Peer-to-Peer Networking and Applications (2020) 13:825–849
https://doi.org/10.1007/s12083-019-00831-2

DPAS: A dynamic popularity-aware search mechanism for unstructured P2P systems

Elahe Khatibi[1] · Mohsen Sharifi[1] · Seyedeh Leili Mirtaheri[1]

Received: 21 February 2019 / Accepted: 28 September 2019 / Published online: 12 November 2019

Abstract
One of the pivotal challenges of unstructured Peer-to-Peer (P2P) systems is resource discovery. Search mechanisms generally utilize blind, or informed search strategies wherein nodes locally store metadata to quicken resource discovery time compared to blind search mechanisms. Dynamic behavior of P2P systems profoundly affects the performance of any deployed resource-searching mechanism and that of the whole system in turn. Therefore, efficient search mechanisms should be adaptable to the dynamic nature of P2P systems whose nodes frequently join and leave the system. Nonetheless, existing informed search mechanisms have failed to accord with dynamicity of P2P systems properly, thereby becoming inefficient. To address this issue, we put forth a new resource-searching mechanism called Dynamic Popularity-Aware Search mechanism (DPAS). Our mechanism estimates the dynamic responsiveness states of candidate nodes to direct search selection process by exploiting temporal number of hits, temporal penalty, and node heterogeneity. Besides, it controls the search scope at each step by estimating both the dynamic temporal popularity of resources and recently obtained results. It thus considers at each step of the search decision-making process to conform itself with the dynamics of P2P systems. Extensive experiments have demonstrated that DPAS has enhanced performance in comparison to other pertinent search mechanisms by virtue of an upsurge in the success-rate and decrease in the response time and bandwidth consumption.

Keywords Unstructured peer-to-peer systems · Resource searching · Dynamic popularity of resources

1 Introduction

Peer-to-Peer (P2P) systems are a kind of distributed system wherein each participating node can simultaneously act as a server and as a client—owing dual functionality; each node is free to join network and interchange resources and services to other nodes spontaneously. P2P systems have often comprised of many nodes that share huge amounts of all types of resources. The main characteristics of P2P systems include scalability, dynamicity, ad-hoc connections, robustness, self-organization, and self-administration. In recent years, P2P systems have gained a lot of attention from both industrial and academic organizations due mainly to the above desirable inherent features; the most popular P2P applications are file-sharing ones. Investigations reveal that a considerable fraction of total Internet traffic is due to P2P file-sharing applications, even more than Internet traffic due to Web applications [1–4].

Buford et al. [1] have classified P2P systems into structured and unstructured based on how they control resource placement and form their network topologies. Structured systems exploit strict rules to specify both file

✉ Mohsen Sharifi
msharifi@iust.ac.ir

Elahe Khatibi
elahe.khatibi@iust.ac.ir

Seyedeh Leili Mirtaheri
mirtaheri@iust.ac.ir

[1] Distributed Systems Research Lab, School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

locations and peer placement in the topology; this is done usually by using a Distributed Hash Table (DHT) mechanism—a distributed indexing service—resulting in an efficient resource searching [2, 5]. Hence, DHT has placed a substantial strain on the system due mainly to the necessity for compulsory self-organizing method of overlay maintenance. In contrast, topologies of unstructured P2P systems are arbitrary and suffer from inefficient resource searching mechanisms. In this regard, coming up with an efficient resource searching is both lifeblood and the most paramount challenge in any given unstructured P2P systems [2].

In this paper, we focus on resource searching in heterogeneous, decentralized unstructured P2P systems. Unstructured P2P systems like Gnutella do not have any restriction on the placement of resources in the system—thereby, enjoying lower maintenance under churn. In other words, they do not consider any correlation between the placement of resources and the network topology, thereby lacking any knowledge about the location of stored files. Hence, most of these systems exert kind of flooding to search for resources and as such create huge network traffic [3, 6].

To counter flooding, many resource-searching mechanisms [7–14] have been proposed for unstructured P2P systems with the purpose of reducing traffic overhead in the primary Gnutella flooding mechanism; these search mechanisms include blind and informed ones. In a blind search mechanism [15–17], nodes do not store any information about locations of resources. Existing mechanisms falling into the blind group still are afflicted with low performance due to their random nature, and inadaptability to different degrees of popularities of requested resources; moreover, blind mechanisms generate large amounts of network traffic. In informed mechanisms [15–18], nodes locally store some information regarding their neighbors, and upon receiving a query message, they opt for a subset of their high-scored neighbors to forward the query to the best ones [8, 9].

Although informed search mechanisms can significantly reduce network traffic, they suffer from important flaws. Current informed resource searching mechanisms do not well accord with alternations in the popularity of resources; the root cause of the fluctuation in resource popularity is in skewed data access patterns in P2P systems. To achieve the best tradeoff between search performance and search cost, any resource searching mechanism should adaptively operate with regard to variable popularities of resources. Search performance is defined as the gained success-rate and search cost, the latter of which is equal to the number of generated messages [2, 19–21]. Furthermore, resource-searching mechanisms must take into account diverse load status of nodes [22, 23]; as a matter of fact, when a node becomes overloaded, it fails to handle received queries, causing a sudden drop in system performance [13–17]. Several critical factors may cause the candidate node to be overloaded like node heterogeneity, node degree, and resource popularities. Additionally, informed mechanisms are not properly adaptable to the dynamic nature of the P2P systems in which nodes frequently join/leave and insert/delete their resources; this results in degradation in system performance swiftly. In fact, stored information at each node may become rapidly invalid; invalid information should be purged, and therefore updated information about either new incoming peers or inserted resources should be stored as well [15–17].

To address the shortcomings of resource searching mechanisms in unstructured P2P systems attributed to inadaptability of these mechanisms to the dynamicity of P2P systems, we present a Dynamic Popularity-Aware Search mechanism (DPAS) for unstructured P2P systems. We focus on files as resources shared by nodes, although our mechanism applies to other kinds of resources too. DPAS is fully distributed, dynamic, scalable, and informed. It enhances searching process by virtue of the following five techniques.

Firstly, each node builds a neighbor table containing information about its neighbors, from which queries are sent to the best neighbors in terms of response capability.

Secondly, DPAS exploits negative feedbacks to update information in the neighbor tables dynamically considering the varying popularities of requested resources as well as nodes' responsiveness states. As a result, DPAS is a new dynamic mechanism that considers both nodes heterogeneity and system erratic conditions.

Thirdly, DPAS makes the best online decision at each search step grounded on the environment's conditions as well as information obtained during recent searches. To do so, by virtue of profiled information during recent searches, it does estimate the popularity of requested resources; therefore, DPAS can define the proper value for Time-to-Live (TTL) and subsequently the number of required queries to be forwarded at the next step. Thus, DPAS behaves differently towards the manifold popularity of requested resources and a different number of obtained results.

Fourthly, DPAS uses a dynamic estimation of node's responsiveness status and node's usefulness. To this end, we estimate node's working load and consider such situations as deletion of node's resources and leaving of the node from the systems; in this respect, DPAS can cover dynamic conditions of unstructured P2P systems as well as high heterogeneity among nodes in terms of responsiveness status. Usefulness and responsiveness of candidate nodes are determined before forwarding queries to them to avoid forwarding queries to overloaded or useless nodes. DPAS applies adaptive penalty and dynamic ranking reduction techniques with regard to system status to reflect recent modifications in nodes' states.

Fifthly, temporal parameters with higher weights on more recently collected data in neighbor tables are introduced. The rationale behind this notion is that a lifetime of most nodes in P2P systems is short [3] and that old information is usually invalid. In light of these efficient techniques, DPAS prevents a drastic plunge in overall system performance; performance degredation is highly probable when incorrect hops are chosen during resource searching due mainly to out-of-date information.

We show experimentally how much DPAS improves the performance of resource searching in terms of success-rate, bandwidth consumption, and response time.

To put it in a nutshell, unstructured P2P technologies and their concomitant concepts due to their inherent characteristic are found overwhelmingly fruitful in a plethora of interactive online and Internet applications; these include file-sharing, video streaming, Voice-over-IP applications, Massive Multiplayer Games (MMG), and Online Social Networks (OSN) [24]. The search mechanism is the mainstay of any given unstructured P2P systems to both maintain and boost the system performance. Given these points, we have been spurred to propose a new search mechanism for unstructured P2P systems to ameliorate deficiency of previous mechanisms. To the best of our knowledge, DPAS is the first search mechanism trying to reap the rewards of the above-mentioned techniques; hence, DPAS takes into account the near-real-time information in diverse parts of their algorithm, including candidate neighbor selection, estimating the number of queries coupled with their attendant TTL values for the next hop, and assessing candidate node status. Consequently, in the interest of search performance assurance, DPAS can involve and reflect dynamicity of P2P systems—a sweeping and underlying factor in the whole system's productivity. In this regard, DPAS has surpassed their counterparts.

The remainder of the paper is organized as follows. Section 2 summarizes related works. Section 3 introduces the main parts of DPAS mechanism. Section 4 reports simulation results and some performance measurements. Section 5 draws our conclusion and outlines future works.

2 Related work

We classify resource-searching mechanisms in unstructured P2P systems into five groups: blind, informed, group, hybrid, and bio-inspired meta-heuristic mechanisms.

2.1 Blind search mechanisms

The first group called *blind* mechanisms produce a great deal of network traffic by sending queries to many nodes blindly [2, 13, 14] [25–27]. In the k-walker random walk mechanism [3]—a blind mechanism—the query originator randomly sends a query message to k number of its neighbors. The mechanism has low performance due to its random behavior and its inadaptability to varying popularities of requested resources. A newer version of BFS like Alpha Breadth-First Search (α-BFS) mechanism [28] has tried to diminish the network resources wastage. Even though this mechanism reduces the average message traffic compared to the flooding scheme, it still generates too much overhead. α-BFS suffers from uninformed query forwarding causing performance degeneration.

There are also some blind resource searching mechanisms that act based on the popularity of requested resources [1, 15, 29], one of which is Expanding Rings (ER) [1, 15]; ER is a blind mechanism that aims to solve the overshooting problem, although it is not too successful. Overshooting happens when additional unwanted results are returned, leading to system resource wastage. ER considers the popularity of resources by defining a set of TTLs during consecutive BFS searches at increasing depths. At first, it forwards a query with a small value of TTL, and search terminates if sufficient resources are found after a defined waiting time. Otherwise, searching is continued by initiating another BFS search with a bigger TTL depth than the previous one. ER suffers from flaws such as repetitive forwarding of queries to nodes having already processed the query, blind query forwarding, ignorance of nodes' heterogeneity during query forwarding, and high network traffic due to flooding the query at each step.

Another blind mechanism that exerts the popularity of requested resources is called Dynamic Querying

(DQ) [15]. DQ first sends a probe query with a small value of TTL to a handful of neighbors. If the desired number of results is not returned, it initiates an iterative process. At each iteration phase, the value of TTL for the next query is estimated based on the number of returned results by which the query is sent to the next neighbor. Search process stops when the desired number of results is received, or all neighbors have been contacted. DQ assumes some conditions for the system that may not always hold due to the heterogeneity of the system. For instance, it presumes that it has always high degrees nodes at its disposal to increase control over the number of hits in a query; this means that nodes with equal degrees are always linked to candidate nodes, and that an equal number of results are always returned from super-nodes that have received the query. As a result putting such strict constraints on the network prevents DQ from being widely adopted. In this mechanism, query forwarding is done blindly and it suffers from long response time, which is due to its sequential operations.

The Equation Based Adaptive Search mechanism (EBAS) [30] is a blind mechanism that uses the requested resources' popularities too. It creates a popularity table in each node to store popularity estimation of each resource in the network based on feedbacks from previous searches. EBAS chooses TTL and a number of random walkers based on information in popularity tables. It assumes that there is some restricted number of resources in the system. This assumption does not always hold in most P2P systems wherein the lack of global information exists. In addition, it forwards the queries blindly and does not consider the temporal popularities of resources and their variation in different conditions. Additionally, this mechanism does not consider nodes' different degrees of responsiveness when forwarding query messages.

Adaptive Resource-based Probabilistic Search mechanism (ARPS) [31] is another blind mechanism that uses the requested resources' popularities. It creates a popularity table in each node for the queries the node produces itself or queries that are received by that node. Each node estimates the resource popularity in the network based on the feedbacks obtained from previous searches; then, each node calculates the query's forwarding probability based on its own degree and the popularity of the requested resource. The main drawback of ARPS is the way it forwards the queries blindly. It does not consider important parameters such as nodes' different responsiveness capabilities when forwarding query messages.

2.2 Informed search mechanisms

The second group of resource searching mechanisms is the *informed* mechanisms, wherein each node amasses and stores some information that is helpful during the search process. Each node sends a query to the nodes that have better responsiveness and are most likely to return wanted results [32]. The informed Directed BFS mechanism (DBFS) [8, 15] has tried to reduce search response time while ignoring important parameters like nodes' changeable responsiveness when it forwards query messages. More precisely, each node forwards a query merely to some of its neighbors that have returned more results during previous searches. In Local Indices [15]—an informed mechanism—each node indexes resources of nodes within a specified radius and answers queries on behalf of those nodes. It uses the BFS mechanism for resource searching, but only nodes in a specified depth, process a received query. It thus achieves higher success-rate due to its indexing method, but at the cost of generating higher network traffic. The network traffic is due to the flooding process that takes place per node leave/join in order to keep correct indexes. In the informed Routing Indices mechanism (RI) [15], each node stores file metadata for each of its outgoing paths and forwards a query to the neighbor having the highest number of documents nearby. Flooding process is required to update nodes' storages after nodes leave/join the system, or when they insert/delete their resources. In Gianduia (GIA) [15], another informed mechanism, a requester node exploits biased random walks to find requested resources by forwarding queries to neighbors that have lots of indexed resources as well as high degrees; the degree of a node is defined as the number of its neighbors. This mechanism also utilizes a topology adaptation algorithm that puts most nodes in short distance of high capacity nodes. The adaptation algorithm imposes extra network traffic. In Intelligent Search Mechanism (ISM) [15, 33] whose goal is to diminish network traffic of flooding mechanisms, queries are forwarded according to their similarities to previous successful queries. Information regarding neighbors and similarity of responded queries are stored in each node; LRU mechanism is used to omit old and invalid information. ISM has critical hurdles, as it does not use negative search feedbacks to update profiles. This means that it does not consider both nodes and resources, which leave the system. It also does not take into account varying popularities of resources, overloading probabilities of candidate nodes, the bandwidth of nodes, and the uptime of nodes during the selection of candidate nodes for next-hop forwarding. Some other new resource discovery mechanisms like Flooding with Random Walk with Neighbors Table (FRWNT) [34], leverages a combination of the blind as well as informed search mechanism across resource searching. FRWNT also languishes due to being negligent on updating information, the popularity of resources, and distinct capacity of each node.

There are some informed search mechanisms deeming either an updating scheme or popularity of resources. Because our suggested mechanism intends to assuage shortcomings of already-proposed search mechanisms by reaping the rewards of both updating scheme and popularity of resources, we assimilate an exhaustive review of informed mechanisms; these informed search mechanisms place a high premium on the aforementioned concepts hereunder. Several informed search mechanisms exploit some type of updating scheme in dynamic P2P environments [35, 36]. Adaptive Probabilistic Search mechanism (APS) [15] is one such mechanism that deploys k-walker random walk mechanism to forward probabilistically a query to k neighbors based on previous results. It only uses one updating mechanism and applies it to different P2P dynamic situations; application of a single updating mechanism does not suit dynamic unstructured P2P systems. In addition, it ignores nodes' heterogeneity and resources' varying popularities during the selection of candidate nodes in forwarding query messages. Another problem occurs when walkers collide, especially when the requester node's degree is less than k. There are also other mechanisms with their own updating schemes. For example, in [37], the query is forwarded to the neighbor that has previously returned the related results in a short period. In 6Search mechanism (6S) [38], each node directly connects to the nodes that have previously returned the desired results; moreover, each node forwards a received query to the best neighbors based on both their previous responses and types of their shared resources. The above-mentioned mechanisms do not deem diverse dynamic parameters during decision making to forward a query at each step, including variable system conditions and different responsiveness capabilities of nodes. In the Hybrid Periodical Flooding mechanism (HPF) [15], which is an informed mechanism, a specified number of queries should be forwarded at each step based on the TTL value and each query receiver node degree. HPF uses two metrics for neighbor selection, namely communication cost (i.e., the physical shortest path between a node and its neighbor), and the number of shared files on each neighbor node. However, it does not consider resources' popularities and the candidate nodes responsiveness states when forwarding queries.

2.3 Group-based search mechanisms

Group-based mechanisms comprise the third group of resource searching mechanisms. In these mechanisms, nodes with similar resources form a system group. A query is sent to one of the groups that are more probable to respond to the query [39]. Interest-Aware Social-Like Peer-to-Peer (IASLP) model for social content discovery has been suggested. IASLP has exploited the group-based approach to identify relevant as well as useful content so as to find the requested resource [40]. Group-based mechanisms are more accurate but are less efficient, especially in systems with limited but varieties of resources at each node. This inefficiency is due to the complication in defining similarities between nodes.

2.4 Hybrid search mechanisms

The fourth group of resource searching mechanisms is *hybrid* mechanisms that use the positive features of existing searching mechanisms in both structured and unstructured P2P systems. This means utilizing flooding for searching popular resources and DHT in the case of rare ones [41, 42]. This group tries to foster searching success-rate, especially for rare requested resources while maintaining the success-rate within an acceptable threshold for popular requested resources [43]. In these mechanisms, both construction and maintenance of the structured overlay impose excess overhead.

2.5 Bio-inspired meta-heuristic search mechanisms

In addition, we have also considered a fifth group—called bio-inspired meta-heuristic group. Some researchers have strived to deploy organic and biological concepts—bio-inspired mechanisms—to wrestle with issues in computer science domains, including search algorithms [44–48]. In this regard, some search mechanisms [44–48] have been inspired from humoral immune system (IBS) [46], structure of fungi, or cellular slime mold life cycles (Dictyostelium discoidem) (SMP2P) [44, 45]; the latter proposed mechanism suffers from considering some rules for peer position and overlay network in order to map slime mold bio-mechanism to P2P lookup operation; this brings about extra overhead. Other mechanisms use food-foraging behavior of either bees or ants—whose focus is on using swarm intelligence to resolve complex problems [3, 47, 48]. Take food-forager ants analogous to query, tries to search for desired resource by depositing some chemical substance, namely pheromone as trials, for instance. IACO search mechanism [7] utilizes inverted ant colony optimization to take into account load-imbalance issue as compared to ACO [48], by diminishing the pheromone impact on used paths. IACO fails to adapt to frequently changing conditions of P2P systems by appraising a myriad of contributing factors at each search step. Moreover, IACO is negligent in considering candidate node status before sending queries. Accordingly, the system suffers from additional network traffic, overloaded nodes, dwindling success-rate, and prolonged response-time.

3 DPAS mechanism

In this section, we present our new resource searching mechanism called DPAS. Resource popularities change in P2P systems as system conditions vary. Thus, it is required to define the TTL values of queries and the number of forwarded queries at each step; these calculations should be done based on system conditions and resource popularities to prevent drops in system performance. DPAS considers the mentioned factors to control and constrain the search scope. Briefly, the novelty of DPAS is that it pushes itself to the limits so as to estimate both the search scope and the best candidate nodes at each search step in a near-real-time manner; to do so, DPAS defines some temporal parameters parallel to adapting to the dynamic nature of P2P systems.

Our mechanism prevents overshooting by dynamically estimating the search scope at each step based on current popularities of requested resources, the number of obtained results and the responsiveness of neighboring nodes. In addition, DPAS considers accurate factors in its decision-making procedure in selecting candidate-neighboring nodes to which forwards the queries. Hence, DPAS exhibits a high success-rate.

Message queues of nodes with popular resources usually get full quickly because each queue has a finite capacity, and the usually extreme number of queries is received for the popular resources. When the incoming message-rate exceeds the node's capacity, the node gets overloaded and drops any subsequent messages sent to it. Furthermore, when the overloaded nodes drop the incoming messages, i.e., query or response message, they are not able to forward these messages to other nodes; as a result, the success-rate and consequently the whole system performance drop rapidly. In addition, when the degree, ranking or usefulness of a node are high, the number of received queries and responses as well as the number of forwarding queries by that node increase. The usefulness of a node is calculated based on the node's previous responses to its neighbors' queries. Furthermore, a huge number of queries and response messages can overload any node in the search path that has the mentioned characteristics. This is especially more probable to happen for low-capacity nodes. In light of these, we consider all of the mentioned contributing factors in our mechanism as a whole to estimate response capability of each candidate node before forwarding the query to them; this is one contribution of our work. In the system that we model, we consider two finite message queues for each node, one for received query messages and another for received response messages. If any of these queues get full, subsequently received messages are dropped. The important goals of DPAS include increasing the success-rate and accuracy as well as decreasing the bandwidth consumption and response time while adapting to dynamic P2P system conditions in terms of nodes frequent leave and join and also insertion or deletion of resources. Each node executes DPAS mechanism locally. DPAS mechanism comprises of two main schemes, namely dynamic neighbor table scheme and search scheme. Figure 1 illustrates the visual overview of DPAS's indispensable components.

In the first scheme, namely dynamic neighbor table, each node stores some information about the recently answered queries via its neighbors in a table. This information is used in the second scheme, namely a search scheme, to select the best candidate neighboring nodes. Dynamic neighbor table scheme uses the following two mechanisms: Adaptive Replacement Mechanism and Dynamic Updating Mechanism.

Adaptive replacement mechanism This mechanism is used to replace the old values in the dynamic neighbor table with newly gathered information whenever a node's dynamic neighbor table gets full. To achieve this, two parameters, namely query popularity and responsiveness abilities of nodes are considered.

Dynamic updating mechanism This mechanism updates the stored information of the table by applying the feedback it receives from recent queries. There are three feedback messages, namely *query-hit*, *rank-setting*, and *updating messages*. Using these messages, the rankings of nodes are modified to reflect the varying system conditions.

The following pseudocode demonstrates the process of dynamic neighbor table scheme.

Pseudocode_1.

```
Dynamic Neighbor Table Scheme //incorporating both Adaptive Replacement and
Dynamic Updating Mechanisms
        If Dynamic Neighbor Table is full
            call Adaptive_Replacement()
        End if
    Else If Receive any feedback message msg (query-hit, rank-setting,
    updating-message, query) // functionality of Dynamic Updating Mechanism
            If msg is query-hit
                Update value of temporal-total-number-of-hits
                parameter
            End if
            If msg is rank-setting //implication of infertile search path
                Update value of Temporal Ascending Penalty parameter
            End if
            If msg is updating-message // implication of corrupted or low-
            quality resource
                Update value of temporal-number-of-invalid-hits
                parameter
            End if
            If msg is query which bears a striking resemblance to previously
            stored queries
                Update value of temporal-number-of-queries parameter
            End if
            If msg is query and receiver possesses the requested resource
                Update value of temporal contribution level parameter
            End if
        Procedure Adaptive_Replacement() // Adaptive Replacement
        Mechanism
        begin procedure
        select row as a victim_row whose value $U_{row(i)}$ is minimum
        $U_{row(i)} = W_1 * \frac{Temp\_Popularity_{row(i)}}{Max\_Temp\_Pop} + W_2 * \frac{Nodes\_States}{Max\_Nodes\_States}$ // formula (5)
        end procedure
    End if
```

The second schema—search scheme—uses the dynamic information in the neighbor tables to select the best candidate nodes; these so-called best candidate nodes are more capable of responding, and enjoying the higher probability to own the desired requested resource—when a new query is received. It uses the following two mechanisms: Dynamic Estimation Mechanism and Dynamic Status-Aware Search Mechanism.

Dynamic estimation mechanism Each node uses this mechanism upon receiving a query to estimate the number of forwarding queries and query's TTL value for the next step; these calculations are done based on both node's dynamic neighbor table's information and the number of already returned results.

Dynamic status-aware search mechanism (DSAS) This mechanism selects the best neighbor nodes to forward them the current query considering their responsiveness. This mechanism uses dynamic neighbor table's information as well. The following pseudocode depicts the searching process of the DPAS mechanism—functionality of above-mentioned search scheme.

Pseudocode_2.

```
Search Scheme //entailing Dynamic Estimation as well as Dynamic Status-Aware Search
Mechanisms

        1.  Receive a query
        2.  If the number of needed results or TTL value equals zero then
                    Do not send the query further
        3.  Else

            4. Opt for the k most similar queries to the currently received query via
            cosine similarity function
            5. Calculate the number of queries and TTL value for the next hop, via
            Equations (13), and (16) // Dynamic Estimation Mechanism
            6. Send the query to the highest-rank neighbor by Equation (23) //Dynamic
            Status-Aware Search Mechanism

        End if
```

The two schemes, namely Dynamic Neighbor Table Scheme as well as Search Scheme and the mechanisms they use–including Adaptive Replacement Mechanism, Dynamic Updating Mechanism, Dynamic Estimation Mechanism, and Dynamic Status-Aware Search Mechanism (DSAS)—are presented in detail in the following subsections.

3.1 Dynamic neighbor table scheme

In our proposed mechanism, DPAS, each node has a Dynamic Neighbor Table (DN-Table) for storing information on its neighbors. In the dynamic neighbor table scheme, dynamic updating mechanism uses the information stored in this table. Each row of this table, for each node, is allocated to a self-produced or received query by that node and consists of a list of neighbors that have previously responded to this query. This response is either generated by the neighbor itself or it is routed through the neighbor; in unstructured P2P systems, query-hit messages traverse the same path as that of the query but in reverse direction. Query's keywords and its *temporal-number-of-queries* parameter are stored in each row. For each row, the *temporal-number-of-queries* parameter contains the number of other received queries that are supposed to be similar to that row query.

Each node stores extra-temporal information on its own resources in related resources rows of the table to estimate resources popularities; resource popularity specifies the

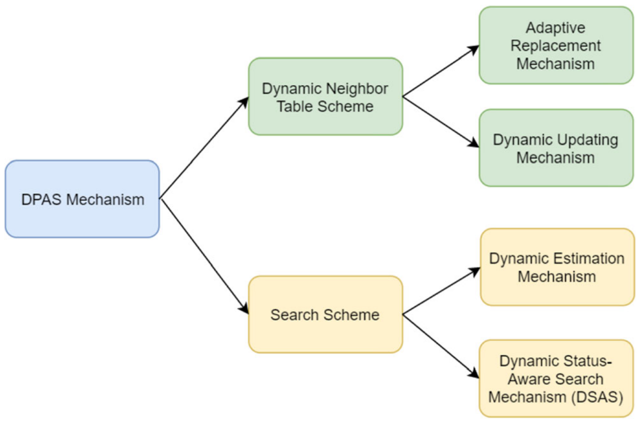

Adaptive Replacement Mechanism
Dynamic Neighbor Table Scheme
Dynamic Updating Mechanism
DPAS Mechanism
Dynamic Estimation Mechanism
Search Scheme
Dynamic Status-Aware Search Mechanism (DSAS)

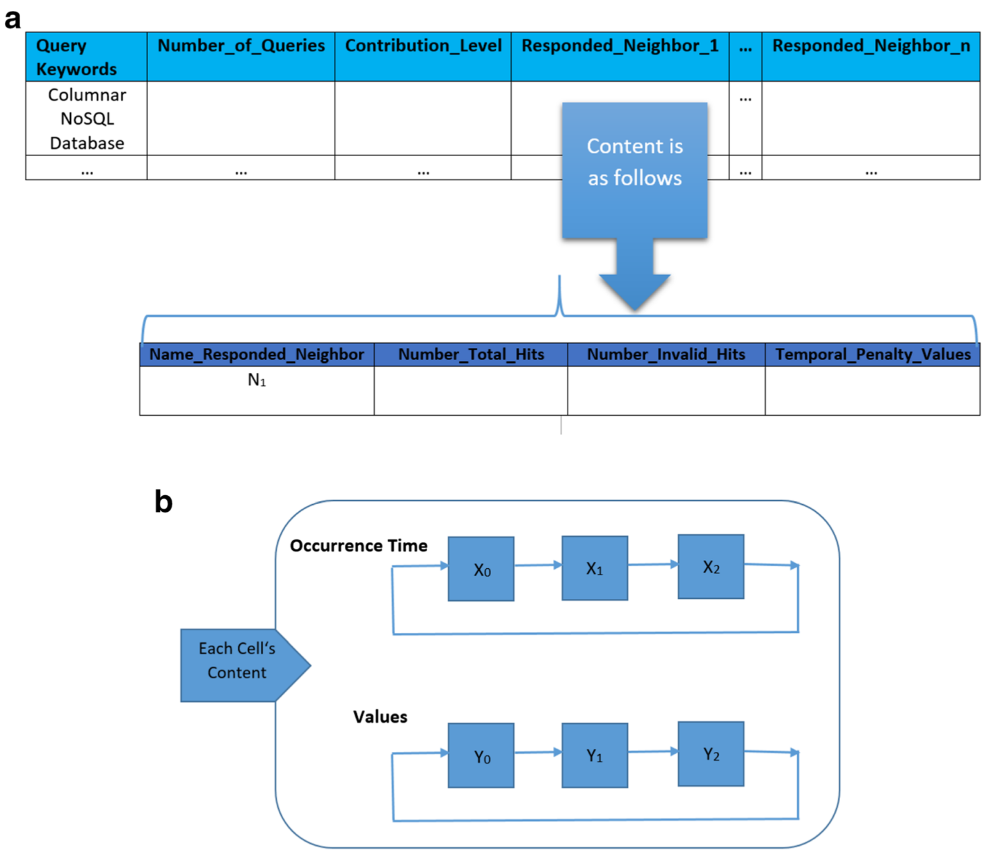
Fig. 1 Visual Overview of DPAS Mechanism and its Attendant Components

number of nodes that should process the query. This information is in fact about the contribution level of the previous requester nodes who have already received the mentioned resources. By contribution level, we estimate the probability that the requester node shares the received resource in the system. Each cell of the mentioned neighbor list contains a sub-list containing some items; these items include the name of the responding neighbor node, *temporal-total-number-of-hits*, *temporal-number-of-invalid-hits* that are not supposed to be valid based on their quality, and the value of temporal adaptive ascending penalties for the related query. Each node also stores both name and size of files shared by its neighbors by asking them to do so. Each node uses this information to calculate loads of its neighbors, *node state*, as we explain later. All items in the DN-Table, except query keywords, are temporal parameters. These temporal paramteres are used to record changes in system conditions like nodes leaving/entering the system or deletion/insertion resources. Hence, the more recent such information is collected, the more effective they would be.

Items in the DN-Table are calculated based on the received queries, their relevant query-hit messages, and their number of related returned results (as defined in more details in later sections). Figure 2a shows the structure of DN-Table coupled with illustrative values; moreover, Fig. 2b depicts the content of each cell of DN-Table.

Let us now explain how temporal items in DN-Table are initialized and weighted. We have designed a temporal "Dynamic Weighted Interval Set" mechanism that uses distinct sets for recording the temporal and weighted values of each of the items; each set records the values for that item at a given time upon the occurrence of a given event. To this end, we consider x interval sets. Each of these interval sets continues for t units of time and a pointer is used to demonstrate the current set. We have implemented these sets by storing related information in two data structures (Fig. 3), namely a cyclic list. These cyclic lists record the first event occurrence

Fig. 2 a. Structure of Dynamic Neighbor Table b. Content of Each Cell of Dynamic Neighbor Table



Query Keywords	Number_of_Queries	Contribution_Level	Responded_Neighbor_1	...	Responded_Neighbor_n
Columnar NoSQL Database				...	
...	...	...		...	...

Name_Responded_Neighbor	Number_Total_Hits	Number_Invalid_Hits	Temporal_Penalty_Values
N_1			

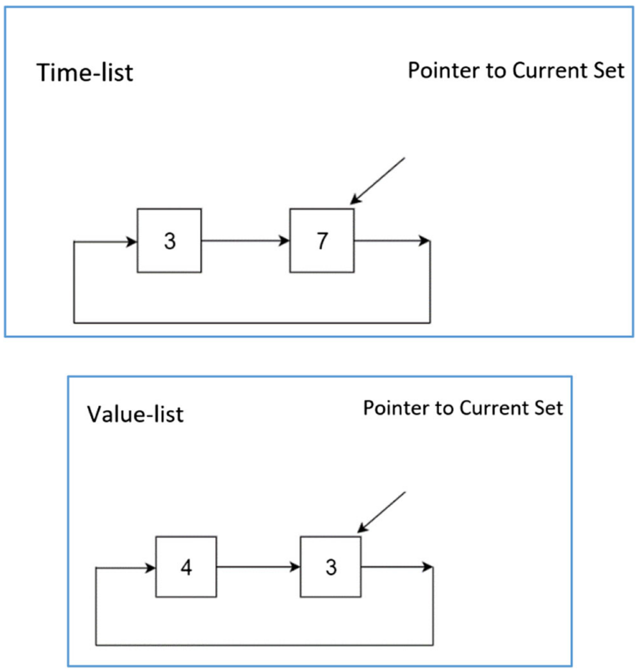


Fig. 3 Cyclic lists for temporal items with illustrative values

time and another cyclic list to record the item value [49]. We call these lists *time-list* and *value-list*, respectively. The size of each list is *x*.

Whenever a related event for each temporal item in the DN-Table is triggered, in each entry of the *time-list*, its related *value-list* entry is set. The time of the first occurred event in each of the intervals is also recorded in a *time-list* entry in terms of that *value-list* entry. These sets are designed to be cyclic. After reaching the end of each *time-list* entry, the next entries of both lists are started, and the two related pointers are set to these newly arrived entries. A *time-list* entry ends if by receiving an event the subtraction of the value of *time-list*'s current entry from current system time (i.e., time of received event) is bigger than *t* units of time. If the current entry of the *time-list* is the last one, and the mentioned previous condition is held, by going to the first entries of both lists again, the values of both lists entries are reset based on the received related event.

Events triggering changes to temporal items are as follows:

1. An event for the *temporal-total-number-of-hits* item, which is triggered by receiving query-hit messages.
2. An event for *temporal-number-of-invalid-hits* item, which is triggered by downloading a resource with low (i.e., imperfect or incorrect) quality.
3. An event for *temporal-number-of-queries* item, which is triggered by receiving a new query if there is any similarity between the current received query and previously-stored queries.
4. An event for temporal contribution level item, which is triggered when the owner of a resource receives a query, messages for that resource. Therefore, the contribution level of requester node is recorded in resource owner node table based on information that is stored in the query message.
5. An event for temporal ascending penalty item that is triggered by receiving rank setting messages.

Parameters *x* and *t* are systems defined. For giving more weight to more recent events, we subtract the time in which the above-mentioned events have triggered from the current system time; afterward, we utilize the inverse of this result as a weighting value for temporal items as it is described in details in the following example. Because each node does these operations locally, there is no need for global time. We use temporal items and their attendant weighting schemes in most of the mechanisms such as in replacement, updating and search mechanisms.

To clarify the above points, let us consider an example for setting the *temporal-total-number-of-hits* item i.e., *Temp_Total_Hit(responder_neighbor,* query*)*. Suppose that *t* and *x* are set to 3 and 2, respectively. Node n_1 has received 2 query-hit messages from its neighbor n_2 for the stored query q_1 in its DN-Table. As each query-hit message can contain more than one result, we assume the first message is received at time $t_{i=0} = 3$ where *i* is the index of the current entries in both of the mentioned lists; moreover, the message includes 4 results as *Temp_Total_Hit(i = 0)* is set to 4. Therefore, the first entry of *temporal-total-number-of-hits*'s relevant *time-list* is set to 3 and the first entry of its relevant *value-list* is set to 4. The second query-hit message is received at time $t_{i=1} = 7$, and it includes 2 results as *Temp_Total_Hit(i = 1)* is set to 2. Therefore, the current system time has the value of 7 (*current_time* = *7*) when this event has occurred; because the subtraction of the time-list's current entry from current system time is bigger than *t*, i.e., (7–3) > 3, we go to the next entries of both lists. Figure 3 shows the filling process of the entries in both of the mentioned lists.

In order to give weight to the *temporal-total-number-of-hits* item, we divide the content of the value-list entries by the subtraction of their relevant time-list entries' values from the current system time. Assuming that current system time in node n_1 is equal to 10, the weighted value of *temporal-total-number-of-hits* for neighbor node n_2 is:

$$Temp\_Total\_Hit(n_2, q_1) = \sum_{i=1}^{x} \frac{Temp\_Total\_Hit(i)}{current\_time - t_i} = \frac{4}{(10-3)} + \frac{2}{(10-7)}. \quad (1)$$

By this way, the effect of old value-list's entries is reduced.

In Eq. (1), the *Temp _ Total _ Hit*(*i*) variable is *temporal-total-number-of-hits* value of the *i-th* value-list's entry and t_i is its related event occurrence time in the *i-th* time-list's entry. The related event is triggered by receiving a query-hit message for each row of the table. Other temporal items can be calculated similarly using their own relevant events.

3.1.1 Adaptive replacement mechanism

Because of the limited capacity of nodes' dynamic neighbor tables, we have to think of an efficient replacement policy to put in a new incoming query in place of the old ones in a table, in case the table is full. We call this policy Adaptive Replacement mechanism (AR) and replace entries based on both temporal as well as variable files' popularities and *Node_Status* parameter. By popularity, we mean how often a specific resource is requested, and in fact, it shows the interest of other nodes to have this resource. The reason for considering the popularity factor is that most queries in P2P systems target popular resources. We have provided the "*Temporal Popularity*" parameter, i.e., *Temp_Popularity*, to execute the replacement policy for the rows of tables. This parameter is calculated by Eq. (8) and Eq. (9) that are presented later.

In addition, we pay attention to recent responsiveness states of responding neighbor nodes when deleting DN-Table rows and when choosing a candidate neighbor node for the next-hop forwarding. The reason for considering recent responsiveness states of responding neighbor nodes is that there may be a situation wherein a requested resource is popular, but selected candidate nodes cannot respond because their states have changed or their responsiveness levels have dropped.

Due to the dynamic nature of P2P systems, it is probable that nodes leave the system, delete some of their popular resources, or become overloaded (i.e., this may occur due to incoming message traffic or the traffic of downloading each node files); as a result, responsiveness states of nodes degrade. Therefore, we consider responsiveness states of responding neighbor nodes in each row of a DN-Table. In this regard, we can select the row that relates to responding neighbors with minimum responsiveness states to be replaced. We show responsiveness states of all nodes in each table row with *Nodes_States* parameter that is calculated by Eq. (2). The responsiveness state of each node is based on two parameters, namely *Overloading_Probability* (i.e., the probability of nodes becoming overloaded) and *usefulness*. The first parameter in Eq. (2) is the *Overloading_Probability* parameter that denotes the probability of each row's overloaded responding neighbor node. The *Overloading_Probability* parameter is used to estimate the load that is imposed on each responded neighbor node; this load includes incoming message load (i.e., query or query-hit messages) and the load that is due to downloading the node's files. This parameter is calculated via Eq. (24) and is explained in details later in subsection 3.2.2. In the calculation of the *Overloading_Probability* parameter, we avoid calculating the *Pop_Current_Query* parameter in Eq. (25)—pointed out in subsection 3.2.2—because it does not design for AR mechanism. The *Usefulness* parameter defined by Eq. (3) using penalty values calculated in the Dynamic Updating mechanism (DU) to be described in details in subsequent sections—in subsection 3.1.2. Briefly, a penalty value is assigned to a node that on the one hand has a positive value showing its base usefulness status but at the other hand its resources are not accessible (i.e., when resources are deleted or the node has left the system or has failed). By subtracting the penalty values from the base value, the usefulness status or *usefulness* parameter for the node is calculated.

$$
\begin{aligned}
&Nodes\_States \\
&= \left[\left(1-\frac{\left(\sum_{i=1}^{n} Overloading\_Probability_{P_i}\right)}{Max\_Overloading\_Prob}\right)+\frac{\left(\sum_{i=1}^{n} Usefulness_{P_i}\right)}{Max\_Useful\quad ness}\right]
\end{aligned}
\tag{2}
$$

$$
Usefulness = (Initial\_Score - [Asc\_Penalty](P_i)) \tag{3}
$$

$$
Asc\_Penalty = \sum_{j=1}^{x} \frac{A\beta(j)}{current\_time - t_j} \tag{4}
$$

Parameter P_i in Eq. (2) and Eq. (3) denotes the i-th responding node of each row and parameter n denotes the number of responding nodes in each row. In addition, *Asc_Penalty* in Eq. (3) denotes the temporal ascending penalty value for each responding node that is calculated by Eq. (4). In Eq. (4), is the ascending penalty value for the *j-th* value-list's entry and t_j is its related event occurrence time for the *j-th* time-list's entry. The related event for the temporal ascending penalty parameter, in fact, receives a rank setting message for each row of the table. For calculating all temporal parameters, we sum up their weighted values in distinct interval sets. For example, consider we have 3 interval sets ($x = 3$) with ascending penalty values =1, =3 and $A\beta(j = 2) = 4$ (j is the index of list's entry). Assuming the occurrence time values are $t_{j=0} = 4$, $t_{j=1} = 7$ and $t_{j=2} = 12$ and the current system time in node n_1 is 19, the final value of temporal ascending penalty parameter is calculated by Eq. (4).

$$
\sum_{j=1}^{x} \frac{A\beta(j)}{current\_time - t_j} = \frac{1}{(19-4)} + \frac{3}{(19-7)} + \frac{4}{(19-12)}
$$

The *Usefulness* parameter should have a positive value, so we subtract the summation of penalty values from a big positive value *Initial_Score*. Our simulations reveal that setting *Initial_Score* to 100 produces the best results. After calculating the *Nodes_States* value, we use it in Eq. (5) to select the victim row. Equation (5) is a utility function that selects the row with minimum utility value; this means the row with minimum values for both temporal popularity and *Nodes_States* parameters to be replaced by the new received query. W_1 and W_2 are system-defined parameters that weight the parameters and show the level of importance of these factors in the system and $W_1 + W_2 = 1$ always holds.

$$
U_{row(i)} = W_1 * \frac{Temp\_Popularity_{row(i)}}{Max\_Temp\_Pop} + W_2 * \frac{Nodes\_States}{Max\_Nodes\_States} \tag{5}
$$

In the denominator part of the first and second expressions of Eq. (5), we put the maximum value of temporal popularity and *Nodes_States* parameters among each table row *m* of the DN-Table, respectively. This is done to normalize these parameters as their values are from different ranges. These values are calculated by Eq. (6) and Eq. (7).

$$Max\_Temp\_Pop = \max_{\forall 1 \leq m \leq last\_row} Temp\_Popularity_{row(m)} \tag{6}$$

$$Max\_Nodes\_States = \max_{\forall 1 \leq m \leq last\_row} Nodes\_States_{row(m)} \tag{7}$$

3.1.2 Dynamic updating mechanism

There are different reasons when no response is returned from a node that is expected to return a response. No response from a candidate node path may be due to the zero value of *TTL*, indicating an end to the search process. No response may also occur when some node in the search path has left the system, or when it has deleted the requested resource. Considering these different situations, DN-Table is updated as follows.

To reflect the dynamicity of the system due to the departure of nodes, resource deletion or changes in users' favorites, we use a dynamic updating mechanism in order to update information that is stored in each node's DN-Table. Users' variant favorites lead to different query generation rates for resources that demonstrate the popularity level of each resource. In this mechanism, if a query is forwarded to a selected candidate node, and the last node in the search path has no relevant resource while its received query has the value of $TTL = 0$, a rank setting message is sent from this last node in the search path in reverse direction to the requester node. Therefore, each node *A* that receives the rank setting message registers a penalty value for the sender node *B* with regard to the value of *TTL* at the time *A* had sent the query to *B*, namely the *MY_TTL* value. This way, the penalty values for the intermediate nodes along the inverse path of search path increase as we get nearer to the query originator node.

It is worth noting that to cover the dynamicity of P2P systems, we introduce the "*Temporal Ascending Penalty*" parameter. We allocate weights to the penalty values by using a temporal mechanism and initializing them by their pertinent event; this means, their event is receiving a rank setting message. In other words, by receiving a rank-setting message, each node uses the relevant *MY_TTL* value to update the value of the temporal ascending penalty parameter for the current interval of the related row and its relevant neighbor, from which the rank setting message is received. In addition, if the result of subtraction of the time-list's current entry from the system time is bigger than the interval length *t*, we go to the next entries of both of the mentioned lists and set the time-list's current entry to current system time. The key point behind this scheme is that nodes that have received the query with $TTL = 0$ still can be a candidate for later queries while only their ranks are decreased by a small value. This mechanism helps in AR process to delete useless paths.

In addition, if the query generator receives a corrupted or low-quality resource, it sends an update message in the path through which it has received the mentioned requested resource. Every node that receives this update message and the query generator node, reduces the rank of the node that they had already selected to forward the related query. This is done by adding one value to the value-list's current entry of the *temporal-number-of-invalid-hits* parameter for the related row and its relevant neighbor, which had sent the query to it previously. In this case, the receiving of an update message by each intermediate node is the event of the *temporal-number-of-invalid-hits* parameter. In addition, if the result of subtraction of the time-list's current entry of the *temporal-number-of-invalid-hits* parameter from the system time is bigger than the interval length *t*, we go to the next entries of both of the mentioned lists and set the time-list's current entry to the current system time. After a while, if some useful nodes join a previous useless path, the effects of these temporal penalties get lower and lower due to the adaptive nature of our mechanism.

3.2 Search scheme

Consider the problem of searching for a copy or multiple copies of a resource in a large, connected unstructured P2P network. Each node can start to search for a resource by creating a query message with both a specified number of needed results and a specified TTL to indicate the termination condition. Then, it forwards the query by virtue of dynamic and online decision-making mechanism, which are explained later in details in Subsections 3.2.1., and 3.2.2 accordingly. The requester node calculates the proper number of queries to be forwarded to the next step; this is dones by considering the number of needed results that is stored in the query message and estimating the popularity of the requested resource via collected information in its DN-Table.

The popularity estimation mechanism is explained later (Section 3.2.1). To select candidate nodes to forward the query to them, the requester node dynamically chooses nodes with the most responsiveness ability to the current query. Every query receiver node that owns the requested resource also sends the response message in the reverse direction of the search path to the query originator node. Intermediate nodes can update their DN-Tables' information too based on this received response message.

Every node that receives the query message calculates some value through the dynamic TTL estimation mechanism and subtracts this value from TTL. If the result is greater than zero, the search continues in the same manner as explained above. In addition, each query receiver node should estimate the number of needed queries for the next step and selecting the best candidate neighbor nodes as well. Dynamic TTL estimation mechanism is explained later (Section 3.2.1). Each query message has also a field called the Request History List in which the name of query receiver nodes is stored. The main thrust behind this field is to prevent repetitive forwarding of that query message to nodes that already have received the query.

The DPAS mechanism works well in both cases of searching for a copy or N copies of a requested resource. However, it does not use the same search policy for the case of a popular resource or a rare resource searching. This is done to adapt to different dynamic P2P environment conditions and as a result preventing performance degradation or incurring excess overhead. To this end, we propose solutions for the overshooting problem, i.e., preventing from excess returned results and spreading the query message more than necessary. This way, system resource consumption is reduced. In other words, DPAS controls the search scope and the number of generated query messages at each search step; DPAS cotrols search scope based on an estimation of the popularity level of the requested resource and the number of local results received during previous search steps. In addition, due to the parallel spread of query messages that belong to the same search process, every query receiver node can update its information about the remaining number of needed results by online contact with the query generator node after a specified number of hops passed by the query. This is done by subtracting the number of results received by the query generator node from other search paths, from the original number of needed results; this results in the termination of the search process when no other result is needed.

3.2.1 Dynamic estimation mechanism at each step

Resource popularities are not static and may vary as nodes leave/join and resources are inserted, deleted, or replicated. In addition, different nodes may have different favorites for each file, resulting in different incoming query rates for that file. Therefore, the dynamic popularity of resources should be estimated adaptively, as it is a driving factor in the imposed load on the nodes. That is, dynamic popularities of resources have great consequences on nodes' workloads based on different incoming query-rates with consideration to the popularity level of resources.

There are usually many replicas for popular resources, so they are found easily. In this case, if the search process terminates after obtaining the desired number of results, the number of generated messages, and consequently the search cost is decreased. Conversely, when searching for low popularity (i.e., rare) resources, we should send more query messages to find the desired number of results. Therefore, the DPAS mechanism adaptively considers the current popularity of each resource into account to improve the search performance, and reduce the search cost.

Every node that receives a query message first calculates the popularity of the requested resource *rr* via Eq. (8) by using the information stored in its DN-Table. This is done to determine the number of nodes that should process the query at the next step based on the estimation of the requested resource's popularity. This means that the TTL value and the number of query processing nodes for the next step is calculated based on resource popularity. If the requested resource is not in the query receiver node's DN-Table, the query receiver node uses the cosine similarity function to select the k most similar queries rr_{j} $(0 < j < k)$ to current query *rr*. These similar queries are stored in the node's DN-Table. In this case, the popularity of the requested resource is calculated by weighted averaging of the popularities of these k similar rows using Eq. (9). This is because of the fact that usually similar resources have similar popularity levels and when a resource becomes popular, similar resources become popular too. If the requested resource is in the query receiver node's DN-Table and $j = 1$, we use Eq. (8) to estimate the popularity of the requested resource.

$$Temp\_Popularity_{rr} = \left[\frac{\sum_{i=1}^{n} Real\_Temp\_Hit(P_i)}{n} + Dynamic\_Feat + \left(\frac{\sum_{i=1}^{m} Contribution(P_i)}{m}\right) {*}W_c\right] \tag{8}$$

$$Temp\_Popularity_{rr} = Weighted.Average\left(\left[\frac{\sum_{i=1}^{n} Real\_Temp\_Hit(P_i)}{n} + Dynamic\_Feat + \left(\frac{\sum_{i=1}^{n} Contribution(P_i)}{m}\right) {*}Qsim(rrj, rr)^{a}\right)\right. \tag{9}$$

The popularity of a requested resource is calculated by Eq. (8) or Eq. (9) based on three main factors, namely *Real_Temp_Hit*, *Dynamic_Feat,* and *Contribution* level of each node as follows. In Eq. (8) and Eq. (9), the first expression, namely the *real-temporal-number-of-hits* denoted by $Real\_Temp\_Hit(P_i)$ for each responder neighbor node P_i, is calculated by Eq. (10). Firstly, we calculate the final value of the number of invalid returned results from P_i, i.e., $Invalid\_Temp\_Hit(i)$, by taking into account the temporal feature as well. Secondly, we calculate the final value of the total number of returned results by P_i, i.e., *temporal-total-number-of-hits*; needless to say, temporal feature is also applied; the aforementioned two final values are derived from the first and second expressions of Eq. (10), respectively. Ultimately, we subtract these two values. By giving more weight to more recent data as explained in Section 3 on the temporal mechanism, we increase the effect of new data compared to old gathered data. Of course, all returned results are not valid results. They may be corrupt or have low quality. Therefore, we define the concept of results-quality by the parameter *temporal-invalid-number-of-hits* to cover low quality returned results. We consider the number of hits when calculating the popularities of resources. This is because often many responses are returned when searching for popular resources, and there is a direct relation between the popularity of a resource and the number of returned results for that resource.

$$Real\_Temp\_Hit(P_i) = \left(\sum_{i=1}^{x} \frac{Temp\_Total\_Hit(i)}{current\_time - t_i} - \sum_{i=1}^{x} \frac{Invalid\_Temp\_Hit(i)}{current\_time - t_i}\right) \quad (10)$$

The final value of the *real-temporal-number-of-hits* parameter is derived by averaging the *real-temporal-number-of-hits* parameter values of each of the P_i responder nodes for the equivalent row to the current query in Eq. (8). For the case that there is no equivalent row to the current query, the final value of the *real-temporal-number-of-hits* parameter is derived by averaging the *real-temporal-number-of-hits* parameter values of each of the P_i responder nodes for each of the k similar rows to the current query in Eq. (9). Parameter n in the first statement of Eq. (8) and Eq. (9) indicates the number of responder nodes for the related row.

The P_i parameter in the above-mentioned equations, namely (8), (9), and (10) denotes the i-th responder neighbor node of each similar/equal query row to the current query; the n parameter defines the number of responder neighbor nodes in each row. In Eq. (9), rr_j stands for the *j-th* similar query that is responded by the P_i neighbor node. The received response from P_i is either generated by the P_i itself or is routed through it; in unstructured P2P systems, query-hit messages traverse the same path as the query is routed but in the reverse direction. By $Real_Temp_Hit(P_i)$we mean the *real-temporal-number-of-hits* parameter value for each query rr_j which is responded by neighbor node P_i.

In Eq. (10), and $Invalid\_Temp\_Hit(i)$ are *temporal-total-number-of-hits* and *temporal-invalid-number-of-hits* parameter values for the *i-th* value-list's entry and t_i is their related event occurrence time for the *i-th* time-list's entry. The related event for the *temporal-total-number-of-hits* parameter receives a query-hit message for each row of the table. In other words, any node that receives the query sends the query-hit message in the reverse direction of search path to the query originator if it has the requested resource; furthermore, the intermediate nodes will change *temporal-total-number-of-hits* parameter value in their tables to show this new query-hit.

Each query-hit message can include more than one result if its related resource owner has more than one related resource. The number of results for each query-hit message is stored in its query-hit message's field named *result-set*. In this case, the receipt of a query-hit message by an intermediate node constitutes the *temporal-total-number-of-hits* event parameter. Therefore, the value of the *temporal-total-number-of-hits* parameter for the related row and its relevant responded neighbor, from which the query-hit message is received, gets updated. Its new value is derived from adding the value of the *result-set* field in the query-hit message to the value stored in the value-list's current entry of the *temporal-total-number-of-hits* parameter. The related event for the *temporal-invalid-number-of-hits* parameter is also the receipt of a low-quality resource that was explained in Section 3. Calculation of all temporal parameters and definitions of other parameters in Eq. (10) and the rest of equations, namely (11), (12), and (20) are similar to those for the *temporal-total-number-of-hits* parameter example explained in Section 3.

Given the facts that TTL has different values in different queries or response messages, the number of queries forwarded at each step is different, nodes frequently leave or join the system, and resources are added or removed from the system, the number of recently returned results for a query is not the exact estimation of the popularity of the requested resource. For example, it is possible that while a node in a previous search process for a possibly popular resource receives some queries, no good or not enough results are received because the TTL value has reached zero.

In other words, when we want to estimate precisely the popularity of a requested resource, we must take into account the above-mentioned situations by proposing the *temporal number_of_Queries* parameter; this means the number of recently received queries for each requested resource. By this parameter, we consider two cases that affect the popularities of resources. The first case is when a sufficient number of results is not returned due to limited search scope of previous searches. The second case relates to the estimation of the number of returned results for a popular resource; this number grows in comparison with the previous number of returned results by increases in the popularity of resource as time passes.

Given the above reasoning, we use another parameter namely, *Dynamic_Feat* (i.e., the second parameter of Eq. (8) and Eq. (9)) to calculate the popularity in Eq. (8) and Eq. (9). This new parameter is calculated via Eq. (11). In Eq. (11), we have introduced the *temporalnumber_of_Queries*" parameter in order to count the number of recently received queries for each table's row. This parameter is initialized to zero for all rows of DN-Table, and its value is set later when the related event is triggered. The event for this temporal *number_of_Queries* parameter is defined as follows.

When a node receives a query, if there are some similarities between the currently received query and the stored queries in the DN-Table, the event for the current interval set of related row's temporal *number_of_Queries* parameter is triggered. Therefore, the value-list's current entry related to the temporal *number_of_Queries* parameter i.e., $number\_of\_Queries(i)$, is increased by one; this increment is done for each similar/equal stored queries of the table. The final weighted value of the temporal *number_of_Queries* parameter is calculated by Eq. (11); the definitions of other parameters in Eq. (11) are similar to the *temporal-total-number-of-hits* parameter example. However, the fact is that this parameter does not has a direct effect on the estimation of resource popularity value, so a weighted variable W_d is used to reduce its effects.

$$Dynamic\_Feat = \left(\sum_{i=1}^{x} \frac{number\_of\_Queries(i)}{current\_time - t(i)}\right) * W_d \tag{11}$$

The third parameter in Eq. (8) and Eq. (9) is the "*contribution level*" of each receiver of requested resources (i.e., requester node). This parameter is calculated as follows. For each row in the node's DN-Table if the owner of the table also owns the resource of this row, we consider the contribution level of the requester nodes in calculating the resource popularity parameter too. When a node downloads a file, it shares this replica with other nodes if it is not a free rider, so its contribution level increases. The more the contribution levels of nodes are, the more replicas of resources are available. Therefore, the contribution level parameter can influence the estimation of resources popularities.

As it has been shown that many nodes in P2P systems are free riders [15], we consider the "contribution level" parameter that takes into account how much nodes share their received resources. It has been shown that many nodes in P2P systems are free riders [15] that do not share their downloaded resources with other nodes in the system. For calculating the contribution level parameter, we first calculate the contribution level of each node at each interval by dividing the number of its uploads over the number of its downloads. This is done in the nominator part of Eq. (12). Then, the overall contribution level is calculated by summing up the results of dividing the number of uploads by the number of downloads of each node at different time intervals. If a node's contribution level is small, the probability that this node shares its resources with other nodes is low too. The number of each requester node's downloads and uploads are stored in the query message that is sent by that node. The event for this temporal *contribution_level* parameter is defined as follows. When a node receives a query and it has the requested resource, the event for the current interval set of related row's temporal *contribution_level* parameter is triggered. Therefore, the value of *contribution_level* parameter for the related row gets updated. Its new value is set by adding the contribution value of requester that is stored in the query message by the value stored in *value-list*'s current entry of the *contribution_level* parameter. Final weighted value of the temporal *contribution_level* parameter is calculated as in Eq. (12).

$$Contribution = \left(\sum_{i=1}^{x} \frac{contribution\_level(i)}{current\_time - t(i)}\right) \tag{12}$$

In the third expression of Eq. (8) or Eq. (9)—in subsection 3.2.1, we average the value of contribution level parameters of m nodes that have received the resource in order to calculate the final value of contribution level parameter for the related row. We use the weighted variable W_c to decrease the effect of this parameter as it has no direct effect on the calculation of resource popularities. It is worth noting that after calculating the values of the three mentioned parameters in Eq. (8) or Eq. (9), and before applying their weighted variables, the values of these parameters are mapped to the maximum value of these three parameters.

Equation (9) calculates the popularity of a requested resource, when the exact requested resource is not in the table, by selecting the k queries rr_j $(0 <= j <= k)$ most similar to the current query rr; this is done by using the cosine similarity function and multiplying the summation of all the mentioned parameters to the similarity level of each row rr_j that is denoted by the $Qsim(rrj, rr)^{\alpha}$ parameter. This parameter is calculated as presented in [50] and shows the similarity score between rr_j and rr. Parameter α $(0 < \alpha < 1)$ is set to show the importance degree of the most similar previous responded queries. This way, the effects of most similar rows are significant. Finally, by weighted averaging of the popularity values of these k similar rows, each of which is estimated via Eq. (9), the final popularity value of the requested resource is calculated. In all temporal parameters mentioned here, variable x denotes the number of time interval sets and each of these interval sets continues for t units of time.

After calculating the popularity of a requested resource by each query receiver node based on the resource popularity value, that node can estimate the number of required queries and the queries' TTL values that should be forwarded to the next step. Equation (13) calculates the number of required queries that should be forwarded to the next step using two parameters, namely *Required_Results* and *Temp_Popularity*; moreover, Eq. (14) calculates the number of remaining results that must be acquired. The *Required_Results* parameter represents the number of remaining required results and the

Temp_Popularity parameter denotes the popularity of the requested resource that is estimated by Eq. (8) or Eq. (9). In Eq. (13) we round the result to upper bound (ceil).

$$No\_of\_Queries = \left\lceil \frac{Required\_Results}{Temp\_Popularity * W_p} \right\rceil \tag{13}$$

$$Required\_Results = no\_needed\_Results - no\_Obtained\_Results \tag{14}$$

The two parameters in Eq. (13) help to reduce the number of queries to be forwarded at each step if the expected number of results is low or the popularity of the requested resource is high; the number of queries is increased, i.e., when the popularity level is low or the expected number of results is high. It is thus possible for Eq. (13) to give wrong estimations that is probably due to Eq. (9) that estimates the popularities of requested resources by using their similar responded queries residing in the DN-Tables.

To reduce the effect of wrong estimations by Eq. (13), we use the weighted variable W_p for the popularity parameter. Variable W_P is dynamically calculated at each step based on the acquired results via Eq. (15). When the lower number of results are expected, the value of this variable is increased, affecting more strongly on the popularity parameter in the calculation of Eq. (13). By adapting the variable W_p to different system conditions, we can avoid search performance drops. The initial value of W_p is 0.5. If the query, itself is available in the DN-Table, the popularity is calculated by Eq. (8) and W_p is set to 1 as there is no need for it.

$$W_p = 0.5 + \frac{no\_Obtained\_Results}{no\_Needed\_Results} \tag{15}$$

In Eq. (14) and Eq. (15), *no_Needed_Results* denotes the number of results that user expects to receive; this value is stored in the query message and is initialized by the query generator node. The *no_Obtained_Results* variable denotes the total number of local results received in each search path started from the query generator node towards candidate nodes driven by the query message; the value of this variable is also stored in the query message with a default value of zero. Every time a query is received by a node, the value of *no_Obtained_Results* in the query message is updated by summing up the value of this variable; this value is stored in the query message, with the number of results that the query receiver node has; the query with this new value is sent to other candidate nodes.

After forwarding the query in *n* hops, the query receiver node contacts the query originator node and gets informed about the updated value of *no_Obtained_Results*; *no_Obtained_Results* denotes the total number of received results from other search paths that have started from the requester node. In our simulations that are reported later in Section 4, we experimentally show that by setting the value of *n* to 4, the best results are achieved.

If the number of queries that should be forwarded is more than the degree of a query receiver node, we forward the maximum possible queries that are equal to the node's degree. We also define two thresholds for the number of forwarding queries at each step. These thresholds are defined as an upper threshold and a lower threshold to prevent huge overheads or low success-rate. In our simulations that are reported later in Section 4, we have set these thresholds to 10 and 2 respectively to produce the best results. If the *No_of_Queries* parameter produces a value bigger than the upper threshold, it is set to an upper threshold value; if its value is smaller than the lower threshold, it is set to a lower threshold value.

Equation (16) calculates the value of the query's TTL for the next hop. We have defined default value, namely 10, for the initial value of TTL for each query through simulations. At each step, using Eq. (16), a value is calculated based on the popularity of the requested resource, i.e., *Temp_Popularity* parameter, and the number of obtained results, i.e., *Obtained_Results* parameter, and then this value is subtracted from the current value of the query's TTL, i.e., *Current_TTL*. The two involved parameters in Eq. (16) help to reduce the TTL value of the forwarded query at each step from a large value to shorten the search path length if the number of obtained results is high or the popularity of the requested resource is high. When the inverse situation happens, the TTL value is decreased from a small estimated value. In other words, if the *Obtained_Results* parameter has a high value, and the *Temp_Popularity* parameter has a high value, i.e., the requested resource has high popularity, the *Current_TTL* variable is decreased from a value bigger than one. On the other hand, the *Current_TTL*'s value is decreased from a value lower than one if the first parameter' value is low, and the last parameter's value is low, i.e., requested resource has low popularity.

We consider two weighted variables for the *Obtained_Results* and *Temp_Popularity* parameters; due to significant effect of *Obtained_Results* parameter, the value of its weighted variable is bigger than the weighted variable of the *Temp_Popularity* parameter. In Eq. (16), the value of *Max_Temp_Popularity* is estimated by each node via table information. The value of the two parameters in Eq. (16) is always in [0,1], but as we want to apply their importance by considering the current value of TTL at each step, we map the summation of them to the *Current_TTL* value. This is done by multiplying the mentioned summation to *Current_TTL*. In addition, in order to terminate the search process in a logical manner and avoid system resource wastage, we choose a suitable lower threshold l to map the mentioned summation. Simulation results (reported in Section 4) show that setting l to 2 produces the best results.

$$Next\_TTL = Current\_TTL - \left[\frac{Obtained\_Results}{Required\_Results} * W_1 + \frac{Temp\_Popularity}{Max\_Temp\_Popularity} * W_2\right] * (Max(l, Current\_TTL)) \tag{16}$$

Search is stopped at any step when the value of the required results parameter estimated by Eq. (14) for the query receiver node gets equal or less than zero, or TTL is equal or less than zero in the next hop. By this flow, it is possible that searching for rare resources continues in an excessive number of hops because of the process of getting a zero value for TTL continuously for a long time. To limit the search in this case and prevent unnecessary traffic, we introduce a variable called *TTL_limit* that is put in the query message; so, if the query's number of passed hops exceeds the *TTL_limit* value, the search process is terminated. In simulations (reported later in Section 4) we found that setting the *TTL_limit* variable to 12 leads to the best performance.

3.2.2 Dynamic status-aware search mechanism

Any node that receives a query sends a query-hit message in the reverse direction of the search path to the query originator if it has the requested resource. In addition, every query receiver node defines the proper number of queries to be forwarded and the value of the query's TTL for the next hop as explained in Section 3.2.1, if stop conditions of search process are not held. Then with regard to nodes' heterogeneities, the query receiver node selects some candidate neighbor nodes with maximum abilities to respond to the current query. The number of selected candidate nodes is equal to the number of forwarding queries.

The number of neighbor nodes being forwarded the query is specified by the *dynamic estimation* mechanism. Neighbor nodes are selected based on information in DN-Table of the query receiver node as well as their dynamic scores and their overloading probability. First, we describe the calculation of the dynamic ranking process. When a node receives a query, it first uses the cosine similarity function to select the k most similar queries rr_{j} $(0 < j < k)$ to current query rr. These similar queries are stored in the DN-Table. Assuming each query rr_{j} is previously responded by a neighbor node P_i, we calculate P_i's dynamic score for the received query. The received response from P_i either is generated by the P_i itself or is routed through it. The dynamic score is calculated by Eq. (17) using *score* and *usefulness* factors as follows.

We calculate the first expression of Eq. (17) using Eq. (18) by multiplying the value of *temporal-Real-number-of-hits*, i.e.,$Real\_Temp\_Hit(P_i, rrj)$, to the similarity value of each query rr_{j} i.e.,. Next, we calculate the summation of different values of these multiplication results for all queries rr_{j} responded by P_i. Finally, we add the normalized result with the normalized summation of *usefulness* value in Eq. (16). In the mentioned equations, rr_{j} stands for the *j-th* query that is responded by the node P_i, and the value of is taken from [33]; it calculates the similarity score between rr_{j} and rr. Parameter α $(0 < \alpha < 1)$ is set to show the importance degree of the most similar previous responded queries. Also, the value of $Real\_Temp\_Hit(P_i, rrj)$ is derived from Eq. (10).

In Eq. (10), we use the *temporal-number-of-hits* concept to give different weights to the number of hits achieved at different times that were explained in Section 3.1.1. *Usefulness* that is used in the second expression of Eq. (17) is calculated by Eq. (19) similar to Eq. (3), which was explained in Section 3.1.2. There are, however, two differences. We calculate the summation in the second expression of Eq. (19) based on each node P_i and responded query rr_{j}. In addition, we weigh penalty values with the expression, based on the similarity score between rr and rr_{j} to reduce the enhanced effect of the penalty parameter, i.e., *Asc_penalty*; this is because of the fact that *Asc_penalty* is less important than the *score* parameter. The value of the *Asc_penalty* parameter in Eq. (19) is calculated via Eq. (20) that indicates the weighted temporal ascending penalty for node P_i and every query rr_{j}. By *Aβ_rrj(n)*, we mean the ascending penalty value for each query rr_{j}. The calculation in Eq. (20) is similar to the one described in Section 3.1.2.

$$DynamicScore(P_i, rr) = \left[\frac{Score(P_i, rr)}{Max\_Score} + \frac{Useful\quad ness}{Max\_Useful\quad ness}\right] \tag{17}$$

$$Score(P_i, rr) = \left(\sum\nolimits_{\forall rrj\ was\quad answred\ by\ P_i} Qsim(rrj, rr)^{\alpha} * Real\_Temp\_Hit(P_i, rrj)\right) \tag{18}$$

$$Useful\quad ness = \left(Initial\_Score - \sum\nolimits_{\forall rrj}(Asc\_penalty * [1 - Qsim(rrj, rr)^{\alpha}])\right) \tag{19}$$

$$Asc\_penalty = \sum\nolimits_{n=1}^{x}\left(\frac{A\beta\_rrj(n)}{current\_time - t_n}\right) \tag{20}$$

In the denominator part of the first and second expressions of Eq. (17), we put the maximum value of *score* and *usefulness* parameters among responded neighbor P_i of DN-Table, respectively. This is done to normalize these parameters as their values are from different ranges. These values are calculated via Eq. (21) and Eq. (22).

$$Max\_Score = \max_{\forall m \in Responded\_neighbor\,P} Score(P_m, rr) \quad (21)$$

$$Max\_Useful\quad ness = \max_{\forall m \in Responded\_neighbor\,P} Useful\quad ness(P_m) \quad (22)$$

Finally, the query receiver node sends the query to the nodes with the highest ranks given by Eq. (23), namely H_1 utility function that calculates the nodes' utilities. In Eq. (23), we try to select the nodes with the most responding ability based on two factors including dynamic score and the probability of candidate nodes not to become overloaded i.e., *Overloading_Prob*.

$$H_1 = W_1 * \frac{DynamicScore(P_i, rr)}{Max\_DynamicScore} + W_2 * \left(1 - \frac{Overloading\_Prob(P_i)}{Max\_Overloading\_Prob}\right) \quad (23)$$

In the second expression of Eq. (23), the *Overloading_Prob* variable estimates the probability of a node to be overloaded. When the workload of a node is more than its capacity, the response time of forwarded queries to this node will significantly increase; this node is considered as an overloaded node. In other words, if any node gets overloaded, it cannot handle any received query (i.e., it cannot process the query and response to download connections); so, in order to prevent sudden system performance drop, we should not send the query to overloaded nodes.

Therefore, we consider overloading states of responded neighbor nodes to select the neighbor nodes that can respond to the query. In this way, we want to avoid forwarding a query to an overloaded node. Calculation of the *Overloading_Prob* value is done using Eq. (24), and the estimation of node load is done in the numerator part of Eq. (24); in Eq. (24), the current load of node is defined based on two parameters; The first parameter is the first expression in the numerator part of Eq. (24), and it is shown by *downloading_Overhead* that indicates the overhead of downloading when other nodes download this node's files. The second parameter is covered by the second and third expressions in the numerator part of Eq. (24) i.e., *Overflow_Prob_Queues* and *Pop_Current_Query*; these paramteres indicate the imposed load of incoming message rate and the probability of node's queues (i.e., query queue and query-hit queue) being full, respectively. We divide the mentioned estimated load value in the numerator part of Eq. (24) by nodes' capacities to take into account heterogeneity in nodes' capacities; hence, we can select the node with the most responsiveness ability. In the dominator part of Eq. (24), nodes' capacities are considered that are estimated based on nodes' bandwidth, disk speed, and their processing power [50]. Each query receiver node calculates the *Overloading_Prob* parameter for each of its neighbor node P_i (i.e., the responder of each of the k most similar queries to current query) besides the dynamic score parameter as follows.

$$Overloading_{Prob(P_i)} = \frac{\left[\frac{downloading\_Overhead}{Max\_downloading\_Overhead} + \frac{Overflow\_Prob\_Queues}{Max\_Overflow\_Prob\_Queues} + \frac{Pop\_Current\_Query}{Max\_Pop\_Current\_Query}\right]}{Node\_Capacity(Pi)} \quad (24)$$

Because there are variant sized files in the system, their retrieval overheads are different. The first parameter of the numerator part of Eq. (24) that is itself calculated by Eq. (25) estimates the imposed load to each candidate neighbor node P_i (i.e., owner of files) when other nodes download its files. As is shown in Eq. (25), this parameter is estimated for each node P_i by multiplying the value of each file's size in P_i node i.e., *size_file*$_i$, to the temporal popularity value of each *file*$_j$ i.e., $Temp\_Popularity_{file_i}$. Next, we calculate the summation of different values of these multiplication results for all files owned by P_i. In Eq. (25), both files' sizes (i.e., retrieval overhead of file) and the number of neighbors' files can be found via pong messages. Moreover, the name of neighbors' files can be found by asking neighbors about the names of their files via messages. In addition, files' popularities are calculated via Eq. (9) or Eq. (10) as we previously explained in Section 3. In brief, if each of P_i's file name is between queries names in the DN-Table of query receiver node, the popularity of this file will be calculated by Eq. (9). If the file name is not in the DN-Table, the file popularity will be estimated using Eq. (10) by selecting k most similar query rows to this file name.

$$Downloading\_Overhead = \sum_{\forall file_i\, owned\, by\, P_i} \left[size\_file_i * Temp\_Popularity_{file_i}\right] \quad (25)$$

Generally, the number of forwarded queries to a node is related to the node's degree, search process frequency (i.e., the popularity of requested resources), and the rank of that node (i.e., the node's previous responses to the queries). In the numerator of Eq. (24), the second parameter namely *Overflow_Prob_Queues* shows the probability of any node's query and response message queues getting full that can lead to node become overloaded. In this case, later incoming messages are thrown out. When a node is overloaded, it drops the received query/query-hit messages so these queries are not spread between other nodes; moreover, the query-hit messages do not reach the destination. As a result, the success-rates of search processes, and consequently the whole system performance drops rapidly.

We calculated the *Overflow_Prob_Queues* parameter using Eq. (26) by estimating the average number of forwarded queries, and consequently a number of forwarded response messages to each candidate neighbor node P_i. This is done based on each resource's popularity level and the rank of each candidate neighbor node for that resource. By resource we mean the responded queries rr_g $(0<=g<=last\_row)$ in each query receiver node's DN-Table, because each query receiver node sends the received query based on query massage and information in the DN-Table. In addition, each node can get aware of its neighbors' degrees and capacities by asking them for this information.

$$Overflow\_Prob\_Queues = \left(\sum\nolimits_{\forall rrg} Temp\_Popularity_{rrg} * DynamicScore(P_i, rrg)\right) * degree_{Pi} \tag{26}$$

In Equation (26), the $Temp\_Popularity_{rr_g}$ parameter that is calculated by Equation (9) indicates the popularity level of each responded query rr_g in each row of the current query receiver node's DN-Table. In Equation (26), we show the rank of each candidate node P_i based on the *DynamisScore*(P_i, *rrg*) parameter for each responded query rr_g stored in the query receiver node's DN-Table. Concepts and calculations of *DynamisScore*(P_i, *rrg*) are similar to Equation (17). But here there are two differences; the first difference is that instead of using rr_j in Equations (18), (19) and (20) as a similar query to current query, we use rr_g when calculating *DynamisScore* in Equation (26) to show each responded query in rows of table answered by P_i. In addition, here the value of expression $Qsim(rrj, rr)^{\alpha}$ shown in Equation (18), is equal to one because we want to calculate the parameters for responded queries rr_g existing in the table.

Briefly, the rank of a node is high if the node has returned responses that are more recent and has lower penalty values. The more the rank and the degree of a node are, the more forwarded query and response messages to this node are; this leads to rapid overloading of that node. As mentioned earlier, the rank of a node is also increased if this node is placed near nodes with high rank and popular resources. In Equation (26), we multiply the summation of calculated parameters for each query row rr_g, and for node P_i to the candidate node P_i's degree i.e., the $degree_{P_i}$ parameter. This way, we involve the probability of forwarding messages from other neighbors of P_i to it.

The third parameter in the numerator of Equation (24) is calculated by Equation (27). In Equation (27), we estimate the popularity of currently requested resource and then the probability of forwarding query/query-hit messages for this currently requested resource from other neighbors of each candidate node P_i to it. Forwarding the query/query-hit messages for the currently requested resource has a direct effect on the related candidate node P_i to become overloaded and its queues to become fully populated.

$$Pop\_Current\_Query = Temp\_Popularity_{rr} * DynamicScore(P_i, rr) * degree_{Pi} \tag{27}$$

In Equation (27), the $Temp\_Popularity_{rr}$ parameter shows the popularity of current query *rr* that is calculated based on the information stored in the query receiver node's DN-Table using Equations (9) or (10) explained earlier. Concepts and calculation of the *DynamisScore*(P_i, *rr*) parameter in Equation (27) are similar to Equation (18) explained in subsection 3.2.2. However, here there is one difference that is the value of expression $Qsim(rrj, rr)^{\alpha}$ that is calculated in Equation (19), here is set to one only if the current query *rr* is in the table; this is because in this case, rr_j in Equations (18), (19) and (20) is itself the current query *rr*. If the current query *rr* is not in the table, we select the *k* most similar queries rr_j $(0<j<k)$ to current query *rr* from the query receiver node's DN-Table, and calculations are performed based on Equation (18). If the current received query *rr* exists in the DN-Table, we do not consider it in the calculations of Equation (26).

It is worth noting that before calculating the utility function H_1 for nodes, we first calculated $\left(1 - \frac{OverloadingProb(Pi)}{MaxOverloadingProb}\right)$ for each of the nodes that has responded to the *k* similar queries rr_j; if this result is less than the specified *Buffer_Threshold* value, the node is omitted from candidate nodes list. This threshold has been considered in order to avoid selecting candidate nodes that are going to be overloaded. In the next step, we select nodes with the maximum responsiveness abilities from the remainder nodes via Equation (23). In Equation (23), more weight is given to W_1 in comparison with W_2. That is because we aim to select the nodes that are more probable to respond to the current query based on their recent responses; in the meantime, we want to consider nodes' states in terms of their loads. Simulation results show that setting the *Buffer_Threshold* parameter to 0.103 and W_1 and W_2 to 0.713 and 0.287 respectively produce the best results. In addition, we dynamically change the value of W_1 and W_2 when their related parameter values get close to their maximum values to prevent system performance drop.

In order to consider high dynamicity of P2P systems and the possibility of new nodes entrances or resources insertion, each node periodically forwards each *t-th* received a query to another neighbor using a smart selection process in addition to the previous selection mechanism. This smart selection process chooses the best neighbor node based on two factors of neighbor nodes, namely their degrees and the number of their shared resources. This mentioned process also helps to deal with the partial coverage problem in the search path and prevents from reducing the range of query coverage. The *t* value is defined based on the nodes' entrance rate in the system.

4 Experimental results

To evaluate our proposed DPAS mechanism, we have simulated a P2P networked system based on Gnutella graph in a pure structure in Java programming language in Netbeans IDE—coming up with a discrete-event simulator. We emulated as well as mapped each entity with one class, and stored resources per node by using HashMap structure, the second item of which is a list of DynamicNeighborTable class as well. Moreover, in PeerNode class, for emulating node capacity in terms of both processing and storing messages from one aspect; we allocated two distinct ArrayLists for received queries and response messages.

The simulated network graph had 10,000 nodes with an average degree of 21. Nodes had different capacities, processing powers, degrees, bandwidth, uptimes, and the different number of resources. As stated in [32, 51], if a peer has a resource for query *q*, it has the resources for other similar queries with high probability. Therefore, we have simulated this designed network in a way that peers focus on specific kinds of resources. Each search is for 25 results. Resources were distributed with different replication rates among nodes. Resources had different popularities to influence the number of replicas and received queries for each resource. We modeled the query distribution and replication distribution by using a *zipfian distribution* with parameter $\alpha = 0.82$ to obtain comparable results with [3]. By query distribution, we mean the number of queries generated for each resource. It is clear that popular resources receive higher number of queries compared to rare resources. By replication distribution, we mean the pattern of distribution of replicas of each resource on nodes.

The 10% top ranked resources caused approximately 50% of the total number of stored resources and received approximately 50% of all queries. Considering the way we defined the default value of parameters, the most popular resources were stored in more than 10% of the nodes, whilst the least popular resources were stored only in 0.25% of the nodes.

The results we report in this section are derived from 100 simulation runs and each data point in the figures to follow represents the average of these runs. To cover dynamic situations, we changed the topology of the network graph more than 200 times at each run. These changes included nodes leaving and joining the network, as well as resources added or removed from the network. We got the best results when we set the value of *x* to 3 and the value of *t* to 4-time units. We have thus set the weight in Equation (11) to $W_d = 0.618$ and in Equation (8) and Equation (9) to $W_c = 0.382$.

Table 1 summarizes our simulation parameters accompanied by their default values.

We extensively ran and compared the performance of our proposed DPAS mechanism with those of ISM, IACO, and MBFS mechanisms. We have used the following metrics to compare the performance of these mechanisms:

1) Success Rate: This rate is the ratio of the number of successful searches to the total number of searches in the system.
2) Response Time: This metric is the interval between the time a user has sent the query and the time the result has been returned to that user.
3) Average Query Cost per Result: This metric is the ratio of the total number of generated *query* messages to the total number of returned results in the system.
4) Average Message Cost per Result: This metric is the ratio of the total number of generated messages (any kind of messages) to the total number of returned results in the system.

The first two metrics are quality of service parameters and the rest show the cost of the searching mechanism.

Table 1 Simulation parameters and their default values

Simulation Parameters	Default Values
Number of Nodes	10,000
P2P Model	Pure
Average Node Degree	21
no_Needed_Results	25
TTL	10
Replication Distribution	Zipf($\alpha = 0.82$)
Query Distribution	Zipf($\alpha = 0.82$)
x	3
t	4
n	4
W_d	0.618
W_c	0.3802
W_p	0.5
l	2
Buffer_Threshold	0.103
W_1	0.713
W_2	0.287
Initial_Score	100
Upper_Threshold	10
Lower_Threshold = l	2
TTL_limit	12

We have simulated DPAS, ISM, IACO, and MBFS mechanisms extensively in a dynamically changing system. Here we use the results of these simulations to compare the performance of these mechanisms with respect to the four metrics, namely, *Success Rate, Response Time, Average Query Cost per Result,* and *Average Message Cost per Result.*

4.1 Success rate analysis

Figure 4 shows a higher rate of success for DPAS compared to ISM, IACO, and MBFS especially when the number of queries rises more than 50. This is because DPAS considers nodes with different responsiveness abilities and candidate nodes states in terms of their loads; this trend prevents from sending a query to the overloaded nodes that cannot answer the query. The updating mechanism and temporal penalties it uses contribute to its robustness against system dynamicity while achieving the maximum success rate of 92.7%. Another effective parameter in achieving such a high success rate is the adaptation of the number of forwarded queries and queries' TTL values at each step with the popularity of requested resources and the number of returned results; this avoids excessive query forwarding in the system. Excessive query forwarding creates huge traffic overhead and unnecessarily overloads the nodes who are involved in the search process.

In the ISM mechanism, each query is forwarded to three neighbors that have previously responded to queries most similar to the current query. The low success rate of ISM is due to the lack of any updating mechanism that has made it only applicable to static systems. ISM forwards three queries with a fixed TTL value at each step without considering the popularities of resources and the number of returned results; hence, this brings about both high network traffic and system performance shrinking. This leads to system overloading, and thus overloaded nodes throw out their received query and response messages–system overall performance drops dramatically. As shown in Fig. 4, ISM and DPAS have nearly equal success rates at the beginning. This is because they both select nodes randomly in initial steps.

In the IACO mechanism [7]—relevant parameters derived as mentioned in this work [7], queries have masqueraded as ants in the search of food; the search process stops when either TTL is equal to zero or the ant finds the resource. Meanwhile, IACO modifies the conventional update function of ACO to lessen the impact of pheromone value for the sake of load balancing. Nevertheless, it has been outshone by DPAS, as it employs a few numbers of ants without concerning about popularity of the requested resource or the number of already-found resources;it has failed to learn from their previous searches intelligently as well as in a near-online fashion. In fact, it takes some time for ants to update their knowledge regarding updating the previous-seen search paths. What's more, DPAS rather than IACO uses similarity function to opt for the best candidate neighbors—which is of paramount importance in its superiority. More importantly, DPAS has outdone IACO, as the nature of DPAS is to well adapt to dynamicity of system conditions; take dynamically estimating values for both TTL and the number of queries at each search step, for instance. However, IACO has deployed a fixed number of ants for search process at each step, and due to its evolutionary nature, it has lagged behind DPAS in terms of performance.

In the MBFS mechanism, every node forwards the query to a randomly chosen subset of its neighbors, so no informed decision is made; as a result, the success rate of MBFS becomes lower than the other two mechanisms. The blind selection of neighbors by nodes to forward queries to them results in system congestion since MBFS produces excessive number of messages; hence, nodes' queues get fully populated soon and nodes become overloaded. Like ISM, MBFS uses a constant value for TTL and forwards the queries to half of the neighbors of a query receiver node at each step; this process is

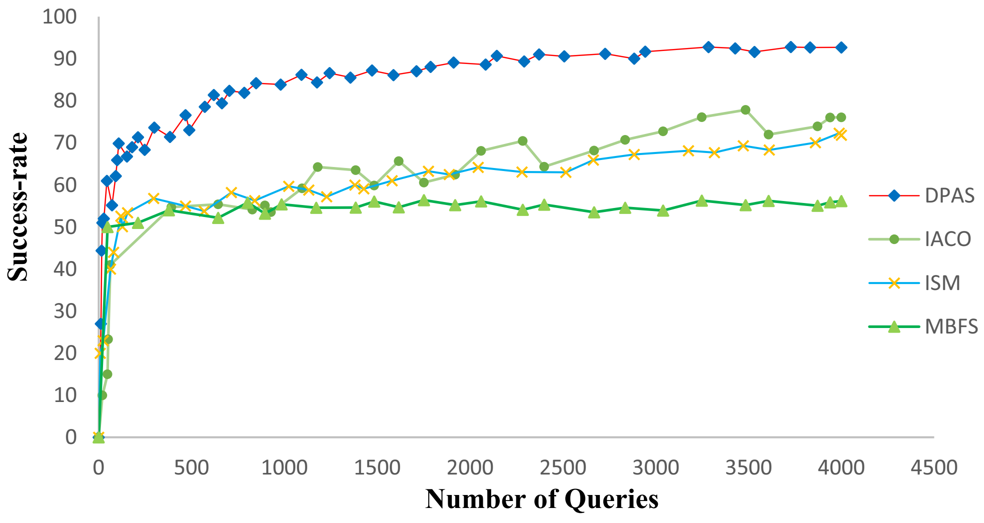


Fig. 4 Success rates of ISM, MBFS, IACO and DPAS mechanisms

done without considering either the popularities of resources or the number of returned results, leading to the low success-rate of MBFS.

4.2 Response time analysis

As Fig. 5 shows, DPAS has a lower (i.e., better) response time than the other two mechanisms. This success is due to the different effective mechanisms that are deployed in our mechanism and because of high success rate, which itself is due to the more appropriate selection of candidate nodes to which queries are forwarded. Our mechanism uses system feedbacks to adapt itself to variant system conditions in terms of nodes join/leave and insert/delete their resources. This predicts future events in the system via dynamic investigation of previous results, thereby forwarding queries to nodes more capable of responding. Response time is increased in ISM because this mechanism neither applies any updating mechanisms nor considers candidate nodes states. At first, ISM and DPAS work similarly.

IACO mechanism has higher response-time in comparison to DPAS since it neglects the popularity of requested resource; with this in mind, it uses more ants—leading to more traffic and overloaded nodes. Moreover, its updating mechanism lags far behind DPAS, as DPAS reaps the benefits of already-searched items swiftly by updating many contributing parameters in search selection process; DPAS tries to move with the times by introducing temporal parameter concept as well; however, IACO has failed to apply such considerations, and thus it lays extra overhead on the system.

MBFS suffers from the high amount of query forwarding, so if system state changes to a congestion state, response time increases. It is worth mentioning that in all simulations we consider latencies due to request processing and query forwarding.

4.3 *Average* query *Cost per Result*

It may be imagined that compared to ISM, updating flow in DPAS results in higher traffic volume. However, if we have a more general look, it can be seen that because of lower success-rate of ISM, repeated searches should start in it to compensate for the previous unsuccessful searches. This leads to a growing hike in both generated messages and bandwidth consumption.

DPAS mechanism adapts itself to requested resource popularity and dynamically specifies TTL and number of forwarding queries based on the mentioned popularity. The number of forwarding queries in ISM is a constant number at each step. Also it continuous searching until TTL = 0 is achieved; it is done even in cases where sufficient results have been already received. The two mentioned operations cause bandwidth consumption and an increment in message production.

Search cost is high in MBFS due to forward lots of queries blindly. In other words, MBFS involves many nodes while searching to improve system performance, but on the other hand, it causes bandwidth consumption and imposes a high amount of load on involved nodes. DPAS mechanism improves overall system performance by decreasing response time and bandwidth consumption and increasing success-rate.

Figure 6 shows the search cost as the average query cost per result. This cost in DPAS is lower than the three other mechanisms as DPAS has increased success-rate and forwards queries by considering resources popularities and variant system conditions. This cost is defined as the number of generating query messages per returned result. IACO creates more search cost in comparison with DPAS since it ignores any calculation regarding the number of forwarded queried and TTL value; moreover, it does not pay attention to select the best neighbors to steer search process scrupulously.

4.4 Average message cost per result

In Fig. 7, the average message cost per result is shown for the four mechanisms. This cost is defined as the number of generating messages (including query, query-hit, rank setting and updating messages) per returned result. DPAS search cost is lower than ISM, IACO, and MBFS which is due to its high

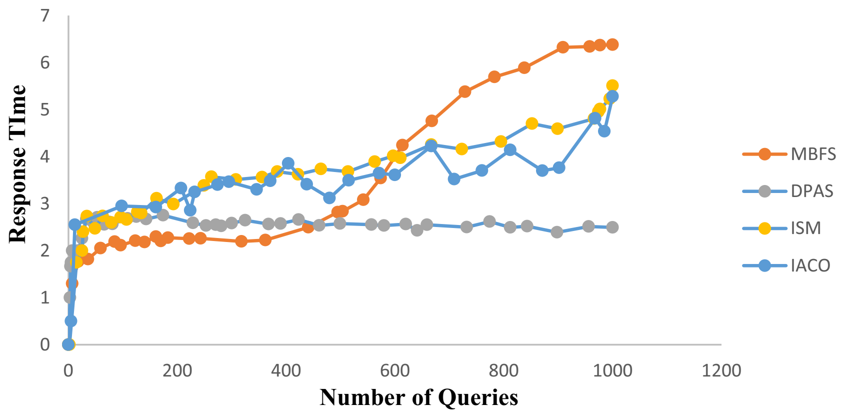


Fig. 5 Search response times of ISM, MBFS, IACO and DPAS mechanisms

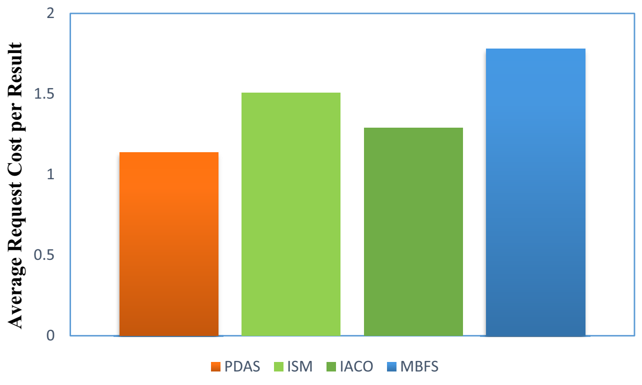


Fig. 6 Average request cost per results of ISM, MBFS, IACO and DPAS mechanisms

success-rate. We mean that because our mechanism does not forward the query to the irrelevant nodes; i.e., nodes without the requested resource, the overloaded nodes, or any knowledge about the location of the requested resource, the average aggregate cost is decreased. In addition, the overall communication and processing cost are reduced.

DPAS decreases the need for repeating searches and the number of generating messages by suitable estimation of more probable responder nodes and well adaptation to different system conditions. In so doing, the requested resource is found in an accurate and speedy manner. Additionally, DPAS terminates the search process when the expected numbers of results have been received, thus diminishing the search cost. ISM mechanism does not apply negative query feedbacks, so many query messages may be forwarded incorrectly to the nodes who have previously left the system, or they have deleted their requested resource. On the other hand, DPAS aims to reduce system traffic and make the system more scalable. MBFS mechanism forwards the query to lots of nodes and as a result, its search cost is high. IACO ends up with more generated traffic in comparison with DPAS, in that it does not consider any estimation with regard to the number of forwarded queried and TTL value. Moreover, due mainly to the lack of both informed decision making for electing the best candidate nodes and similarity of searched-keywords between queries, it produces more traffic and overhead.

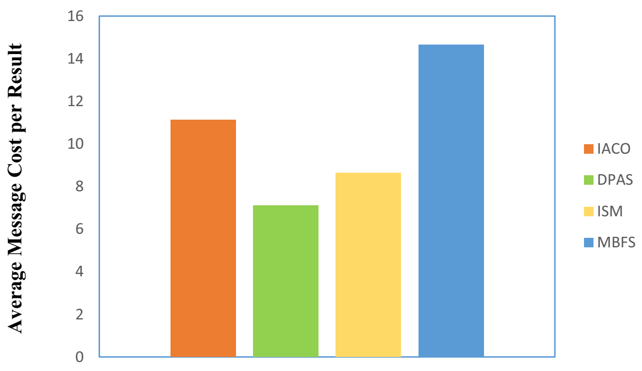


Fig. 7 Average message cost per results of ISM, MBFS, IACO and DPAS mechanisms

5 Future work

For the subsequent works, we intend to apply word2vec [52] and other NLP methods to prophesy the correlation between the different requested resource and their corresponding topics so much so that we can appraise the popularity of alike resources falling into an identical topic beforehand. In light of this, not only can we recommend a group of pertinent resources to the user—maybe request them in the future, but this approach also makes our strategy more efficient and futuristic in a case of ushering search process. Meanwhile, we intend to employ a similar-customized P2P-based RD mechanism in today's big data container-based resource management units, including YARN and Mesos; furthermore, we can entertain Cobb-Douglas utility function in decision-making process to boost both system performance and in turn precision of our formulas. Furthermore, we would like to shift our focus from similar types of resources, files, to manifold types of distributed resources. One improvement to the DPAS mechanism will be involving another factor, namely nodes' uptimes in selecting candidate nodes in Equation (23). Another improvement to our mechanism is the initialization of DN-Table for a newly arrived node. This can be done by sending queries to the neighbors of the newly arrived node and wanting them to send their relevant information after taking them from their neighbors. Moreover, our work can be extended and exploited in E-healthcare as well as IoT systems when it comes to search issue [53–55].

6 Conclusion

Efficient resource searching is one of the main debilitating challenges in unstructured peer-to-peer systems. In this paper, we proposed a new resource search mechanism called Dynamic Popularity-Aware (DPAS) that tried to select the best candidate nodes at each step of the search process while also adapting to system dynamic environments. To this end, DPAS introduced many effective parameters including a temporal number of hits, temporal penalties, temporal dynamic resources popularities, dynamic estimation of nodes states in terms of nodes' load and their dynamic ranking. These parameters were used to intelligently proceed the search process. DPAS adjusted the search scope at each hop by considering the estimated popularity of a requested resource and number of returned results. Simulation results demonstrated the superiority of our mechanism compared to other well-known existing mechanisms in terms of success-rate, response time and bandwidth consumption metrics. For future work, we plan to consider other factors such as a load of a node in selecting candidate nodes when forwarding the query. We also plan to deploy some user profiles to guide the search selection process under user guidelines.

References

1. Buford J, Yu H (2010) Peer-to-peer networking and applications: synopsis and research directions, Boston: Springer. FORMAT
2. Masood S, Shahid M, Sharif M (2018) Comparative analysis of peer to peer networks. International Journal of Advanced Networking and Applications (IJANA) 9(4):3477–3491
3. Shojafar M, Abawajy J, Delkhah Z, Ahmadi A (2015) An efficient and distributed file search in unstructured peer-to-peer networks. Peer-to-Peer Networking and Applications (PPNA) 8(1):120–136
4. Shamshirband S, Soleimani S (2018) LAAPS: an efficient file-based search in unstructured peer-to-peer networks using reinforcement algorithm. Int J Comput Appl:1–8
5. Schmidt C, Parashar M (2004) A peer-to-peer approach to web service discovery. World Wide Web (WWW) 7(2):211–229
6. Ed-daoui I, Hami AE, Itmi M, Hmina N (2018) Unstructured peer-to-peer systems: towards swift routing. International Journal of Engineering & Technology (IJET) 7(2.3):33–36
7. Asghari S, Navimipour N (2019) Resource discovery in the peer to peer networks using an inverted ant colony optimization algorithm. Peer-to-Peer Networking and Applications (PPNA) 12(1):129–142
8. Zarrin J, Aguiar R, Barraca J (2018) Resource discovery for distributed computing systems: a comprehensive survey. Journal of Parallel and Distributed Computing (JPDC) 113(1):127–166
9. Zhen-Wan Z, Peng K, Ren-Jie S (2015) A Survey of Resource Discovery in Mobile Peer-to-Peer Networks, in International Conference on Communication Systems and Network Technologies, Gwalior, India
10. Sharifkhani F, Pakravan M (2013) A review of new advances in resource discovery approaches in unstructured P2P networks, in International Conference on Advances in Computing, Communications and Informatics (ICACCI), Mysore, India
11. Arunachalam A, Sornil O (2015) An Analysis of the Overhead and Energy Consumption in Flooding, Random Walk and Gossip Based Resource Discovery Protocols in MP2P Networks. In International Conference on Advanced Computing & Communication Technologies (ACCT), Haryana, India
12. Lazaro D, Marques J, Jorba J (2013) Decentralized resource discovery mechanisms for distributed computing in peer-to-peer environments. ACM Computing Surveys (CSUR) 45(4):1–40
13. Meshkova E, Riihijärvi J, Petrova M, Mähönen P (2008) A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks. Comput Netw 52(11):2097–2128
14. Kapoor H, Mehta K, Puri D (2013) Survey of various search mechanisms in unstructured peer-to-peer networks. Int J Comput Appl (IJCA) 68(6):21–25
15. Thampi S (2010) Survey of search and replication schemes in unstructured p2p networks, arXiv preprint arXiv, vol. 2, no. 1
16. Ding G (2013) A control theoretic approach to analyzing peer-to-peer searching, in 8th International Workshop on Feedback Computing, San Jose
17. Navimipour N, Milani F (2015) A comprehensive study of the resource discovery techniques in peer-to-peer networks. Peer-to-Peer Networking and Applications (PPNA) 8(3):474–492
18. Palmieri F (2017) Bayesian resource discovery in infrastructure-less networks. Inf Sci 376:95–109
19. Gunopulos D, Zeinalipour-Yazti D (2002) A local search mechanism for peer-to-peer networks, in Proceedings of the eleventh international conference on Information and knowledge management, ACM, McLean, Virginia, USA
20. Fox G (2001) Peer-to-peer networks, Computing in Science & Engineering (CiSE) 3(3):75
21. Li Z (2017) A hybrid peer-to-peer framework for supply chain visibility, Purdue University, West Lafayette: Doctoral dissertation
22. Wu K, Wu C (2013) State-based search strategy in unstructured P2P. Futur Gener Comput Syst (FGCS) 29(1):381–386
23. Mirtaheri S, Sharifi M (2014) An efficient resource discovery framework for pure unstructured peer-to-peer systems. Comput Netw 59:213–226
24. Al-Aaridhi R, Dlikman I, Masinde N (2018) Search Algorithms for Distributed Data Structures in P2P Networks, in International Symposium on Networks, Computers and Communications (ISNCC), Rome, Italy
25. Gaeta R, Sereno M (2011) Generalized probabilistic flooding in unstructured peer-to-peer networks. IEEE Transactions on Parallel and Distributed Systems (TPDS) 22(12):2055–2062
26. Song S, Zeng X, Hu W, Chen Y (2010) Resource search in peer-to-peer network based on power law distribution. In Second International Conference on Networks Security, Wireless Communications and Trusted Computing (Nswctc), Wuhan, Hubei, China
27. Dorrigiv R, Lopez-Ortiz A (2007) Search algorithms for unstructured peer-to-peer networks, in 32nd IEEE Conference on Local Computer Networks (LCN 2007), Dublin, Ireland
28. Jamal A, Teahan W (2017) Alpha multipliers breadth-first search technique for resource discovery in unstructured peer-to-peer networks. Int J Adv Sci Eng Inf Technol 7(4):1403–1412
29. Margariti S, Dimakopoulos V (2015) On probabilistic flooding search over unstructured. Peer-to-Peer Networking and Applications (PPNA) 8(3):447–458
30. Bisnik N, Abouzeid A (2007) Optimizing random walk search algorithms in P2P networks. Comput Netw 51(6):1499–1514
31. Zhang H, Zhang L, Shan X (2007) Probabilistic search in p2p networks with high node degree variation, in IEEE International Conference on Communications, Glasgow, UK
32. Ogino N, Kitahara T (2017) An efficient content search method based on local link replacement in unstructured peer-to-peer networks. IEICE Trans Commun 101(3):740–749
33. Kalogeraki V, Gunopulos D (2002) A local search mechanism for peer-to-peer networks, in Proceedings of the eleventh international conference on Information and knowledge management. ACM, McLean, Virginia, USA
34. Bashmal L, Almulifi A, Kurdi H (2017) Hybrid resource discovery algorithms for unstructured peer-to-peer networks. Procedia Computer Science (PCS) 109:289–296
35. Navimipour N, Rahmani A, Navin A (2014) Resource discovery mechanisms in grid systems: a survey. J Netw Comput Appl 1(41): 389–410
36. Qu W, Zhou W, Kitsuregawa M (2010) Sharable file searching in unstructured Peer-to-peer systems, vol. 51, no. 2, pp. 149–166
37. Hidayanto A, Bressan S (2011) Adaptive routing algorithms in unstructured peer-to-peer(P 2 P) systems. Int J Comput Sci Eng 3(2):487–505

38. Wu L, Akavipat R, Menczer F (2005) 6S: Distributing Crawling and Searching Across Web Peers., in Web Technologies, Applications, and Services, Calgary, Canada
39. Zhu Y, Hu Y (2006) Enhancing search performance on Gnutella-like P2P systems. IEEE Transactions on Parallel and Distributed Systems (TPDS) 17(12). https://doi.org/10.1109/TPDS.2006.173
40. Guo Y, Liu L, Wu Y, Hardy J (2018) Interest-aware content discovery in peer-to-peer social networks. ACM Transactions on Internet Technology (TOIT) 18(3):39–60
41. Yang M, Yang Y (2009) An efficient hybrid peer-to-peer system for distributed data sharing. IEEE Trans Comput 59(9):1158–1171
42. Loo B, Huebsch R, Stoica I, Hellerstein J (2004) The case for a hybrid P2P search infrastructure, in *International workshop on Peer-To-Peer Systems*, Berlin
43. Zaharia M, Keshav S (2008) Gossip-based search selection in hybrid peer-to-peer networks. Concurrency and Computation: Practice and Experience (CCPE) 20(2):139–153
44. Šešum-Čavić V, Kuehn E, Zischka S (2018) Swarm-inspired routing algorithms for unstructured P2P networks. International Journal of Swarm Intelligence Research (IJSIR) 9(3):23–63
45. Šešum-Čavić V, Kühn E, Kanev D (2016) Bio-inspired search algorithms for unstructured P2P overlay networks. Swarm and Evolutionary Computation (SEC) 29(1):73–93
46. Guan Z, Cao Y, Hou X, Zhu D (2007) A novel efficient search algorithm in unstructured P2P networks," in *Second Workshop on Digital Media and its Application in Museum & Heritages (DMAMH 2007), IEEE*, Chongqing, China
47. Krynicki K, Jaén Martínez F (2014) Ant colony optimization for resource searching in dynamic peer-to-peer grids. International Journal of Bio-Inspired Computation (IJBIC) 3(6):153–165
48. Krynicki K, Jaen J, Mocholi J (2013) On the performance of ACO-based methods in p2p resource discovery. Appl Soft Comput 13(12):4813–4831
49. Khatibi E, Mirtaheri S, Khaneghah E, Sharifi M (2012) Dynamic multilevel feedback-based searching strategy in unstructured peer-to-peer systems, in IEEE International Conference on Green Computing and Communications, Besancon, France
50. Chawathe Y, Ratnasamy S, Breslau L (2003) "Making gnutella-like p2p systems scalable," in conference on Applications, technologies, architectures, and protocols for computer communications, Karlsruhe, Germany
51. Chen H, Jin H, Liu Y, Ni L (2008) Difficulty-aware hybrid search in peer-to-peer networks. IEEE Transactions on Parallel and Distributed Systems (TPDS) 20(1):71–82
52. Mikolov T, Chen K, Corrado G, Dean J (2013) Efficient estimation of word representations in vector space, in ICLR Workshop
53. Hamza R, Yan Z, Muhammad K, Bellavista P (2019) A privacy-preserving cryptosystem for IoT E-healthcare, *Information Sciences*
54. Hamza R, Muhammad K, Lv Z, Titouna F (2017) Secure video summarization framework for personalized wireless capsule endoscopy. Pervasive and Mobile Computing (PMC) 1(41):436–450
55. Riad K, Hamza R, Yan H (2019) Sensitive and energetic IoT access control for managing cloud electronic health records. IEEE Access (IEEEAccess) 7:86384–86393

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Ms. Elahe Khatibi She has received her M.Sc. degree in Computer Software Engineering from the School of Computer Engineering of Iran University of Science and Technology; her research interests are in the areas of resource discovery, and peer-to-peer systems.

Mohsen Sharifi is a Full-Professor of System Software Engineering in the School of Computer Engineering of Iran University of Science and Technology. He directs a distributed system software research group and laboratory. His main research interest is in the development of distributed systems, solutions, and applications, particularly for use in various fields of science. The development of a true distributed operating system is on top of his wish list. He received his B.Sc., M.Sc. and Ph.D. in Computer Science from the Victoria University of Manchester in the United Kingdom in 1982, 1986, and 1990, respectively.

Seyedeh Leili Mirtaheri is currently an assistant professor in the Electrical and Computer Engineering Department of Kharazmi University. Her research interests are in the areas of distributed and parallel systems, peer-to-peer computing, cluster computing, mathematics and scientific computing. She received her Ph.D. and M.Sc. in Computer Engineering (Software) from the School of Computer Engineering of Iran University of Science and Technology in 2013 and 2008, respectively. The research work reported in this paper is the result of her cooperation with other authors of the paper during her Ph.D. studies in Iran University of Science and Technology.