Enhancing the Performance of Self-Organizing Superpeer Network

L.N.S.INDUMATHI
M.Tech student(CSE),
GudlavalleruEngineeringCollege
Gudlavalleru, JNTUK.

G.VIJAYA DEEP
Asst. Professor, Dept of CSE
Gudlavalleru Engineering College
Gudlavalleru, JNTUK

Abstract

In peer to peer networks files are distributed among the peers in decentralized manner. Any node can join and leave the network at any time. In superpeer architectures file sharing and load balancing will be handled by the superpeer, which controls all other peers. Issues need to be addressed in these architectures are how client peers related to superpeers, how load is balanced, how superpeers locate files and how to handle node failures. The self-organizing superpeer networks works in decentralized manner and solves these issues. It maintains two caches, one is supeepeer cache located at weak peers and other is file cache located at superpeers. Search, load balancing and update protocols are addressed in this paper. Weak peer selects the superpeer which offers best search performance. Loads are balanced by calculating the effective loads of superpeers. A superpeer gets updated if its error value is larger than predefined value. In this way searching, load balancing and updating are optimized in SOSPNET. Finally it can quickly adjusts to changes, survives even in the case of node failures.

IJERT

1. Introduction

A significant amount of work has been done in the field of optimizing the performance and reliability of content sharing peer-to-peer networks. Among the proposed optimizations, the concept of leveraging the heterogeneity of peers by exploiting high-capacity nodes in the system design has proved to have great potential. The resulting architectures break the symmetry of pure P2P systems by assigning additional responsibilities to high-capacity nodes called superpeers. In a superpeer network, a superpeer acts as a server to client peers. Weak peers submit queries to their superpeers and receive results from them. Superpeers are connected to each other by an overlay network of their own, submitting and answering requests on behalf of the weak peers. Several protocols have been proposed to exploit super-peers. We add to this work the design of a superpeer network capable of optimizing the relationships between peers taking into account their content interests as deduced from their (possibly changing) behavior. We call our architecture the Self-Organizing Superpeer Network (SOSPNET) because the relationships between peers are discovered, maintained, and exploited automatically, with-out any need for user intervention or explicit mechanisms.

While some researchers have focused on exploiting static properties of shared data, also the possibility of utilizing patterns in dynamic peer behavior has attracted the attention of the research community. Such patterns in peer behavior have been reported by several measurement studies, which have revealed correlations between the search requests made by users of popular P2P systems. It was observed that the performance of locating content can be greatly im-proved by grouping peers interested in similar files and routing their search requests within these groups.

The semantic relationships between peers and files can be discovered relatively easily. The biggest challenge is, thus, to build an architecture that maintains and exploits the discovered semantic structure existing in all these semantic relationships. In this paper, we present the design and evaluation of a P2P architecture that combines the homogeneity of peer interests with the heterogeneity of peer capacities to solve the problem of efficient peer relationship management.

The design of our self-organizing superpeer network is guided by the following requirements: First, SOSPNET should be self-organizing in that it is able to discover and exploit the semantic structure present in the network, no matter what the initial topology is. Second, a new peer joining the network does not need to have any knowledge about the system; the longer a peer stays in the system, the more information it can collect and exploit for improving the performance of its searches. Third, the time it takes a new peer to achieve its optimal performance should be minimized.

SOSPNET uses two-level semantic caches deployed at both the superpeer and the weak peer level to maintain relationships between related peers and files. The cache maintained by a superpeer contains references to those files that were recently requested by its weak peers, while the cache of a weak peer stores references to those superpeers that satisfied most of its requests. We propose a novel mixed caching policy that combines the advantages of the traditional least frequently used (LFU) and least recently used (LRU) policies to improve the cache hit rates for less popular files. Furthermore, SOSPNET incorporates in its design a mechanism for balancing the load among super-peers. Load balancing is fully integrated with the content search algorithm and does not require any additional information exchange between superpeers nor a separate, external control component. The load balancing decisions are made independently by individual superpeers based on local information.

We also introduce a general performance model of a P2P system with semantic relations between peers and files based on two 8-month-long measurements of a large P2P network. From the model, we derive a bound on the search performance of a superpeer network using semantic caches.In a series of simulations, we show that the performance of SOSPNET is very close to the theoretical bound. In addition, we evaluate in our simulations the fault tolerance, the clustering properties, and the load balancing capabilities of SOSPNET. Finally, we compare SOSPNET with alternative architectures, assess its responsiveness to peer joins and leaves, and measure the time needed to find an optimal set of connections between peers, which all help in understanding how the system would perform in a real environment.

The rest of the paper is organized as follows: In Section 2, we specify the problem domain and scope of the presented system. Section 3 describes in detail the architecture of our self-organizing superpeer network. Section 4 summarizes the related work. The paper concludes in Section 5 by exploring some opportunities for future work.

IJERT

2. Organizing Peer Relationships

The vast majority of mechanisms for optimizing different performance aspects of P2P networks rely in one way or another on organizing the relations between peers. The relationships are organized by defining for each peer the set of other peers, called its neighbors, it interacts with.

In symmetric P2P networks such as Gnutella, any two peers are potential neighbors. In hybrid approaches such as Napster, all peers have a single neighbor—a central server that keeps information on all peers and responds to requests for that information. In superpeer networks such as Kazaa, and Chord superpeers, neighbors are selected from the set of high-capacity peers called superpeers; low-capacity peers—the client peers—cannot become neighbors.

In this paper, we aim at solving the problems of the existing superpeer networks related to the issue of establishing relationships between peers. Before presenting our approach, we identify the weak points of existing superpeer architec-tures. Each of the popular superpeer protocols proposed in the literature, including Kazaa, and Chord superpeers, makes at least one of the following three assumptions:

1. Every peer is assigned to a fixed, very small number (usually one) of superpeers. Consequently, super-peers become bottlenecks in erms of fault tolerance. Restoring the system structures such as routing tables back to a consistent state after a superpeer crash requires a considerable effort.
2. Peers are assigned to superpeers randomly and statically. The randomness of the assignment is explicit (as in Gnutella) or implicit (as in Chord, where the superpeer selection is based on peer identifiers, which are selected randomly). This static assignment does not adapt to changes in the network structure or in peer characteristics (e.g., content interests).
3. The peer-to-superpeer assignment has the so-called all-or-nothing property. When a peer connects to a superpeer, the latter takes responsibility for all the content stored at the peer. Such an assignment does not take into account the possible diversity of the peer's interests, and makes balancing the load among the superpeers difficult. In the rest of the paper, we show how to overcome all these limitations by introducing our self-organizing super-peer architecture SOSPNET.

3. The architecture of The self-organizing Superpeer Network

In this section, we present the SOSPNET system design. After a general overview of the SOSPNET architecture, we discuss in detail the employed data structures and protocols.

3.1 Architecture Overview

The basic idea behind the system architecture we propose is simple and intuitive. Weak peers with similar interests are connected to the same superpeers. As a consequence, super-peers get many requests for the same files. The request locality suggests the usage of caches that store the results of recent searches. But not only superpeers are responsible for discovering semantic structure in the network. We also allow weak peers to collect statistics about the content indexed by the superpeers. Having this information, weak peers can make local decisions about which superpeers to connect to.

In our architecture, superpeers store the information about the location of the content recently requested by their weak peers. Weak peers, on the other hand, sort the superpeers known to them according to the number of positive responses to their queries, and prefer to connect to superpeers that have satisfied most of their requests.

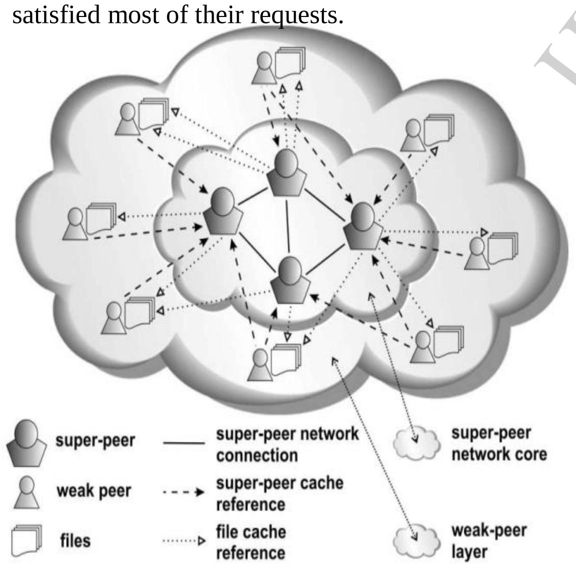

Fig. 1. The structure of SOSPNET.

To accelerate the process of grouping peers with similar interests under the same superpeers, we allow weak peers to exchange their lists of superpeers. More precisely, if a search succeeds, the requesting peer asks the peer that has the requested file for its list of top-ranked superpeers. This list is then merged with the list of superpeers known to the requesting peer. The intuition here is that if both peers were interested in the same file, then it is highly probable that they will share interest for more files in the future.

3.2 System Model

The information stored at a node in our system depends on the type of this node. Each weak peer maintains a superpeer cache, which contains the identities of superpeers (e.g., their IP addresses and port numbers). Each superpeer has a file cache of pointers to files stored at the weak peers.The relationships between SOSPNET peers are presented in Fig. 1.

All items in the superpeer and file caches are assigned priorities, which are nonnegative integer numbers. The priority determines the importance of a particular item, the higher the better. The initial priority assigned to a data item when it is added to the cache and the way the priority is modified upon a cache hit are determined by the caching policy. There are two situations when the priorities are taken into account. First, when the cache capacity is exceeded, the item with the lowest priority is removed. Second, the priorities are used for optimizing query routing. Details are presented in Section 3.4.

The last element of Fig. 1 that has not been mentioned until now is the network interconnecting the superpeers. We do not specify precisely which P2P protocol should be used here. We assume, however, that this protocol can efficiently deal with frequent changes of the information stored at the superpeers. Additionally, we require that the probability that a search succeeds is high when the requested informa-tion is present at least at one of the superpeers. Examples of protocols satisfying these criteria are Gnutella and epidemic-based approaches such as SCAMP.

The load balancing mechanisms of SOSPNET require introducing some specific terminology. We assume that each superpeer specifies its capacity as a value in the interval ð0; 1 ,with higher values assigned to more capable peers. We do not make any further assumptions about the superpeer capacities, which may either reflect static node properties (e.g., CPU speed) or change dynamically based on the current situation in the system (e.g., available bandwidth).

The particular method of computing the capacity values falls outside the scope of this paper. The current load of a superpeer is computed by counting the number of requests processed by the superpeer in a certain time frame called the request history window. The size of the request history

IJERT

window is the same for all superpeers, thus making the current-load values consistent across all superpeers in the system. However, the values of the current load of the superpeers cannot be compared directly, as different superpeers may have different capacities. Instead, we compute for each superpeer the effective load by dividing the current load by the capacity of the superpeer. A superpeer controls its load simply by dropping some of the search requests it receives.The accepted load is defined as the fraction of accepted search requests of those sent to the superpeer.

3.3 Two-Level Caching

The two-level caching architecture represented by superpeer and file caches allows us to separate caching policies that can be optimized for a peer role. In SOSPNET, the superpeer caches of the weak peers and the file caches of the superpeers are controlled according to different caching policies.

The priority of a superpeer in a superpeer cache isincreased by one after every positive feedback provided by this superpeer. This leads to the in-cache LFU [10] policy. The benefit of LFU is its inherent memory property—the priority of a superpeer is determined by the number of successful feedbacks it has provided in the past. The priority changes slowly, so one positive response from an unknown super-peer will not discredit a well-proven superpeer that satisfied many requests in the past, which would be the case if one used a memoryless policy such as LRU.

The caching policy employed for file caches should meet some specific requirements. First, similar to LRU, the file caches of the superpeers have to adapt fast to the changing needs of the weak peers. This is important particularly in the initial stage of the superpeer lifetime, when it is contacted by random peers. Second, like LFU, the file caching policy should keep track of long-term file popular-ity. Addressing the specific requirements of file caches, we propose a mixed caching policy that combines the desired properties of LRU and LFU. According to the mixed policy, if the file pointer is not yet present in the cache, then it is added to the cache with its priority one higher than the highest priority of all other cached items as in LRU. Otherwise, the priority corresponding to the file pointer is increased by one as in LFU.

3.4 Search Protocol

Peers use the information collected during past

```
peer_search(p:  peer, f : file_name):
    for s in p.S ordered according to decreasing priorities do
        q ← super-peer_local_search(s,f)
        if super-peer_local_search succeeded then
            t ← s
            break
    if f was not found until now then
        s ← super-peer in p.S selected randomly with probability proportional to its priority in p.S
        < q, t > ← super-peer _search (s,f)
        If super-peer_search did not succeed then
            return ERROR "File f not found"
    if p.S contains t then
        increase the priority of t in p.S
    else
        insert t into p.S
    merge_super-peer_caches(p, q)
    return q
super-peer_local_search(s: super-peer, f: file_name):
    if an entry < f, q > exists in cache s.F then
        increase the priority of < f, q > in s.F
        return q
    else
        return ERROR "File f not found"
super-peer_search ( s, f ):
    perform a search in the super-peer network to locate
    a super-peer t which has an entry  < f, q > in its cache
    if search succeeded then
        insert < f, q > into s.F
        return < q, t >
    else
        return ERROR "File f not found"
merge_super-peer_caches(p: peer, q: peer):
    for s in q.S do
        if p.S contains s then
            increase the priority of s in p.S
        else
            insert s into p.S
```

Fig. 2. Pseudocode of the superpeer search protocol in SOSPNET.

IJERT

searches to improve the performance of future requests. The contents of the superpeer and file caches are reorganized depend-ing on the feedback provided by peers involved in the search process.

The pseudocode of the search algorithm employed in our self-organizing superpeer network presented in Fig. 2 is divided into four subroutines. The superpeer cache of peer p is denoted by p:S, while the file cache of superpeer s is represented by s:F. The main search algorithm is the function peer_search. When a weak peer p looks for a file f, it first checks the file caches of the superpeers known to it (line 2). Note that p starts with the superpeers with the highest priorities. When the file is found (line 4), a pointer to superpeer s that knows the location of f is stored for future reference (line 5). However, if the file was not found with this method (line 7), the search request is forwarded to one of the superpeers in p's superpeer cache selected according to a random distribution biased toward superpeers with higher priority (line 8). This superpeer is further responsible for locating file f. If the search succeeds, a pair <q; t>, where q is a peer that has f and t is a superpeer that has a pointer <f; q> in its file cache, is returned to p (line 9). At this point, the self-(re)organization process begins. This process is performed in two stages. First, peer p increases the priority of the superpeer t that satisfied the search request (lines 12-15). As a consequence, in the future, p will direct more of its requests to t. Second, p integrates the list of superpeers kept by the weak peer q with its own superpeer cache (line 16). We exploit here a simple, yet powerful principle called interest-based locality [50], which postulates that if p and q are interested in the same file, it is very likely that more of their requests will overlap. It is thus beneficial for both p and q to use the same set of superpeers.

The algorithm of the superpeer_local_search is straight-forward. The search succeeds only if a pointer to file f is present in the file cache of superpeer s (line 19). Before returning the peer q that possesses file f (line 21), the priority of the corresponding cache item is increased (line 20).

The function superpeer_search performs the search in the superpeer network (line 25). Upon receipt of the search results, a pointer to the requested file f and to the peer q holding file f is added to the file cache of s (line 27). The return value of the function (line 28) contains not only the peer q, but also the superpeer t that has a pointer to f in its file cache.

The last function presented in Fig. 2, merge_super-peer_caches, takes two parameters representing two peers p and q. The superpeer cache of peer p is updated with the content of q's superpeer cache (lines 32 and 33). The functionality of merging the superpeer caches is not crucial for the system operation, but it accelerates the process of grouping weak peers under the same superpeers, which improves the search performance.

3.5 Insert Protocol

The file-insert protocol deployed by SOSPNET is very simple. Once in a while, each weak peer sends information on the files which it possesses to one of the superpeers in its superpeer cache. This superpeer is selected randomly with a probability proportional to its priority in the superpeer cache of the weak peer.

3.6 Balancing the Load among Superpeers

Load balancing is critical to the availability, accessibility, scalability, and throughput of a P2P system. Poor load balancing may gradually transform the superpeer network into a backbone network as was observed for Gnutella [13]. The idea here is to avoid overloading individual super-peers, which is the case when some superpeers are getting significantly more queries than others.

Before describing the load balancing mechanism of SOSPNET, we first define the requirements of load balancing for a superpeer network in general. A minimal requirement is to prevent situations in which the load imposed on a superpeer exceeds its capacities. A more advanced load balancing solution can further guarantee that the load assigned to each superpeer is proportional to its capacity.

Finally, the performance overhead and implementation burden incurred by adding the load balancing extensions should be low. In the remainder of this section, we show how the above goals can be easily achieved by exploiting the properties of the self-organizing superpeer network.

At first sight, the load balancing problem that we face in the SOSPNET design seems to be more difficult than in other superpeer networks because the SOSPNET superpeers do not explicitly know their weak peers. Furthermore, in the SOSPNET architecture, the assignment of weak peers to superpeers is not fixed. As a consequence, the superpeers cannot transfer weak peers between each other without the active cooperation of the weak peer layer. Being aware of these limitations, we have built into the search protocol a mechanism that indirectly influences the set of superpeers contacted by the weak peers by discouraging directing requests to overloaded superpeers.

The basic idea behind the load balancing mechanism of SOSPNET relies on the observation

IJERT

that a superpeer may control the number of received requests by affecting its priority in the superpeer caches of weak peers. An over-loaded superpeer can simply start dropping some of the requests, effectively decreasing its priority in the superpeer caches of the requesting peers. As the priority of a superpeer has a direct impact on the probability of that superpeer being selected as a request target, the load imposed on the overloaded superpeer will gradually decrease. Note that if a superpeer s refuses to service a request, then eventually, the client peer will ask another superpeer t to search for the file and to subsequently store a reference in its file cache. In other words, t will eventually take over some of the file references that were cached by s.

The requirement that the load experienced by a super-peer is proportional to its capacity involves relating the effective load of that superpeer to the effective loads of other superpeers in the system. To avoid introducing an independent load-information exchange protocol, we let superpeers gather load values of other nodes while performing searches.

The integration of the SOSPNET load balancing functionality with the search protocol is presented in Fig. 3. The function superpeer_local_search of Fig. 2 is extended with lines 18.1-18.4, which control the fraction of requests that are handled by superpeers. Only a fraction of s:accepted load randomly selected requests are accepted and processed as described in Section 3.4. The remaining requests are dropped, forcing the requesters to decrease the priority of s. If a request is accepted, its time stamp is saved in the request history window denoted by s:W (line 18.4). Request time stamps are used later for computing the current load of the superpeer.

The value of the accepted load of superpeer s is updated every time s discovers another superpeer t during the invocation of superpeer_search by taking into account the load of t in the update_accepted_load function. The values of the effective loads of s and t, denoted by s:effective load and t:effective load, respectively, are computed by dividing the numbers of requests in the request history windows of the two peers by their capacities . The imbalance between the loads of s and t is then quantified by computing the relative difference between the effective loads, which is then used to compute the value of the parameter new accepted load of s. Finally, the accepted load of s is updated by applying exponential smoothing. We use exponential smoothing instead of just replacing the accepted loads with the new values to avoid drastic changes in the accepted loads, giving the system time to adapt to the new settings.

IJERT

```
18   super-peer_local_search(s : super-peer, f : file_name):
18.1      r ← random value from range ( 0, 1 )
18.2      if r > s.accepted_load then
18.3           return ERROR “Super-peer s overloaded”
18.4      add request timestamp to request history window s.W
19        if an entry < f, q > exists in cache s.F then
               …
24   super-peer_search( s, f ):
               …
26        If search succeeded then
26.1           update _accepted_load( s, t )
27             insert  < f , q > into s.F
               …
37   update_accepted_load( s : super-peer, t: super-peer):
38        s.requests ← number of requests in window s.W
39        t.requests ← number of requests in window t.W
40        s.effective_load ← s.requests/s.capacity
41        t.effective_load ← t.requests/t.capacity
42        $\Delta$ ← (t.effective_load – s.effective_load)/
               (t.effective_load + s.effective_load)
43        new_accepted_load ← s.accepted_load + $\Delta$
44        if new_accepted_load > 1 then
45             new_accepted_load ← 1
46        if new_accepted_load < 0 then
47             new_accepted_load ← 0
48        s.accepted_load ← $\beta$ . S.accepted_load +
               (1- $\beta$ ).New_accepted_load
```

Fig. 3. Pseudocode of the superpeer load balancing protocol in SOSPNET.

In one specific case, the behavior of the load balancing algorithm can be confusing. Let’s assume that superpeer s is overloaded and that it has in its cache the pointer <f; q> to file f requested by p. The request will be forwarded to another superpeer, say t. Superpeer t will then perform a superpeer search, find s, store a pointer to f in its own cache, and return <q; s> to p. As a consequence, peer p will increase the priority of s in its superpeer cache. This behavior is counterintuitive as p should be discouraged to contact s in the near future. However, the increase of the priority of s should be interpreted as a one-time trade-off. If a different peer subsequently sends a request for file f to t, superpeer t will satisfy the request from its local file cache. Our load balancing algorithm has, thus, the highly desired property of replicating file pointers cached by the over-loaded superpeers at lighter loaded peers.

The load balancing scheme that we presented here is simple yet powerful and extremely flexible. While many state-of-the-art load balancing algorithms assume that all peers have equal capacities, our self-organizing architecture can deal with arbitrary capacity values and even allows these values to be changed during system operation. The load imbalance caused by a change of the parameters of the superpeer s is automatically taken into account, and the system gradually adapts to the new circumstances. Because neither the weak peers nor the file pointers have to be explicitly reassigned from one super-peer to another, no complex overlay infrastructure such as virtual servers or buckets of file identifiers needs to be introduced.

3.6 Update Superpeer

Once a superpeer S (previously S1) receives UpdateSuperpeer(S1, S2, rtS1, rtS2) from peer P , it saves the previous hop P ′ out of this message and proceeds with the following algorithm:

The arrival of a message increases the total error of the superpeer, proportional to the distance to the peer stimulating the network. If the error increment is greater than a predefined threshold moveThreshold, a super-peer movement is triggered: the superpeer ”moves” closer to the stimulus source. The peer movement is punished by another total error increase to discourage excess of peer migration. If the total superpeer error exceeds a predefined value createNewThreshold,thenCreateSuperpeerInTheMidle is sent to the neighbor with the maximum error value S_{max} to trigger the creation of a new superpeer.

```
1 newError ← $rt_{S}1 * $errorIncrementFactor
2 S.error ← S.error + newError
3 for all f ∈ S.f ingerList do
4       f.age ← f.age + 1
5       if f.age ≥ f ingerDeleteT hreshold then
6           S.f ingerList.remove(f )
7       end if
8 end for
9 if S2 ∈ S.f ingerList then
10      S.getF ingerT o(S2).age ← 0
11 Else
12      S.f ingerList ← S.f ingerList + S2
13 end if
14 if newError ≥ moveT hreshold then
15      S.send(M oveSuperpeer, P ′ )

16      S.error ← S.error + moveP unishment
17 end if
18 Smax ← fingerList.f indM axErrorN eighbor()
19 if S.error ≥ createN ewT hreshold then
20          S.send(CreateSuperpeerInT heM iddle, Smax)
21      S.error ← 0
22 end if
```

Fig 4: pseudocode for Updating Superpeers

IJERT

4. Related Work

The concept of leveraging the heterogeneity of peers by exploiting high-capacity nodes as superpeers has proved to have great potential and has resulted in implementations in popular P2P networks. KaZaa and Morpheus, which are both based on the FastTrack protocol, are widely used file sharing systems that make use of superpeers. Although FastTrack is a proprietary technology with no detailed documentation, it is known that FastTrack peers are automatically elected to become superpeers if they have sufficient bandwidth and processing power (users can disable this feature). A central bootstrapping server provides new peers with a list of one or more superpeers to which they can connect. Superpeers index the files shared by client peers connected to them and proxy search requests on behalf of these peers. All queries are therefore initially directed to the superpeers.

An extension of the basic Gnutella system has an architecture based on ultrapeers, which are conceptually equivalent to superpeers. Any new peer with enough bandwidth and CPU power immediately becomes an ultrapeer and establishes connections with other ultrapeers, forming an overlay network. A new ultrapeer is required to establish a predefined minimum number of connections to client peers.

The Edutella network proposes a superpeer architecture based on the concept of semantic clusters. It creates a logical layer on top of the base P2P network topology by grouping peers with similar content interests.The clustering is performed by matching the semantic information provided by the peers to clusters, with each cluster being maintained by a superpeer. In addition to controlling the internal structure of the cluster, superpeers are responsible for routing messages between peers from different clusters. In contrast to SOSPNET where peers are clustered based on patterns automatically discovered in their requests, in the Edutella network, the policies defining the peer clustering rules have to be defined manually by domain experts.

Superpeer architectures have also been proposed for structured P2P networks. Such architectures group nearby peers based on some criteria, such as network latency or adjacency in the key space, and organize the communication between groups using a superpeer layer. To find a peer that is responsible for a key, the top layer overlay network routes among the superpeers to determine the group responsible for the key, which, in turn, uses an intragroup overlay network to find the target peer. The lookup time in structured superpeer networks depends on the size of the state maintained by each superpeer and on the total number of superpeers. Some architectures are even able to guarantee a constant-time lookup. The rigid organization of content items based on their identifiers in structured P2P networks hampers optimization efforts that exploit the semantic properties of the content.

Exploiting the semantic properties of peers and content has also attracted a significant interest of the research community. Various approaches to capturing the semantic proximity between peers have been proposed. Some of them rely on a predefined ontology.

Unfortunately, defining ontologies is often a manual, time consuming process and the resulting semantic classes have to be continuously adjusted to reflect changes in semantic profiles of the content. Another approach is based on adding semantic shortcuts (i.e., extra links) between peers that share some interest. These links are created dynamically based on the set of most recent downloads, for instance. Such a mechanism is very reactive to evolving download patterns. Nevertheless, the nonintrusive nature of this approach does not allow to exploit available information such as the overlap between caches, which has also been used to approximate the semantic proximity between peers. A refined proximity measure takes into account not only the content of peers' caches but also their generosity and the popularity of shared files. None of the approaches discussed here combines a dynamic discovery of semantic proximity between peers and files with a multilevel P2P architecture as SOSPNET does.

IJERT

5. Conclusions

Self-organizing superpeer network architecture called SOSPNET, was introduced which has unstructured topology. At first peers will have random sets of neighbors and they can make local decisions regarding which superpeer to select based on the information collected during previous searches. Furthermore, we have demonstrated that a new peer can join at any time into the network and can very quickly finds a set superpeers which offers highest performance. Finally, SOSPNET is robust to node failures and can effectively balance the load between superpeers.

6. References

[1] http://gnutella.wego.com, 2007.
[2] http://www.fasttrack.nu, 2009.
[3] http://www.kazaa.com, 2009.
[4] http://www.napster.com, 2007.
[5] http://www.suprnova.org, 2009.
[6] http://www.thepiratebay.org, 2009.
[7] Y. Breitbart, R. Vingralek, and G. Weikum, "Load Control in Scalable Distributed File Structures,"Distributed and Parallel Databases, vol. 4, no. 4, pp. 319-354, Oct. 1996.
[8] V. Cholvi, P. Felber, and E. Biersack, "Efficient Search in Unstructured Peer-to-Peer Networks," Proc. Symp. Parallelism in Algorithms and Architectures (SPAA '04), June 2004.
[9] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," Lecture Notes in Computer Science, pp.46-66, Springer,2001.
[10] B. Cohen, "Incentives Build Robustness in Bittorrent," Proc. First Workshop Economics of Peer-to-Peer Systems, May 2003.
[11] F. Le Fessant, S. Handurukande, A.-M. Kermarrec, and L.Massoulie, "Clustering in Peer-to-Peer File Sharing Workloads," Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS '04), Feb. 2004.
[12] A.J. Ganesh, A.-M. Kermarrec, and L. Massouli, "Peer-to-Peer Membership Management for Gossip-Based Protocols," IEEE Trans. Computers, vol. 52, no. 2, pp. 139-149, Feb. 2003.
[13] T. Decker, R. Luling, and S. Tschoke, "A Distributed Load Balancing Algorithm for Heterogeneous Parallel Computing System,"Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications, Nov. 2000.