Scientific Research Publishing

Communications and Network, 2026, 18(2), 33-44
https://www.scirp.org/journal/cn
ISSN Online: 1947-3826
ISSN Print: 1949-2421

Google-Like Content Lookup in P2P Systems

Anushi Keswani, Apurva Sonavane, Himanshu Gupta

Computer Science, North Carolina State University, Raleigh, North Carolina, USA
Email: avkeswan@ncsu.edu, asonava@ncsu.edu, hgupta6@ncsu.edu

How to cite this paper: Keswani, A., Sonavane, A. and Gupta, H. (2026) Google-Like Content Lookup in P2P Systems. *Communications and Network*, **18**, 33-44.
https://doi.org/10.4236/cn.2026.182003

Received: September 19, 2025
Accepted: May 17, 2026
Published: May 20, 2026

Abstract

The popularity of a peer-to-peer file system continues to grow every day because of its ability to scale, performance and system failure handling. In a P2P network, the users share their resources by distributing them over multiple nodes throughout the entire system instead of a single server. In this paper, we aim to create an algorithm that can index the documents and store them in the peer-to-peer system. We have mentioned our unique approach of trigraph search instead of whole keyword matching based on frequency, and represented some of our results on a local system.

Keywords

Peer-to-Peer Systems, Distributed Hash Tables, Trigraph Indexing, Semantic Search, Hypercube Overlay Networks

1. Introduction

The motivation for this project is to design a Google-like search engine for P2P systems, in which a search returns a list of names that are close to some degree to the search string.

In recent years, Peer-to-Peer (P2P) systems have proved to be one of the most reliable architectures for implementing distributed file systems. Andoutsellis-Theotokis *et al.* [1] define the P2P system as an application that takes advantage of the resources, such as storage, cycles, and content. Another important element of distributed computing is using a large-scale network of machines, which has gained popularity in recent years due to the increasing acclaim of P2P services like Napster [2], Gnutella [3], Kazaa [4] and Morpheus [5].

An important feature of these systems is that they are based on the design of client-server architecture that provides the base to system features like caching, replication and availability. However, the growth of the internet has triggered the growth of the P2P systems as well. In a P2P system, a user becomes a part of the

system and shares the resources by directly exchanging their information/documents with others. The information is distributed amongst various nodes of the system, hence the system should be highly scalable, available and distributed.

The self-organizing capacity of a P2P network is one of its key features, as the topology can change as nodes enter or leave the system. The nature of the nodes that are part of the P2P architecture contributes to the key aspects such as scalability, self-organization and fault tolerance.

The P2P systems need to select appropriate nodes/peers to operate and must also tolerate churn, defined as rapid node joins and departures [6]. A brute approach for this type of system is to select a node by disseminating the entire list of nodes. A scalable approach is to build a sparse graph structure for all the participating nodes and then traverse through the graph to do the node selection. The Yoid approach [7] is based on a multicast protocol where the algorithm constructs a random graph and performs a random walk to find a suitable node that might match our results. There are numerous recent protocols that use this random walk for node selection, such as Bullet [8], Chainsaw [9] and Chunkyspread [10]. Gnutella unstructured file architecture networks make use of a random selection component [3] while searching for the desired nodes to obtain the files.

2. Background

There are various types of P2P system architecture, such as Unstructured P2P architecture, Structured P2P architecture and Hybrid architecture. The details of these structures and their limitations are discussed in detail in the following sections.

2.1. Unstructured Architecture

These are systems where there is no centralized directory, since each peer acts as a client and a server at the same time. Gnutella [3] is an example of this architecture, where the network is formed by nodes that join and leave the system. Several factors motivate the adoption of decentralized networks such as privacy control, availability, scalability, security, and reliability. On the downside, to find a file in a decentralized network, a node must query its neighbors. In its basic form, this method is called Flooding and is extremely unscalable, generating large loads on the network participants.

2.2. Structured Architecture

Structured Architectures are generally based on Distributed Hash Tables (DHT). DHTs are hash tables stored on all the peers of the P2P system. They can have different structures such as a ring structure or they may use the chord mechanism to have shortcuts across the ring to each other.

The way the chord mechanism works is that the keys for the item to be looked up and the peer ID are hashed similarly and stored with the value as the IP address. Every peer will have neighbors. If the value of the neighbor is just larger than the

item to be looked up, then the system looks for the item in this peer. If the item is not present in the peer, the item does not exist in the system.

Additionally, every peer will keep a track of the peer before it in the ring, the peer after it and the peer two steps ahead of it. If a peer is deleted, the peer that is two steps after this peer will be made the next peer and the peer after this will be the peer that comes two steps after. Additions are made in sequence as per the peer ID key.

2.3. Hybrid Architecture

In a Hybrid architecture, some functionality of the system is centralized while some is not. The Modern P2P systems like Napster and Pointera follow this type of hybrid architecture. In these types, some nodes have special functionality; for example, they might index files held by a set of users. When a user searches for a file at these special nodes, they find a file of interest, then they download it directly from that peer. These systems perform better than pure systems because some tasks (like searching) can be done much more efficiently in a centralized manner.

Bit-Torrent follows a Hybrid P2P architecture where there is a central entity that is responsible for providing parts of the offered services.

Components of BitTorrent file sharing:

1) Tracker: Its main task is to allow peers to communicate with each other. Tracker itself doesn't hold any file copy or information. But when a New Peer comes to it with their desired download file, their IP and Port, then tracker provides them a list of peers downloading the same file.

2) A Torrent file: This is a Meta file with a .torrent extension. It has encoded information about the tracker URL, the hash of file pieces for verification and the name of the file.

3) Seeder: This is a peer with the whole file. It keeps on uploading the file until all others have some parts and until the whole file is available collectively.

4) Leecher: These peers don't have the complete copy of the file. They get the list from tracker of the peers that have their missing pieces. It validates the file by the hashes in the torrent file. Once it gets the full file, it becomes a seeder.

BitTorrent uses a piece selection algorithm to ensure all pieces are available by replicating the pieces quickly. There are 4 policies for these algorithms.

Tit for Tat

Choking

Advantages of this structure:

1) Suitable for very large files like software updates;

2) the more the traffic, the better the file sharing.

Disadvantages of this structure: 1) The IP and information are publicly available to the tracker, which is unsafe; 2) failure of the tracker can lead to failure of the entire system.

Bottleneck Torrents file shared are sometimes greater than 100 GB and thou-

sands of users downloading these huge files simultaneously take up a lot of Bandwidth over the Internet. This is a reason most ISPs throttle traffic over Bit torrent (Wait for draft of a paper to solve this problem).

2.4. Design Points in P2P Systems

2.4.1. Churn Protection

Daniel Stutzbach *et al.* [6] define churn as an inherent property of a P2P system that occurs due to the rapidly joining and leaving of nodes in the system.

2.4.2. Security

Security is the main goal for any system. As parts of the P2P infrastructure are accessible to everyone, a different security philosophy is required to protect against data and routing manipulation (e.g., Sybil attacks). An encryption mechanism, different coding schemes can help secure the system

2.4.3. Privacy

To protect privacy despite the fact that it is stored on unreliable nodes, data needs to be encrypted all the time, possibly using full encryption.

2.4.4. Bandwidth

The bandwidth between nodes is limited, opposite to high-bandwidth connections within a data center. This requires specific algorithms.

$$\mathcal{L} = -\sum_{\tau \in \mathbb{D}^+ \cup \mathbb{D}^-} \left(y_\tau \log\left(s_{\tau 0}\right) + \left(1 - y_\tau\right) \log\left(s_{\tau 1}\right) \right)$$

2.4.5. Heterogeneous Nodes

In contrast to a traditional data center, in P2P networks, the peers are usually heterogeneous in terms of memory, disk space, processor speed, bandwidth, and idle time. Hence, it is necessary to take all the above factors into consideration while designing a P2P system.

2.4.6. Node Volunteer Participation

An important key factor to be considered is the volunteering nature of any given node in the system, *i.e.*, they can join or leave at any given point in time. Therefore, the system should be designed to keep it robust to function even at undesirable removal of nodes at any given moment in time.

3. Related Work

Random Node selection is an important selection process that requires doing random walks over the P2P network topological tree. Gnutella unstructured networks use this random selection process while searching for nodes. [3] Additionally, there have been a few studies, such as GIA [11], which help in improving this file search by using random walks.

I. Stoica *et al.* [12] propose a Distributed Hash Tables (DHT) structured P2P network system where an identifier is assigned to any node that is joining the net-

work. Usually, these identifiers are assigned at random. Further, they are selecting a random value from the DHT space and route to that particular node. However, this method proves to be too exhaustive if we are trying to search for a particular node. Another problem that arises due to the random selection in DHTs is if the identifiers are not assigned appropriately.

Liang Yao *et al.* [13] use Knowledge Graphs (KGs) in conjunction with BERT to implement semantic search within sparse graph structures. A knowledge graph is a multi-relational graph in which entities are represented as nodes and relations as edges, commonly modeled as triplets (head entity, relation, tail entity). Their approach leverages a triplet loss function together with a BERT-based model to identify semantically relevant search results.

The KG-Bert algorithm is able to find one of the attributes given the other two. This can prove to be a key feature while designing a P2P system that works on semantic search to pull up documents related to the search query. However, ff search engines were merely returning results based on keywords, you wouldn't like the search results you were getting. The very best results can only be returned by taking into consideration the semantic factors of a page.

In existing works that we have seen above, an identifier is necessary that yields the search result in a limited time frame. Modern keyword indexing systems often rely on hashing-based vector representations to efficiently map textual features into compact numerical spaces [14]. We hope to come up with a method that will allow us to use keywords for search instead of the whole identifier. We hope to come up with a method that will allow us to use keywords for search instead of the whole identifier. An inverted index methodology has been proposed previously such that the keywords and its objects that contain those keywords have been paired with each other. Finding an intersection of different keywords would yield us the same object which would point us in the right direction. This is called an inverted index. If we query on keywords and perform the right operations that would give us the intersection, we could come up with an object [15] [16].

This method has a lot of redundancy as many keywords point to the same object but if the object goes down, there is a single point of failure. The redundancy serves little purpose. Also, some words occur more often than others and this creates a hot spot within the network. Such a redundancy also makes deleting and inserting difficult.

The author also tries to use the concept of inverse document frequency to figure out the keywords that really matter and not query on filler words that are really common.

The paper tries to represent every object as an r-bit vector. The r-bit vector tries to represent every object as a point in a hypercube. The paper tries to achieve load balance. The index of a keyword is stored by a bunch of nodes. If a keyword is popular, there are more nodes managing its index. Also, since there are so many nodes managing the index of a keyword, a failure in one of them would not stop the system from working. We avoid a single point of failure.

Given a set of keywords, it will be easier to find the object associated with it. Now, this paper got rid of identifier search that was previously used in structured peer-to-peer search. This makes insert and delete really easy, too. So, every object has a set of keywords associated with it. The author mentions that we must also consider other objects that have this subset of keywords in their keyword set. If the keywords associated with an object are a superset of the search keywords, this object should also come up as a result of the search. Hence, a large set of keywords would increase the specificity of the objects searched. Just like other schemes of DHTs [12] [15], every node is given a unique ID and every object is also given a unique ID. The range of numbers for IDs has to be longer than the peers currently in the system as peers can leave the system and when a new peer enters, they need another ID. An ID cannot be repeated in this system. This is done to ensure that the system has a mechanism for finding out if a node has gone down; if it has, another peer will act as a surrogate to this node and act as a backup. This is the way the author suggests avoiding a single point of failure.

An r-dimensional hypercube has 2^r nodes in it. Every node has an identifier that is r bits long. We can generate 2^r different combinations in a binary string. The way it works is that for two nodes u, v in a graph V, the i^{th} bit in u and v is going to be different. This signifies that they are neighbors because of the i^{th} bit.

Most of the methodology for insert, delete, and search is borrowed from this paper [17]. Hybrid and comparative P2P architectures have also been explored in prior work. Yang and Garcia-Molina [18] compare structured and hybrid peer-to-peer systems to evaluate scalability trade-offs. Loo *et al.* [19] propose hybrid search infrastructure designs that improve lookup efficiency under dynamic conditions. Klumpp [20] provides a broader overview of file-sharing architectures and their network implications. These works further motivate the need for scalable, fault-tolerant decentralized search mechanisms.

4. Methodology

The most common issue in all the structures above is that when the file name given by a client is not exact, it becomes challenging to match it in the P2P system nodes. The unique approach selected by us is explained below to tackle this issue.

4.1. Create Tri Graphs

A *trigraph* is defined as a contiguous sequence of three characters extracted from a string using a sliding window of length three. Given a string $S = c_1 c_2 \cdots c_n$, the trigraph set is $\{(c_i, c_{i+1}, c_{i+2}) \mid 1 \le i \le n-2\}$ after removing whitespace characters.

In this work, punctuation and special characters are retained, while spaces are ignored. Punctuation often carries semantic or syntactic information (e.g., hyphenated terms, version identifiers), whereas whitespace does not contribute to lexical similarity and may introduce artificial trigraph boundaries.

Trigraphs provide a balance between semantic discrimination and robustness to noise. Unigrams lack sufficient contextual information, while higher-order n-

grams increase dimensionality and sparsity. Trigraphs capture local structure while remaining resilient to typographical variations, making them suitable for approximate search in decentralized systems.

We briefly discussed how Google uses the Page Rank Method and knowledge graphs for semantic searching and mapping of the input keyword to the list of documents stored. We also read how BERT in Knowledge Graph is implemented using three tails (h, r, t) to find out the relations.

Based on this, we decided to go for Tri-Graph word frequency. The algorithm will process all the input documents stored in the P2P system at any moment. We will start scanning the document 3 characters at a time. We will consider various punctuation and special characters, but will ignore any spaces. For maintaining the consistency of the trigraphs, we will consider lowercase characters only, hence allowing us to prevent different keywords because of the case in characters. Now each document is mapped offline to the number of trigraphs it contains from our exhaustive list or if new trigraphs are found, they will be added to the system. However, one limitation of this approach is that if we keep adding all trigraphs, the list can become too exhaustive and we can overflow our memory. Therefore, to prevent this, we also consider the frequency of the trigraphs to reduce the size of trigraphs list. We will try to avoid some very common trigraphs like "the", "and", which will be present in almost all documents.

Now, in our dictionary, we store the key as the trigraphs and the value as the list of documents containing those trigraphs. Then we further convert this to a vector, which allows us to perform search queries faster. Words can be converted into vectors using the Hashing Vectorizer [14]. We prefer to use Hashing Vectorizer because it does not require any vocabulary or grammar, which perfectly aligns with the peer-to-peer concept we are trying to achieve. These trigraphs will be converted into vectors so that we can map them to the various identifier nodes in our hypercube.

The motivation to vectorize these Tri-graphs generated is that it is easy to find a bit difference in n-dimensional vectors. This will facilitate us to put the nodes that have similar trigraphs close to each other. Additionally, it makes joining and assigning peers to our hypercube system a lot easier. Once we have generated the trigraph vectors for a new document, we can check in our hypercube to find the most similar vector where we can place this particular document. This will also help in expanding our search functionality, as we can also provide suggestions by querying the adjacent nodes. This will also enable us to show any suggestions for some trigraphs if the exact trigraph is not found in our system.

Our approach now focuses on the trigraphs search and guides the client to the list of documents containing those trigraphs.

The trigraph generation process implemented in our system is shown in **Figure 1**. The algorithm iterates through each document character sequence, normalizes the input to lowercase, removes whitespace, and extracts overlapping trigraphs for indexing.

Figure 2 illustrates the frequency distribution of selected trigraphs across the document corpus, highlighting the dominance of common trigraphs such as “the” and “and”, which are intentionally filtered to reduce index size.

4.2. Code

```
class TriGraph:
    def trigraph_create(self,f):
        import matplotlib.pyplot as plt
        import io
        curr = dict()
        f = io.open(f, mode="r", encoding="utf-8")

        text_file = f
        s = text_file.read()

        for i in range(0,len(s),3):
            place_holder = ""
            for j in range(i,i+3,1):
                if j < len(s):
                    place_holder+=s[j]
            if place_holder in curr:
                curr[place_holder]+=1
            else:
                curr[place_holder] = 1
        return curr
```

Figure 1. Code snippet for generating trigraphs.

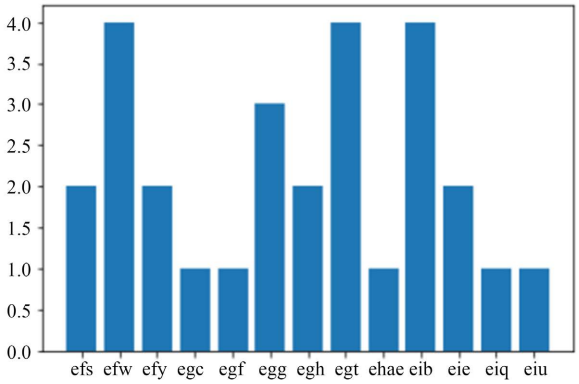

Figure 2. Sample trigraphs with their frequency.

4.3. Trigraph Vector Mapping and Node Assignment

Algorithm 1: Mapping a Document to a Hypercube Node

Input: Document D, hypercube dimension r
Output: Selected hypercube node ID and stored document reference
$T \leftarrow$ Extract trigraph set from D
$V \leftarrow$ Compute frequency vector over the filtered trigraph space
Normalize V
$id \leftarrow$ Hash V using a locality-preserving hash function to produce an r-bit ID
if *collision at id* **then**
  Probe adjacent hypercube nodes until an empty slot is found
Store the document reference at the selected node

4.4. Nodes in Hypercube

We have evaluated different measures on how we can store these trigraphs in our system. The hypercube approach seems to solve the issue of storing our trigraphs in the system. We consider an *r*-bit hypercube, which will have 2^r nodes in it. Each node in our hypercube will be identified by a vector identifier. The difference between two adjacent nodes in the hypercube should not be more than 1 bit. This will allow us to store nodes with similar trigraphs in the adjacent node locations in the hypercube. There might be some popular trigraphs whose vectors can be stored on more than one node in our hypercube.

Whenever a new peer joins our system, they are going to be assigned a unique ID. This unique ID will be constructed using the most common trigraph vector present on that system. Hence, each node in our hypercube will be identified by an *r*-bit vector which is based on the trigraphs on the particular node. Each node will maintain a Hashing List of the trigraphs and the location of the documents where these trigraphs are present.

When the user comes up and searches for any word, we will create trigraphs for the search query. We will obtain the vector of these trigraphs. If the search query has *n* trigraphs, then we will create *n* different vectors V_s^n where each vector V_i depicts i^{th} trigraph vector. We will try to match this trigraph vector with the node vector of our hypercube. For each trigraph from our search query T_s, we can get a list of node N_1^s where these trigraphs are present. The combination of these node vectors and trigraph vector will help us identify which peers have those documents containing similar words. We will then obtain an intersection of these Node vectors to get the IDs where all these trigraphs are present. If there is any ID, where all the trigraphs are present, we will return the IDs where the maximum trigraphs are present to return the closest list of peers' documents.

Consider we have a hypercube of 10 dimensions, then we can $2^{10} = 1024$ nodes in our hypercube. The user search query S_Q consists of three trigraphs T_1 T_2 T_3. Once we query these trigraphs, we get the results as follows:

$$T_1 = \left[N_1T_4, N_3T_7, N_{12}T_{43}\right]$$

$$T_2 = \left[N_3T_7, N_{13}T_{24}, N_{17}T_4\right]$$

$$T_3 = \left[N_1T_4, N_3T_7, N_{34}T_2\right]$$

If we do an intersection of the results obtained, we can see that the entire search query is present at N_3T_7 -Node 3 and the trigraph vector 7. Now, since each node maintains a hash table for its Trigraph vector, we can query Node 3, Trigraph vector 7 to get the peer locations of all the documents that have the search query S_Q.

4.5. Insert and Delete Operations

Every object is associated with one node and other nodes would have references to this node associated with it. If one gets to access the reference of the object, they can reach the object. This is ensured by making sure that between any node *u*, *v*,

there is always a path that goes from u to v in the graph. This collection of references makes it easy to navigate the network smoothly. We are using the DOLR scheme (Distributed Object Location and Routing) to locate, insert, read, and delete objects.

1) **Insert:** If one wishes to insert a copy of an object into a node x, the original location of the object and the node that holds the original object (object, node *u*) act as a reference to the object. This reference now has to be stored in node *x*. The reference is first looked up using L (object). Once the reference is found, we traverse from u to x and store (object, node u) as a reference to the object. This is how inserting a reference works. The way an insert for an original object works, in the node *u* is that if we have the function L (object) that returns us the value of None, then we automatically understand that this object is the first of its kind and store a reference to it.

2) **Delete:** The way a delete operation works is that it follows the same procedure as L(object) to look up a reference and it deletes that object. If references to this object exist elsewhere, if there are copies, the delete operation does not go through.

4.6. Complexity Analysis

Let N be the number of peers and r the hypercube dimension.

Insertion: Trigraph extraction takes $O(|S|)$ time for a document of length $|S|$. Mapping and routing require $O(r)$ hops.

Search: Query routing requires $O(r)$ messages, compared to $O(\log N)$ in traditional DHTs.

Deletion: Deletion follows the same routing path as insertion, requiring $O(r)$ hops.

Compared to inverted-index approaches, the proposed method avoids centralized index maintenance and reduces update overhead under churn.

4.7. Mapping Keywords to Objects

Every keyword is mapped to a mathematical space W. that assigns it an integer. After hashing this integer that is assigned to the keyword, we should be able to fetch the ID of the node u that holds the object that the keywords are associated with. This mechanism assigns a hypercube for a set of keywords associated with K. The function that keeps a track of the keywords hashing to a node keeps an index handy that holds the value (keyword set, object). Once the node that is responsible for the object is found, looking for the object that we require from this node should be easier. The node is obtained by the hash function of the keyword. On giving some keywords, the user will get some objects. When the user wants to fine-tune their search, they can give more keywords to achieve better results.

4.8. Overcoming Hurdles

The hypercube overlay provides scalable routing as the network grows, requiring

only local neighbor state. Node joins and failures affect only adjacent peers, avoiding global reconfiguration. Redundant trigraph references stored on neighboring nodes ensure continued query resolution under churn.

One of the hurdles we have to consider is whether a system goes down. In the hypercube approach, each system is assigned a node where it can have numerous trigraphs. If a node goes down, we will delete all the hashing tables on that particular node and can remove that particular node from our hypercube. Since each node operates independently, the other nodes keep on working, hence preventing system failure. If some documents go missing on the particular system, we can recompute its node vector based on the existing trigraphs and can reassign that system to a new node. This will keep our system working dynamically by updating the trigraphs whenever a new document is added to the peer-to-peer system or some document is removed from the system.

5. Conclusion

This paper presented a decentralized, Google-like content lookup scheme for peer-to-peer systems based on trigraph frequency indexing and hypercube-based overlay routing. By combining approximate string matching with structured peer organization, the proposed approach supports fuzzy keyword search while remaining robust to churn and avoiding single points of failure. Although large-scale evaluation is left for future work, the design demonstrates a scalable and fault-tolerant alternative to traditional DHT-based search mechanisms.

Acknowledgements

We would like to thank Dr. Khaled Harfoush for his continued guidance on this project throughout the semester. This has enabled us to learn a lot of new concepts and propose this system.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Androutsellis-Theotokis, S. and Spinellis, D. (2022) A Survey of Peer-to-Peer File Sharing Technologies. Athens University of Economics and Business White Paper (WHP-2002-03).

[2] Napster Inc. https://www.scribd.com/document/456414891/P2P-Design-Document

[3] Ripeanu, M. (2001) Peer-to-Peer Architecture Case Study: Gnutella Network. *Proceedings First International Conference on Peer-to-Peer Computing*, Sweden, 27-29 August 2001, 99-100. https://ieeexplore.ieee.org/document/990433

[4] Kazaa. https://www.sciencedirect.com/topics/computer-science/kazaa

[5] Morpheus. https://www.morpheus.com

[6] Stutzbach, D. and Rejaie, R. (2006) Understanding Churn in Peer-to-Peer Networks. *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, Rio

de Janeiro, 25-27 October 2006, 189-202. https://doi.org/10.1145/1177080.1177105

[7] Francis, P. (2000) Yoid: Extending the Internet Multicast Architecture. https://www.researchgate.net/publication/228368945_Yoid_Extending_the_internet_multicast_architecture

[8] Kostić, D., Rodriguez, A., Albrecht, J. and Vahdat, A. (2003) Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles—SOSP*'03, Bolton, 19-22 October 2003, 282-297. https://doi.org/10.1145/945472.945473

[9] Pai, V., Kumar, K., Tamilmani, K., Sambamurthy, V. and Mohr, A.E. (2005) Chainsaw: Eliminating Trees from Overlay Multicast. In: Castro, M. and van Renesse, R., Eds., *Peer-to-Peer Systems IV*, Springer, 127-140. https://doi.org/10.1007/11558989_12

[10] Venkataraman, V., Yoshida, K. and Francis, P. (2006) Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast. *Proceedings of the* 2006 *IEEE International Conference on Network Protocols*, Santa Barbara, 12-15 November 2006, 2-11. https://doi.org/10.1109/icnp.2006.320193

[11] Zhu, Y.W. and Hu, Y.M. (2003) Enhancing Search Performance on Gnutella-Like P2P Systems. *IEEE Transactions on Parallel and Distributed Systems*, **17**, 1482-1495.

[12] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F. and Balakrishnan, H. (2001) Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *Proceedings of the* 2001 *Conference on Applications*, *Technologies*, *Architectures*, *and Protocols for Computer Communications*, San Diego, August 2001, 149-160. https://doi.org/10.1145/383059.383071

[13] Yao, L., Mao, C.S. and Luo, Y. (2019) KG-BERT: BERT for Knowledge Graph Completion. arXiv: 1909.03193.

[14] Pedregosa, F., Varoquaux, G., Gramfort, A., *et al.* (2011) Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research*, **12**, 2825-2830.

[15] Joung, Y.-J., Fang, C.-T. and Yang, L.-W. (2005) Keyword Search in DHT-Based Peer-to-Peer Networks. 25*th IEEE International Conference on Distributed Computing Systems* (*ICDCS*'05), Columbus, 6-10 June 2005, 339-348. https://ieeexplore.ieee.org/document/1437097

[16] Reynolds, P. and Vahdat, A. (2003) Efficient Peer-to-Peer Keyword Searching. In: Endler, M. and Schmidt, D., Eds., *Middleware* 2003, Springer, 21-40. https://doi.org/10.1007/3-540-44892-6_2

[17] Zhou, F., Zhuang, L., Zhao, B.Y., Huang, L., Joseph, A.D. and Kubiatowicz, J. (2003) Approximate Object Location and Spam Filtering on Peer-to-Peer Systems. In: Endler, M. and Schmidt, D., Eds., *Middleware* 2003, Springer, 1-20. https://doi.org/10.1007/3-540-44892-6_1

[18] Yang, B. and Garcia-Molina, H. (2001) Comparing Hybrid Peer-to-Peer Systems. 2001 *Proceedings of the* 27*th VLDB Conference*, Roma, 11-14 September 2001, 561-570.

[19] Loo, B.T., Huebsch, R., Stoica, I. and Hellerstein, J.M. (2004) The Case for a Hybrid P2P Search Infrastructure. *Proceedings of the Third International Conference on Peer-to-Peer Systems*, La Jolla, 26-27 February 2004, 141-150. https://dl.acm.org/doi/10.1007/978-3-540-30183-7_14

[20] Klumpp, T. (2013) File Sharing, Network Architecture, and Copyright Enforcement: An Overview. *Managerial and Decision Economics*, **35**, 444-459.