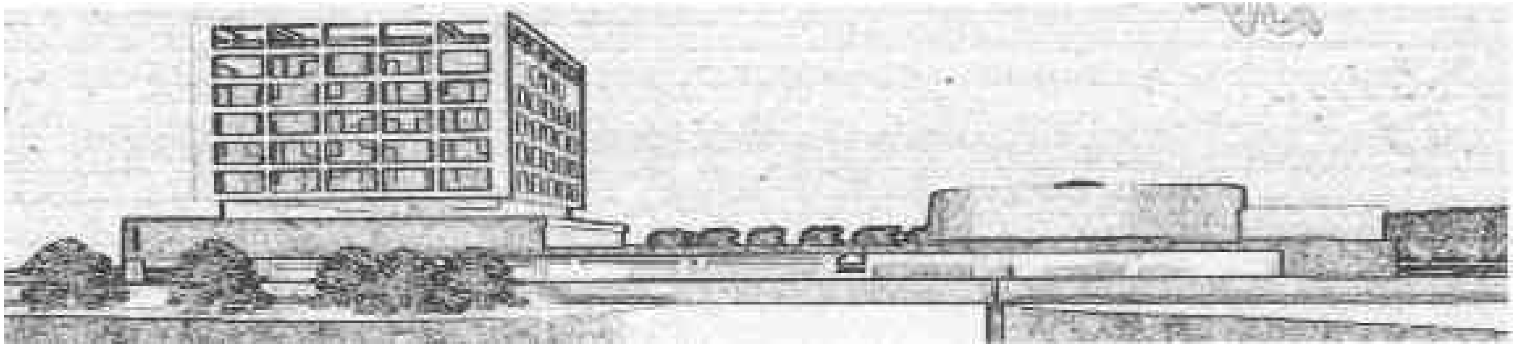
VOLUME V, NUMBER 8



REPORTS ON SYSTEMS AND COMMUNICATIONS

Self-Adaptable Overlays for Unstructured Peer-to-Peer Networks

LUIS RODERO MERINO
LUIS LÓPEZ
ANTONIO FERNÁNDEZ
VICENT CHOLVI

Móstoles (Madrid), Spain, May 2005
Depósito Legal: 50653-2004
ISSN: 1698-7489

Self-adaptable overlays for unstructured peer-to-peer networks

Luis Rodero Merino, Antonio Fernández, Luis López,
{lrodero,anto,llopez}@gsyc.escet.urjc.es
Laboratorio de Algoritmia Distribuida y Redes
Universidad Rey Juan Carlos
Tulipán S/N
28933 Móstoles (Madrid)

Abstract *This paper faces the problem of searching in unstructured peer-to-peer (P2P) networks. It introduces a novel paradigm of self-organized dynamic overlays whose topology adapts to the particular traffic offered to the network. The proposed mechanism uses random walks for resource location and, under this assumption, can be proven to drive the system to an optimal configuration when the network is in very high or in very low load conditions. A fully functional implementation of a P2P system based on this approach has been developed. We present its principles, architecture, and some experimental results obtained through real executions of the application.*

1 Introduction

Peer-to-peer (P2P) systems have obtained great popularity during the last few years. The best known and most widely used application of P2P technology is the sharing of contents. This application was first explored by Napster [16], which grew extremely fast captivating thousands of users worldwide and raising an increasing interest in the scientific community. Other P2P solutions have appeared incorporating new ideas, like Gnutella [14], where, unlike Napster, no central server is present and search is done in a totally decentralized manner, or Kazaa [15], where several nodes act as "superpeers", with different behavior and mission than the rest of participants.

Based on these experiences, some authors define a P2P systems as "a distributed [system], consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources such as content, CPU cycles, storage and bandwidth, capable of adapting to failures and accommodating transient populations of nodes while maintaining acceptable connectivity and performance, without requiring the intermediation or support of a global centralized server or authority" [1].

If we look into the mechanism used to locate contents, P2P systems may be classified into two categories:

- **Centralized**. A central repository stores an index of all resources available in the network, with the location of those resources (which nodes hold them). All nodes register their resources and address their searchs to this repository.
- **Decentralized**. There is no central repository. Queries are forwarded through the network.

Clearly, decentralized systems are closer to the idea of P2P. They are less vulnerable to attacks or censorship, and have better scalability since the P2P system tasks are shared among all network participants. Decentralized systems may themselves be classified by the way new resources are located in the network:

- **Unstructured networks**. The placement of resources is not related with the network topology. Nodes are also added in a non regular way. Examples are Gnutella [14] and Kazaa [15].
- **Structured networks**. Resources are placed at precise locations. The location for every resource is computed by a hash function. Search is done by means of a distributed hash table (DHT). Examples are Chord [13], Pastry [12], Tapestry [17], CAN [10] and Kademlia [9].

Unstructured networks are, nowadays, the most used for content sharing. They are simple to manage and seem to deal better with the organization overhead under high churn rate of peers. On the other hand, structured networks are more efficient locating resources and their search mechanisms scale much better than the techniques used by unstructured networks. Remark that, on the latter, location of resources is not deterministic, content and location are not related and there is not a tight control on the overlay topology either. In this situation, two basic search mechanisms are frequently used.

The first search mechanism is flooding. By this technic each peer broadcasts its queries to all its

neighbors. If some neighbor has the resource, it replies to the query source. If not, it forwards the query to all its neighbors again. There are two kinds of flooding techniques, Breadth First Search (BFS) and Depth First Search (DFS). In both cases, search is limited by a TTL mechanism. The main drawback of flooding is the lack of scalability (for a deeper discussion on this topic see [11]). The second search mechanism is the use of random walks. In this case, nodes forward each query to only one peer, chosen randomly among its neighbors. These messages are tipically called "walkers". The requesting node sends $k \geq 1$ walkers. Each walker will follow its own path. This mechanism introduces less communication overhead compared with flooding, but it may also take longer to solve queries. In [4, 7] we can find some comparisons of both techniques in different network topologies and conditions, concluding that random walks seem to be a promising technique suitable to solve the scalability problems of flooding in unstructured networks. The cost to pay is an increase on the search time for certain topologies.

In this paper we present a P2P system based on a self-adapting overlay topology, that changes dynamically depending on the load on the network. Topology moves from a starlike to a random like one as the load increases. Note that previous results [5] prove that the starlike topologies are optimal (in terms of search time) for non-congested systems and random-graph-like systems are optimal for high loads. For intermediate loads, hubs (nodes that have many more incoming connections than the average) appear. Hubs have a wide knowledge about the network contents, but are not central (not all nodes are connected to them), for this reason, they allow solving queries in few hops without becoming as congested as central nodes.

This paper is organized as follows: first we introduce the concept of *Dynamic Adaptable Network Overlay* and present some related work. Then we explain an architecture and a communication protocol implementing this concept. Finally we present some experimental results and obtain conclusions.

1.1 Dynamic Adaptable Network Overlays

When dealing with P2P unstructured networks, it is well known that the efficiency of random walks in the search process is highly dependent on the overlay structure of the system. The approach traditionally used in the literature to model this begins by assuming that nodes know their own resources plus the ones held by their immediate neighbors. In this case, if some node becomes a central node (all participants are connected to it) it will know all the resources present in the whole system and will be able to correctly answer all queries. In a starlike topology a few nodes become central and all nodes in the system are connected only to them. Hence, all searches are solved in just one hop.

With this argument we understand that, in a non congested scenario, the optimal topology is a highly polarized starlike structure. However, this situation is inefficient if congestion considerations become relevant, since the central node may become overloaded. This is supported by the results in [7], where it is shown that high-degree nodes (those having most connections) support most of the shared load. Moreover, it has been proved [5] that, under the searching model we propose, the optimal network topology is a homogeneous-isotropic one in the presence of severe congestion.

1.2 Related work

There are other similar proposals in the literature of P2P systems where network topology changes to adapt itself to the network load. In all of them random walks are used as the search mechanism, and networks adapt themselves tending to a power-law topology. For example, Lv *et al.* [8] present a system where a flow control mechanism avoids nodes to become overloaded and changes the topology making messages to flow toward nodes with higher capacity. To achieve this, every node periodically tracks the messages it receives. If the node is overloaded, it redirects the most active neighbor (the one sending more queries) to another neighbor with higher spare capacity. By this way, high-capacity nodes tend to have higher degrees, and no node is overloaded. This solution, nonetheless, requires every node to know the state of all its neighbors, which introduces an important communication overhead. The simulation results presented in that work do not take into account this overhead. In any case, their results support the idea of improving P2P networks efficiency taking advantage of nodes heterogeneity as we do.

Chawathe *et al.* in [2] propose a system called Gia that strives to avoid overloading any of the nodes by explicitly accounting for their capacity constraints. Walkers are explicitly forwarded to high-degree nodes, which should be more likely able to answer, but having into account their capability constraints. An active flow control avoids overloading hot spots: one node can send messages to some neighbor only if it has notified the sender that it is willing to accept it. Topology is also adapted in a continuous manner. Nevertheless, their mechanisms introduce more network overhead than ours. In our system, when some node becomes overloaded, their neighbors disconnect by their own, so no explicit flow control is needed. Besides, in Gia nodes need to be aware of the state (number of connections) of their neighbors. The performance in a real scenario, where

the added overhead should be considered, is unknown.

Both solutions are interesting and show the potential of using random walks over power-law networks built by *dynamic network topologies.* Nevertheless, our proposal simplifies the proposed mechanism, adapts better (optimally) to extreme traffic conditions and offer real benchmark results obtained through the execution of a fully functional application.

1.3 Reconnection mechanism

The P2P system we propose is based on the creation of a dynamic adaptable overlay using a mechanism similar to the one presented in [3]. With this mechanism, the interconnection topology changes dynamically in a periodic way, so that nodes reconnect their links using a particular algorithm to choose wich nodes to connect to. This algorithm is based on an *attachment kernel* Π_i, which determines the probability of a particular node to be connected/rewired to node i. The proposed kernel has the form

$$\Pi_i \propto k_i^{\gamma_i} \tag{1}$$

where k_i denotes the number of links of node i and

$$\gamma_i = \begin{cases} 2 \text{ if } c_i \leq \textit{threshold} \\ 0 \text{ otherwise} \end{cases} \tag{2}$$

where c_i refers to the load on node i. Originally this was measured in queries per unit of time, but our implementation uses the number of pending queries as load metric (see section 4.2). Nodes can store as many queries as needed, but can deliver, on average, a limited number of them at every time step. If the node receives more queries than it can process it gets *collapsed.*

The rationale behind Equation 1 and 2 is explained as follows. First, we note that by taking a value of $\gamma_i = 0$ for all nodes, we obtain a random topology (intuitively, all nodes have the same probability of being chosen for a new connection [6]); in turn, if the value of γ_i is strictly greater than 1, we obtain a starlike (the more connections one node has, the more likely it will be chosen by other nodes, finally building a starlike topology [6]). Consequently, in [3] is established that the value of γ_i will be either 2 if the node is not collapsed and 0 otherwise. Thus, the network will evolve towards a random-like topology when the nodes become collapsed, or towards a starlike topology otherwise. It is easy to realize that the value of γ_i for not collapsed nodes has a strong impact on the way topology evolves. In section 4.1 we explain and justify the value of γ_i we have used in our experiments.

2 System implementation

We have implemented a P2P system where nodes use the reconnection mechanism described in section 1.3.

2.1 Communications protocol

Nodes communicate by a simple protocol, with the following types of messages:

CONNECT Petition to stablish a connection with another node.

ACCEPT Reply sent to a CONNECT message if the node accepts the connection. Nodes will exchange their lists of resources afterward.

REJECT Reply sent to a CONNECT message if the node rejects the connection.

RESOURCES_LIST Message containing part of the resources list of the sending node.

DISCONNECT Disconnection message.

LOOK_FOR_NODES Message to look for other nodes. It has a TTL.

NODES_FOUND This message is sent to the original sender of an associated LOOK_FOR_NODES message, containing the list of nodes traversed by the latter, when its TTL reaches 0.

LOOK_FOR_RESOURCE Message for searching resources. It has a TTL.

RESOURCE_FOUND When some resource is found, a RESOURCE_FOUND reply is sent directly to the query source node.

RESOURCE_NOT_FOUND When the TTL of some search gets to 0, a RESOURCE_NOT_FOUND message is sent directly to the query source node.

Messages are sent using UDP. We ensure that the resulting UDP packet size is smaller than the typical MTU to prevent fragmentation. We have implemented an ACK mechanism to add some reliability to the communication. This ACK mechanism is optional, except for the messages to manage connections: CONNECT, ACCEPT, REJECT, DISCONNECT, RESOURCE_LIST.

Connections are directed, i.e., if some node N_1 connects to another node N_2, $N_1 \rightarrow N_2$, then the connection is said to be an N_1 outgoing connection to N_2 and an N_2 incoming connection to N_1. Nodes can send queries through any of their incoming or outgoing connections, and only through

them. Nodes can only close their outgoing connections. It is possible that, having already a connection $N_1 \rightarrow N_2$, N_2 connects to N_1, $N_1 \leftarrow N_2$. This would mean that if N_1 disconnects from N_2, communication between these nodes would still be possible through the connection $N_1 \leftarrow N_2$.

To ease the comprehension of this section we give some notation. $out(N_i)$ denotes the set of nodes that have an incoming connection from N_i; $in(N_i)$ denotes the set of nodes that have an outgoing connection to N_i; $conn(N_i) = out(N_i) \bigcup in(N_i)$

2.2 Reconnection process

To keep adapting the network topology to the load, a reconnection process is triggered periodically. All nodes have a constant number of outgoing connections, that are (possibly) changed at every reconnection. The reconnection process is done in three steps: (1) A list of candidate nodes is obtained by sending a LOOK_FOR_NODES message and waiting the NODES_FOUND reply. (2) The set of nodes to which to connect is computed by the *attachment kernel* function. Those nodes are chosen from the list of candidates. (3) The new outgoing connections to the chosen nodes are built. If an old outgoing connection was in the list of chosen nodes, then the connection is just kept. The node disconnects from the rest of old outgoing connections.

2.3 Resources

Each node N_i has a local set of resources, denoted $localRes(N_i)$. It also knows, for each node it is connected with, the resources that node has. That knowledge is lost when connections are closed. We denote the total set of resources known by N_i as $knownRes(N_i) = localRes(N_i) \bigcup (\bigcup_{N_j \in conn(N_i)} localRes(N_j))$

When a node N_i starts the search for some resource res, it first checks if $res \in knownRes(N_i)$. If not, a LOOK_FOR_RESOURCE query is built and sent to some randomly chosen $N_j \in conn(N_i)$. All nodes are able to start, process, reply, and forward queries. When one node N_j receives a query started by another node N_i for some resource res, it first checks if $res \in knownRes(N_j)$. If so, a RESOURCE_FOUND reply is sent directly to N_i. If not, the search TTL is decremented by 1. If the TTL becomes 0, then a RESOURCE_NOT_FOUND reply is sent to N_i. Otherwise, the LOOK_FOR_RESOURCE message is forwarded to some neighbor $N_k \in conn(N_j)$ chosen at random.

3 Node architecture

The system is implemented in Java. The node design is shown in Fig. 4. The system is formed by the following parts:

Communications Layer It takes care of sending and receiving messages. It implements the ACK mechanism.

Packet Listener . It continously gets incoming messages through the **Communications Layer** module.

Messages Queue Set It holds all the incoming messages that are waiting to be processed. Messages are held in three queues, each one with a different priority. The *Connections Queue* holds connection process messages and has the maximum priority; the *Searchs Results Queue* holds NODES_FOUND, RESOURCE_FOUND and RESOURCE_NOT_FOUND messages and has medium priority; the *Search Petitions Queue* holds LOOK_FOR_NODES and LOOK_FOR_RESOURCES and has the minimum priority.

Message Attender Thread Pool Pool of threads that process messages read from the **Messages Queues Set** module.

Connected Nodes Set It contains the list of incoming and outgoing connections of this node. It also holds the list of resources of every connected node.

Dynamic Network Topology Kernel It implements the *Kernel Reconnection Function.* Every time a reconnection process is run, this module computes the nodes the system should connect to.

Forward Resolver When a message is forwarded this module is called to decide which node the message must be sent to.

Connections Accepter It implements the connections acceptance policy. When a CONNECT message is received this module decides whether the connection is accepted or not.

Reconnecter Periodically starts a new reconnection process. It uses the **Node Finder** to get the list of candidates.

Node Finder It looks for other nodes, sending search messages and collecting the results.

Resource Finder It looks for resources, sending search messages and collecting the results.

4 Experimental setup

A set of experiments have been run with the system described above. The results are shown in section 5. In this section we describe the setup of the experiments.

Experiments were done with 42 nodes in a cluster of 8 PCs. There were 6 virtual nodes by computer, the PC left was used to host a Subscription Service.

Each node held a set of 2000 resources, $|localRes(N_i)| = 2000$. Every resource is held only by one node: $localRes(N_i) \bigcap localRes(N_j) = \emptyset, \forall N_i, N_j, N_i \neq N_j$.

ACKs are not used in communications. Each node reports its state to the Subscription Service every 15 seconds, and triggers a new reconnection process every 30 seconds. Each node keeps three outgoing connections.

Queries are done by *virtual users*. There is one per per node, and all make queries with a fixed frequency, that varies for each experiment. The *virtual user* chooses uniformly and randomly which resource to ask for among all the resources in the network.

The congestion threshold also changes for each experiment. A lesser threshold means that nodes become collapsed more easily. Then, for the same query frequency (same load on the network), the topology will be more decentralized.

We have added an artificial delay to every query processing to increase reply times and be able to overload high degree nodes with less participants in the network. This delay consists simply on executing an empty loop every time two resources are compared. The loop iterations number is an experiment parameter.

Experiments last 2 hours each. Queries launched during the first 10 and last 50 minutes are discarded from the results.

4.1 Nodes connections

The communication protocol makes possible for some node to reject a connection request. For this experiments, nodes were configured to accept all connections.

The γ_i value for the *attachment kernel function* has been set to 2.5 instead of 2 as in the theoretical model of [3]. The reason was to increase the attractiveness of high degree nodes so topology would evolve faster and these nodes would keep their incoming connections. Note that if $k_i^{\gamma_i}$ does not reach high enough values, then nodes will disconnect from central nodes or hubs with a non negligible probability.

Normally, each node decides which other nodes to connect to by applying the *Kernel Function* to a list of candidates. Thus, an important point is how to obtain that list of candidate nodes.

We have developed a Subscription Service where all nodes can subscribe and send information about their state. When nodes subscribe, they also get some experiment parameters. The Subscription Service is also used to provide the list of neighbors a new node must connect to at start time. In all experiments the network topology starts in a random state. Nodes also send information about their state (congestion and connections) periodically. Thus, we avoid using LOOK_FOR_NODES messages, as we want to evaluate the influence of resources searchs only. Peers ask the Subscription Service which nodes they must connect to in every reconnection.

4.2 Congestion computation

There is one important different between our implementation and the model presented in [3]: the way node's load is computed. In [3], a node's load is the number of queries that node has received during the last unit of time. Nonetheless, we realised than in heterogeneus networks nodes congestion depends not only on the messages received, but also on the processing capacity of those nodes. The more capacity some node has, the less congested it will get for the same number of messages received.

We found out that a more realistic congestion metric is the number of pending messages. If two nodes receive the same amount of queries, the queue size will be smaller for the node with higher processing capacity. If two nodes are equally able to process messages, then the node that receives more messages will have a bigger queue.

5 Experimental results

5.1 Topology adaptability

First, we show how network topology adapts itself depending on the network load.

Initially, topology is in a random state. Then, it evolves and changes as reconnections are done. New connections are stablished depending on the *kernel function*. When the network load is small, nodes tend to form a starlike topology, see for instance Figure 1, where three nodes have become central. This topology is built with threshold (this is, node capacity) of 10, and load of 5 petitions per minute and node.

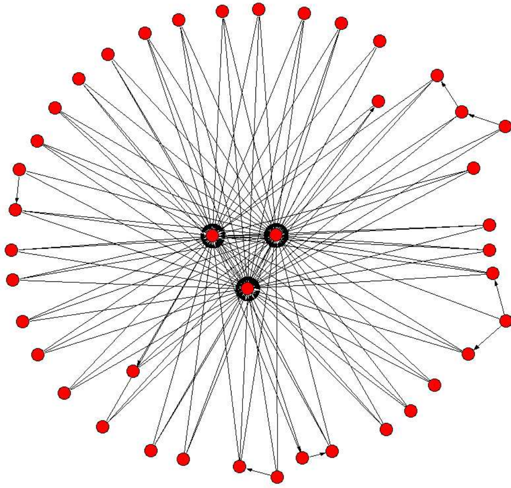


Figure 1: Topology with threshold 10 and 5 petitions per min

With the same threshold, but increasing the load to 10 petitions, we have the topology show in Figure 2. Here, we see that the network is not totally centralized. There are nodes that have many connections (hubs) but are not central.

Finally, we increase the load to 15 petitions per minute, resulting in the topology shown in Figure 3. Here the network is totally random. No node has many connections because it inmediately becomes overloaded and neighbors disconnect from it.

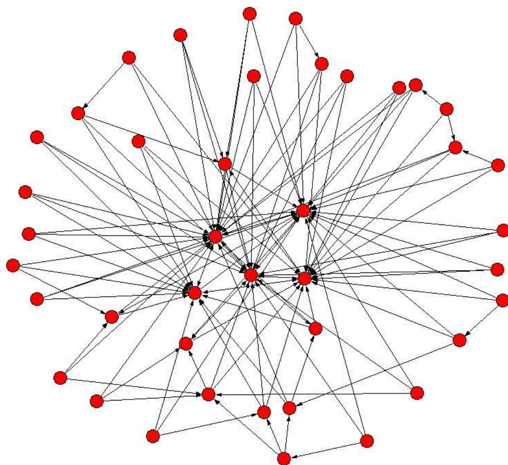


Figure 2: Topology with threshold 10 and 10 petitions per min

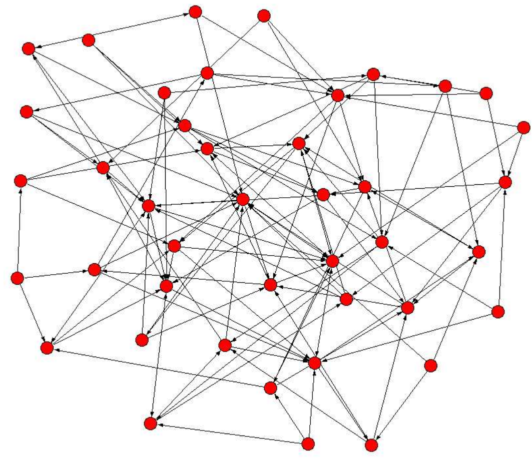


Figure 3: Topology with threshold 10 and 15 petitions per min

5.2 Search performance

Now, the search mean time is evaluated. Experiments have been executed for different network loads and nodes thresholds. The network load is measured in terms of petitions per minute triggered by each node.

In Figure 5 and 6 we present the results of experiments for loads of 2, 4, 6, 8, 10, and 12 petitions per minute, and with thresholds of 0, 10, 50, 100, 1000, 10000, and 1000000. Note that setting the nodes threshold to 0 means that nodes will always be collapsed, and the topology will be random. To force a centralized topology, the nodes threshold is set to 1000000 (nodes will never have a congestion greater than that). In experiments with intermediate thresholds, the topology is centralized with low network load, moving to a random state as the load increases.

For low load, we see how the random network (the one with threshold 0) has the worst mean search time. The other networks are all centralized and their search times are similar.

Another thing to note, is that the centralized network becomes the worst one at load 10, since it has the biggest mean search time. At the same time, the random network is the fastest. The other networks lay between these two. This agrees with [5].

As the network load increases, the systems with high thresholds get slower at a higher rate than the systems with low thresholds. The random network keeps being the fastest, and the centralized the slowest. But it appears that networks converge as there is more load on the network. This is logical, the bigger the load, the closer the topologies are to the random state for all thresholds (remember that random is optimal).

Nonetheless, three experiments appear with results that do not seem to fit to what theoretically we should get. See, in Figure 5, the results for experiments with load 6 and threshold 10, load 8

and threshold 100, and load 8 and threshold 50. Their mean search times are the biggest, while these times should be between those for random and central networks. We have found out a side effect, not foreseen in previous literature, that penalizes certain threshold networks (depending on the load) and could explain these results. Normally topologies tend to form central nodes or hubs. The more connections some node has, the more nodes will try to connect to it. But when there is a high query load on the network, hubs will not be able to process queries fast enough. When finally the hub becomes congested (by any congestion computation metric used), it sharply stops being attractive and its neighbors will disconnect from it, then it will lose all the knowledge about those nodes. The problem arises here: it becomes a "regular" node, but most likely it still has many queries on its queue, queries that will take a large time to process and that it will likely not be able to complete itself. When hubs appear and disappear at a high rate then this effect penalizes total search times. A solution to this problem could be the use of resources caches in nodes.

Finally, we see that we have found one threshold, threshold 10, that seems optimal or close to optimal for all loads (save for load 6, as is explained above). This suggests that, depending on some parameters of the network, there is certain threshold that minimizes the search times for all loads.

Is important to note that our experimental results agree with those obtained by simulations in [3], where our reconnection model was introduced. This encourages us to keep working on this line of research.

6 Conclusions and future work

In this paper we have introduced our implementation of a P2P system with Dynamic Topology Adaptation. The results show how the topology adapts itself depending on the load of the network, trying to keep close to an optimal topology. This system does not require nodes to keep information about their neighbors state, and does not need a explicit flow control mechanism so the total network overhead is smaller than in other systems (e.g. [8] and [2]) that also use dynamic topologies.

Nonetheless, this system is far from optimal. There are many improvements that can be done for a better performance. Some of them are: avoid sharp topology changes, as they can have impact on search times; use resources caches on nodes; use Bloom filters to compact the list of resources of neighbor nodes; evaluate other forwarding policies that could have better performance than random forwarding; and look for better node search methods (like agents).

References

[1] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.

[2] Yatin Chawathe, Sylvia Ratnasamy, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM 2003)*, pages 407–418, Karlsruhe, Germany, August 2003.

[3] V. Cholvi, V. Laderas, L. López, and A. Fernández. Self-adapting network topologies in congested scenarios. *Physical Review E*, 71(3), 2005.

[4] George H. L. Fletcher, Hardik A. Sheth, and Katy Borner. Unstructured peer-to-peer networks: Topological properties and search performance. In *Proceedings of the Third International Workshop on Agents and Peer-to-Peer Computing*, New York, New York, USA, July 2004. To be published by Springer.

[5] R. Guimera, A. Diaz-Guilera, F. Vega-Redondo, A. Cabrales, and A. Arenas. Optimal network topologies for local search with congestion. *Physical Review Letters*, 89, November 2002.

[6] P. L. Krapivsky, S. Redner, and F. Leyvraz. Connectivity of growing random networks. *Physical Review Letters*, 85:4629–4632, November 2000.

[7] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, pages 84–95, New York, New York, USA, June 2005.

[8] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can heterogeneity make Gnutella scalable? In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 94–103, Cambridge, USA, March 2002.

[9] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, pages 53–65, Cambride, USA, March 2002.

[10] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A

scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM 2001)*, pages 161–1672, San Diego, California, USA, 2001.

[11] Jordan Ritter. Why gnutella can't scale. no, really. Technical report. Electronic format in http://www.darkridge.com/ jpr5/doc/gnutella.html.

[12] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, Heidelberg, Germany, 2001.

[13] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM 2001)*, pages 149–160, San Diego, CA, USA, 2001.

[14] The gnutella website. http://www.gnutella.com.

[15] The kazaa website. http://www.kazaa.com.

[16] The napster website. http://www.napster.com.

[17] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, University of California, Berkeley, 2001.

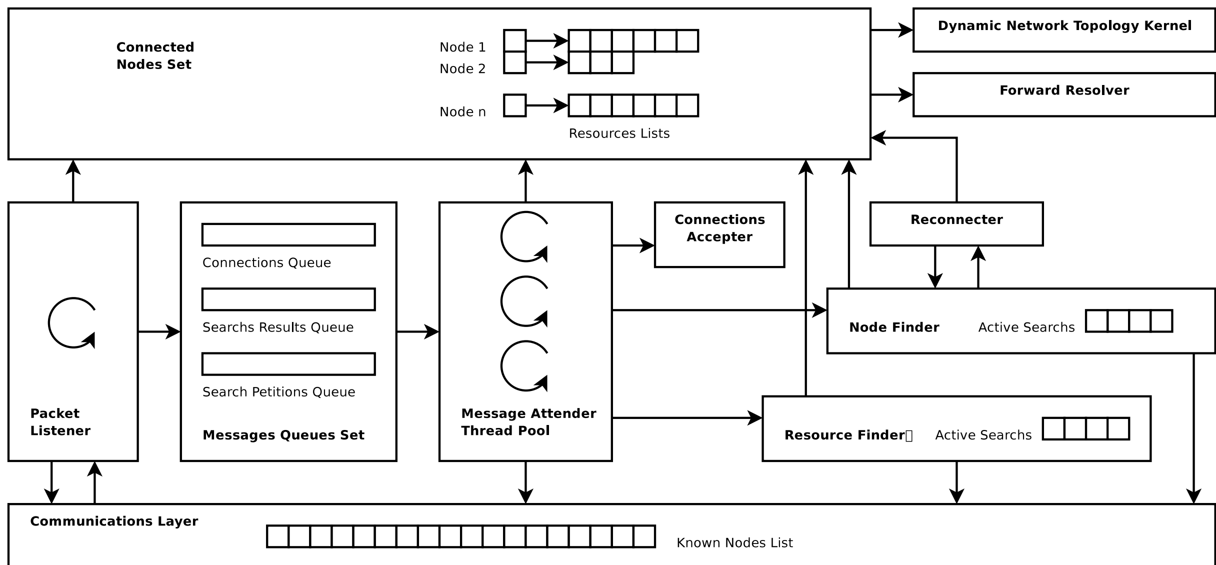


Figure 4: Node Architecture

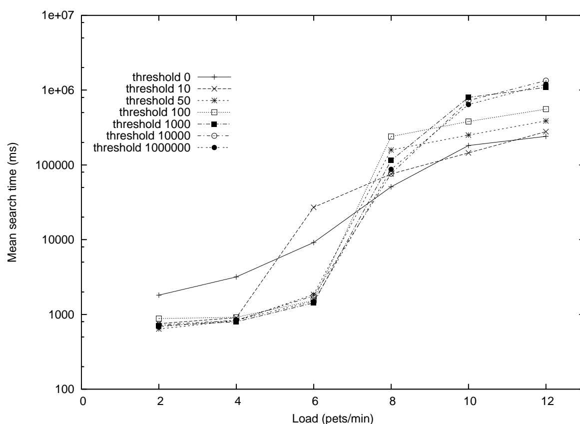


Figure 5: Mean times against load, for different thresholds (log scale)

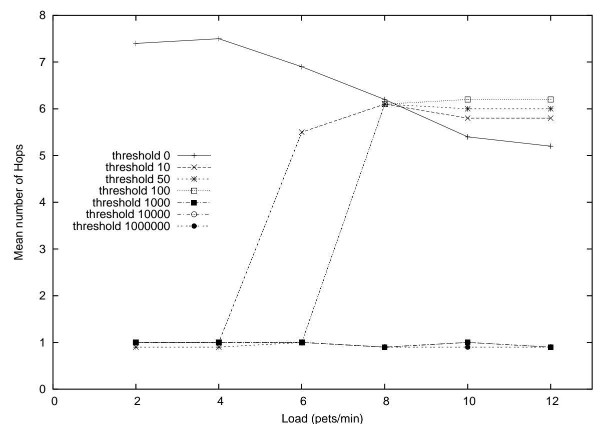


Figure 6: Mean number of hops against load, for different thresholds