user requests.

This paper describes the design of the real-world distributed search engine *YaCy*[3]. YaCy is an open source project founded in 2003 with the goal to provide users private and censorship-resistant web search. The largest public YaCy network is commonly referred to with the name *freeworld*. To the best of our knowledge, the freeworld network is the largest distributed search engine with several hundred of active users every day. In order to study YaCy's source code at runtime, we set up our own YaCy distributed search engine in the PlanetLab[4] network. Finally, we verified our understanding with the YaCy developer community.

II. Description of YaCy

YaCy is written in Java and thus runs on most common platforms, such as Windows, Linux and Mac. The most common use case of YaCy is to participate in a distributed network which shares a distributed index with all other YaCy peers. Therefore, every peer maintains local databases that store parts of the index and all local databases together form the entire (distributed) index. Consequently, YaCy will engage in remote connection in order to store or retrieve data from the distributed index or to reply to other peer's requests. However, it is also possible to either run YaCy in a standalone manner without participating in a network with others or to participate in a YaCy network, but to keep the node in *stealth mode* which temporarily disables any remote communication. For the case in which a user wants its YaCy peer to engage in remote

[1] http://www.washingtonpost.com/wp-srv/special/politics/prism-collection-documents/

[2] http://news.bbc.co.uk/2/hi/technology/4645596.stm

[3] http://www.yacy.net/en/index.html

[4] http://planet-lab.eu/

communication, every peer can be in three different modes. The mode of a node is determined by the access the peer has to the network and consequently also the actions a peer can perform. The different node types are:

- Virgin: A peer that is not able to connect to the public network and runs YaCy in a standalone manner. Consequently, this peer is not able to perform remote searches, remotely store content to the distributed index or to receive requests from other peers. However, a virgin peer is able to lookup and store information from/to the local databases.
- Junior: A peer that cannot be accessed via other peers, but is able to initiate connections. This is usually, because the peer is behind a firewall that prevents incoming connections. A Junior peer is able to initiate connections and is thus able to perform searches and store content to the distributed index. Furthermore, a junior peer is able to perform all actions of a virgin node, i.e. accessing the local databases.
- Senior: A senior is able to both, contact other peers in the network and be contacted by other peers, respectively. Consequently, a senior peer is able to perform all actions of a junior node and additionally is able to receive requests from other peer's.

YaCy further defines another role of a peer, the *principal peer*. This peer is a node in senior mode, but also plays a particular role in YaCy's network maintenance, which we will introduce in the following section in detail.

A. Network Maintenance

YaCy defines its own address space and tries to keep every node informed about all the other nodes that are currently online. Further, it defines protocols for joining and leaving the network.

1) Address Space in YaCy: YaCy is a mix of a structured and unstructured P2P network. While it does not implement DHT routing, like for example Chord [8], it structures all peers and data similarly to Chord in a ring structure and is thus not an entirely unstructured network like for example Gnutella [9]. YaCy defines its own address space, the *YaCy hash* which is a 12 character string of an alphabet that contains 64 characters. Consequently, the YaCy address space is $64^{12} = 2^{72}$. In the following we will use the term *YaCy hash* and *hash* interchangeably. YaCy defines a order of characters which we present here it here in ascending order: "*ABCD...Zabcd...z0123...9-_*". These characters are stored in the array *chars* and consequently:

$$\begin{aligned}
\text{chars}[0] &= \text{'A'} \\
\text{chars}[1] &= \text{'B'} \\
&\vdots \\
\text{chars}[26] &= \text{'a'} \\
&\vdots \\
\text{chars}[52] &= \text{'0'} \\
&\vdots \\
\text{chars}[63] &= \text{'_'}
\end{aligned}$$

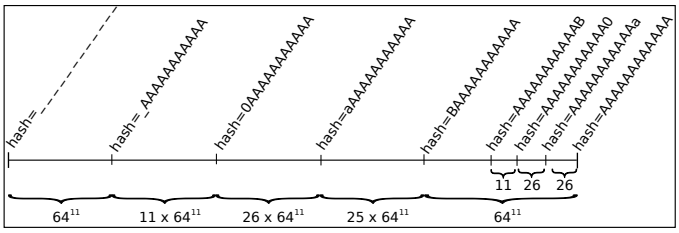
In Figure 1a we illustrate the entire address space from the smallest YaCy hash (12 '*A*' characters) to the largest YaCy hash (12 '_' characters) on a few example hashes.

As we will see later in Section II-B, YaCy requires to convert URLs, arbitrary strings and long integer to YaCy hashes. Therefore, YaCy implements the following functions:

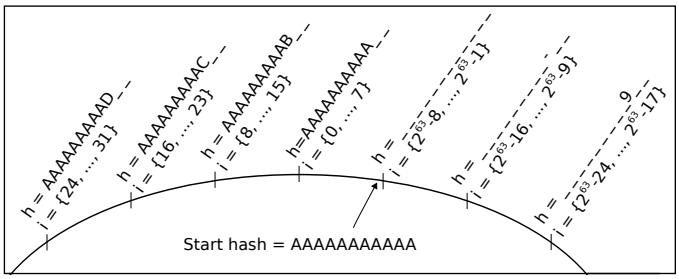
$f_{i\to h}$	Converts a long integer to a YaCy hash.
$f_{s\to h}$	Converts a string to a YaCy hash.
$f_{\text{URL}\to h}$	Converts a URL to a YaCy hash.
$f_{h\to i}$	Converts a hash to a long integer.

YaCy requires a bijective mapping from long integers and YaCy hashes. However, the number of all possible hashes are larger as the numbers in a long integer. In particular, a long integer has 64 bit and thus contains 2^{64} numbers but there are 2^{72} different YaCy hashes. In order to obtain a bijective mapping between long integers and YaCy hashes, YaCy reduces the domain of both. For long integer, YaCy ignores the 3 least significant bits and the most significant bit, resulting in $\frac{2^{64}}{2^4} = 2^{60}$ numbers. For YaCy hashes, YaCy fixes the two least significant characters to the '_' character. This results in $\frac{64^{12}}{64^2} = 64^{10} = 2^{60}$ many hashes, to which we will refer as *anchor points*. As a result, 8 consecutive numbers (three least significant bits are ignored) map to one anchor point (hash that ends with two '_' characters) and vice versa. In Figure 1b we illustrate the first 4 and last 3 anchor points in the YaCy ring together with their corresponding groups of 8 integers. In Figure 1c, we illustrate the 4096 (64^2) YaCy hashes between two consecutive anchor points. In order to convert a long integer to a YaCy hash (i.e. YaCy anchor point), YaCy implements the function $f_{i\to h}$, which we illustrate in Figure 2a. We omit the the rather complex details of the function $f_{h\to i}$ due to space constraints. However, please note that the functions $f_{h\to i}$ and $f_{i\to h}$ are the inverse of each other, i.e. $f_{i\to h}(f_{h\to i}(h)) = h$ and $f_{h\to i}(f_{i\to h}(i)) = i$ for any long integer i and hash h.

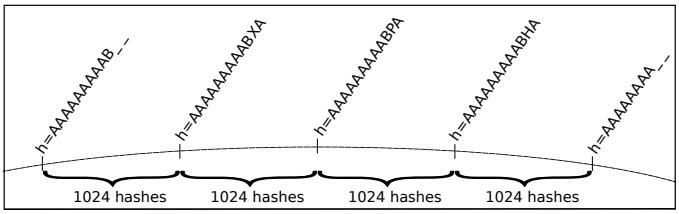
The function $f_{s\to h}$ converts an arbitrary long string s to a YaCy hash h. We illustrate this function in Figure 2b. In the first step the string is hashed using the md5 hash function. As a result, we obtain a 128 bit string of which we take the first 72 bit and group them into 12 blocks with a size of six bits. Similar to the function $f_{i\to h}$, the function $f_{s\to h}$ takes the value of every 6 bit block to determine a character of the YaCy alphabet. Finally, we obtain the corresponding 12 character YaCy hash for the input string s.



(a) The YaCy address space consists of $64^{12} = 2^{72}$ hashes. We first illustrate the number of YaCy hashes for the least significant character. According to the alphabet, there are 26 hashes between the hash "*AA...A*" and the hash "*A...Aa*"; another 26 hashes between "*AA...a*" and "*AA...0*"; and furthermore 11 hashes between "*AA...0*" and "*AA...AB*". Furthermore, we illustrate the number of hashes for the most significant position. There are 64^{11} many hashes from hash for every increment in the most significant position of the hash. For example, there are 64^{11} many hashes between the hash "*A...A*" and the hash "*BA...A*" and 26×64^{11} hashes between the hash "*aA...A*" and the hash "*0A...A*".



(b) Hashes h of the anchor points in the YaCy ring representative for the first 4 and last 3. The first 4 anchor points lie after the start hash *AAAAAAAAAA* in counter clockwise direction. The last 3 anchor lie before the start hash in counter clockwise direction. Each hash represents to a set of 8 numbers i. There are 4096 YaCy hashes between every anchor point.

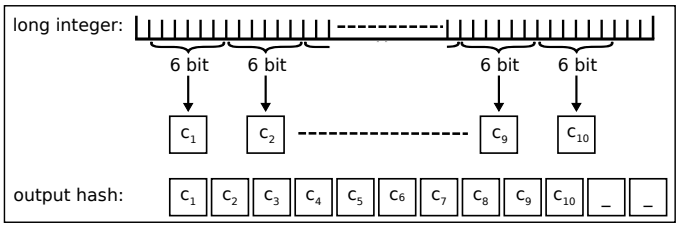


(c) Illustration of the 4096 hashes between two consecutive anchor points "*AA...A__*" and "*AA...AB__*."

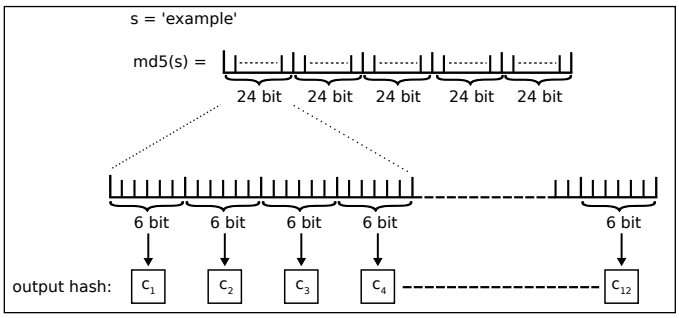
Fig. 1: Overview of the YaCy address space (Figure 1a), anchor points in the network (Figure 1b) and space hashes between them (Figure 1c).

TABLE I: Parts of a URL in YaCy terms.

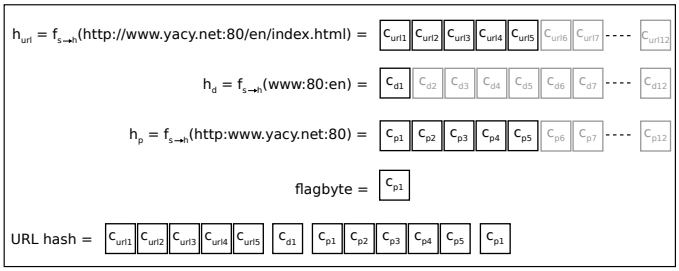
Part	Example
URL	http://www.yacy.net:80/en/index.html
Host	www.yacy.net
Domain	yacy
Sub-domain	www
Port	80
Protocol	http
Root path	en



(a) Conversion of a long integer to a hash. The function $f_{i \to h}$ omits the most significant bit and the 3 least significant bits. The remaining 60 bits are separated into 10 parts, each 6 bits long. Each 6 bits block determines the character by using the six bit number as index in the array *chars*.



(b) Conversion of a string to a hash. The function $f_{i \to h}$ takes the 72 most significant bits of the output of the md5 hash function and groups them into blocks of 6 bits. Finally, every 6 bit block is converted to one character of the YaCy alphabet as in $f_{i \to h}$.



(c) The function $f_{\text{URL} \to h}$ on the example http://www.yacy.net:80/en/index.html.

Fig. 2: Functions $f_{i \to h}$, $f_{s \to h}$ and $f_{\text{URL} \to h}$.

The function $f_{\text{URL} \to h}$ converts a URL into a YaCy hash. Therefore, YaCy splits a URL into separate parts which we summarize in Table I. In order to compute the hash of a URL, YaCy concatenates the following hashes:

- First 5 characters of $f_{s \to h}(\text{url})$
- First 2 characters of $f_{s \to h}(\text{sub-domain:port:rootpath})$
- First 5 characters of $f_{s \to h}(\text{protocol:host:port})$
- YaCy *flagbyte* character

In Figure 2c we illustrate the function $f_{\text{URL} \to h}$ on the example http://www.yacy.net:80/en/index.html. The *flagbyte* character aggregates information about the URL in a 6 bit string. These bits are set depending on the following information: URL's top level domain; HTTP or non-HTTP resource; and length of domain string. The 6 bits are used as index for the *chars* array and thus determine the character of the YaCy alphabet.

TABLE II: Summary of information in a seed.

Seed entry	Description
Peer hash	12 character, unique identifier of the peer
IP	The public IP address of the peer
Mode	Senior, Junior or Virgin as described above
Flag	Root-mode, remote index, remote crawling, direct connect
RWI Count	Approximate number of websites this peer stores
Port	Port YaCy listens on for incoming connections
Search tag	User defined tags for information available on the node
Version	Version of the YaCy software
Last seen	The time stamp a peer has been seen the last time by another peer
Birth date	The time stamp a peer joined the YaCy network the first time

TABLE III: Overview of flags a peer can set.

Flag	Description
Root-mode	Set if last peer ping was finished in less than one second
Remote index	A peer is willing to accept remote storage requests
Remote crawling	A peer accepts to crawl a website for another peer
Direct connect	Whenever a peer had a direct connection with another peer.

2) Peer Connectivity: Every YaCy peer separates all the other peers in two local sets. Firstly, the *active set* contains all the other peers in the YaCy network which are currently online. Secondly, the *passive set* contains all the other peers that are currently offline and were online in the past month. If a peer is offline for longer than one month, the peer is removed from the passive set.

YaCy aims at keeping all online peers aware of each other. This is achieved by two mechanisms. Firstly, via the principal peers. Those peers store their *seedlist* on a *bootstrap machine*. The seedlist contains the seeds of all the remote peers a peer is aware of. Every YaCy peer regularly downloads the seedlist from the bootstrap machines and thus learns the peers the principal is aware of. A seedlist contains the *seeds* of all peers in the network that are currently online and are offline for no longer than a day. We summarize the most important seed information in Table II. We provide a further overview on the seedlist field *flag* in Table III. The second mechanism to keep all online nodes aware of each other is the *peer ping* mechanism. Every senior node sends a peer ping message to the three peers that have the oldest *last seen* value, i.e. last seen tags that lie the most in the past. A junior node pings the 20 youngest peers that have the youngest last seen value, i.e. last seen tags that are the most recent. Both, senior and junior node only send a peer ping message to peers in the local active set. A peer ping is performed every 30 seconds and whenever a peer A pings another peer B, peer B answers with the seeds of the 20 youngest peers with respect to the last seen tag.

3) YaCy Cluster: The YaCy software enables everyone to set up her own distributed search engine, which is independent from the freeworld network. This serves people that for example need a search engine in their local intranet and do not want to use commercial solutions, such as Google Enterprise Search[5]. However, YaCy further offers users to add clusters to the freeworld network in order to enable a more federated approach. For example, if a user maintains one YaCy peer or a network of YaCy peers that share an index on a particular topic, this user might be willing to share this index in the freeworld network as a specialized *cluster* under two assumptions: Firstly, the data in the cluster is persistent, i.e. the data will never be transferred away to other peers not belonging to the cluster. Secondly, no data that has been crawled by another peer outside of the cluster is stored at one of the cluster peers.

A cluster owner is able to guarantee that all her cluster peers only communicate among each other by defining the particular peers that belong to the cluster. This is realized via a shared list of IP addresses among all the nodes that belong to the cluster. However, this approach has the disadvantage that the cluster peers will never engage in any communication with other peers outside of the cluster and thus will also never share their index with the rest of the world. Therefore, the YaCy configuration file contains two settings. Firstly, the cluster node owner can disable a peer to accept remote data transfers from outside of the cluster. As a result the *remote index* flag of this peer will be disabled (see Table III) and then the peer is referred to as *Robinson peer*. Secondly, the cluster node owner is able to configure a cluster node such that it will never transfer any data to other nodes outside of the cluster. Finally, YaCy provides a measure to indicate that a cluster has an index on a particular topic. This is done by the *search tag* of a peer (see Table II).

4) Peer Hash Computation: A YaCy peer's position in the YaCy address space is determined by its *peer hash*. Peer hashes are persistent and thus, a peer only generates a peer the first time it joins the YaCy network. For any subsequent join, a peer reuses its peer hash. For the first join, a peer is in principle free to choose a peer hash, but honest peers are following a particular algorithm. We illustrate the process of a peer hash computation in Figure 3. In order to compute the peer's position, the peer enumerates all free spaces (gaps) between any two consecutive peers in the YaCy ring and sorts them in descending order with respect to the size of the gaps. The peer subsequently tests every gap in descending gap size. For every gap, the peer tosses a coin and chooses to join the gap with probability $\frac{1}{2}$. After the peer chose a gap, the peer divides the entire gap into 8 equally sized parts and randomly picks a position that falls in one of the inner 6 parts. The peer takes over the first two characters of the randomly computed position. The remaining 10 characters of the position are randomly chosen. Finally, the peer stores its freshly generated peer hash on the local hard disk.

5) Peer Join and Leave: The first action of a peer that joined a YaCy network is to obtain an overview about the other peers being online in the network. Therefore, a peer downloads the seedlist from the bootstrap machine(s) and updates its local seedlist. Subsequently, the peer pings the 20 youngest peers with respect to the lastseen tag. The newly joined peer receives the seedlist from all the 20 nodes and is thus able to further update its local seedlist.

A peer that leaves the YaCy network does not send any goodbye messages to other peers, but simply leaves the network. Consequently, there is no difference if a peer leaves the network or when the peer crashes. When a peer is no longer in the network, other peers might still send requests to this peer. When they find this peer to be offline, they move the peer from the local active set to the local passive set.

[5]http://www.google.com/enterprise/search/

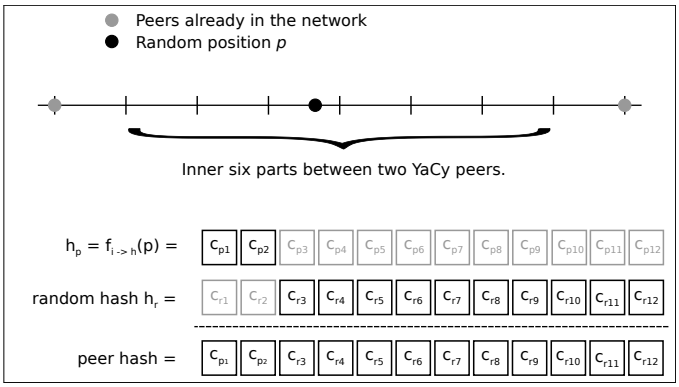


Fig. 3: Computation of a peer hash. A peer chooses a gap and divides it into 8 parts. Subsequently, the peer compute a random hash h_p that falls into the 6 inner parts. Finally, the peer hash consists of the first two characters of h_p and the 10 last characters of a random hash h_r.

B. Maintenance of the Distributed Index

The main purpose of YaCy is to maintain an index for websites. In the following we will explain how YaCy peers gather information about websites (crawling), how these peers distribute the crawled information and finally, how this information can be retrieved by searching peers. YaCy allows users to use a list of *stop words*. These stop words are filtered out prior to any processing of websites and user requests, respectively.

Every YaCy peer has two local databases: A database for reverse word indexes (RWI) [10] and one Solr[6] database. The former is a mechanism implemented by YaCy and the latter is a open source search platform from the Apache Lucene™ project. In the following, we will address the role and the functionality of both database for storage and search. The combination of all RWI and Solr databases from all the peers in the network, build the RWI and Solr distributed index. For both, the RWI distributed index and Solr distributed index, YaCy's stores RWI entries and Solr document at peers whose peer hash is close to the hash of the RWI entry.

1) Crawling: The first step for maintaining an web index is to crawl websites documents and to extract the important information. All YaCy peers are able to crawl website documents and there are three cases in which a YaCy peer initiates a crawl. Firstly, if the user enters a website URL in the YaCy crawler. In this case, the user deliberately intends a certain website to be part of the YaCy index. Secondly, if the user sets YaCy as her local HTTP proxy. In this case YaCy automatically crawls every website the user visits, either by entering a URL in her browser or when the user clicks on a link. YaCy maintains several rules in order to avoid private websites to be crawled and states that "No personal or protected page is indexed; such pages are detected by Cookie-Use or POST-Parameters..." We note that this does not avoid the crawling and indexing of websites with sensitive content. However, a user explicitly set YaCy to be her local HTTP proxy and thus that the user is aware that all websites she visits (either via entering a URL or clicking on a link) are being indexed and stored in the YaCy index. Finally, if a peer has less than $15,000$ websites indexed, the peer falls into *greedy learning* mode. In this mode every time a peer retrieves a website via a YaCy search, all embedded external links are being crawled. Please note that greedy learning can be set off by the user but is enabled by default. For crawling a user can define a *depth* value, which is the number of links the crawler follows starting on the crawled website. Consequently, for any website A, a depth value of 0 means that the crawler only crawls the website A and a depth value of for example 1 means the the crawler crawls the website A and all the websites that appear as a link on A. The default depth value is 3 but might be changed by the user. In greedy learning and in the proxy case we have depth $= 0$.

Once a website has been crawled, the obtained information need to be converted for the RWI and Solr database. The RWI database stores *RWI entries* which are of the form $f_{s\to h}(\text{word}) \to f_{s\to h}(\text{URL})$ and indicates that a particular word occurs at a given URL. During the crawling process of a website, YaCy creates for every word (except stop-words) one RWI entry. For example, if the website www.yacy.net contains three words (free, their, censor), the resulting RWI entries would be:

$$f_{s\to h}(\text{free}) \to f_{\text{URL}\to h}(\text{http://www.yacy.net})$$
$$f_{s\to h}(\text{their}) \to f_{\text{URL}\to h}(\text{http://www.yacy.net})$$
$$f_{s\to h}(\text{censor}) \to f_{\text{URL}\to h}(\text{http://www.yacy.net})$$

After a peer created all RWI entries, it converts the website into a condensed form. This condensed document contains all important information of the website document in a structured way. For example, information such as: text encoding; website title; HTML meta data; text, clickable links, language of the website document; advertisement links. Subsequently, this document is put into the Solr database, which automatically creates the proper *Solr document*[7].

2) Knowledge Distribution: After a peer crawled a website document and created the respective RWI entries and Solr document, the RWI entries and Solr document are transferred with *DHT transfer jobs*[8]. The DHT transfer job transmits the RWI entries and Solr document to the three closest peers at the desired position in the YaCy network. We present the YaCy function which computes this position in Figure 4.

This mechanism ensures that other peers in the network know which peers they have to contact in order to efficiently retrieve information for a given search term. YaCy thus follows approaches like in [8]. However, unlike works as [8], YaCy allows a peer to opt out of the storage of remote RWI entries with the remote index flag (Table III). Only if a peer has set the remote index flag, other peers will send DHT transfer jobs to this particular peer. Please note the if a peer is among the closest peers for a particular RWI entry, but does not accept

[6]http://lucene.apache.org/solr/

[7]Please note that In YaCy version 1.4 there is a similar document for RWI entries, the *YaCy document*. However, in version 1.5 the Solr document is also used for the description of RWI entries and we thus omit the explanation of the outdated YaCy document

[8]Please note that YaCy does not implement any DHT routing and we only use the term *DHT transfer job* in order to adopt YaCy terminology.

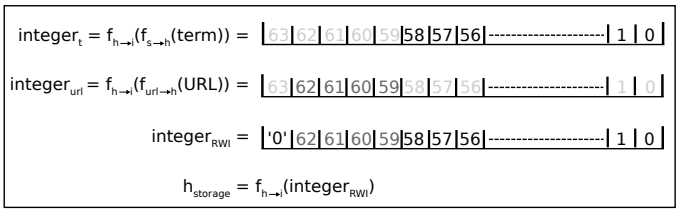


Fig. 4: Computation of the hash to store a RWI entry. Please note that the most significant bit in integer$_{\text{RWI}}$ is 0 by definition.

DHT transfer jobs, this peer is ignored and the next closest peer is chosen.

YaCy peers engages every 15 seconds in a DHT transfer job. However, a DHT transfer job might get postponed if the peer's resources are currently exhausted. The exact steps for a DHT transfer job are not important for this work and we omit this rather complex part of YaCy. However, please note that the DHT transfer jobs ensure that the data eventually arrives at the three peers that allow remote index and whose respective peer hash is the closest to the hash of the RWI entry.

For every DHT transfer job, the sending peer removes the transferred RWI entries but keeps the Solr document. Indeed the sending peer is not the closest peer with respect to the kept Solr document and this is contrary YaCy's approach to store information at the respective closest peers. However, in its current version, YaCy implements this exception and a peer never deletes a Solr document. The receiving peer stores the received RWI entry in its RWI database. Further, it checks if it has the Solr document of the respective URL in its Solr database. If that is not the case, it requests the respective Solr document and stores it in the Solr database as well.

3) Search: When a user performs a search in YaCy, first the local RWI and Solr databases are checked for matching entries. Further, remote search requests are sent in order to retrieve information from the distributed index. YaCy provides two different remote search requests: *YaCy search request* and *Solr search request*. For any request, YaCy creates the respective *candidate set* that include the peers which the request is sent to. For a YaCy search request, the searching peer transmits the hash of the search term $f_{s\to h}(\text{term})$ along with the request. For a Solr search request, the search term is sent as plaintext.

The candidate set of a YaCy search request is different depending on whether the user's search term consists of one or several search terms. First, for every search term in the user's search string, YaCy search initiates a *primary search* to all peers in the candidate set. In the case of a one term search string, for example *Jediism*, the requesting peer retrieves all the available RWI entries in the network that match this particular term. Consequently, YaCy search terminates after this step. However, in the case of a search string that consists of more than one search term, for example *holy places*, primary search is used to learn what peers store RWI entries for all the terms in the user's search string. In particular, for our example, YaCy retrieves all RWI entries for the terms *holy* and *places*. Subsequently, YaCy search engages in *secondary search* to all peers that prove to have RWI entries for both *holy* and *places*. For the YaCy search request, a peer is asked for at most 10 replies and is granted a timeout of 3 seconds in the primary search phase. For the secondary search phase, every peer in the candidate set is granted a timeout of 6 seconds and is asked for at most 20 replies.

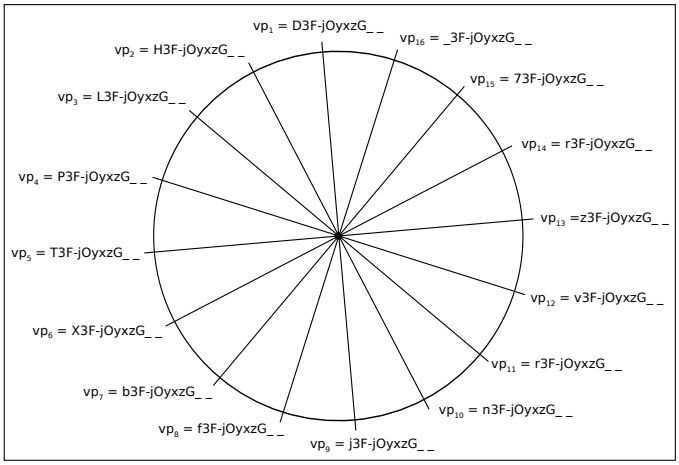


Fig. 5: The 16 vertical partitions for the term *Jediism*.

YaCy partitions the network into a fixed number of partitions and refers to them as *vertical partitions*. The exact number of vertical partitions can be defined, but every node in the network needs to use the same number. YaCy's freeworld network uses 16 vertical partitions and we thus use this example for the rest of this paper. The number of vertical partitions defines the number of positions in the network at which information matching a user's search request can be found. This is a result of the DHT transfer jobs, as we have seen in Section II-B2 (Figure 4): The integer integer$_{\text{RWI}}$ which determines the hash at which a RWI entry is stored ($f_{i\to h}(\text{integer}_{\text{RWI}})$) is computed as follows: the 59 least significant bits are determined by $f_{h\to i}(f_{s\to h}(\text{term}))$ and the bits $60-63$ are determined by $f_{h\to i}(f_{\text{URL}\to h}(\text{URL}))$. As a result, RWI entries for the same term are stored at any of the 16 possibilities given by the first 4 bits of $f_{h\to i}(f_{\text{URL}\to h}(\text{URL}))$ and all the RWI entries for a particular URL are stored at locations which share the first 4 bits of $f_{h\to i}(f_{\text{URL}\to h}(\text{URL}))$. In particular, the positions for a search term are computed as follows:

$$i_t = f_{h\to i}(f_{s\to h}(\text{term}))$$
$$\text{hash}_i = f_{i\to h}(j||\{i_t\}_{4,\ldots,63}) : j \in \{0,1,\ldots,15\}$$

Where j is the binary representation of the integer numbers $0,1,\ldots,15$ and $\{i_t\}_{4,\ldots,63}$ are the bits 4 to 63 of the bit string representation of the number i_t. In Figure 5 we illustrate the 16 vertical partitions for the search term Jediism. Please note that all vertical partitions are a hash with two '_' at the end, because the last step was to use the function $f_{i\to h}$.

YaCy search candidate set (1 search term): The closest 2 peers to every vertical partition of the search term that have the remote index flag set[9]. Further, take the 5 peers that have

[9]Please note that the number of peers to be contacted at the closest position can be changed in a configuration file. However, the value 2 currently hard coded and thus overwrites the configuration file value.

the most RWI entries stored (highly loaded peers) and add each of them with probability 80% to the candidate set

YaCy search candidate set (more than 1 terms): Firstly, for every search term, take the closest 2 peers to every vertical position. Secondly, add around 24% of the most heavy loaded peers. In particular, the peer starts to take the 30% most heavily loaded peers in the network and adds every peer with probability 80% to the candidate set.

For a Solr search request, the respective Solr search candidate set is built and a the searcher sends the search request along with all search terms. Solr is capable of compiling a reply for several search requests and regardless of the length of the user's search term, the reply is sent back. Solr search does only have one phase and in this every peer is asked for at most 10 responses. The timeout is 3 seconds.

Solr search candidate set: The creation of a Solr search candidate set consists of two phases. At first, if the network is large enough, 20 peers are added to the candidate set. Therefore, a peer computes a random hash and, decremental from this hash, checks this peer to be suitable for a Solr search request. A peer is suitable for a Solr search request if: The peer's root-mode flag is set and the peer was not added to the YaCy search candidate set. In case all other peers in the network have been checked and the candidate set size is less than 20, random peers that are not in the YaCy candidate set are added until the Solr candidate set has a size of 20. In the second phase the candidate set is extended in two more steps. Firstly, 50% of all Robinson peers (peers that do not have the remote index flag set) in the network are added with 50% probability to the Solr candidate set. Secondly, all Robinson that have a search tag matching one of the search terms are added to the candidate set. Adding Robinson peers (particular those with a matching search tag field) to the Solr search candidate set tries to exploit the possible existence of specialized clusters as we have introduce in Section II-A3.

After the results for both, YaCy and Solr search, are available at the searchers machine, YaCy aggregates and ranks the results according to their estimated importance. We introduce the ranking mechanism in Section II-C.

C. Ranking of Search Results

After a peer received a set of search results via inspecting the local database and performing a YaCy and a Solr search, YaCy needs to rank the received search results. This is achieved in two steps: Pre-ranking and post-ranking.

In the pre-ranking, YaCy assigns both, YaCy search results and Solr search results an initial *ranking score*. This score depends on several features whose impact has different order of magnitude. We present them in Table IV for both, YaCy and Solr ranking. Please note, that the pre-ranking score also depends on the arrival time of the search results. This is because some ranking features are normalized with the respective value from other search results (marked with a asterisk symbol in Table IV). Since YaCy aims at presenting search results to the user as fast as possible, YaCy does not wait until all the search results have been received, but rather starts to assign ranking scores of intermediate search results. Consequently, for the same search with the same results, the ranking is likely to be different, because the different arrival time of the search results caused the utilization of different normalization values.

TABLE IV: Overview of website features that matter for YaCy and Solr pre-ranking. We enumerate the features in descending impact, i.e. the smaller the number, the higher the impact. Features that are marked with a * are normalized by other search results. Please note that some features do not exist in Solr, but only in YaCy ranking and are hence indicated with the '-' symbol.

	YaCy Ranking	Solr Ranking
Whether the term appears in the HTML title	1	1
Whether keyword exists in URL	2	2
Whether keyword appears as anchor text in a hyperlink	3	-
Shortness of URL*	4	3
Density of non stop words*	5	4
Earliness of the keyword appearing in a sentence*	6	-
Ratio of keywords by all words appearing on the website*	7	7
Number of sentences that contain the keyword*	8	-
Number of sentence on the website*	9	-
Number of inbound links*	10	-
Number of times the keyword appears on the website*	11	-
Total number of words in the website	12	9
Whether the keyword is highlighted	13	8
Number of outbound links*	14	6
Date*	15	5

After the pre-ranking, the peer might verify that the gathered information match the search query. This is configurable by the user. Therefore, the peer checks whether every search term appears in the URL meta data (i.e. URL, URL title, author or subject). If a search term appears in the URL meta data, this search result is considered to be valid and no further tests for this search result are performed. In the case that there is one or several search terms that do not appear in the URL meta data, the peer downloads and inspects the website itself. Please note that this feature can be turned off by the user.

For every pre-ranked and verified search result, YaCy engages in post ranking. In this phase the ranking score of a search result might get increased depending on: i) whether the search term appears in the URL; ii) whether the search term appears in the HTML title tag; iii) *Citation count* which is a measure of similarity of the currently checked website and all the other websites in the peer's local URL database.

ACKNOWLEDGMENT

The authors thank Michael Christen, the initiator of the YaCy project and the developer of the YaCy architecture. His comments and feedback during the process of this work proved to be very helpful and was highly appreciated.

REFERENCES

[1] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[2] R. Jones, R. Kumar, B. Pang, and A. Tomkins, ""I know what you did last Summer": query logs and user privacy," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, ser. CIKM '07. New York, NY, USA: ACM, 2007, pp. 909–914.

[3] B. Tancer, *Click: What millions of people are doing online and why it matters*. Hyperion, 2008.

[4] E. Pariser, *The filter bubble: What the Internet is hiding from you*. Penguin UK, 2011.

[5] S. Iyer, A. Rowstron, and P. Druschel, "Squirrel: a decentralized peer-to-peer web cache," in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, ser. PODC '02. New York, NY, USA: ACM, 2002, pp. 213–222.

[6] S. Orlando, R. Perego, and F. Silvestri, "Design of a parallel and distributed web search engine," *arXiv preprint cs/0407053*, 2004.

[7] P. Melliar-Smith, L. Moser, I. Michel Lombera, and Y.-T. Chuang, "itrust: Trustworthy information publication, search and retrieval," in *Distributed Computing and Networking*, ser. Lecture Notes in Computer Science, L. Bononi, A. Datta, S. Devismes, and A. Misra, Eds. Springer Berlin Heidelberg, 2012, vol. 7129, pp. 351–366.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160.

[9] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," in *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, 2001, pp. 99–100.

[10] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Comput. Surv.*, vol. 38, no. 2, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1132956.1132959