PAPER SUBMISSION TO JOURNAL OF COMPUTER COMMUNICATIONS

Special Issue of Elsevier Computer Communications on Self-Organization and Self-Management in Communications – Taking Vital Steps Towards Realizing Autonomic Networking

Load-driven Neighbourhood Reconfiguration of Gnutella Overlay

Evangelos Pournaras [a], Georgios Exarchakos[b], Nick Antonopoulos[c]

a. University of Surrey, Guildford, Surrey GU2 7XH, United Kingdom, email: e.pournaras@few.vu.nl
b. University of Surrey, Guildford, Surrey GU2 7XH, United Kingdom, email: g.exarchakos@surrey.ac.uk
c. University of Surrey, Guildford, Surrey GU2 7XH, United Kingdom, email: n.antonopoulos@surrey.ac.uk

Corresponding Author: Georgios Exarchakos

Mailing address:

Computing Department
School of Electronics and Physical Sciences
University of Surrey
Guildford, Surrey
GU2 7XH
United Kingdom

Telephone number: (+44) 1483 68 9104

E-mail address: g.exarchakos@surrey.ac.uk

Abstract: Unstructured P2P networks support distributed applications whose workload may vary significantly over time and between nodes. Self-optimizing systems try to keep the load in the network balanced despite the frequent load fluctuations. Several P2P systems exhibit a number of related features but fail to avoid centralization under high-load situations. ERGO aims to balance the overloaded nodes by rewiring some of their incoming links to underloaded ones via a set of interconnected servers which index the underloaded nodes. In two simulated environments, ERGO load-balancing on Gnutella network increases the balanced nodes and network availability by preserving its efficiency and even reducing its messages.

Keywords: peer-to-peer, self-optimizing, load-balancing, rewiring, resource management.

1 Introduction

Evolving self-configurable decentralised P2P networks may experience high load situations depending on the distributed application they support. Nodes (or peers) receiving many queries per time unit, beyond their capabilities, may queue or drop some of them or even fail. Assuming a uniform distribution of queries generation in the network, popular nodes (with many incoming links) receive more queries than the unpopular ones.

Defining the load as the number of queries received and node capacity as "*the number of queries that the client can handle*" [1] per time unit, the nodes can be distinguished into overloaded, normally loaded and underloaded. A node is overloaded if the received queries are more than an upper threshold but yet lower than its capacity. If the received queries are less than a lower threshold, then the node is underloaded and if between the two thresholds, then normally loaded. Any extra query, when its capacity is fully used, gets dropped.

Dropped queries are not forwarded further. Resources located on nodes beyond the overloaded one cannot be discovered unless the same query is regenerated and/or finds another path. The resources of underloaded nodes do not sufficiently contribute to the network discovery mechanism and/or may increase the average path length and, thus, the latency of the queries in the network. Load-balancing techniques aim to remove this inequality between nodes by moving load from overloaded to underloaded nodes.

This paper addresses some of the key features of Autonomic Computing as defined and analysed by IBM in [2]. One of these is self-optimization, “Self-optimise: to tune and balance workloads to maximise the use of information technology resources”. We propose ERGO, a load-balancing mechanism that works on top of the Gnutella [3] overlay. It is a set of interconnected virtual servers that receive advertisements of underloaded nodes and requests for help from overloaded ones. The request travels from server to server until it is satisfied. As soon as an underloaded node willing to take over the excess load is found, the incoming links of the overloaded one are redirected to it. The term ‘request’ is used to distinguish the Gnutella queries from ERGO ‘requests for help’.

Gnutella is the basis of many unstructured, hierarchical or flat organizations with high applicability to real-world systems. The motivation to work on Gnutella was its resilience to node failures, good efficiency on requests for popular resources and ability to handle complex requests. Load-balancing schemes have been proposed mainly for structured P2P networks but recent efforts try to tackle the issue on unstructured ones, as well.

2 Related Work

There are two main classes of P2P network topologies: structured and unstructured. Although load-balancing techniques have been applied to both, this research focuses on unstructured topologies, with particular emphasis on Gnutella I.

Structured P2P networks such as CAN, Pastry, Tapestry, Chord, Kademlia and Viceroy [4], assign an ID from an address space to every node; a key from the same space to every resource, and map each tuple {key, resource} onto a specified ID. Given the keys, this type of overlay network can perform efficient discovery but requires a copy or pointer to each resource at the node which is responsible for this key of the resource. They are theoretically well-founded topologies and guarantee successful discovery with exact recall if the requested resource is available in the network. These system topologies are based on DHTs. Each node has a list of pointers to neighbours as further as possible, and implement distributed algorithms to update these pointers whenever the topology changes. This updating scheme may produce many messages in case of intermittent behaviour of nodes.

Latency becomes a considerable problem in large structured networks; that is networks with a very large number of nodes as each query is routed to the next set of intermediate nodes as far as possible and, thus, producing long-distance network traffic and delays. Low-capacity nodes can introduce further delays as they can easily become overloaded even if they do not process the queries but instead just forward them. Query routing is based on simple keys and, thus, structured P2P overlays do not support complicated queries.

The authors of [5] address the problem of uneven key-space partitioning in DHTs. According to [5], some machines may receive more keys during mapping by a factor of $O(\log n)$ on average, where n is the number of nodes. Furthermore, even if the address space is evenly distributed, the uneven distribution of keys may cause overloading. The algorithms are based on the idea of virtual nodes that form a real node. The distribution of keys is based on the Markovian properties and the algorithm achieves $O(\log n)$ degree per real node; $O(\log \log n)$ lookup hops and constant factor load-balancing. The item balancing algorithm is based on movements of underloaded nodes into areas of address space occupied by excessive number of items. The use of moving virtual servers to modify the regions of the ID space associated with each node is used in [6] and [7]. The architecture proposed in [8] balances the length of intervals in which a node can enter or leave the network.

Unstructured P2P overlays organise the nodes in a random flat or hierarchical graph that is decoupled from the location of the resources. Replication methods may increase the availability of a resource within the network by copying the same resource to several places [9], thus making the discovery of popular (well-replicated) resources more efficient with high recall [10], as is the case for the majority of requests. No unstructured overlay guarantees successful discovery. However, it usually supports complicated queries and is highly tolerant to node failures. In flat random organisations, if a node is unable to satisfy a query, it forwards it to one or more neighbouring ones. They produce many unnecessary messages and may result in flooding the network and revisiting nodes [1].

In case of hierarchical random (sub-strand of unstructured P2P) overlays, certain nodes (super-nodes) take over the indexing of resources. Every node discovers the location of the requested resource with the aid of the super-nodes and then may directly access it. The interconnections between individual super-nodes, between super-nodes and nodes and between simple nodes form a random network with a few nodes more popular than others. While this scheme reduces the number of messages required, the fewer super-nodes in the network means that the more the network suffers from single-point of failure issues and scalability limits [11].

Recent research has drawn less attention on load issues of unstructured P2P, too. Phenomena like Free-Riding and uneven distribution of resources [12] over the nodes may transform a random Gnutella into a star network [10]. At a given TTL, a query travelling within a star network visits few nodes thus reducing the recall of the network, particularly for the sparse resources. Adar and Huberman [12] observed in 2000 that 66% of the nodes were sharing no files and 73% of them just the 10% of the files in the network. In 2005 a similar study [13] proved that these percentages were 85% and 86% respectively. Therefore, the Gnutella network starts exhibiting client-server characteristics reducing its scalability and availability.

In an attempt to provide QoS guarantees on unstructured P2P networks, Hughes, Warren and Coulson in [14] presented AGnuS, a set of subsystems lying on top of Gnutella. AGnuS implements four mechanisms which collaboratively achieve the QoS requirements: load-balancing, automated caching of the most popular resources, content-based routing by monitoring the neighbours file-type densities and file filtering. For load balancing purposes, each node maintains more (eight) neighbours than in original Gnutella but forwards to four only chosen based on their current load (the less the better). Although this technique improves availability and shortens the average path length (thus reduces latency), it cannot prevent the network from turning into a star or power-law topology and, thus, becoming intolerant to targeted attacks. Furthermore, AGnuS does not provide any mechanism to relieve the overloaded nodes when query rerouting based on the neighbours load cannot help.

Gia [1] is another system that targets the scalability problems of Gnutella. The developers achieve improvement by using topology adaptation, flow control and replication algorithms. Each Gia node calculates a satisfaction level of its neighbour list and adds a new one with higher capacity and degree than another in the list. This process continues until its list satisfaction reaches the optimum value (1). Based on the flow control mechanism, each node sends periodic tokens to its immediate neighbours. A node can forward only one query to the token originator. The distribution of tokens follows the capacity distribution among the neighbours of a node. This architecture significantly changes the Gnutella protocol as the searching algorithm is random walkers. Gia does not provide any mechanism to relieve already overloaded nodes and its mechanisms show a preference to high capacity nodes that could result in a star topology. Both AGnus and Gia systems target the source of the queries rather than the receivers.

While the ROME [15] architecture tackles load problems in Chord rings using a central server monitoring all nodes of the ring, G-ROME [16] builds an unstructured overlay of ROME servers to move underloaded nodes among independent rings. This is complementary to the load-balancing techniques on single rings and the idea is further used in the current work: add more nodes into overloaded areas (rings in case of G-ROME and unstructured neighbourhoods in case of ERGO).

Although CAN network is a structured P2P system, most load-balancing mechanisms for DHTs are not directly applicable to semantic-CAN applications [22]. In [23], effort has been put on efficiently routing queries towards reducing the number of duplicates. Moreover, in [21] the load generated by inquiries from other nodes in the network is balanced (close to the objective of our work). Imbalance issues, due to values-based data mapping, are also addressed in [22]. The authors propose a solution that monitors the load; peers join highly loaded areas and keep information about other overloaded peers. Multiple zones are supported with load-aware joins and zone splits of highly loaded peers.

In P2P networks, the load generated by the requests is not the only critical performance issue. Peers keep data and resources that have to be available to other peers. Content-based P2P applications can create overloaded peers either as their content is semantically important for the rest peers in the network, or because the replication mechanism is not intelligent enough to predict bottlenecks in both peers and the network as a whole. In [24], the issue of content-oriented XML retrieval is outlined. The authors propose a P2P structured architecture in which the network and the local indexes retained are reorganised to provide scalability, robustness and load-balancing. The mechanism decreases the bandwidth consumption and enables parallel computing.

Finally, the MINERVA system [25] uses the concept of local indexes to form a global index of meta-information in the P2P network, in order to provide information discovery. This information concerns compact statistics and QoS information. The results show that the discovery success is improved, load-balancing is achieved through this meta-information exchange and indexes are maintained.

3 ERGO: Enhanced Reconfigurable Gnutella Overlay

ERGO is a system designed to function on top of Gnutella aiming to keep the load of nodes within certain user-defined load thresholds. In principle, the proposed architecture rewires any incoming link (cause of excessive load) of overloaded node (requestor), to other nodes (responders) that can handle both their own and new load. Once this process has been completed, both the requestor and responder have a load below their upper threshold.

3.1 Architecture Overview

ERGO does not modify the functionality of the overlay that it is applied to, but instead monitors and reconfigures the incoming connections according to the load generated from the nodes on the starting point of these links. A node chooses a number of incoming links to redirect to other nodes that can handle their load. The model also provides the mechanism that allows overloaded nodes discover the underloaded ones and perform the rewiring.

This mechanism is built on top of the Gnutella overlay as a set of arbitrarily interconnected virtual servers that keep record of the underloaded nodes and process requests for help from the overloaded ones. From this point onwards, the terms 'virtual server' and 'server' will be used interchangeably. A server that receives a request and cannot provide help (find an underloaded node that can take over the excessive load of the requestor) forwards the same request to one random neighbouring server. When an appropriate node is discovered, the rewiring takes place. Figure 1 illustrates the basic components of the architecture.

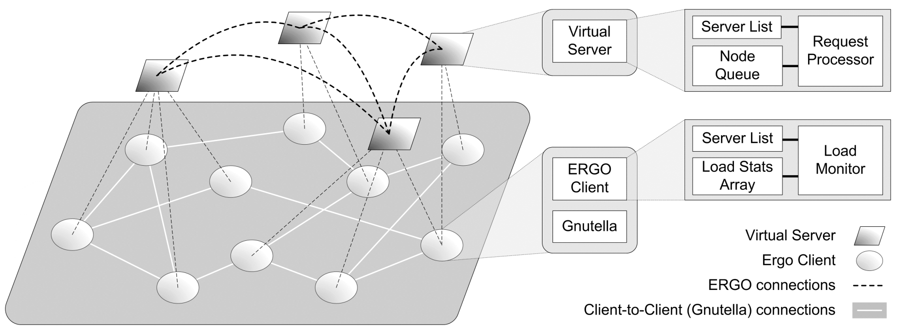


Figure 1: ERGO architectural components

ERGO uses the concept of hierarchical unstructured P2P overlays to reduce the number of load-balancing messages. An ERGO virtual server may be installed on an independent machine but it may also operate as a separate module in parallel to the same Gnutella and

ERGO client. While the virtual servers are selected by their performance profile, every Gnutella client is monitored by one ERGO client advertised in one of the virtual servers. The most important structures of the Server are the *Server List*, *Node Queue* and *Messages* (requests) *Processor*:

- **Server List**: a list of other addresses of servers that it is connected to. This list is used when a request cannot be satisfied by the local server and is forwarded to another random one. This data is retained and updated during the functionality of the network.
- **Node Queue**: a component that keeps record of the underloaded nodes in a queue. Each record has the address, current load and upper threshold of the underloaded node. The higher the distance of the current load of the node from the upper threshold, the closer to the top of the queue it is placed.
- **ERGO Messages Processor**: the component that processes all the requests and responses which the server receives.

ERGO clients have three components to detect load changes and to keep record of the servers they can access to request help:

- **Load Monitor**: a component which reacts on overloading/underloading situations. When the load of a node surpasses the upper threshold, it triggers the load balancing procedure. When the load is below the lower one, it sends an advertisement to a random server.
- **Server List**: is the same list as the one of virtual servers, a list of servers able to support load-balancing requests and to advertise underloaded nodes. Each time the node becomes underloaded, a random server is selected, thus distributing the ERGO load to several servers.

- **Load Statistics Array**: it has a record for every incoming link: the address of the node on the other end (load originator), and the load received from this link.

Both clients and servers use the GWebCache system [17] to build up their server lists. Initially, a list is used as a starting point to help an ERGO client or server join the network. Subsequently, when they cannot access the majority of the servers in their lists, they send a new request to GWebCache. ERGO clients use the response originators to replace failed servers in their lists and only if this is not enough, they make a request to GwebCache.

The following sections present an analysis of the architecture starting from messages exchanged description; load monitoring; request forwarding; resource discovery and finally rewiring. The last issue concerns failures and extreme conditions that may occur in ERGO.

3.2 Interaction Messages

ERGO architecture introduces a number of messages as some of its actions require communication between its entities - virtual servers and clients. This communication takes place with the advertisements, the request from and response to the overloaded node, and the rewiring request to the load originator. Figure 2 lists the interaction messages used in ERGO.

Message	**Description**
advertisement	the advertisement of an underloaded node to a server.
request	the request generated by the overloaded node and forwarded to the overlay of servers containing the value of the excessive load to be served.
response	the response from the server to overloaded node with one underloaded node that can take over the excessive load.
rewire	command to the load originators to rewire their links from the overloaded nodes to the discovered underloaded one.
ack	used by both servers and nodes to acknowledge the other part that an action is complete or not.

Figure 2: The list of the ERGO messages and their use

Figure 3 presents the sequence of message exchanges between two servers; the underloaded; overloaded and load originator nodes.

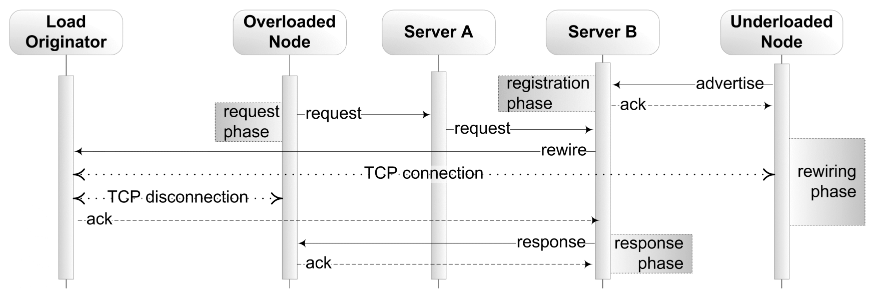


Figure 3: The sequence diagram of ERGO interaction messages between servers and nodes

Each ERGO interaction message has a header and a payload. The size of the header is fixed and has four fields. However, the payload is variable both in number of fields and size. The ERGO message template is shown below:

Header				Payload
Message ID:16byte	TTL:1byte	Type:1byte	Length:4bytes	…

Figure 4: ERGO message template

- **Message ID**: a 16-bit message unique identifier. The Gnutella ID generation process is used for this ID.

- **TTL**: maximum number of steps the message can travel within the overlay. This is equal to the TTL used in Gnutella messages. Each entity, upon receiving a message, has to reduce the number in this field by one.

- **Type**: a code specifying the type of the message payload (ack:0x00; advertise:0x01; request:0x02; response:0x03; rewire:0x04)

- **Length**: integer representing the length of the payload. This makes each message distinguishable from the next one.

The message payload uses four fields depending on their type:

- **Port** [2 bytes]: the listening port of the message originator at which an incoming connection can be accepted.

- **IP** [4 bytes]: the IP address of the message originator.

- **Load (LD)** [2 bytes]: the number of Gnutella queries a node handles or can handle within a time unit.

- **OK** [1 byte]: a boolean value to respond with 'true/false' on a request.

Analytically, the table shown in Figure 5 gives detailed structure for each message.

Type	Payload	Description
0x01	IP PORT LD LD	Fixed-size payload with four fields. IP and PORT are necessary for opening a connection to the underloaded node. The third field is the overloading threshold of the node (the minimum load that would overload it). The forth field is the current load.
0x02	IP PORT LD $^{2\ldots*}$	Variable-size payload with two or more triplets [IP, PORT, LOAD]. The first triplet is the IP, PORT and excess load of the overloaded request originator. The following triplets are the IP, PORT and load produced by the incoming links and are used by the server to choose the links to be rewired.
0x03	IP PORT	Fixed-size payload with one [IP,PORT] tuple for the discovered underloaded node.
0x04	IP PORT IP PORT	Fixed-size payload with two [IP, PORT] tuples. The first tuple is about the underloaded node that will accept the rewired links and the second about the overloaded node from which the links will have to be disconnected.
0x00	OK	Fixed-size acknowledgement payload with a true/false value.

Figure 5: The ERGO messages payload analysis

3.3 Load Monitoring

By monitoring the load generated by the Gnutella incoming links, an ERGO client may create four events that in turn trigger the load-balancing mechanism. These events happen when the load passes from the underloaded or overloaded to the normally-loaded area and vice versa. Figure 6 illustrates a visualization of the thresholds.

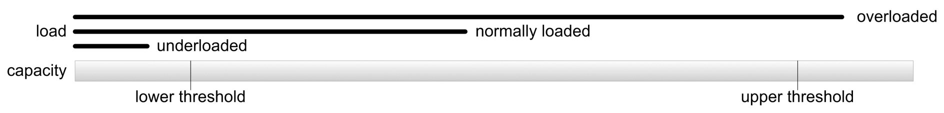


Figure 6: Thresholds of ERGO nodes

If normally loaded, the ERGO client has to stop any load-balancing activity. It periodically sends advertisements to a server from its Server List if it is underloaded and requests if it is overloaded. By monitoring Gnutella, the ERGO client keeps statistical data (load) for all the incoming links based on which it takes all its decisions.

3.3.1 Load Statistics

ERGO client calculates the load of a node by summing up the load received by all the incoming links within a time unit. It determines the portion of its capacity that is used and thus it can take corrective actions. The aim of ERGO is to perform load-driven rewiring of the incoming links. Therefore, the client needs to know the load received per link.

The Load Statistics Array is a table that has as many records as the incoming links of the node. Each record contains the address and the number of Gnutella queries originated from that link within the last time unit. This array is updated by creating a new record when a new incoming link is established or after rewiring actions.

3.3.2 Underloaded Nodes Advertisements

Once the load has dropped below the lower threshold (underloaded node), the ERGO client constructs an 'advertisement' message and chooses a random server from the Server List to sent it to. While the node remains underloaded, the same message is periodically sent to the same server. These heartbeats keep the information state of the underloaded node state information in the server up-to-date, thus retaining the best-goes-first ordering scheme.

The heartbeat stops when the load gets more than the lower threshold. The last 'advertisement' message has a negative number in its forth field indicating that the node is not underloaded anymore. Another random server is only chosen when the load falls below the lower threshold again. This rule prevents the underloaded node from sending advertisements to various places reducing the number of messages.

There is a TCP connection between ERGO client and server. If one of them fails, the connection is dropped. In case of server failure, the client removes it from its Server List and another random server is selected for all its future ERGO operations. If a client fails, it is removed from the server Node Queue.

3.3.3 Request Generation

The area between the upper load threshold and the maximum capacity of the node works as a buffer zone before the node starts rejecting queries. To avoid this situation, the ERGO client starts generating requests seeking an underloaded node that could handle its excess load, that is load above the upper threshold. Each new request is cached and forwarded to a random server of Server List (or more than one). The cache keeps the requests for time equal to TTL hops of request forwarding between servers plus two hops of response travelling.

When the lifetime of a request in the cache expires, or a response is received, it is removed from the cache. If the load is still over the upper load threshold, a new request is generated and sent to another random server (or another subset with potentially increased size).

3.4 Target Node Discovery

Every request originated from an ERGO client is forwarded between ERGO servers. Each server tries to fully satisfy the requested load using the first underloaded node of Node Queue only, thus accelerating the discovery process. The two conditions to be fulfilled by this selection prevent cascading effects of overloading situations since not only

1. must the overloaded node become normally loaded, but
2. the transferred load in the selected underloaded node must not exceed the upper threshold as well.

Each request carries the excess load of the requestor and its Load Statistics Array. The server attempts to find any subset of nodes in this array by always satisfying the two conditions. If the first node can fully satisfy the requested load, the Node Queue reserves the discovered node and the walker stops. ERGO does not support partial answers, thus avoiding deadlock situations with node reservations in multiple servers and retaining the Gnutella clients node degrees constant.

The first node has the highest distance between current load and upper threshold. If it can not satisfy these requirements then no one else in the same Node Queue can. The TTL field of the received request gets reduced by one, and if non-zero, the same request is cached for TTL time units and is forwarded to a single random neighbouring server. The forwarding mechanism in the servers overlay is random k-walkers with the 'k' equal to the number of servers initially selected by the request originator.

3.5 Neighbourhood Rewiring

Both rewiring and response actions are undertaken by the server that found an appropriate underloaded node in its Node Queue. Rewiring the load originators precedes response to the request originator.

The responder server builds as many rewiring messages as the number of nodes to be rewired, including the addresses of the overloaded and underloaded nodes. These messages are sent to the load originators that in turn shift the connections from the overloaded nodes to the underloaded ones and positively acknowledge to the server that they have finished the rewiring.

After the acknowledgements from all the load originators have been received, the server responds with the address of the discovered underloaded node to the overloaded one as a confirmation that the process is complete. Then the node clears its cache from the request. If it is still overloaded, then a new process is initiated. This is the fundamental reason that the response phase follows the rewiring.

3.6 Failures and Extreme Situations

A server failure affects neighbouring servers, underloaded and overloaded nodes connected to it. These nodes remove the failed server from their Server Lists. If the server fails during the advertisement and request phase, another server is chosen by the clients and neighbouring servers for their ERGO communication. The server may fail during rewiring or response phases and some ‘rewire’, ‘response’ and/or ‘ack’ messages may never reach their destination. There is no recovery mechanism or protocol. The ERGO approach is attempt-based. Furthermore, actions can not be cancelled: the ERGO architecture reverses no rewiring actions or response message if not all the acknowledgements are received.

When a discovered underloaded node fails, its TCP connections with the Server List and neighbours are lost. At this point, the server who has a record in its Node Queue for that node removes it. A load originator does not disconnect from the overloaded node if it has not already established a connection to the underloaded one. Therefore, no rewiring takes place and no response is sent to the overloaded node that will finally remove the last request from the cache when it expires.

Defining the ‘weakness’ as ‘the failure of an overloaded node to initiate the procedure of load-balancing, or the unsuccessful logical movement of nodes that cause the overload’, the following scenarios highlight the weaknesses of the ERGO architecture:

- the virtual server has no registered underloaded nodes and the TTL of the request is 0;
- the virtual server fails to satisfy the conditions and although the TTL is non-zero, it has no neighbouring virtual servers in its Server List;
- the virtual server has found an underloaded node in its queue, but it is no longer underloaded. This can happen between two advertisements but the architecture will let the rewiring actions take place. It is assumed that this case is rare.

4 ERGO Overlay Maintenance and Concept Analysis

This solution introduces another overlay layer in top of Gnutella. There are several reasons why this solution can provide a flexible environment for various applications and network infrastructures. Heterogeneity characterises network infrastructures and this overlay aims to be adaptive considering different computer systems. Some scenarios have been identified concerning the type of machines that can manage the load-balancing actions and the particular conditions under which they can contribute to the performance improvement of the network. Two solutions are proposed about the problem of this overlay maintenance and issues are raised on how cost-effective ERGO can be in different systems and by considering client's side as well.

4.1 The virtual servers

The term *virtual server* is inspired by the nature of these machines that can be one of the following:

1. Dedicated powerful machines (servers or mainframes);
2. Clients with ERGO additional functionality;
3. Virtual Organizations.

In the first case, the network benefits from the classic client-server model by exploiting the high performance profile of dedicated machines that support the load-balancing requests. They may also have other roles in the network serving other autonomic self-* properties.

In the second case, if clients fulfil some minimum performance requirements, they can contribute to the ERGO overlay. ERGO client can be configured by the user to act as a virtual server. In order the user to be motivated to enable its client to act as virtual server, some special benefits can be given, such as higher download speed, access to more peers for better query success. In another mode, the client could decide by its own to act as a virtual server when utilising network configuration.

In the final scenario, the virtual server could be part of a virtual organization formed by more than one machine. The load-balancing actions are policy-driven and they are provided as services. Organisations define what these services can offer to other virtual organizations, or to clients.

4.2 Layer Maintenance

Two mechanisms are proposed for maintaining the ERGO layer. They follow different concepts and the investigation of their effectiveness is topic of future work.

The first mechanism benefits from the discovery messages and answers to build embedded advertisements of virtual servers within the messages of the overlay protocol. In the case of ERGO a field can be added in Queries and Queries Hits representing a list of virtual servers. The list is updated during their propagation and, specifically, when they visit a node. If this node is a virtual server, it adds itself to this list and also checks if there are any other nodes available to add to its Server List. In this way, nodes update their Server Lists dynamically during the searching process.

This solution does not introduce any additional messages. However, existing protocols need modification in their messages that also should carry additional information. The discovery success is closely related to the number of queries in the network. In this case, we consider the number of query generations and their TTL. The greater the number of queries in the network the better the virtual servers' discovery is. If the number of queries is relatively low, by definition, the need for load balancing is lower.

Alternatively, another solution could be incorporated in ERGO and not in the underlying overlay. In this method, the ERGO is initialised with all the nodes as virtual servers. This means that the Server List is equal to the Neighbour List. An alternative method is to fill the Server List from a bootstrap node. If one node stops acting as a virtual server (administration configuration, client configuration or policy change respectively for the virtual servers) and receives a request for help from an overloaded node, it sends an ERGO message informing the originator that it is no longer a virtual server. It also keeps the address of the node that it refused to help. The originator deletes the current virtual server and looks for another one in its Server List. If the configuration or policies change again, enabling the virtual server role of the node, it sends notifications (using an ERGO message) to all the nodes it refused to help that it is again available for them. If a node falls in the state being without servers, it can request available stable servers (server with low refuse rate) from its neighbours. In this way, the Server List is adapted even in extreme conditions.

This maintenance mechanism is part of future work. The messages specification, time convergence and switching modelling of nodes between the two overlays must be investigated. However, we believe that it corresponds well to our existing system, and can work efficiently.

4.3 Cost-effectiveness issues

Considering the cost-effectiveness of our solution from the Gnutella client perspective, there is no significant cost that could influence its performance. Advertisements are sent by underloaded nodes, the buffer between overload and capacity thresholds works as a prediction of performance bottleneck, and the only cost we should seriously consider is the updates in the Load Statistics Array. The number of updates is equal to the number of queries received. The ERGO algorithm alleviates this load exactly. This can be additionally translated into processing alleviation in nodes. Additional processing cost is attributed to clients as their in-degree increases.

The processing cost of the virtual servers depends on the following:

- their in-degree. It is related with the ratio of virtual server to the total nodes in the network;
- the in-degree of the overloaded nodes that belong to their incoming links (bigger Load Statistics Arrays to process);
- the convergence balancing time.

In ERGO, the investigation of the processing cost can be formalised be considering the cost generated by the additional structures that are introduced. Then the cost is quantified as the O(k) complexity accessing a structure. Following the above analysis, from the client's perspective, the cost for accessing the Load Statistics Array and the Server List is calculated. From the server's perspective, the cost of the previous two is taken into account together with the cost maintaining the consistency of the Node Queue. The calculation of processing cost is part of the experiments that were run. However, due to space limitations, graphs are not presented showing detailed results. Figure 7 summarizes the range of values for the two simulation environments that are described in section 5.3 with 30 and 300 servers. The values refer to the average cost per node per cycle.

	(300,10,50,60)	**(30,10,50,60)**	**(300,20,70,90)**	**(30,20,70,90)**
Client	[0.98, 2.84]	[0.97, 3.17]	[0.05, 2.90]	[0.96, 3.14]
Server	[0.02, 1.17]	[0.05, 1.70]	[0.03, 0.17]	[0, 1.63]

Figure 7: The range of processing cost in client and server for two simulation environments and different number of servers in the network

The numbers in the table represent the range of cost expressed as number of accesses in the new introduced structures of ERGO. The range concerns the minimum and maximum values during the experiments. The variation in the range depends on the number or queries generated and the rewirings that happen during the iterations. The column labels refer to the number of virtual servers (underlined number) and the load thresholds as they are defined in section 5.3.

From the table, it seems that the processing cost is insignificant. Even when the node acts as server the overhead is small. The number of servers affects the values, with fewer servers increasing the cost. This is because they receive average more requests for help as they have higher in-degree. The load thresholds affect more the server and to a small degree the client.

The communication cost of ERGO is extensively described in section 5.4 and in the last two graphs of Figure 12.

5 Simulations and Evaluation

This section presents the evaluation phase that was based on a Java implementation. The aim of the experiments was to identify the effect of ERGO system on Gnutella. The following subsections describe the simulator environment, analyse the metrics that were measured, the experimental environment and outline the results.

5.1 Simulator

The experiments and evaluation were based on a Java simulator. From development perspective, its uniqueness lies on the use of Java Data Structure Language (JDSL) [18]. From architecture perspective, we created an event-driven simulator with events corresponding to autonomic properties. The simulation maturity will be monitored in parallel with our future research.

The criteria of a valid simulation program and a valid simulation experiment are discussed in [26]. There are several well known simulation environments in research community, such as PeerSim [19]. However, our system and its features required a simulation environment with the criteria of *extensibility*, *configurability* and *interoperability*, which we have distinguished from the criteria list illustrated in [26]. These characteristics have been difficult to be found, in the required degree, in existing simulation environments. Furthermore, the use of JDSL not only provides a higher flexibility when using various data structures that do not exist in the standard Java library such as graphs, queues, dictionaries, etc., but also these structures create a high performance environment. For example, they provide best-possible asymptotic time complexity for every supported operation and caching. More about the architecture and implementation of the simulator can be found in [20].

5.2 Performance Metrics

The chosen metrics that are described below outline the evaluation of ERGO key features. The list is not an exhaustive one, though more results can be found in [20]. Considering the load-balancing aim of the system, some overloaded or fully loaded and underloaded nodes are expected to become balanced when using ERGO system. This possibly comes at a cost in messages. The experiments are also expected to give more insight into the ERGO effect on the query success rate. The metrics used are the following:

- **Node profiles**: apart from the profiles 'underloaded', 'balanced' and 'overloaded' defined by the load thresholds, there is one more profile: the 'fully loaded'. This refers to nodes that use all their capacity and cannot receive and process more messages.
- **Query Success**: measures the changes of the Gnutella query success rate.
- **Availability**: it is the percentage of the load of underloaded and balanced nodes over the total load in the Gnutella network.
- **Number of messages**: Gnutella and ERGO total messages (queries, queries hits and load-balancing messages) are calculated as a metric of expressing the communication cost of our proposed algorithm.

5.3 Environment Characteristics

A number of experiments presented in this section aim to evaluate ERGO performance and compare it with Gnutella I. ERGO affects not only the network topology but the performance of Gnutella as well. The initial configuration of the network and node profiles is as follows:

- 3000 Gnutella and thus ERGO clients as well. 300 of these are also ERGO servers. They are assumed to be stable. In some of the experiments the ratio of virtual servers/total nodes is varied as an examination of the effect in availability and query success.

- Neighbour and Server Lists have three Gnutella neighbours and ERGO servers respectively. All nodes have the same capacity and load thresholds.

- The flooding mechanism of Gnutella has a TTL = 3 and the 1-walker of ERGO overlay a TTL = 4.

- There are 50 000 files available in the Gnutella network, uniformly distributed among nodes, and described with 1000 keywords. Each file is described by one keyword. For the sake of the experimentation of ERGO functionality, the files are not replicable and thus when discovered they are not copied into the query generator.

- All experiments are executed in 100 iterations, simulating 100 time units. All the queries, both Gnutella queries and ERGO requests generated within an iteration, are completely processed. This means that the queries travel TTL steps within the network and all their corresponding messages are processed within the same iteration.

- In every iteration, a number of new queries (load) are generated and forwarded in the network. This load is not the same throughout the experiments, but has peaks, valleys and constant parts to simulate ERGO and Gnutella behaviour in more realistic situations.

- Each overloaded node sends a request to an ERGO server only once per iteration.

The two simulated environments differ in the capacity and load thresholds of the nodes. In the first experiment, the lower, upper and capacity thresholds are 10, 50 and 60 and on the second one 20, 70 and 90 respectively. The first one strains the ERGO functionality, whereas the second one provides more flexibility. Both of these configurations are also simulated on pure Gnutella network (without the use of ERGO) and compared to the ERGO-enabled one.

5.4 Results and Evaluation

By applying the above configuration of the experiments on the simulator, this section presents the measurements of the metrics and the results on a set of graphs. Most of them have two y-axes; the second right-side one is always used for displaying the total queries generated per iteration.

The first set of graphs refers to node profiles. Stack area plots are used to present the number of nodes for different node profiles on top of the other. Figure 8 illustrates the results for Gnutella and ERGO network with load thresholds [10, 50, 60] and Figure 9 with [20, 70, 90]. In these four graphs the second y-axis is also used for displaying the number of successful queries.

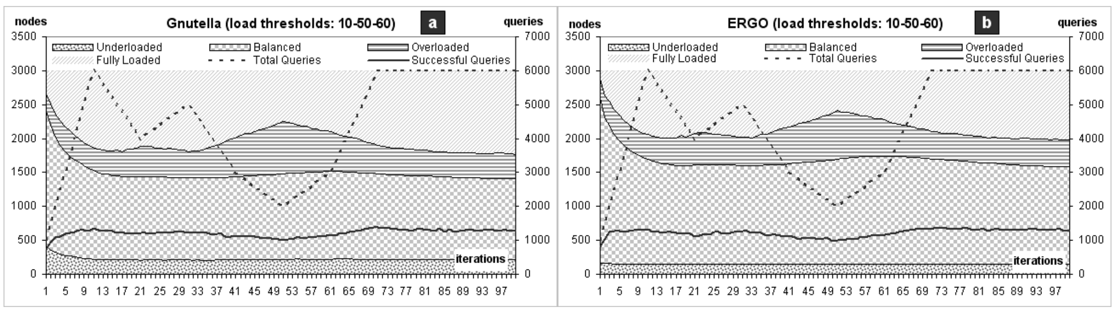


Figure 8: Node profiles on Gnutella and ERGO networks based on load thresholds [10, 50, 60].

In Gnutella system (Figure 8(a)), the overloaded and fully loaded nodes outnumber both the balanced and underloaded by approximately 10%. Enabling the ERGO system (Figure 8(b)), the underloaded and fully loaded nodes get reduced by 2.5% and 7% respectively for the benefit of the balanced nodes whereas the overloaded ones seem to remain the same. This reveals that underloaded nodes are discovered and exploited. Furthermore, equal number of nodes switch between full loading and overloading state. In load valleys, the overloaded and balanced nodes tend to increase. While the load decreases some fully loaded become overloaded.

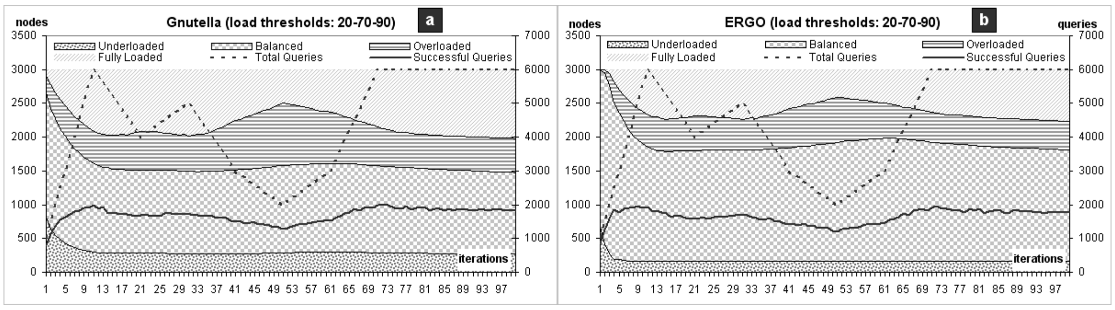


Figure 9: Node profiles on Gnutella and ERGO networks based on load thresholds [20, 70, 90].

Figure 9, shows similar results for the second experiment. However, the balanced nodes in both Gnutella and ERGO networks are respectively 1% and 7% more than in the first environment as the nodes have more capacity. The difference in balanced nodes between Figure 9(a) and 9(b) is 6% higher than between 8(a) and 8(b). Given the lower and capacity thresholds of nodes, ERGO system reduces the over- and fully loaded nodes 5% more than in the first experiment (Figure 8). This is because the higher the capacity thresholds, the more possible the load-balancing requests are to be satisfied by the higher number of underloaded nodes in the second environment. All the above four subgraphs present a small impact of ERGO on the search efficiency of Gnutella and the effect is clarified in the next experiments.

The availability measurements depict the network (global view) performance improvement and robustness of Gnutella within the ERGO enhancement. Figure 10 illustrates the measurements in the two different environments. In this experiment we varied the number of servers in order to view the effect of different ratios. The first observation is that ERGO availability is increased (depending on the server ratios) 3% - 34% and 6% - 25% compared to Gnutella in the two environments respectively. In every case, it is always higher. Another interesting point is that the Gnutella of the second environment, in which nodes have higher capacity, still has worse performance compared to the ERGO performance of the first environment, which is 1% - 32% better (depending again on the server ratios).

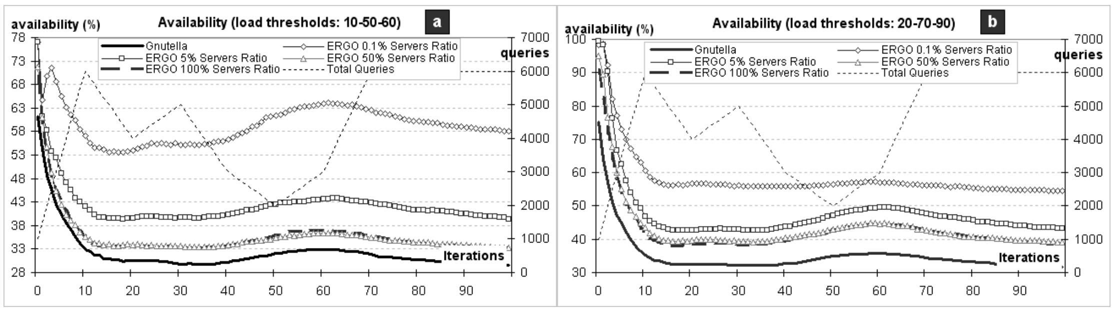


Figure 10: Network Availability in Gnutella and ERGO in the two simulation environments with varied servers/total nodes ratio

The reason for the different values under different server ratios is related to the knowledge of each individual server. When the number is low, the information about the available resources is more centralised. The server has a better global view of the network. This comes with a high processing and load cost in these machines. On the other hand, when the number of servers is relatively high, the information is more distributed, and each server has only a small fraction of knowledge about the network. This difference decreases 12% in the second environment. This is due to the fact that the network has more capacity and the ERGO exploits the flexibility to balance the overloaded peers.

Figure 11 presents the results for query success. The effect of ERGO on it is relatively small with a tendency of a slight reduction (0.43% - 0.8% average in the two environments) as the ERGO becomes more centralised (decreased servers ratio).

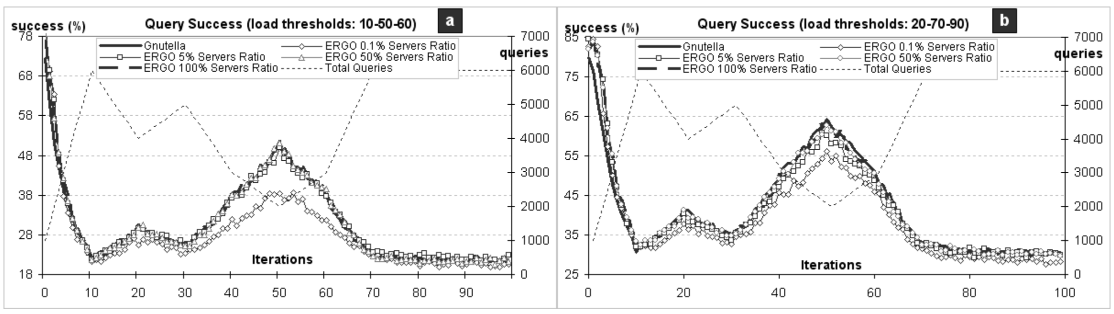


Figure 11: Query Success Rate in Gnutella and ERGO in the two simulation environments with varied servers/total nodes ratio

ERGO rewires load originators to underloaded nodes. The higher the lower threshold of a node is, the more such rewiring actions it can accept. Moreover, the higher capacity and upper threshold the nodes have, the less load overpasses their upper threshold. Therefore more small chunks of load are rewired. This process makes underloaded nodes with many incoming links popular, and thus more queries are dropped if they have already visited the same node. The effect seems to fade out as the server ratios decrease.

Comparing Figures 10 and 11, our conclusion is that there is a convergence value of servers ratio on which ERGO achieves the optimum performance (approaching Gnutella query success and having the maximum availability). In the experiments, it seems that this convergence value is 5% servers ratio. Further formalisation is a topic of future work.

The side-effect of rewiring, with the small reduction in query success, is explained in the following figures, as well. Figure 12 displays the number of messages generated throughout the experiments in both the Gnutella and ERGO systems, as well as the number of query messages transmitted in the Gnutella network.

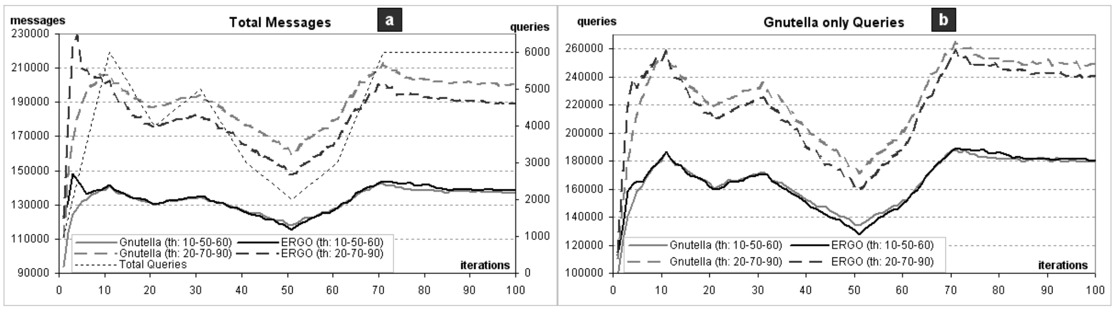


Figure 12: Number of messages and queries on both experiments for Gnutella and ERGO networks.

The total messages (Figure 12(a)) generated in the two systems are 25%-35% fewer in the first experiment compared to the second as the Gnutella queries are also almost equally fewer (Figure 12(b)) as well. This is explained by the fact that the fully loaded nodes that are responsible for dropping queries are greater in number and the system availability is lower. Figure 12 also shows that the number of messages is lower in ERGO system than in Gnutella. This difference is clearer in the second environment with nearly 10% decrease. As shown in Figure 12, the rewiring makes underloaded nodes more popular, especially when their lower and upper thresholds are relatively high compared to their capacity. This means that their in-degree increases, the queries revisit the same nodes and considering the queries drops Gnutella nodes perform when they receive the same query, their number decreases. On the other hand, in Gnutella without ERGO functionality, in-degree is not modified and remains random during simulations. Given the definition of the iteration in simulations, the ERGO messages can increase significantly if these requests are sent more frequently.

6 Conclusions and Future Work

Self-optimization is a vital feature towards Autonomic Computing. ERGO tries to apply this concept on unstructured P2P networks and uses a Gnutella network as a case-study. AGnuS [14] and Gia [1] are existing efforts to address the load-balancing issues of such networks but they do not seem to achieve this goal in certain situations.

ERGO builds an unstructured overlay of servers that keep record of the underloaded nodes and receive requests from overloaded ones. They rewire the nodes that produce the excess load on the overloaded nodes to move part of the load to the discovered underloaded ones. This technique allows more queries to be processed as the number of nodes that are fully loaded decreases. Simulations show that ERGO balances a significant number of nodes and equally benefits the system availability. Moreover, the query success is retained close to that of Gnutella. The rewiring process, under certain conditions, can make underloaded nodes popular acting like hubs. Thus, a query has increased probability to revisit the same underloaded node. In this case, this node drops the query and stops early the forwarding and as a result the number of messages decreases.

As documented in [10], the Gnutella topology has a tendency to become a star network that has high capacity, stable nodes in the centre. ERGO works as an opposing force to this tendency as it makes underloaded nodes more popular. Important steps for future work are the experimentation with a more dynamic topology (with node failures) and full Gnutella protocol simulation, so that the ERGO effect on Gnutella is more accurately analysed. The layer maintenance described briefly in Section 4 is also a topic of future investigation towards the further enhancement of ERGO. Another aspect of future experimentation is the variance of load thresholds in the network between different clients and different servers. New dimensions in the rewiring process can be added, including content or trust-based rewiring.

7 References

[1] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, S., Shenker, Making Gnutella-like P2P Systems Scalable, In Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (Karlsruhe, Germany, August 25 - 29, 2003). SIGCOMM '03, pp. 407-418, ACM Press, New York, NY.

[2] IBM, "An architectural blueprint for Autonomic Computing," Accessed: 30-09-2007, Ref. Type: Electronic Source <http://www-03.ibm.com/autonomic/pdfs>

[3] The Gnutella Protocol Specification 0.4, Accessed: 30-10-2006, Ref Type: Electronic Source < http://www.stanford.edu/class/cs244b/gnutella_protocol_0.4.pdf >

[4] K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim, A survey and comparison of peer-to-peer overlay network schemes, Communications Surveys & Tutorials, IEEE (2005), pp. 72-93.

[5] D. R. Karger, M. Ruhl, Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems, Theory of Computing Systems 39 (6) (2006) 787-804, Springer New York.

[6] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, I. Stoica, Load balancing in dynamic structured P2P systems, INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies 4, pp. 2253-2262, Hong Kong, China, 7-11 March 2004.

[7] Y. Zhu, Y. Hu, Efficient, proximity-aware load balancing for DHT-based P2P systems, IEEE Transactions on Parallel and Distributed Systems 16 (4) (2005) 349-361.

[8] M. Bienkowski, M. Korzeniowski, F. M., Heide, Dynamic Load Balancing in Distributed Hash Tables, Lecture Notes in Computer Science (3640) (2005) 217-225, Springer-Verlag.

[9] E. Cohen, S. Shenker, Replication strategies in unstructured peer-to-peer networks, SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications 32 (4). (October 2002), pp. 177-190, Pittsburg PA, USA, 19-23 August 2002.

[10] H. Chen, H. Jin, J. Sun, D. Deng, X. Liao, Analysis of large-scale topological properties for peer-to-peer networks, CCGrid 2004. IEEE International Symposium on Cluster Computing and the Grid, (2004) pp. 27-34, Chicago, Illinois, USA, 19-22 April 2004.

[11] G. Fakas, B. Karakostas, B., A peer to peer (P2P) architecture for dynamic workflow management, Information and Software Technology 46 (6). (2004) 423-431.

[12] E. Adar, B., Huberman, Free Riding on Gnutella", First Monday, Oct. 2000.

[13] D. Hughes, G.; Coulson, J. Walkerdine, Free riding on Gnutella revisited: the bell tolls?, IEEE Distributed Systems Online 6 (6) (Jun. 2005).

[14] D. Hughes, I. Warren, G. Coulson, Improving QoS for peer-to-peer applications through adaptation, FTDCS 2004. Proceedings. 10th IEEE International Workshop on Future Trends of Distributed Computing Systems 2004, pp. 178-183, Suzhou, China, 26-28 May 2004.

[15] J. Salter, N. Antonopoulos, ROME: Optimising Lookup and Load Balancing in DHT-Based P2P Networks, PDPTA'05: Proceedings of The 2005 International Conference on

Parallel and Distributed Processing Techniques and Applications, pp. 699-702, Las Vegas, Nevada, USA, 27-30 June 2005.

[16] G. Exarchakos, N. Antonopoulos, S. Salter, G-ROME: Semantic-driven Capacity Sharing among P2P Networks, Journal of Internet Research 17 (1) (2007) 7-20.

[17] Gnutella Web Caching System. Accessed: 29-09-2007, Ref. Type: Electronic Source http://www.gnucleus.com/gwebcache

[18] R. Tamassia, M. T. Goodrich, L. Vismara, M. Handy. An Overview of JDSL 2.0, the Data Structures Library in Java, 2005. Accessed: 01-08-2007, Ref. Type: Electronic Source http://www.jdsl.org/other_modules/overview/overview.pdf

[19] PeerSim: A Peer-to-Peer Simulator. Accessed: 06-12-2007, Ref. Type: Electronic Source http://peersim.sourceforge.net/

[20] E. Pournaras, Enhanced Reconfigurable Gnutella (ERG): Dynamic Performance Improvement of Gnutella Networks, Master Thesis, University of Surrey, 2007

[21] D. Takemoto, S. Tagashira, and S. Fujita, Distributed Algorithms for Balanced Zone Partitioning in Content-Addresable Networks, In Parallel and Distributed Systems, 2004, ICPADS 2004, Proceedings, pages 377-384, 2004

[22] O. D. Sahin, D. Agrawal, and A. E. Abbadi, Techniques for Efficient Routing and Load Balancing in Content-Addressable Networks, Fifth IEEE International Conference in Peer-to-Peer Computing, PAGES 67-74, 2005

[23] S. Ratnasamy, A Scalable Content-Addresable Network, PhD Thesis, University of California, Berkeley, 2002

[24] J. Winter and O. Drobnik, Peer-to-Peer Cooperation for Content-Oriented XML-Retrieval, International Workshop on Peer-to-Peer Computing for Information Search

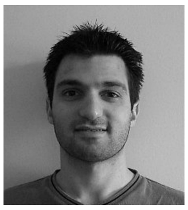
(P2PSearch 2007) at Future Generation Communication and Networking (FGCN2007), Jeju-Island, Korea, Dec. 2007

[25] MINERVA Project, Accessed 06-12-2007, Ref. Type: Electronic Source http://www.mpi-inf.mpg.de/departments/d5/software/minerva/

[26] N. Ting, A Generic Peer-to-Peer Simulator, In Proceedings of the 2003-2004 Grad Symposium, CS, Dept, University of Saskatchewan, April 7-8, 2003

8 Biographies

Evangelos Pournaras is currently a Ph.D. student in the Department of Computer Science in Vrije Universiteit. His research interests include self-organization, resource discovery, aggregation and optimization in peer-to-peer networks. Contact him at the Department of Computer Science, Vrije Universiteit De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands; e.pournaras@few.vu.nl



Georgios Exarchakos is a Ph.D. student in the Department of Computing at the University of Surrey. His research interests include network management and resource discovery in peer-to-peer networks. Contact him at the Department of Computing, University of Surrey, Guildford, Surrey GU2 7XH, United Kingdom; g.exarchakos@surrey.ac.uk

Nick Antonopoulos is currently a senior lecturer in the Department of Computing at the University of Surrey. His research interests include emerging technologies such as web services and peer-to-peer networks,

software agent architectures and security. Contact him at the Department of Computing, University of Surrey, Guildford, Surrey, GU2 7XH, United Kingdom; n.antonopoulos@surrey.ac.uk