

# Optimizing Peer Relationships in a Super-Peer Network

Paweł Garbacki and Dick H.J. Epema  
Delft University of Technology  
{p.j.garbacki,d.h.j.epema}@tudelft.nl

Maarten van Steen  
Vrije Universiteit Amsterdam  
steen@cs.vu.nl

## Abstract

*Super-peer architectures exploit the heterogeneity of nodes in a P2P network by assigning additional responsibilities to higher-capacity nodes. In the design of a super-peer network for file sharing, several issues have to be addressed: how client peers are related to super-peers, how super-peers locate files, how the load is balanced among the super-peers, and how the system deals with node failures. In this paper we introduce a self-organizing super-peer network architecture (SOSPNET) that solves these issues in a fully decentralized manner. SOSPNET maintains a super-peer network topology that reflects the semantic similarity of peers sharing content interests. Super-peers maintain semantic caches of pointers to files which are requested by peers with similar interests. Client peers, on the other hand, dynamically select super-peers offering the best search performance. We show how this simple approach can be employed not only to optimize searching, but also to solve generally difficult problems encountered in P2P architectures such as load balancing and fault tolerance. We evaluate SOSPNET using a model of the semantic structure derived from the 8-month traces of two large file-sharing communities. The obtained results indicate that SOSPNET achieves close-to-optimal file search performance, quickly adjusts to changes in the environment (node joins and leaves), survives even catastrophic node failures, and efficiently distributes the system load taking into account peer capacities.*

## 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 (P2P) networks [11]. Among the proposed optimizations, the concept of leveraging the heterogeneity of peers by exploiting high-capacity nodes in the system design has proven to have great potential [19]. The resulting architectures break the symmetry of pure P2P systems by assigning additional responsibilities to high-capacity nodes called *super-peers*. In a super-peer network, a super-peer acts as a server to *client (ordinary, weak) peers*. Weak peers submit queries to their super-peers and

receive results from them. Super-peers 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 [12, 19]. We add to this work the design of a super-peer network capable of optimizing relationships between peers taking into account their content interests as deduced from their (possibly changing) behavior. We call our architecture the *Self-Organizing Super-Peer Network (SOSPNET)* because the relationships between peers are discovered, maintained, and exploited automatically, without any need for user intervention or explicit mechanisms.

While most of the interest of the research community is still focused on exploiting static properties of shared data, the first protocols utilizing patterns in dynamic peer behavior have recently been proposed [2, 18]. Such patterns in peer behavior have been reported by several measurement studies [8, 9], which have revealed correlations between the search requests made by users of popular P2P systems. It was observed that by grouping peers interested in similar files and routing their search requests within these groups, the performance of locating content can be greatly improved [9]. The semantic relationships between peers and files can be discovered relatively easily [2, 18]. The biggest challenge is, thus, to build an architecture that maintains and exploits the discovered semantic structure. 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 SOSPNET is guided by the following set of requirements. The self-organization property of SOSPNET requires that the system is able to discover and exploit the semantic structure present in the network no matter what the initial topology is. A new peer joining the network has no knowledge about the system and is connected to a set of randomly selected nodes. The longer a peer stays in the system, the more information it can collect and exploit for improving the performance of its searches. The time that it takes a new peer to achieve its optimal performance should be minimized.

SOSPNET uses two-level semantic caches deployed at both the super-peer and the weak-peer level to maintain relationships between related peers and files. The cache maintained by a super-peer contains references to those files which were recently requested by its weak peers, while the cache of a weak peer stores references to those super-peers that satisfied most of its requests. The initial design of a system deploying two-level semantic caching was explored by us in [7]. The current paper extends this work in several ways. First, we improve the algorithm for establishing relationships among peers by allowing weak peers to contact each other directly and exchange information on other peers in the network. Second, we present a mechanism for balancing the load between super-peers. Third, we introduce a model of a P2P system with semantic relations between peers and files based on the measurement data of a large P2P network, which we use to evaluate the search performance, 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 neighbors for each peer, which all helps to understand 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 SOSPNET. Section 4 introduces a model of P2P networks with semantic relationships between peers and files based on real-world traces. This model is further used in Section 5 to evaluate the performance of our architecture. The paper concludes in Section 6 by exploring some opportunities for future work.

## 2. Organizing peer relationships

The vast majority of mechanisms 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 the *neighbors*, it interacts with.

In symmetric P2P networks such as Gnutella [3], any two peers are potential neighbors. In hybrid approaches such as Napster [1], all peers have a single neighbor — a central server that keeps information on all peers and responds to requests for that information. In super-peer networks [19] such as Kazaa [8], Gnutella ultrapeers [15], and Chord super-peers [12], neighbors are selected from the set of high-capacity peers called super-peers; low-capacity peers — the client peers — cannot become neighbors.

In this paper we aim at solving the problems of the existing super-peer networks related to the issue of establishing relationships between peers. Before presenting our approach we identify the weak points of existing super-peer

architectures. Each of the popular super-peer protocols proposed in the literature, including Kazaa, Gnutella ultrapeers, and Chord super-peers, makes at least one of the following three assumptions:

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

In the rest of this paper we show how to overcome all these limitations by introducing our self-organizing super-peer architecture, SOSPNET.

## 3. Architecture of SOSPNET

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 proposed by us is simple and intuitive. Weak peers with similar interests are connected to the same super-peers. 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. We also allow weak peers to collect statistics about the content indexed by the super-peers. Having this information, weak peers can make local decisions about which super-peers to connect to.

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

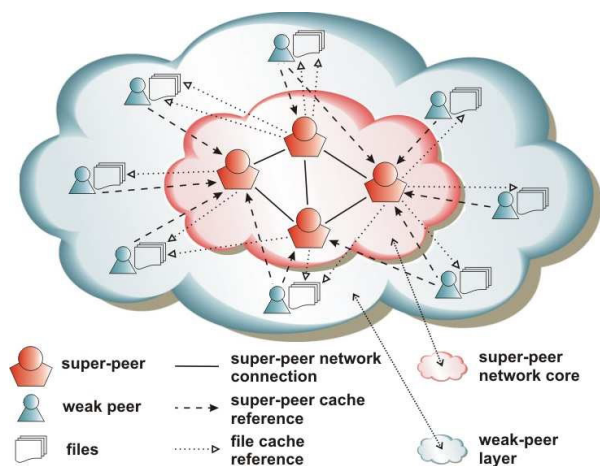


Figure 1. The structure of SOSPNET.

### 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 *super-peer cache* which contains the identities of super-peers (e.g., their IP addresses and port numbers). Each super-peer has a *file cache* of pointers to files stored at the peers. The relationships between SOSPNET peers are presented in Figure 1.

All items in the super-peer and file caches are assigned *priorities*, which are non-negative 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 3.3.

The super-peer and file caches are controlled according to different caching policies. The priority of a super-peer in a super-peer cache is increased by one after every positive feedback provided by this super-peer. This leads to the *in-cache least frequently used (LFU)* policy. The priority of a file pointer in a file cache is modified according to the *mixed* policy [7]. If the file pointer was already in the cache then the corresponding priority is increased by one. Otherwise, the file pointer is added to the cache with its priority one higher than the highest priority of all other cached items. The mixed caching policy has been proven to have many desirable properties, such as high cache hit ratios for less popular files. Additionally, the mixed policy guarantees that almost every file in the system is indexed by at least one super-peer. For details we refer the reader to [7].

The flexibility provided by the caches eliminates all three weak points of existing super-peer designs mentioned in Section 2. First, by manipulating the size of its super-peer

cache, a weak peer may decide to how many super-peers it is connected. The more connections maintained by the peer, the better is the resilience to crashes of multiple nodes.

Second, the problem of static peer-to-super-peer assignment is solved by the policy used for the super-peer caches. This policy prefers super-peers indexing content that is close to a user's interests. Possible changes in user interests or in the type of files cached at the super-peers result in restructuring the connections between peers.

Third, the all-or-nothing property is replaced with a property that we refer to as *partial responsibility*. The super-peers in our system index individually selected files rather than the entire sets of files stored at their weak peers. This type of architecture can deal with a situation in which a single weak peer has files of different semantic types. Pointers to these files can then be cached by different super-peers.

The last element of Figure 1 that has not been mentioned until now is the network interconnecting the super-peers. We do not specify precisely which of the P2P protocols should be used here. We assume however, that this protocol can efficiently deal with frequent changes of the information stored at the super-peers. Additionally, we require that the probability that a search succeeds is high when the requested information is present at least at one of the super-peers. Examples of protocols satisfying these criteria are Gnutella and SCAMP [6].

The load-balancing mechanisms of SOSPNET require introducing some specific terminology. We assume that each super-peer specifies its *capacity* as a value in the interval  $(0, 1]$ , with higher values assigned to more capable peers. We do not make any further assumptions about the super-peer 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 super-peer is computed by counting the number of requests processed by the super-peer in a certain time frame called here the *request history window*. The size of the request history window is the same for all super-peers, thus making the current-load values consistent across all super-peers in the system. However, the values of the current load of the super-peers cannot be compared directly, as different super-peers may have different capacities. Instead, we compute for each super-peer the *effective load* by dividing the current load by the capacity of the super-peer. A super-peer controls its load simply by dropping some of the search requests. The fraction of accepted search requests among those sent to the super-peer is defined as the *accepted load*.

### 3.3. Search protocol

Peers use the information collected during past searches to improve the performance of future requests. The contents

of the super-peer and file caches are reorganized depending on the feedback provided by peers involved in the search process.

```

1 peer_search(p : peer, f : file_name):
2   for s in p.S ordered according to decreasing priorities do
3     q ← super-peer_local_search(s,f)
4     if super-peer_local_search succeeded then
5       t ← s
6       break
7   if f was not found until now then
8     s ← super-peer in p.S selected randomly with probability
      proportional to its priority in p.S
9     < q, t > ← super-peer_search(s,f)
10    if super-peer_search did not succeed then
11      return ERROR "File f not found"
12    if p.S contains t then
13      increase the priority of t in p.S
14    else
15      insert t into p.S
16    merge_super-peer_caches(p, q):
17    return q
18 super-peer_local_search(s : super-peer, f : file_name):
19   if an entry < f, q > exists in cache s.F then
20     increase the priority of < f, q > in s.F
21     return q
22   else
23     return ERROR "File f not found"
24 super-peer_search(s, f):
25   perform a search in the super-peer network to locate a super-peer
      t which has an entry < f, q > in its cache
26   if search succeeded then
27     insert < f, q > into s.F
28     return < q, t >
29   else
30     return ERROR "File f not found"
31 merge_super-peer_caches(p : peer, q : peer):
32   for s in q.S do
33     if p.S contains s then
34       increase priority of s in p.S
35     else
36       insert s into p.S

```

Figure 2: Pseudo-code of the search protocol in SOSPNET.

The pseudo-code of the search algorithm employed in SOSPNET presented in Figure 2 is divided into four sub-routines. The super-peer cache of peer  $p$  is denoted by  $p.S$ , while the file cache of super-peer  $s$  is represented as  $s.F$ .

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 super-peers known to it (line 2). Note that  $p$  starts with the super-peers with the highest priorities. When the file is found (line 4), a pointer to super-peer  $s$  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 super-peers in  $p$ 's super-peer cache selected according to a random distribution biased towards super-peers with higher priority (line 8). This super-peer is further responsible for locating file  $f$ . If the search succeeds, a pair  $\langle q, t \rangle$ , where  $q$  is a peer that

has  $f$  and  $t$  is a super-peer that has a pointer  $\langle f, q \rangle$  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 super-peer  $t$  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 super-peers kept by the weak peer  $q$  with its own super-peer cache (line 16). We exploit here a simple, yet powerful principle called *interest-based locality* [16], 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 super-peers.

The algorithm of the **super-peer\_local\_search** is straightforward. The search succeeds only if a pointer to file  $f$  is present in the file cache of super-peer  $s$  (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 **super-peer\_search** performs the search in the super-peer network (line 25). Upon receipt of the search results, a pointer to the requested file  $f$  and to the peer  $q$  holding file  $f$  are 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 super-peer  $t$  that has a pointer to  $f$  in its file cache.

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

### 3.4. Insert protocol

The 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 super-peers in its super-peer cache. This super-peer is selected randomly with a probability proportional to its priority in the super-peer cache.

### 3.5. Balancing the load among super-peers

Before describing the load-balancing mechanism of SOSPNET, we first define the requirements of load balancing for a super-peer network in general. A minimal requirement is to prevent situations in which the load imposed on some super-peers exceeds their capacity limits. A more advanced load-balancing solution can further guarantee that the load assigned to each super-peer is proportional to its capacity. Finally, the performance overhead and implementation burden incurred by adding the load-balancing exten-

sions should be low. In the remainder of this section we show how the above goals can be easily achieved by exploiting the properties of SOSPNET.

At first sight the load-balancing problem that we face in the SOSPNET design seems to be more difficult than in other super-peer networks because the SOSPNET super-peers do not explicitly know their weak peers. Furthermore, in the SOSPNET architecture, the assignment of weak peers to super-peers is not fixed. As a consequence, the super-peers cannot transfer the weak peers between each other without the active cooperation of the weak-peer layer.

The basic idea behind the load-balancing mechanism of SOSPNET relies on the observation that a super-peer may control the number of received requests by affecting its priority in the super-peer caches of weak peers. An overloaded super-peer can simply start dropping some of the requests, effectively decreasing its priority in the super-peer caches of the requesting peers. As the priority of a super-peer has a direct impact on the probability of that super-peer being selected as a request target, the load exercised on the overloaded super-peer will gradually decrease. Note that if a super-peer  $s$  refuses to service a request then eventually the client peer will ask another super-peer  $t$  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 super-peer to the loads of other super-peers in the system. To avoid introducing an independent load-information exchange protocol, we let super-peers gather load values of other nodes while performing the search.

The integration of the SOSPNET load-balancing functionality with the search protocol is presented in Figure 3. The function **super-peer\_local\_search** of Figure 2 is extended with lines 18.1 to 18.4, which control the fraction of requests that are handled by super-peer  $s$ . Only a fraction of  $s$ .*accepted\_load* randomly selected requests are accepted and processed as described in Section 3.3. The remaining requests are dropped, forcing the requesters to decrease the priority of  $s$ . If a request is accepted, its timestamp is saved in the request history window denoted by  $s.W$  (line 18.4). Request timestamps are used later for computing the current load of the super-peer.

The value of the accepted load of super-peer  $s$  is updated every time  $s$  discovers another super-peer  $t$  during the invocation of **super-peer\_search** (line 26.1) by taking into account the load of  $t$  in the **update\_accepted\_load** function. The values of effective loads of  $s$  and  $t$ , denoted by  $s$ .*effective\_load* and  $t$ .*effective\_load*, respectively, are computed by dividing the number of requests in the request history windows of the two peers by their capacities (lines 38

```

18 super-peer_local_search( $s$  : super-peer,  $f$  : file_name):
18.1  $r \leftarrow$  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  $\langle f, q \rangle$  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  $\langle f, q \rangle$  into  $s.F$ 
  ...
37 update_accepted_load( $s$  : super-peer,  $t$  : super-peer):
38  $s$ .requests  $\leftarrow$  number of requests in window  $s.W$ 
39  $t$ .requests  $\leftarrow$  number of requests in window  $t.W$ 
40  $s$ .effective_load  $\leftarrow s$ .requests/ $s$ .capacity
41  $t$ .effective_load  $\leftarrow t$ .requests/ $t$ .capacity
42  $\Delta \leftarrow (t$ .effective_load  $- s$ .effective_load)/( $t$ .effective_load
  +  $s$ .effective_load)
43  $new\_accepted\_load \leftarrow s$ .accepted_load  $+$   $\Delta$ 
44 if  $new\_accepted\_load > 1$  then
45    $new\_accepted\_load \leftarrow 1$ 
46 if  $new\_accepted\_load < 0$  then
47    $new\_accepted\_load \leftarrow 0$ 
48  $s$ .accepted_load  $\leftarrow$ 
    $\beta \cdot s$ .accepted_load  $+$   $(1 - \beta) \cdot new\_accepted\_load$ 

```

Figure 3: Pseudo-code of the super-peer load-balancing protocol in SOSPNET.

to 41). The imbalance between loads of  $s$  and  $t$  is then quantified by computing the relative difference  $\Delta$  between the effective loads (line 42), which is then used to compute the value of the parameter  $new\_accepted\_load$  of  $s$  (lines 43 to 47). Finally, the accepted load of  $s$  is updated by applying exponential smoothing with weighting factor  $\beta \in (0, 1)$  to the current value of the accepted load and  $new\_accepted\_load$  (line 48). 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.

In one specific case the behavior of the load-balancing algorithm can be confusing. Let's assume that super-peer  $s$  is overloaded and that it has in its cache the pointer  $\langle f, q \rangle$  to file  $f$  requested by  $p$ . The request will be forwarded to another super-peer, say  $t$ . Super-peer  $t$  will then perform a super-peer search, find  $s$ , store pointer to  $f$  in its own cache, but return  $\langle q, s \rangle$  to  $p$ . As a consequence, peer  $p$  will increase the priority of  $s$  in its super-peer 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 tradeoff. If a different peer sends subsequently a request for file  $f$  to  $t$ , super-peer  $t$  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 overloaded super-peers 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 super-peers is automatically taken into account, and the system gradually adapts to the new circumstances.

## 4. Performance model

This section presents the data models that we use in evaluating the performance of SOSNET.

### 4.1. Notation

It has been observed [9] that user content interests as well as file popularities in file sharing P2P networks are not independent. The similarities in user request patterns can be modeled by assigning semantic types to both files and peers.

We use the symbols  $D$ ,  $U$ , and  $N$  to denote the total number of files (data items), of peers (users), and of semantic types in the system. The number of files and peers of (semantic) type  $n \in \{1, \dots, N\}$  are denoted by  $d_n$  and  $u_n$ , respectively. Files of type  $n$  are numbered sequentially from 1 to  $d_n$ . Each peer periodically generates a request for a file, which is selected according to a distribution that depends on the peer's type only. We denote by  $p(m)$  the overall probability that a random peer requests a file of type  $m$ , and by  $p(m, k)$  the overall probability that a random peer asks for the  $k$ -th file of type  $m$ , for  $k = 1, \dots, d_m$ . Note that the distribution of  $p(m)$  can be computed from the values  $p(m, k)$  in the following way:

$$p(m) = \sum_{k=1}^{d_m} p(m, k). \quad (1)$$

Further, we use the symbol  $p_n(m, k)$  to denote the probability that a peer of type  $n$  requests the  $k$ -th file of type  $m$ . We will define this probability in Section 4.2.

### 4.2. Models of the semantic structure

In our experiments we use four datasets to model file popularities. The first two of them are based on real-world traces, while the second two are created synthetically. There are two reasons for using both models. First, a broader spectrum of the simulation data increases the credibility of our results. Second, we use the opportunity to assess the usefulness of synthetic datasets in the evaluation of system designs based on the semantic paradigm.

#### 4.2.1. Model based on real traces

Before presenting the method of computing the distributions  $p(m)$  and  $p(m, k)$  from the actual data traces, we describe how we have obtained these traces. For a total

period of eight months we have collected the download statistics provided by the `suprnova.org` (Feb – April 2004) and `piratebay.org` (Nov 2006 – May 2006) websites, which at the time of gathering the data were the most popular [14] websites used for searching files in the BitTorrent [5] network. BitTorrent is currently the largest P2P network with over one third of the world's P2P traffic [13]. Each file registered at `suprnova.org` or `piratebay.org` is categorized by human volunteers called moderators. The number of categories defined by moderators equals 198 and 40 for `suprnova.org` and `piratebay.org`, respectively. We treat the categories defined by the moderators as the semantic types.

For each of the 24,081 files registered at `suprnova.org` and 164,821 files available at `piratebay.org` we were able to obtain the number of peers downloading this file. The fraction of downloaders for a file can be interpreted as the file popularity  $p(m, k)$ . In order to reduce the influence of temporal interest localities such as flashcrowds [14] on the value of  $p(m, k)$ , we compute for each file the average number of downloads observed during the whole measuring period. The average is obtained by dividing the total number of downloads of the file by the duration of the period in which the file was accessible for download.

Although collecting the access patterns for a particular file is possible, obtaining complete statistics about the content downloaded by a specific peer is infeasible. First, many users are behind NAT boxes which prevents us from discovering their peer IP addresses. Second, we cannot guarantee that a user is not using other websites to look for the files he is interested in. Consequently, the behavior of users needs to be modeled synthetically, taking, however, file popularities into account.

We propose the following formulas for  $u_n$  and  $p_n(m, k)$ :

$$u_n = p(n) \cdot U \quad (2)$$

$$p_n(m, k) = \begin{cases} (1 - \alpha) \cdot p(m, k), & m \neq n, \\ \left[ (1 - \alpha) + \frac{\alpha}{p(n)} \right] \cdot p(n, k), & m = n. \end{cases} \quad (3)$$

Eq. (2) says that the number of users of a certain semantic type is proportional to the popularity of this type. The parameter  $\alpha \in [0, 1]$  in Eq. (3) characterizes how strong the interest of users is for files of their own type. When  $\alpha$  equals 0, peers of all types behave indifferently, while at the other extreme with  $\alpha$  equal to 1, peers of type  $n$  request only files of type  $n$ .

Clearly, the values  $p_n(m, k)$  define valid probability distributions as it can be easily shown that the sum  $\sum_{m,k} p_n(m, k)$  equals 1 for every  $n \in \{1, \dots, N\}$ . A very important property of our model of the semantic structure is that the frequency of queries to files generated by all the peers in the system follows the distribution  $p(m, k)$ . It can

be shown that this property is preserved by the distribution  $p(m, k)$ .

#### 4.2.2. Synthetic model

The synthetic model of the semantic structure which we use in our experiments was previously introduced in [7]. This model assumes that the numbers of files of each semantic type are the same, and that the distribution of the file popularity within one type, the file popularities without type partitioning, and the numbers of peers of each type follow Zipf's law. We note that most related studies have assumed a Zipf distribution (e.g., [4]), with the notable exception of an evaluation of Kazaa [8] that tends to indicate that content popularity follows a different type of distribution.

The numbers of files and peers of each semantic type, and the request characteristics in the synthetic model are given by the following formulas:

$$u_n = \frac{U}{n \cdot H_N} \quad (4)$$

$$d_n = \frac{D}{n \cdot H_N} \quad (5)$$

$$p(m) = \frac{1}{m \cdot H_N} \quad (6)$$

$$p(m, k) = \frac{1}{m \cdot H_N} \cdot \frac{1}{k \cdot H_{d_m}} \quad (7)$$

$$p_n(m, k) = \begin{cases} \frac{1}{k \cdot H_{d_m}} \frac{1-\alpha}{m} \cdot \frac{1}{Z}, & m \neq n, \\ \frac{1}{k \cdot H_{d_n}} \left( \alpha + \frac{1-\alpha}{n} \right) \cdot \frac{1}{Z}, & m = n, \end{cases} \quad (8)$$

where  $H_i = \sum_{j=1}^i 1/j$  is the  $i$ -th harmonic number, and  $Z$  is a normalizing constant chosen so that  $\sum_{m,k} p_n(m, k)$  equals 1 for  $n = 1, \dots, N$ . It can be shown that  $Z$  equals  $(1 - \alpha) \cdot H_N + \alpha$ , independent of  $n$ .

For the sake of comparison with the trace-based datasets, we generate two synthetic datasets which we shall further call `suprnova_syn` and `piratebay_syn`. The numbers of files and semantic types in the synthetic datasets are the same as in the corresponding trace-based datasets.

#### 4.3. Optimal caching performance

Having the formal description of the model of the semantic structure, we can compute the optimal performance of the caches deployed in SOSPNET. More precisely, the caching performance is defined as the probability that a random file request can be satisfied by one of the super-peers known directly to (stored in the super-peer cache of) the requesting peer. The upper bound for the caching performance can be computed analytically. We call this bound the *optimal caching performance*.

In an ideal situation, each weak peer  $p$  has its own "private" set of super-peers that cache pointers to files which are most likely to be requested by  $p$ . Let's assume for simplicity that the sizes of all super-peer caches in the system

are equal to  $\sigma$ , and the sizes of all file caches are equal to  $\phi$ . Consequently, the super-peers of  $p$  can index in total at most  $\sigma \cdot \phi$  unique files. Now we only have to find the set of  $\sigma \cdot \phi$  files which are most likely to be requested by  $p$ . According to Eq. (3), the probability that the  $k$ -th file of type  $m$  is requested by  $p$  is  $p_n(m, k)$ , where  $n$  is the semantic type of  $p$ . We sort all values  $p_n(m, k)$  in a descending order:  $p_n(m_1, k_1) \geq p_n(m_2, k_2) \geq \dots \geq p_n(m_D, k_D)$ ,  $(m_i, k_i) \neq (m_j, k_j)$  for  $i \neq j$ . The probability that a file requested by  $p$  is satisfied by one of its super-peers, denoted by  $ocp(n)$  (optimal caching performance of a peer of type  $n$ ) can be expressed as

$$ocp(n) = \frac{\sum_{i=1}^{\sigma \cdot \phi} p_n(m_i, k_i)}{\sum_{i=1}^D p_n(m_i, k_i)} = \sum_{i=1}^{\sigma \cdot \phi} p_n(m_i, k_i). \quad (9)$$

The optimal caching performance of the whole system,  $ocp$ , is the weighted average of the values  $ocp(n)$ , where the weights represent the numbers of peers  $u_n$  of type  $n$ :

$$ocp = \sum_n \frac{u_n}{U} \cdot ocp(n) = \sum_n p(n) \cdot ocp(n). \quad (10)$$

### 5. Performance evaluation

We have built a discrete-time simulator to evaluate SOSPNET. In the simulations we use the model of the semantic structure introduced in Section 4.

#### 5.1. Experimental setup

The simulated system consists of 100,000 peers and 1,000 super-peers. The selection of the number of peers relative to the number of super-peers is guided by what was learned from the study of Kazaa [10] — the most popular super-peer network ever deployed, in which the peer-to-super-peer ratio is around 100. The sets of files and semantic types are obtained with the method described in Sections 4.2.1 and 4.2.2. The numbers of peers of a particular semantic type follow the distribution defined by Eq. (2). The value of the parameter  $\alpha$  is set to 0.8. The size of the super-peer cache in any weak peer is 10 while the size of the file cache in any super-peer is 1,000. Before the simulation starts, all the super-peer caches have been filled with the identities of super-peers selected randomly and uniformly from the set of all super-peers. The file caches are initially empty. Each peer initially stores 10 files selected randomly, taking into account the file popularities and peer type. The super-peers are organized into a Gnutella-like network. Simple request flooding is employed for locating files that are not found in the local file caches.

The simulation is executed in phases. In each phase, every weak peer requests one file. The target of the request is determined by the distributions  $p_n(m, k)$ . Although in SOSPNET searching and load balancing are integrated in one protocol, in the experiments we evaluate these two mechanisms separately by disabling load balancing during all experiments but the last one.

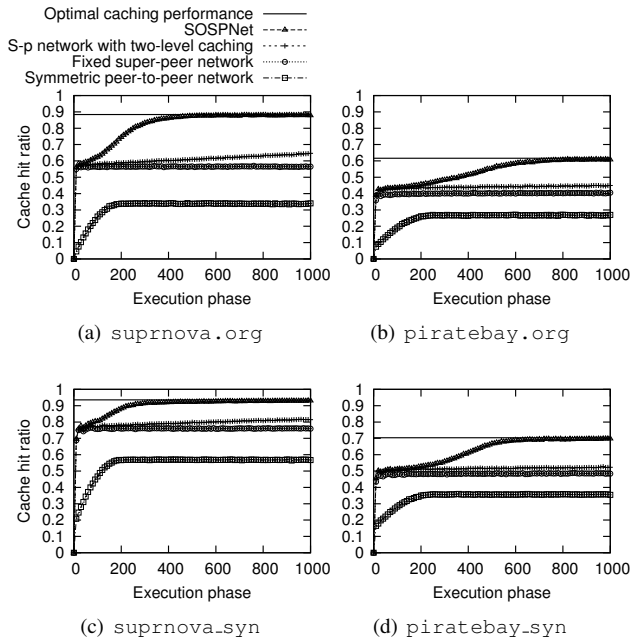


Figure 4. Cache hit ratio for four types of P2P architectures exploiting the semantic relations between peers. The labels in the legend are organized to reflect the order of the lines in the plots.

## 5.2. Results

This section presents the results of the experimental evaluation of SOSpNET.

### 5.2.1. Caching performance

In the first series of experiments we compare the performance of searching in SOSpNET with three other systems that exploit a semantic structure in the P2P network.

The first of these systems, the *symmetric peer-to-peer network*, does not make use of super-peers. Similarly as in [18], each peer in this network maintains a cache of nodes that answered their requests in the past. The caching policy used here is the same as the policy of the super-peer caches in SOSpNET. The size of the peer caches is set to 40. The total number of items cached in the symmetric network is 4,000,000 (100,000 peers with caches of size 40 each), which is twice as high as the total number of items cached in SOSpNET, which is 2,000,000 (100,000 super-peer caches of size 10 each, and 1,000 file caches of size 1,000 each).

The second system, the *fixed super-peer network*, used for comparison exploits super-peers, but assumes that the set of super-peers assigned to a weak peer is fixed. Similar to SOSpNET, weak peers are initially assigned a list of 10 randomly and uniformly selected super-peers, but this list stays unmodified during the whole simulation. The fixed super-peer architecture is comparable with the Gnutella network with ultrapeers [15], with the addition that the responses to peer requests are cached. The size of the file cache at each super-peer is the same as in SOSpNET, and

equals 1,000.

Finally, the third reference system, the *super-peer network with two-level caching*, follows the approach described in [7], exploiting two-level caching of semantic information, without weak peers exchanging information about their super-peers.

Figure 4 presents a comparison of the performance of SOSpNET and the three reference systems. The results are shown separately for all four datasets. For each execution phase we present the fraction of search requests that are satisfied by one of the peer's direct neighbors (cache hit ratio). The direct neighbors in the symmetric system are the nodes stored in the peer's cache. In the super-peer architectures, the direct neighbors are the super-peers contained in the super-peer cache. To improve the clarity, we only plot every fifth point. The solid line represents the value of the optimal caching performance given by Eq. (10).

All three super-peer architectures outperform the symmetric design. The cache hit ratios of the self-organizing, two-level caching based, and fixed super-peer networks are very similar in the early execution phases. However, at some point, around phase 30, the performance of the fixed system stabilizes. This is the point where the items in the file caches are arranged optimally and further improvement could only be done by modifying the peer-to-super-peer assignments. In the two-level caching network, weak peers are not allowed to merge the contents of their super-peer caches, which has a significant performance impact. Among the evaluated systems, only SOSpNET reaches the theoretical performance upper bound.

An interesting observation regarding the optimal caching performance introduced in Section 4.3 can be made at this point. The value of  $ocp$  is defined with the simplifying assumption that peers of different types are using distinct sets of super-peers. Consequently, the value estimated by Eq. (10) may be higher than the actual achievable performance of a SOSpNET-like system where super-peers are shared among peers of different semantic types. The experimental validation using a realistic system model shows, however, that this is not the case. The fact that the cache hit ratio of SOSpNET converges to  $ocp$  has two consequences. First, the formula in Eq. (10), which gives the performance upper bound, is also an accurate approximation of the actual value of the optimal caching performance. Second, the performance achieved by SOSpNET is close to optimal.

### 5.2.2. Peer joins and leaves

The number of execution phases needed for SOSpNET to achieve its peak performance does not say much about the bootstrapping period of an individual peer. In the next experiment we measure how long it takes for a peer joining the system to find its optimal set of super-peers. This time we only use the `suprnova.org` dataset. We

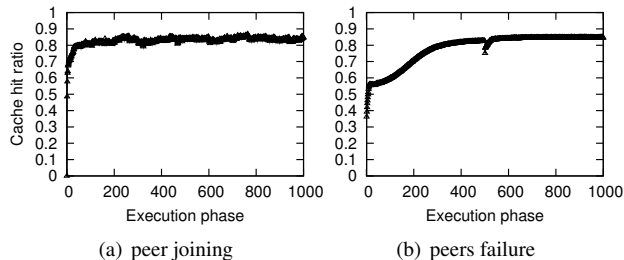


Figure 5. The caching performance of a weak peer joining the system (a), and the drop in performance when half of the peers simultaneously fail in phase 500 (b).

assume that the new peer joins the network when the system has reached its maximum performance, i.e., in phase 1,000 of the experiment described in Section 5.2.1. Figure 5(a) shows how fast the cache hit ratio of the new peer increases with the phase number of this peer.

We expect that SOSPNET is resilient to node failures (unexpected leaves) because of the redundancy built into the architecture. A weak peer that loses some of its super-peers can still connect to the network using the remaining nodes in its super-peer cache. We simulate a catastrophic system failure by killing simultaneously half of the weak peers and half of the super-peers, both selected randomly and uniformly. As a consequence, on average 50% of the pointers stored in super-peer and file caches become invalid. We measure how long it takes before the system replaces the broken references with valid ones.

Figure 5(b) shows how the performance of the system is affected by the failure of 50% of the nodes in execution phase 500. In spite of the scale of the failure, the cache hit ratio does not decrease much. In the next 30 phases, the performance of the system returns to the performance level observed before the failure.

### 5.2.3. Clustering of peers and content

In Section 3.1 we claimed that the mutual dependency established between the super-peer and the file caches results in semantically related peers and files being clustered together. Here we validate the correctness of this claim. We first investigate the correlation between the super-peer caches of weak peers of the same semantic type. Also in this experiment we use the `suprnova.org` dataset.

Figure 6(a) presents the value of the *peer clustering coefficient* defined for each semantic type. The semantic types are sorted according to the decreasing values of the clustering coefficients. The peer clustering coefficient is the average number of identical items in the super-peer caches of peers of one semantic type divided by the size of the super-peer cache. The statistics are collected in execution phase 1,000. For the sake of comparison, we also present the peer

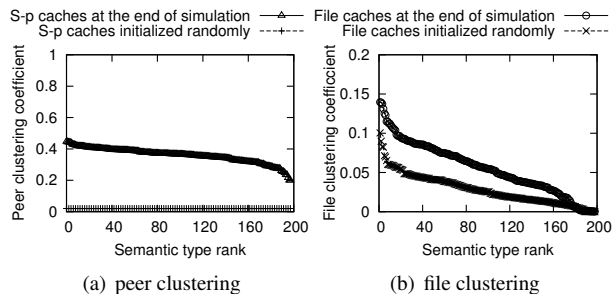


Figure 6. The correlation between the super-peer caches of the weak peers of a certain type (a) and between the items in file caches of the super-peers (b).

clustering coefficient computed at the beginning of the simulation for randomly initialized super-peer caches.

A high value of the peer clustering coefficient indicates that the locality property of the weak peer requests is indeed exploited by SOSPNET. For more than 90% of the types, peers of this type have on average at least 3 (out of 10) identical items in their super-peer caches, which is remarkable given that there are 1,000 super-peers in the system.

In the following experiment we evaluate the correlation between the semantic types of files in the file caches. We define the *file clustering coefficient* of a semantic type as the average of the Jaccard's coefficients [17] of pairs of files of this type. The Jaccard's coefficient is a commonly accepted measure of similarity between sample data. The Jaccard's coefficient of two files is the ratio between the number of co-occurrences of both files and the total number of individual occurrences of these files in the file caches. The values of the file clustering coefficients observed in execution phase 1,000 are compared in Figure 6(b) with clustering coefficients of file caches initialized with pointers to files selected randomly with a bias towards more popular files. Also in this figure the semantic types are sorted according to the decreasing values of the clustering coefficients. The use of biased instead of uniform random distribution during the file cache initialization makes the comparison more representative for an environment where the number of file occurrences in the system is proportional to its popularity.

The smaller extent of file clustering in SOSPNET compared to peer clustering is expected. The targets of peer requests are not limited to files of one semantic type. Even if all peers of one semantic type use the same super-peers, the file caches of those super-peers will still maintain pointers to files of different types.

### 5.2.4. Load balancing

In the last experiment we activate the load-balancing functionality of SOSPNET to assess its ability to deal with super-peers with heterogeneous capacities.

Every super-peer is randomly and uniformly assigned one of four capacity groups. The capacity values of the

Super-peer capacity	Number of super-peers	Average effective load	Standard deviation	Cache hit ratio
0.25	242	172.516	30.212	0.843
0.5	252	163.754	20.864	0.835
0.75	243	148.226	19.185	0.847
1	263	159.374	21.549	0.851

Table 1. The performance of the load-balancing algorithm of SOSPNET.

number of super-peers in each group are presented in Table 1. The results were obtained for the `suprnova.org` dataset. All super-peers start with the same value of the accepted load equal to 1, resulting in super-peers accepting all requests. The parameter  $\beta$  controlling the speed of convergence of the exponential smoothing employed to correct the accepted loads of the super-peers (see Section 3.5) is set to 0.9. We let the simulation run for 1,000 phases before measuring the average effective load, the standard deviation of the effective load, and the cache hit ratio of the super-peers in the different capacity groups.

Ideally, the effective loads of all super-peers in the system should be the same. The results of the experiment, which are presented in Table 1, indicate that the load-balancing mechanism of SOSPNET is able to distribute the system load among the super-peers according to their capacities. Furthermore, the low value of the standard deviation indicates that there are no significant differences in the amounts of load assigned to super-peers with the same capacities. Finally, the last column of Table 1 shows that the load-balancing mechanism does not significantly affect the search performance by retaining cache hit rates which are close to those observed when load balancing was disabled (see Section 5.2.1).

## 6. Conclusions

We have introduced a self-organizing super-peer network architecture called SOSPNET built on top of an unstructured topology with semantic correlations between peers and files. Starting with random sets of neighbors, peers are always able to find the super-peers which guarantee the highest performance of their searches. All decisions in our system are made locally by each peer based on the information collected during previous searches. We have also proposed a novel performance model of a P2P network where peer requests exhibit semantic patterns. Through simulations with real-world trace-based data, we have shown that content in SOSPNET can be located very efficiently. Further, we have demonstrated that a new peer that joins the system can very quickly find the set of super-peers that guarantee the highest performance. Finally, we have shown that our system is resilient to catastrophic failures, and that it supports super-peers with heterogeneous capacities by controlling the amounts of load delegated to individual super-peers.

## References

- [1] <http://www.napster.com>.
- [2] Y. Busnel and A.-M. Kermarrec. Proxsem: Interest-based proximity measure to improve search efficiency in p2p systems. In *ECUMN 2007*, Toulouse, France, February 2007.
- [3] Y. Chawathe, S. Ratnasamy, L. Breslau, and S. Shenker. Making gnutella-like p2p systems scalable. In *SIGCOMM 2003*, Karlsruhe, Germany, August 2003.
- [4] V. Cholvi, P. Felber, and E. Biersack. Efficient search in unstructured peer-to-peer networks. In *SPAA 2004*, Barcelona, Spain, June 2004.
- [5] B. Cohen. Incentives build robustness in bittorrent. In *1st Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, May 2003.
- [6] A. Ganesh, A.-M. Kermarrec, and L. Massouli. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2), February 2003.
- [7] P. Garbacki, D. Epema, and M. van Steen. Two-level semantic caching scheme for super-peer networks. In *IEEE 10th International Workshop on Web Content Caching and Distribution*, Sophia Antipolis, France, September 2005.
- [8] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP 2003*, Bolton Landing, NY, October 2003.
- [9] S. B. Handurukande, A. M. Kermarrec, F. Le Fessant, L. Massouli, and S. Patarin. Peer sharing behaviour in the edonkey network, and implications for the design of serverless file sharing systems. In *EuroSys*, Leuven, Belgium, April 2006.
- [10] J. Liang, R. Kumar, and K. W. Ross. The kaza overlay: A measurement study. *Computer Networks, Special Issue on Overlays*, 49(6), October 2005.
- [11] D. S. Milojevic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. Technical Report HPL-2002-57, HP, March 2002.
- [12] A. T. Mizrak, Y. Cheng, V. Kumar, and S. Savage. Structured superpeers: Leveraging heterogeneity to provide constant-time lookup. In *3rd IEEE Workshop on Internet Applications*, San Jose, CA, June 2003.
- [13] A. Parker. The true picture of peer-to-peer file-sharing, panel presentation during WCW'05, September 2005.
- [14] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *IPTPS'05*, Ithaca, New York, February 2005.
- [15] A. Singla and C. Rohrs. Ultrapeers: Another step towards gnutella scalability. Technical report, 2002.
- [16] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient content location using interest-based locality in peer-to-peer systems. In *INFOCOM'03*, San Francisco, CA, April 2003.
- [17] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, April 2005.
- [18] S. Voulgaris, A.-M. Kermarrec, L. Massouli, and M. van Steen. Exploiting semantic proximity in peer-to-peer content searching. In *FTDCS 2004*, Suzhou, China, May 2004.
- [19] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *IEEE International Conference on Data Engineering*, Bangalore, India, March 2003.