Hoverlay: A Peer-to-Peer System for on-demand Sharing of Capacity across Network Applications

by Georgios Exarchakos

Submitted for the degree of Doctor of Philosophy
within the University of Surrey

Department of Computing
School of Electronics and Physical Sciences
University of Surrey
Guildford, Surrey GU2 7XH, U.K.

April 2009

To my parents and siblings

Acknowledgements

Throughout these three and a half years of my research project, Dr Nick Antonopoulos was my supervisor and mentor; an *always-there* person to discuss new ideas, problems and solutions. He is the person who very successfully guided me throughout my research; a sagacious and honest person who offered long hours almost every day on brain-storming. His trust and consistency were proved valuable characteristics for a prosperous collaboration and helped me guess his expectations before even we discuss matters. He was always there when I needed some help, a person to talk about both research and personal issues willing to provide all his support. No extra words would express my appreciation to Nick apart from one thing; I am now his son's godfather.

Apart from the daily communication I had with my supervisor, many people, colleagues and friends, have significantly contributed to my research with discussions, feedback and support in numerous occasions. I would like to particularly thank Kan Zhang, Athanasios Pavlou, Dimitris Fotiadis, Alexandros Marinos, Athena Eftychiou and Stefan Stafrace for spending some of their time to provide comments on my thesis. Finally, I am grateful to my parents and siblings for their endless help as none of the following pages would ever exist without them.

Many thanks to all and be sure that every single page reminds me your help.

Abstract

Heterogeneous distributed applications deployed on different networks may have variable network throughput requirements during their lifetime frequently swapping between underloaded and overloaded situations. This study proposes a model for on-demand capacity sharing among networks allowing their sizes to adapt based on their workload fluctuations. Its aim is to keep all nodes normally loaded and reduce the messaging cost of deployed applications and focuses on those with highly dynamic workload fluctuations, bursts of traffic and/or massive node failure rates.

Conceptual migration of capacity from one network to another may improve spare capacity utilization and network reliability even in high workload situations. Hoverlay is the proposed two-tier Unstructured P2P management architecture which realises capacity sharing between networks in three steps: *publishing*, *discovery*, and *commissioning* of capacity. While the first two refer to capacity publishing and discovery mechanisms the last one fetches and commissions it to the requesting network. This architecture facilitates the cooperation of heterogeneous networks each of which is represented by a single server on an interconnected server overlay.

The second contribution of this thesis is the proposed search mechanisms "Stalkers" deployed on Hoverlay to support the intermittent behaviour of service capacity. The idea behind this is the use of fresh and/or incoming server-to-server links so that queries can trace resource migrations. There are three variations of this technique: k-Stalkers, FireStalkers and FloodStalkers aiming at improving number of messages, query latency and success rate.

A number of experiments were carried out and showed that Hoverlay increases node utilization and allows the underlying network to resize based on its workload. Node (capacity) migration between networks saves the overlay from multiple queries and increases query success rate reducing its average latency if compared to the competitive Flock of Condors system. A number of experiments to evaluate Stalkers mechanisms showed that all the three variations of Stalkers achieve their aim in environments of highly dynamic resources such as Hoverlay. They outperform Flooding and k-Walkers in success rate and latency keeping the number of messages relatively low. The thesis concludes with its findings and contributions as well as proposals for further work.

List of Publications

Book Editing

1. N. Antonopoulos, G. Exarchakos, L. Maozhen, and A. Liotta, Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications, IGI-Global, 2009.

Journal Publications

1. R. Vinod, G. Exarchakos, N. Antonopoulos, "Performance-driven Optimisation of Gnutella Reconfiguration," *Journal on Peer-to-Peer Networking and Applications*, [accepted].
2. L. Exarchakos, M. Leach, and G. Exarchakos, "Modelling electricity storage systems management under the influence of demand-side management programmes," *International Journal of Energy Research*, vol. 33, 2009, pp. 62-76.
3. E. Pournaras, G. Exarchakos, and N. Antonopoulos, "Load-driven neighbourhood reconfiguration of Gnutella overlay," *Journal of Computer Communications*, vol. 31, Aug. 2008, pp. 3030-3039.
4. G. Exarchakos and N. Antonopoulos, "Resource Sharing Architecture For Cooperative Heterogeneous P2P Overlays," *Journal of Network and Systems Management*, vol. 15, 2007, pp. 311-334.
5. G. Exarchakos, N. Antonopoulos, and J. Salter, "G-ROME: semantic-driven capacity sharing among P2P networks," *Journal of Internet Research*, vol. 17, 2007, pp. 7 - 20.

International Conferences

1. G. Exarchakos, N. Antonopoulos, and K. Zhang, "Firewalks: Discovery Mechanism for Non-replicable Reusable Resources," *Proceedings of the Seventh International Network Conference (INC 2008)*, Plymouth, UK, 2008, p. 65.

2. K. Koukoumpetsos, N. Antonopoulos, K. Zhang, and G. Exarchakos, "Improving Availability of Mobile Code Systems by Decoupling Interaction from Mobility," *ICN 2008. Seventh International Conference on Networking*, Mexico: 2008, pp. 602-607.

3. G. Exarchakos and N. Antonopoulos, "Non-replicable reusable resources discovery on scale-free Peer-to-Peer networks," *DEST 2008. 2nd IEEE International Conference on Digital Ecosystems and Technologies*, Phitsanulok, Thailand: 2008, pp. 28-33.

4. G. Exarchakos, J. Salter, and N. Antonopoulos, "Semantic Cooperation and Node Sharing Among P2P Networks," *Proceedings of the Sixth International Network Conference (INC 2006)*, Plymouth, UK, Plymouth, UK: 2006, pp. 11-19.

5. N. Antonopoulos, G. Exarchakos, and K. Zhang, "Sharing Supply-Demand Trends for Efficient Resource Discovery in GRID Environments," *Proceedings of the Seventh International Network Conference (INC 2008)*, Plymouth, UK, 2008, pp. 125-137.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As a plethora of various distributed applications emerge, new computing platforms are necessary to support their extra and sometimes evolving requirements. Symmetrically, more powerful and diverse platforms provide material and motivation for new ideas and applications. This research derives its motive from deficiencies of real networked applications deployed on platforms unable to fully support their characteristics and proposes a network architecture to address that issue. Falling into the area of distributed computing, it focuses on sharing non-replicable intermittent resources over large-scale and volatile decentralised heterogeneous and multi-administrative environments utilising low cost, in installation and maintenance, platforms.

The throughput requirements of an application may largely vary over time and, thus, abruptly overwhelm the network with messages (*overloaded* situation) that an existing infrastructure could be unable to handle. This phenomenon may introduce delays in network responsiveness and deterioration of deployed services quality. On the contrary, low traffic leaves network resources under-utilised thus reducing their profitability (*underloaded* situation). In certain cases, services provided by two different networks may be in competition. That is, while one of them experiences high load, the other may keep all the equipment fully functional even if underutilised. Existing architectures try to improve the performance of a distributed application assuming that required resources *a)* are locally available, *b)* can perform as expected under well-defined conditions and/or *c)* rely on successful management. Distributed application architectures deployed on unreliable nodes usually focus on computationally heavy tasks and introduce redundancy on well-selected nodes to ensure task completion and load-balancing.

Transfer of spare capacity from an underloaded network to an overloaded one can be profitable for both networks. This thesis presents **Hoverlay**, a novel interconnection platform of different networks which provides appropriate mechanisms for sharing, discovering and exchanging of nodes on demand. Its design respects a potential heterogeneity of nodes and their ownership, takes over their management and aims at fast discovery at low cost. It is a network topology adaptation and resource discovery issue approached with logical movement of appropriate nodes from providing to requesting systems so that highly demanding areas of the network are quickly satisfied.

Its predecessor, G-ROME [42], was a more application-specific architecture based on the same idea of nodes migrating between networks to relieve workload bursts. However, G-ROME had a set of weaknesses largely addressed by CSOA (Capacity Sharing Overlay Architecture) described in [40]). This thesis presents a more complete view of CSOA and renames it with a more unique name Hoverlay. Its name is an indication of its functionality:*"proceed through an area in search of a prey or quarry"* [1]. It is an unstructured peer-to-peer (P2P) network built on top of other underlying ones which uses Stalkers distributed algorithms to locate resources; CSOA was not using specially designed search algorithms optimized for such environments.

Its business applicability may vary depending on the diversification and lifetime of services sought by requestors. Hoverlay design eases the support of short-term transactions between a variety of heterogeneous services. For instance, it is an architecture suitable for networked applications involving a collection of services, not necessarily the same, created to complete a single task; its lifetime coincides with that of the task it processes.

1.1 Defining Service Capacity

In the context of large-scale resource sharing, a resource, according to Vanthournout et al. [102], *"is any source of supply, support, or aid that a component in a networked environment can readily draw upon when needed"*. Yang et al. [107], define service capacity as a resource that represents the throughput of the network (the number of served requests per peer) in peer-to-peer file sharing environments where processing is not the predominant shared resource. They distinguish the *aggregate upload* from *aggregate download service capacity* with the former being *"the overall achievable throughput the system can offer to downloading peers interested in a given document"* while *"the average download throughput achieved per peer, which might be roughly estimated as the aggregated upload service capacity normalized by the number of downloading peers"* being the latter. Others (i.e. [54]), use just the term *throughput* as the aggregate download rate of file-sharing peer-to-peer applications. Both of these definitions assume that processing requirements are much less compared to bandwith ones.

In this research, service capacity refers to a more generalised context (not only file sharing) and is closely connected to the throughput of a network. In high load situations, more nodes able to receive, process and respond to requests are necessary to handle part of that load. Thus, a definition of *service capacity* that could fit in that networked environment would be the following: *service capacity is the number of requests a node can receive, process and respond within a time unit.* However, this definition, though comprehensive, is not normalised; network bandwidth measurements can be relatively objective but measuring CPU performance cannot rely solely (if at all) on its clock speed. Nodes with similar CPU clock may vary in their CPU performance but it gives a rough idea of its capabilities. With the aim of supporting networked applications

[1]stalk. Dictionary.com. *Dictionary.com Unabridged (v 1.1).* Random House, Inc. http://dictionarvery frequently y.reference.com/browse/stalk (accessed: November 06, 2008).

and their traffic, the proposed architecture assumes that bandwidth throughput is more important than CPU and that most modern CPU architectures can efficiently handle those applications. Therefore, service capacity is represented as a tuple of bandwidth and CPU throughput in kbits-per-second (kbps) and million cycles-per-second (MHz), respectively.

This definition helps the classification of nodes based on their current load. An *overloaded* node receives more requests for service capacity than it can handle whereas an *underloaded* one has almost all its capacity unoccupied. *Normally loaded* nodes are able to serve current load without any extra remote help but keeping their service capacity busy. A whole network is *overloaded* if one or more nodes are overloaded but there are not any underloaded ones to take on their excessive load and *underloaded* if the other way round. On any other condition, it is *normally loaded.* Networks can be in any of these conditions irrespective of the relation between their global load and capacity. For instance, a network with a single overloaded and all the remaining normally loaded nodes remains overloaded though its excess workload could be split among others without overloading them. Resources can take over extra capacity via Hoverlay only if underloaded. Nodes monitor their service capacity and detect their load status based on upper and lower thresholds set by their administrators.

The availability of service capacity depends not only on node failures but its usage, too. It is a resource that cannot exist in replicas and allows a single task at a time to access and use it; busy capacity is unavailable. As global workload increases within a system, the available service capacity becomes scarcer. Assuming, without loss of generality, that a node provides a single and unified portion of its service capacity, node and service capacity may be used interchangeably this point onwards.

1.2 Research Question

Under certain conditions, some nodes may have to serve traffic that exceeds their service capacity. Even if it is optimally used and the load uniformely distributed among nodes, new capacity becomes crucial in case this traffic continues to grow. Though all the resources within an organisation are well administered, they are limited; new investements are required to serve excessive load produced either by internal or external users. On the contrary, remote outer-organizational resources may be insufficiently maintained and administered but enough to serve extra workload. The problem addressed in this thesis is three-faceted: **publication**, **discovery** and **commission** of service capacity. *Publication* refers to resource representation and advertising, *discovery* relates with mechanisms deployed for locating resources and *commission* embody capacity migration from providing to requesting environments and the ways that capacity may be used. Therefore, the research question of this thesis is the following:

How can remote service capacity efficiently relieve on demand overloaded network nodes given its highly intermittent availability?

While High Throughput Computing (HTC) platforms and Grids have high operational and maintenance costs, existing Peer-to-Peer (P2P) architectures are designed and built to support a limited set of applications. Many Grid and HTC systems are mature technologies that realise efficient resource sharing but require well administered and maintained resources, they have weak scaling properties and/or are not resilient to intermittent behaviour of the nodes [62], [76]. P2P Networks can continue working properly even if some of their nodes experience frequent failures (fault-tolerance). They are networks built to support a specific application that can easily scale from small to large sizes sharing among nodes the cost of deploying such an application. Currently, a small range of P2P applications are supported concerning mainly content management (file sharing), parallel processing independent tasks and collaborative applications (instant messaging, VoIP, office applications, limited support on P2P Gaming) [78]. This research tries to relax scalability and applicability constraints of Grids and P2P Systems.

Unlike a P2P system, a Grid can support a variety of different applications but is not as scalable as a P2P network. Grids facilitate this diversification of deployed applications due to their generic advertisements of their resources and to their job submission services. In P2P networks there is no clear differentiation between the application and network configuration; the deployed application is a distributed algorithm dedicated to configure and manage the network. Hoverlay focuses on networked applications (e.g. P2P) and the load they produce rather than, though without precluding, heavy computational tasks. Service capacity is an appropriate resource for supporting such applications as it allows addressing their load via a job submission toolkit. For instance, if a P2P application is heavily loaded, fresh capacity may join its network by running its client piece of software. This resource exchange between different environments demands support of node heterogeneity for two reasons: *a)* shared service capacity is not fixed or guaranteed and the node providing it may have completely different specification compared to others, *b)* does not come from a similar environment (may have previously served a completely different application) as that of its requestor. Therefore, any system used to share such resources should be designed assuming certain specification of nodes, behavioural patterns of their users, well defined fixed and guaranteed capacity or specific characteristics of applications they served in the past.

A distributed application may experience traffic bursts due to sudden users' unexpected behaviour. An extensive survey from Streamcheck Research revealed that an increasing number of video streaming providers suffer from flash crowds especially in peak hours [1]. For instance, in top news providers like CNN, MSNBC and ABC, because of traffic bursts, the "breaking stories are often unreachable during peak viewing hours" and the users experience content outages and longer startup times. BBC announced that on 7th July 2002 the download time of webpages was eight times longer, a quarter of requests for webpages were not served and "*email traffic reportedly doubled in response to congestion on mobile networks*" [2].

Not only is there no system to adequately address the problem but also many network applications seem to experience extreme workloads, posing a high maintenance and administration cost. Instances of network application clients increase with the average connection speed and penetration of Internet into the global population. New users bring new behavioural patterns and cultural habits making application usage patterns more stochastic. The research question becomes timely considering the lack of a widely accepted solution to those problems.

Streaming servers and video repositories have high capital and installation cost, processing and bandwidth requirements for live and/or on-demand services and may become a substantial maintenance burden for several content providers. Hoverlay can dynamically allocate and use existing infrastructure of companies that build different applications on different networks. Using a Hoverlay server per application, capacity can be outsourced from one network to another based on their requirements.

1.3 Design Methodology

The Hoverlay design follows a step-by-step *problem identification→solution→evaluation* with feedback approach. Initially, a good problem and objectives definition as well as identification of most important parameters are necessary to set the scene of this research question, to decompose it into more fundamental problems and to identify any system design requirements. Analytical overviews of existing approaches to those problems provide useful information about what should be avoided, adopted or adapted. Finally, solutions proposed to any of these subproblems aim at improving system efficiency and should not generate new problems that would cancel any efficiency improvements achieved.

Evaluation is an important step since it may depict potential deficiencies of proposed solutions and provide an indication of a better approach. Results from the evaluation step may feed back to problem specification and solution steps with useful conclusions and allow them to appropriately adapt. Hoverlay is expected to be deployed on large-scale environments with high topology change rates. Experimentation in real environments is impractical because of the number of nodes, their heterogeneity, application and user behaviour diversity required. Therefore, its evaluation used appropriate simulations of such environments allowed quick parameter modifications and results collection.

Several existing P2P simulators were tested such as PeerSim [64] and P2PSim [52]. PeerSim is a Java-based simulator that started supporting unstructured overlays after the first simulations and experiments of the current research. P2PSim is a good P2P simulator but supports only structured networks and its authors stopped maintaining it the year the current project started. Therefore, a new simulator called *Omeosis* for hybrid Unstructured P2P Networks was built in C++, based on Object-Oriented programming principles. Though appropriate libraries of PeerSim for Unstructured networks appeared in the meantime, there was already enough material to use and build on top of first Omeosis versions. Swapping to PeerSim would require familiarizing with PeerSim libraries and converting the C++ code into Java; continuing developing Omeosis was a more convenient path. Omeosis is able to simulate random and simultaneous message delivery on random hybrid unstructured P2P networks, node migrations, different random topologies (scale-free, uniform random degree distributions) and has an extensibility manager to install and handle different searching and rewiring techniques.

The generic design of Omeosis also allowed the simulation of existing systems which are used as benchmarks for Hoverlay evaluation. Node migrations between networks interconnected via a Peer-to-Peer overlay is a main difference from other Grid-based systems. *Flock of Condors* (see section 1.4) is a well-known and mature system built with similar purposes but migrates jobs intead of resources. This worked as the benchmark to evaluate the main principles of the proposed architecture.

Stalkers is another facet of the current thesis' contribution; they are a set of search mechanisms designed to improve Hoverlay performance. Omeosis was further extended to support both Stalkers and existing search algorithms for the evaluation of the former in the context of Hoverlay.

1.4 Background Overview

Networking is about sharing resources by replicating, moving or remotely accessing them. Each network has different properties that depend on its topology. This section provides an overview of the basic features of architectures that enable resource sharing. Any centralized approach of sharing underlying heterogeneous and intermittent resources would suffer from high workload generated by frequent queries and advertisements of requested and available capacity respectively. High rates of leave/join actions of those nodes would cause extra significant update overhead. In case of a failure of the central manager, no capacity sharing would be possible practically disconnecting all resources. A type of distributed architectures that can scale to large networks using low-cost non-centrally managed resources (nodes) is Peer-to-Peer Networks. The nodes of these networks may belong to any user and are usually built on weak infrastructure. Their topology can be classified into the *Unstructured* and *Structured* ones: the former construct a network of arbitrarily and directly connected nodes whereas the latter a network of which the connections among nodes are determined by a well defined algorithm.

Structured P2P Networks assign keys to resources, identifiers to nodes and map those keys to specified IDs via a hash function. Thus, each node manages a certain range of keys with the resources they represent either hosting them or as their proxy to the overlay. Structured resource discovery mechanisms use this hash function to hash the requested resource key and locate its hosting node ID. P2P systems such as CAN [83], Chord [96], Pastry [87] and Tapestry [109] can guarantee successful discovery if the requested resource is available in the network within $O(log\, n)$ messages [74]. However, latency becomes a considerable problem in case of large networks (huge number of nodes) since each query is routed to the next intermediate nodes as far as possible producing long-distance network traffic and delays. Low-capacity peers can introduce further delays since they can easily become overloaded even if they don't process the query but just forward it. Query routing is based on simple keys and exact matching of key hashes and thus structured P2P overlays do not support complicated queries.

Unstructured P2P overlays organise the peers in a random flat or hierarchical graph which is decoupled from the location of the resources. Any unstructured overlay does not guarantee successful discovery even if the requested resource exists in the system but it usually supports complicated queries. Adopting a more distributed approach using Unstructured P2P Networks solves the single-point-of-failure and reliability problems of the centralised one. Replication of resources [75], [101] may increase the throughput performance of the overlay network since it increases the availability of the same resource [107]. Advertisements [110] or gossiping techniques may direct the query faster to the resource provider thus reducing its latency.

However, replication or gossiping/advertisement as well as informed resource discovery techniques of P2P Networks are not applicable to service capacity discovery. Gossiping techniques disseminate information about a resource (e.g. its location, capacity) assuming that it rarely changes. Service capacity may frequently change over time practically making this information rather unsuitable for advertising this type of resource for discovery purposes. In contrast to other resources (i.e. files), service capacity cannot be replicated but only reused. That is, only a limited number of users may use it at any given time [20]. These properties make it a dynamic with high failure rates resource and thus, its discovery difficult. Organizing underlying nodes in a Structured P2P would require the use of its lookup function each time a node joins the system resulting in a high maintenance cost.

Resource volatility is again the main reason why multiple advertisements widely distributed among nodes and informed search techniques are not preferred. Service capacity availability frequently changes over time and thus any advertisements or information about its status become quickly invalid or expensive to update. Hoverlay design is based on a decentralised indexing service, resources are registered with a single entity and the deployed discovery mechanism tries to guess resource locations without any assistance from special-purpose statistical data.

High Throughput (HTC) and Grid Computing (Grid) platforms are built to share any kind of resources focusing mainly on reusable ones. HTCs are aggregations of very powerful machines in a single or federated administrative domain and/or require good resource management [62]. Existing research on high throughput computing has produced several solutions to the issue of reusable resources discovery especially storage capacity and CPU cycles. Condor [73], [99] is one of the most mature high throughput computing technologies. It is a distributed job scheduler providing the appropriate services for submission and execution of jobs on remote idle resources even if they are under different administrative domains. A central manager receives the advertisements of available resources and tries to submit the queued jobs to the appropriate ones, which report back to the manager their execution state regarding each job. The central manager along with the idle resources constitutes the condor pool.

Flocking [39], [38] was introduced to statically link several condor pools and share resources between them. It requires a manual configuration of neighboring pools thus limiting the adaptability of the system in case of dynamic changes in resource availability. It is also assumed that pool managers run on reliable machines [29] since their failure can prevent execution of new jobs. These constraints can be relaxed with a Pastry [87] self-organizing overlay of Condor pools [29]. The Condor pools are organized into a ring and proactively advertise their idle resources to their neighbours so that they can choose these advertised resources whenever necessary. Unfortunately, this P2P-based flock of Condors requires a substantial maintenance overhead for updating the proximity-aware routing tables since it is based on the advertisements of available resources. If the availability of the resources very frequently changes, these updates need to be also frequent and therefore introduce high maintenance costs.

An important feature of Condor Flocking is that the execution machines are always managed by the same managers. Thus, every new discovery of similar remote resources by the same manager follows the same procedure. Given that the required service capacity could frequently exceed the locally available one, a local manager would forward equally frequent queries seeking almost the same amount of capacity; thus resulting in a significant number of messages.

The benefits of P2P overlays [74], [14] for the discovery of reusable resources have been identified and used in P-Grid. P-Grid, identifying the update overhead posed by resource advertisements on DHT-based overlays, uses a tree-based distributed storage system for maintaining them [5]. Resource providers locate in this tree the requestors they can serve and offer themselves for use. While other structured P2P networks hash the indexing keys, thus limiting searching capabilities, P-Grid enables complex queries. The organization of this overlay raises a number of concerns about its scalability in case of large highly dynamic networks since an update action of one advertisement could propagate to many peers. OurGrid [26] is another effort to combine the maturity of Grids in resource sharing with the benefits of P2P Networks on decentralised resource discovery.

Sensor Networks is also a field that uses the benefits of P2P Networks to achieve reliable cooperation of networked sensors. Recent research on P2P-based end-to-end bandwidth allocation [31] proposes a wireless unstructured overlay of sensors. Initially a central peer possesses all the bandwidth and distributes it on-demand via queries broadcasted to all peers. Gradually, this centralised architecture converges into a pure Unstructured P2P network as bandwidth moves away from that central peer. This system cannot be applied to the case of service capacity sharing since it makes an important assumption: the available bandwidth within the whole network is known a-priori and initially centrally stored. Requests for bandwidth are one-hop broadcastings; in wireless environments these broadcastings introduce the same cost in messages either aimimg at reaching one or more nodes.

All the systems described above are efficient in the context they were developed for but they are insufficient in the context of service capacity. Network characteristics may change extremely fast so that any advertisement and/or indexing scheme could result in frequent updates with a high cost on messages.

1.5 Architectural Principles

Any system aiming to share service capacity over large-scale heterogeneous environments has to follow some guidelines and respect certain constraints. Hoverlay is to be deployed on a network of random nodes without guarantees for proper administration and predictable usage patterns of their local resources.

Its acceptance depends on, among other factors, respect of resource ownership. Resource owners need to feel in control of their own resources and allowed to use them whenever necessary. Tasks originated from a local node are to be executed in priority against remote ones as that would save the network from frequent invocations of discovery mechanisms. A node may also abruptly withdraw from a network without any notification to any other entity violently terminating current tasks. Hoverlay outsources free (unused) only service capacity to avoid conflicts with tasks currently in execution status; a way to monitor resources availability is necessary. As soon as a task from a local node arrives, some of the current tasks may be terminated, if a node becomes overloaded, so that the local resources are provided back to their owner. Meanwhile, the discovery mechanism is triggered to seek for extra service capacity to handle that excessive load.

Large-scale networks of heterogeneous nodes may experience unpredictable workload; two separate areas (set of nodes within a radius from the center) may serve significantly different load. This is a characteristic with which Hoverlay needs to deal and efficiently achieve service capacity transfer between these two areas. Though they may be quite distant, their bridging and capacity migration needs to be completed as fast and with as low number of messages as possible. Moreover, nodes with frequent requests for extra capacity may significantly increase message costs and thus, they should be served faster than others. Ideally, shared capacity distribution should follow that of requests within the system so that the per-node success rate is uniform across network.

Resource indexing should avoid centralisation. A centralised indexing service may help on fast resource discovery but suffers from a number of problems: it is a '*single-point-of-failure*', traffic '*bottleneck*', expensive to maintain and upgrade. Sophisticated equipment to handle numerous simultaneous requests and highly qualified personnel to ensure security levels and QoS may be required. Potential failure of such a centralised entity would cancel out any communication between nodes and resource discovery becomes impossible. Frequent (as is the case given the service capacity volatility) join/dis-join actions and/or capacity requests from nodes may overwhelm that central server with messages causing latency increase and quality of service deterioration. Decentralised architectures spend more messages to seek for required resources, they do not always guarantee discovery even if those resources are available but are more robust in node failures and all produced traffic is distributed among many nodes.

The proposed architecture has to scale well and achieve good success rate by keeping the messages per request rate as low as possible despite any abrupt changes to network topology. The time interval between a request initiation and a response delivery (*latency*) should also be as low as possible. Otherwise, not only would the quality of service deteriorate but a response would be useless if arrived after the load drops back to normal, too.

1.6 Proposed Approach

As described above, any proposed answer to our research question has to provide an answer to several parts: *publication*, *discovery* and *commission* of service capacity. While resource publication and commission relate mainly to the overlay architecture, resource discovery is an algorithmic problem loosely-coupled to that overlay. Thus, this report approaches these two facets of the research question separately but not independently. Though a variety of discovery mechanisms may be deployed on the same overlay, only some can perform well. Therefore, the Hoverlay design aims at an architecture which takes into account all special features and behaviour of service capacity and eases its fast discovery as analysed before.

It introduces the idea of collaboration between networks, not just between nodes. A network that experiences low workload may provide (outsource) on demand underutilised nodes to another network with overloaded ones so that the former take over all the excessive workload of the latter. All underloaded nodes register with the Hoverlay which overloaded ones use to seek their required excess service capacity via sending appropriate queries. Once the appropriate nodes are discovered, they move to the requesting network. Each node can be used only by one network at a time.

The proposed overlay is an unstructured Peer-to-Peer network of servers. A server represents an underlying network and acts as mediator between service capacity requesters and providers. There is a one-to-many relationship between a server and nodes of its underlying network; it is the single place of the overlay with which an underloaded node registers and from which an overloaded one starts a request. Servers try to satisfy requests with a subset of their registered underloaded nodes and if that is not possible, they forward the same request to their own neighbouring servers. Discovered nodes are transfered from responding to requesting server which becomes their new and only portal. Subsequent registrations of those nodes are directed to that new server until, eventually, they are moved again upon request. Node mobility from one server to another (i.e. from one network to another) is, in fact, equivalent to the migration of its access control to another server.

P2P Overlays support heterogeneity of underlying networks and can reorganise themselves based on load and request distribution even in the presence of frequent server departures and arrivals. Specifically, unstructured ones introduce very low cost (in terms of messages) from random server/node failures while remaining resistant to graph fragmentations. Servers of Hoverlay may implement access policies on nodes of their underlying network which may vary from transparency to highly constrained accessibility; that gives network administrators control over their resources and deployed applications. Node mobility and its registration to the local server increases the probability subsequent requests from an underlying node will be served quickly using local underloaded resources. Moreover, this scheme helps underloaded nodes move to areas of the overlay which produce more requests and thus their distribution be adaptive to the one of global workload.

Therefore, node (service capacity) location has a short lifetime since not only does it allow exclusive access but its access rights are mobile, too; that makes its discovery more difficult. Stalkers is a search algorithm specifically designed to tackle service capacity intermittent behaviour on unstructured P2P overlays. Its principal idea is that request forwarding from server to server should track the movements of nodes. Old provider servers lose their available service capacity as they attract more requesters and links. Based on Stalkers, each server has *outgoing* (created by the server itself upon receiving an answer) and *incoming* (created by others upon receiving an answer from that server) links. This is a collection of search mechanisms based on modifications of k-Walkers using outgoing links to discover recent providers and incoming links for recent requesters which probably have gathered new service capacity and discovered new providers via their latest requests.

The Hoverlay environment was parametrised and evaluated based on a set of metrics. The parameters and metrics that were used are shown below.

1.6.1 System Parameters

- **Failure Rate** represents the join/leave rate of nodes. This rate in P2P networks is not stable either on time or per node. As detailed in [89] and [93], the on-time of a node may vary significantly and may also depend on connection speed of the node.

- **Connectivity Degree** is the number of neighbours of a node. Every node is directly connected to a number of other nodes to forward and receive requests to and from. The connectivity degree influences the success rate of the architecture [89].

- **Network Size** is the number of nodes of a network. The number of requests in an overlay usually increases with that overlay size and, thus it may affect bandwidth consumption and success rate. Depending on the overlay topology, its size may have little effect on the success rate if the Time-to-Live (TTL) (below) parameter is used.

- **Time-To-Live (TTL)** is the maximum number of steps a query can do within an overlay. This sets an upper limit to number of messages generated by the deployed discovery mechanism (i.e. Stalkers). If a query cannot reach the network edges because of the TTL, then its success is not guaranteed even if an answer exists in that network.

- The **search techniques** [74] and [101] are methods used for the forwarding requests from one server to its neighbours. These techniques are a major factor that influences the number of messages produced in the overlay per query. The bigger the TTL the further the termination horizon of query propagation is and thus the more messages per request are spent.

1.6.2 Evaluation Metrics

The behaviour of the proposed architecture will be evaluated based on the following factors:

- **success rate** is the percentage of successful (answered) requests over total number of requests.
- **satisfied capacity** is the total amount of requested service capacity that was satisfied by the system.
- **number of propagated messages** produced by the discovery mechanism.
- **latency** is a qualitative metric that measures the time elapsed from the moment a query leaves a requestor untill the first answer is returned back to it.

1.7 Innovation and Contribution

This research is based on three innovations in the area of resource sharing and discovery. Existing research on dynamic bandwidth allocation in networks focuses on improving bandwidth allocation and load-balancing techniques. However, this research develops a system that discovers new remotely available capacity that can efficiently serve a portion of the excessive requests at a requestor node.

The innovation of our approach to the research question lies on logical migration of nodes from overloaded to underloaded networks. This logical movement reduces reallocation latency of the same or similar free resources to the same network, thus very efficiently supporting frequent small fluctuations in capacity requirements of that network.

Finally, the basic principle behind Stalkers discovery mechanism is innovative since its aim is to detect resource movements (migrations between networks) without using any additional information but rather links lifetimes. It tries to efficiently use both incoming and outgoing links to increase the probability of locating available resources.

Grids and P2P Systems are two resource sharing paradigms which assume resource availability guarantees or deal with different in nature resources. Service capacity is a reusable and non-replicable resource with highly intermittent availability. Commercial P2P systems deal with mainly replicable (file sharing) or consumable (IP Telephony) resources. Shared nodes serving network traffic cannot provide guarantees that they will keep offering the same capacity throughout their lifetime and, additionally, there is no guarantee that this traffic will have a specific pattern and/or CPU, memory requirements. This makes service capacity significantly different from resources in Grid or P2P environments.

This research contributes to the scientific community by identifying and analysing special features of reusable non-replicable resources, comparing them against replicable ones and proposing a set of guidelines for sharing them. Though, Bedrax-Weiss et al. [20] have pointed out these differences, they do not provide any design principles as to how distributed systems should deal with such resources.

1.8 Thesis Layout

Hoverlay is an architecture based on unstructured P2P Networks adopting ideas from Grids and as such it uses the new search algorithm, Stalkers, for resource discovery. The whole thesis is organised in the following seven chapters:

- *Chapter 1 - Introduction* anything upto the current chapter analysing the research question,

- *Chapter 2 - Resource Sharing Environments* offering a literature review of distributed resource sharing systems and the topological requirements and principles of Hoverlay. This chapter starts with the identification of those service capacity features that make it a special resource for sharing in dynamic heterogeneous environments. An analysis of existing resource sharing systems follows focusing on P2P topologies, their discovery mechanisms and their combination with Grids for supporting computational resources. It provides a classification into two categories (blind and informed search) and overview of existing mechanisms used for resource discovery in P2P networks. It also proposes a further subclassification of informed ones based on the source and maintenance of the information they use for query migrations. It finishes with an enumeration of drives and principles based on which the proposed architecture, Hoverlay, is designed.

- *Chapter 3 - Global ROME: Preliminary Model* presenting a preliminary model, G-ROME, which facilitates resource migration between Chord-based underlying networks. It provides the specification of that system and a preliminary evaluation through simulations. However, G-ROME exhibits some weaknesses which are analysed and provide the motive for the Hoverlay design.

- *Chapter 4 - Hoverlay: Architecture Specification* analysing the proposed architecture with its protocols evaluated under various environmental settings and existing search algorithms. It is a detailed description of the Hoverlay specification including its components and their interaction protocols. The chapter continues with its evaluation via simulations and comparisons with Flock of Condors. In high load situations, Hoverlay outperforms its competitor (Flock of Condors) with regards to success rate and query latency without introducing much higher cost in messages.

- *Chapter 5 - Tracing Resource Migration* which analyses the proposed search algorithm evaluated in combination with Hoverlay in various network settings. Based on the literature review of chapter 2, this one starts with the requirements of an efficient search algorithm appropriate for sharing service capacity in decentralised dynamic and heterogeneous networks. It continues with its, Stalkers, generic design and provides three variations depending on the metric it aims at optimizing. After extensive simulations and comparisons with existing search techniques, the experiments show that Stalkers achieve their targets.

- *Chapter 6 - Conclusions and Future Work* concludes this thesis with an overview of its contributions and findings from all the experiments carried out before. It closes with a number of options for extending or improving both Hoverlay and Stalkers efficiency.

Chapter 2

Resource Sharing Environments

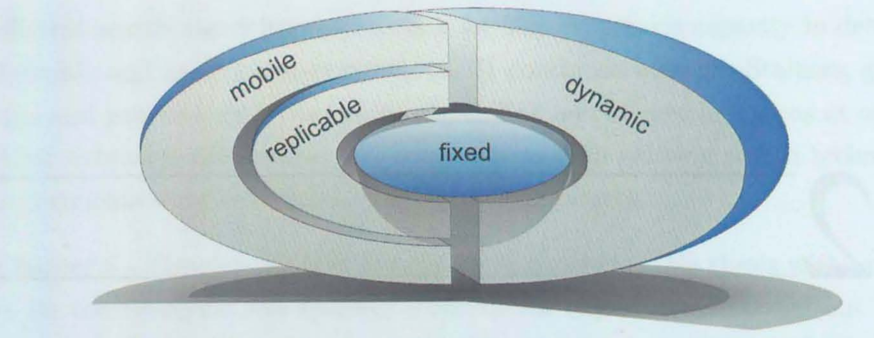
While previous sections offer a clear view of the research topic, this chapter provides an overview of existing work on related issues positioning current research in the appropriate context. It gives important details about related systems and identifies potential shortcomings of existing mechanisms with similar aims. This review is essential to specify a complete list of required design principles and features of the proposed system, Hoverlay, which will help sketching out its architecture.

Initially, this review depicts some special behavioural patterns and features of the resources this study deals with, *service capacity.* It, then, gives some background information on Grids and P2P Networks, two principal distributed resource sharing paradigms. While the former support sharing of resources like service capacity, the latter facilitate the collaboration of nodes in large-scale heterogeneous environments. An extensive analysis of P2P Overlays giving an emphasis on Unstructured ones (a subclass of theirs) and their resource discovery mechanisms precedes a section about attempts to apply their benefits on Grid systems scalability. This chapter tries to address the following questions:

1. how different service capacity is versus other resource types,
2. which existing systems could be used for sharing service capacity and what their weaknesses are,
3. which discovery mechanisms are available for searching resources in decentralised networks,
4. what requirements a service capacity sharing framework has.

It concludes that a decentralised overlay architecture, *Hybrid Unstructured P2P Overlay*, can work as the basis for Hoverlay in order to satisfy the special characteristics of service capacity. Though a classification of and details about existing discovery mechanisms deployed on similar networks are presented within this chapter, the requirements analysis of the proposed one, Stalkers, appears in Chapter 5.

According to [62], any central control of the entire network would limit its scalability and fault-resilience. These two features are the primary targets of its design but



Source: Vanthournout et al. 'A taxonomy for resource discovery' [102]

Figure 2.1: *Vanthournout's Resource Classification*

centralised architectures are weak on both and therefore, practically, cannot constitute a design basis for sharing service capacity.

2.1 Classification of Resources

Every member of a distributed system is involved in some sort of resource exchange. Optimization techniques focusing on the efficiency of a system need to use special characteristics of all exchanged resources. Therefore, understanding the nature of different resource types helps the identification of appropriate resource sharing systems for each one of them. Previous studies provide resource type groupings based on a number of characteristics.

Vanthournout et al. [102] propose a classification of resources based on how frequently their location and properties change. While the **fixed** ones have fixed unique location and/or properties, **replicable** ones have multiple replicas which, in some cases, may be fixed. When resources location is variable they are **mobile** even if replicable or with fixed properties. Resources with variable identifiers and/or properties are **dynamic** though they may have fixed location, too. **Mobile and dynamic** resources refer to all of the above. Figure 2.1 visualises this resource classification.

Holt [59], based on whether a resource can be reused or not, identifies two classes: **reusable** and **consumable**. A reusable resource represents a total number of capacity units each of which can be assigned to a single task at a time. As soon as the task terminates, another one may reuse the same capacity. A task may use and release any portion of the available capacity but no more than the maximum available. For instance, CPU cycles, bandwidth or buffers can be reusable resources. Consumable resources have unlimited number of capacity units; a producer may provide any number of units which disappear from the system as soon as a task consumes them. IP packets or P2P queries are examples of consumable resources. Their fundamental difference is that consumable resources can be used only once whereas the reusable ones may, sequentially only, serve any number of requestors. Therefore, reusable and consumable resources are two mutually exclusive sets.

Bedrax-Weiss et al. in [20] include more classification factors, besides the ones iden-

Factor	**Classes**			
lifetime	consumable ✓	producible	replenishable	reusable
quantity	discrete ✓	continuous		
divisibility	single ✓	multiple		
availability	fixed	variable ✓		
certainty	deterministic	stochastic ✓		
access	shared	exclusive ✓		
aggregation	pooled ✓	non-pooled		

Table 2.1: *Bedrax-Weiss Resource Classification and Service Capacity features*

tified above, and provide mutually exclusive subcategories for each one. Differences on resources lifetime factor give four categories: **consumable** and **reusable**, as analysed before, and **producible** or **replenishable**. While producible resources are produced but never used or consumed within the same system (e.g. human wastes from a household perspective without installed recycling systems), replenishable ones are produced and/or consumed simultaneously or at different times (e.g. human wastes in an urban environment with recycling systems). The quantity of a resource capacity may either be **discrete**, if produced/consumed in chunks only (e.g. IP traffic divided in packets with a minimum length multiple of 1 bit), or **continuous**, if any amount of it (e.g. water in a tank). In terms of divisibility of resource's capacity, it may be **single-**, if consumed as a whole (e.g. an IP packet), or **multiple-**capacity one, otherwise (e.g. hard disk storage space). Resource capacity variations over time give two more categories: **fixed** (no change) and **variable** ones (e.g. an video file and stream respectively). With respect to the accuracy of the capacity measurement, **deterministic** resources (e.g. number of bits in a IP packet) have well-defined capacity but **stochastic** ones (e.g. energy of a electromagnetic photon) a more probabilistic approximation. While **shared** resources (e.g. files in read access mode) can be used by many activities at the same time, **exclusive** ones (e.g. a video stream) by a single requestor at a time only.

Smith and Becker in [94] also distinguish two more categories of resources: **pooled** and **non-pooled** ones if they collectively can compose a bigger resource (pool of resources) or not, respectively. Resources pools, as defined in [94], are resource aggregators and can be viewed as a resource whose capacity is allocated to multiple activites at the same time. Table 2.1 presents the resource classification as proposed by Bedrax-Weiss et al. in [20].

2.2 Features of Service Capacity

This research focuses on sharing service capacity which involves description, publishing and discovery of resources. Though service capacity is tightly coupled to capabilities and location of hardware, its description and publication site are decoupled from any physical location. The current study deals with both its physical instance and reference to that instance. Resource references may be description files (e.g. adverts, specifications, pointers) or database tuples which inherit part of their behaviour from their physical pairs (resource physical instances). Based on Vanthournout's classification, files are replicable and potentially mobile resources whereas computational ones (i.e. CPU, memory, bandwidth) cannot move or exist in replicas but have very dynamic properties. A description may be mobile but there should only be one instance within the whole system as, otherwise, access to that resource would be uncontrollable and the resource overwhelmed by rejecting multiple simultaneous requests. Service capacity is a dynamic resource due to frequent load fluctuations; so is its description since the latter has to reflect important properties of the former. While some physical resources have fixed location their descriptions may not. Following Vanthournout's grouping, service capacity (using its physical instance and references into a bundled form) may be classified into any category apart from the 'replicable' one.

Bedrax-Weiss classification of this resource type helps on the identification of more features important for the system design. Service capacity is associated with computational resources of the network and as such it can only be a reusable resource since e.g. CPU or bandwidth may be reused by a task as soon as the previous releases it. It is also a discrete resource since nodes cannot provide and/or request any amount of shared service capacity but only chunks of it. It may also be partially consumed or as a whole. Given the non-deterministic fluctuations of resource owner's load, it has variable stochastic availability and allows exclusive only access since only a single task at a time may use it. Not only may a node share a set of chunks of its capacity but references to those chunks may be gathered in pools, as well. All features of service capacity based on this classification method appear in Table 2.1 marked with a tick on their right side.

There are two more characteristics successfully depicted by Bustos-Jimenez et al. in [28]. Frequent fluctuations of network load may cause equally frequent requests for extra capacity shortening the lifetime of resource references, too. Short response times are essential to improve Quality of Service as perceived by users. Moreover, deployed search mechanisms look for resources that fulfill a set of requirements ignoring their unique identifier. The aim is to get the requesting task done on time no matter where the extra capacity comes from. Yang et al. [107] have successfully analysed how service capacity evolves with the load of a network in a context of replicable resources. While replication of resources improves their availability and thus service capacity, Hoverlay shares fixed system-wide non-replicable capacity; regardless of the workload, the available capacity is limited.

The source of such heterogeneity is not only the diversity of shared resources but network policies, as well. While files are resources with fixed quality once created, the quality of service capacity depends on system's maintenance. A resource becomes more reliable if it is frequently updated and offers predictable availability. System administrators who want to offer high Quality of Service try to do frequent updates and upgrades and ensure stable availability of the resources. Improving the reliability of a resource usually aims at a profit and leads in a legal binding with consumers [62]. On the contrary, some administrators (e.g. owners of home-office networks), with the aim of maintenance cost reduction, offer out of date resources with poor availability.

We assume that the resources are not always properly updated, upgraded and available to resource requestors. They may fail causing any ongoing activity to abruptly terminate. Their requestors and providers may belong to different administrative domains, have not any incentive to share and are not bound to any regulations; therefore, no sanction schemes can be applied.

2.3 Grid Job Scheduling

Sharing resources is an issue well addressed by high performance computing (HPC) systems. They aim to provide vast, highly available, fault-tolerant computational resources [99]. While Parallel and Distributed Computing platforms enable the aggregation and sharing of several well and centrally administered resources within the same administrative domain, Grids organise resources under various sharing policies and domains. Therefore, Parallel and Distributed systems can not constitute the basis for designing a multi-administrative system for dynamic and reusable resources. However, Grids satisfy some of the constraints set in section 2.2 since they aggregate cross-organisational resources in Virtual Organisations [48].

Successful examples of massive resource sharing and parallel processing are the seti@home [12], genome@home and folding.net [70]. They construct star networks (client-server) to share CPU cycles of nodes running bags of independent tasks but are prone to man-in-the-middle attacks and server failures [15]. A single central entity (server) submits jobs onto underlying machines and all the communication goes to it; nodes work in isolation from each other. Though this topology is tolerant to node failures it has scalability problems because of the central server which has to deal with node registrations, sometimes data-intensive task distribution and results aggregation.

Grids can be distinguished in computational, data and service grids [69]. Data Grids provide the suitable infrastructure for large data management and mining applications. Service Grids aggregate resources, any single machine is unable to provide, into composite services and Computational Grids gather computational capacity much higher than the capacity provided by a single machine of the Grid.

Grid Computing architecture designs assume that shared resources are powerful, diverse, well-administered and their network connection very reliable [48], [62]. The aggregation of appropriate resources in Grid Virtual Organisations has to achieve a certain point of reliability which may require highly reliable resources. The administration policy of Grids is also well defined and centrally controlled ensuring high and uniform availability of resources but high maintenance costs, as well. According to Iamnitchi and Foster [62], Grids deploy a well-administered infrastructure, the use of which has to be profitable for the owner. This may involve human negotiations which may be time-consuming procedures and practically require a certain level of trust between the producers and consumers (contracting parts), thus, reducing the scalability of the system.

Established Grid architectures, among others, are the Globus [47], Legion [56], PUNCH [66] and Sun Grid Engine [51]. They achieve control of ongoing activities via centralised or hierarchical entities [15] which decrease their scalability, adaptability and availability. Therefore, these systems violate the requirement for high scalability and fault-resilience.

While Grids offer good coordination mechanisms for resource sharing, they seem to be weak in the discovery of highly dynamic resources. These platforms moved towards a more decentralised approach to resource discovery to address the problem. For instance, Globus discovery mechanism moved from the centralised indexing MDS to a decentralised one MDS-II (a number of distributed index servers), which, however, offers limited scalability [35].

2.3.1 Condor

Condor is one of the most mature high throughput computing technologies and supports cross-organisational dynamic resource sharing and job scheduling [73], [99]. It is a distributed job scheduler providing the appropriate services for submission and execution of jobs on remote idle resources even if these are under different administrative domains. A central manager receives advertisements from available resources and tries to submit the queued jobs to appropriate resources which report back to manager the execution state of each job.

A central manager along with a set of idle resources constitutes the condor pool which records availability information (using ClassAds) about each resource. These resources though they may be located in different administrative domains, they are considered as local from their manager perspective. Resource requestors submit their jobs in ClassAd description to their managers, which match those ClassAds with their local resources. As soon as a suitable resource is found, the job is allocated to it. Condor provides services for job resubmission or migration via a check-pointing scheme in case of resource failures. Both submission and execution machines run specialised Condor daemons. All resources (busy or free) inform the manager about their status using heartbeats.

However, each manager is a priority-based system and may access only local resources. A job waits in a priority queue till it gets at its top and the central manager finds a match with any of its available resources. Therefore, flocking [38], [39] was introduced to statically link several Condor pools and share resources between them. Condor Flocking aims at increasing resource availability and system efficiency via creating communities of pools. A neighbourhood of pools (interconnected pools) is configured manually thus limiting the system adaptivity to dynamic changes in resource availability and to failures of neighbouring managers.

There are two kinds of flocking techniques: the private and the group flocking [39]. In private flocking, if a job is not satisfied its central manager submits it to every pool of the flock. As soon as a pool allocates a resource for that job and its execution starts, it sends a signal to all the other pools to prevent them from duplicate execution. Once a job execution terminates, that job leaves all the queues in which it is placed. Group flocking uses advertisements of jobs (source-initiative) or resources (server-initiative) to other pools. The final decision as to which resource will be used for the execution is on the central manager of submission machine. In case of source-initiative approach, a pool may send a query for resources to every other pool in its flock, which may respond with an appropriate machine that satisfies the query; initiating pool uses one of the responses it got. In the server-initiated scenario, a pool advertises its available resources to other pools and the execution pool grants permission to the submission pool for accessing that particular resource.

Even with Condor Flocking, each Condor Pool manages the same execution machines and is assumed to be running on a reliable machine [29]. Based on Condor usage patterns [62], a good portion of requests it handles are repeated [62]; thus, every new discovery of the same/similar remote resources by a manager follows the same workflow. Given that the required capacity could frequently exceed local availability, the local manager would forward equally frequent queries seeking extra capacity of similar properties; thus resulting in a significant number of messages.

Though flocking erases the 'single-point-of-failure' issue of Condor system, it cannot create dynamic pool neighbourhoods, uses a single-hop horizon for query forwarding and retains the priority-based execution scheme. Private flocking was quickly abandoned [38] since it introduces many messages to avoid redundant executions of a job. However, group flocking suffers from many updating messages, too, in case of frequent failures of central managers and/or resources. Finally, prioritization scheme of jobs cannot be used for sharing dynamic resources that have immediate need for help.

2.4 P2P Networks

Peer-to-Peer (P2P) networks try to overcome assumptions like high and uniform availability of Grid resources and address solutions to resource discovery problems in highly dynamic environments. They are usually referred to as *overlays*, built on the application OSI layer and serve a single application and share specific resources for which they are usually optimised. Their name derives from the communication type between their members (peers); at their simplest form peers interact without intermediaries (e.g. servers). Every peer (it will be referred as node this point onwards) has a basic characteristic: intermittent membership. A node may frequently and abruptly join/leave a network without notifying other members even if this violates/terminates an ongoing activity.

2.4.1 Node Characteristics

The owners and administrators of nodes are usually anonymous individuals that voluntarily offer their resources using a weak infrastructure (limited bandwidth, storage and CPU capacity and unreliable connectivity). In this context, they do not provide any guarantees on reliability and efficiency of their resources [62] and they usually do not expect income by sharing them. However, the network setup requires minimal effort since there in no central coordinating authority and any change on node membership status requires minimal network reconfiguration.

A node, member of a P2P network, usually exhibits intermittent and untrustworthy behaviour. Defining churn rate as the percentage of nodes joining and leaving a network per time unit, P2P networks experience high such rates [98]; they have not been designed to rely on central entities and thus they are considered to be resilient even to high churn rates. Resource provisioning is not always a natural corollary of P2P membership; that is, a node participating in a P2P network may provide useless or no resources at all. If a P2P requires that a node provides a minimum amount of resources to become a member, there are no guarantees that these resources will be of any use to the remaining P2P community or the provider may frequently change its shared resources to avoid bad reputation mechanisms [13], [45]. In case there are no such requirements, even nodes without any shared resources may join the P2P.

2.4.2 Centralised Indexes

Centralised P2P networks follow the same principles as client-server architectures for resource discovery. A single entity responds to every query originated from a member of this star network. This centralised entity is a single point of failure and vulnerable to attacks like denial-of-service or man-in-the-middle. It is usually a very reliable and powerful machine with good connectivity. In P2P context, its role is to provide either directly the requested information (e.g. location) about it.

Napster [90] is a P2P system that uses a centralised index server to store the location of files shared by every node in that network. A node joins the network by submitting a list of files it shares to a predefined Napster server. That server, upon receiving a query from any node, using its index, creates a list of other nodes that may provide the requested resource and responds back. Requesting node, then, directly connects to one of them to forward the query to and retrieve the file. Though the actual requested file does not reside in Napster server, thus reducing minimum required specifications of that server for storage capacity and bandwidth, the whole network may shut down if the server fails. Napster topology is visualised in figure 2.2.

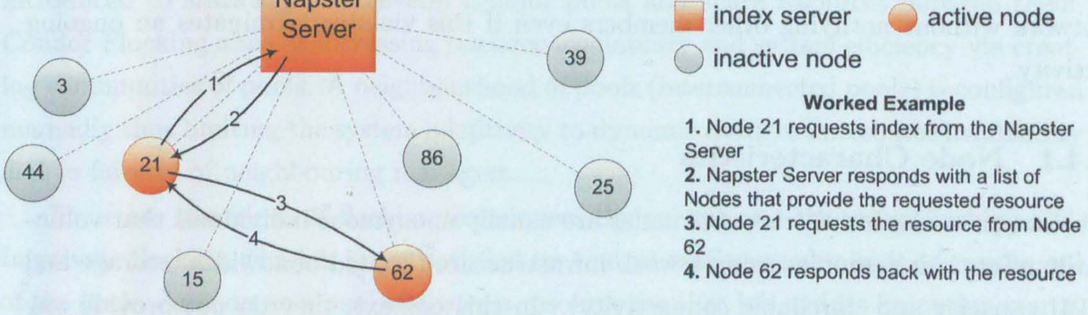

Napster Server
index server
active node
inactive node
Worked Example
1. Node 21 requests index from the Napster Server
2. Napster Server responds with a list of Nodes that provide the requested resource
3. Node 21 requests the resource from Node 62
4. Node 62 responds back with the resource
3
44
21
15
62
86
39
25
1
2
3
4

Figure 2.2: *Centralised Architecture and resource discovery mechanism*

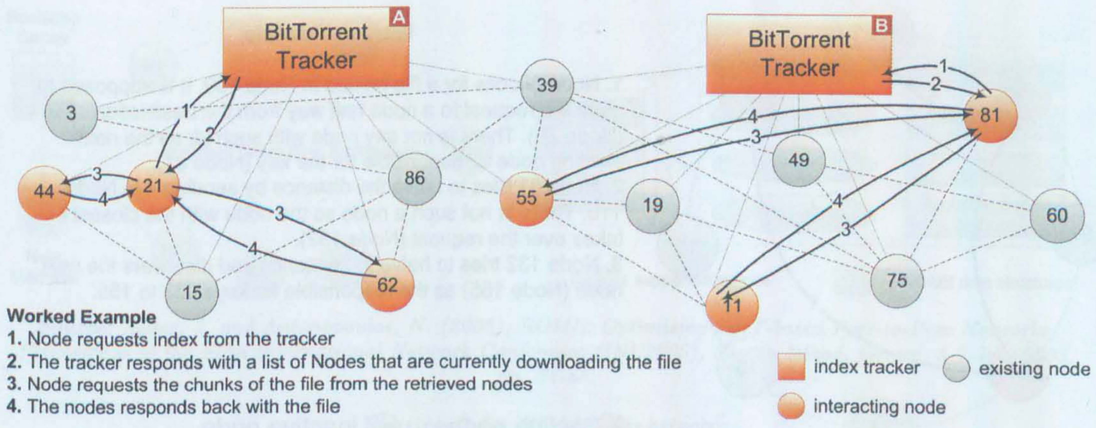

BitTorrent Tracker A
BitTorrent Tracker B
3
44
21
15
62
86
39
55
19
49
81
60
75
11
1
2
3
4
Worked Example
1. Node requests index from the tracker
2. The tracker responds with a list of Nodes that are currently downloading the file
3. Node requests the chunks of the file from the retrieved nodes
4. The nodes responds back with the file
index tracker
existing node
interacting node

Figure 2.3: *Two different BitTorrent networks for different files controlled by two trackers. These networks exist as long as the downloading activities are not complete.*

The concept of centralised index servers is also used in BitTorrent [81] which is mostly deployed for file sharing. It uses regular websites to publish available files in nodes. For every shared file there is a node that controls the distribution of that file, the tracker, which is the centralised entity of the network. A node may share a file by publishing on a website a small *.torrent* file containing a pointer to its tracker, possible names and length of that file. Any requestor of that specific file locates the corresponding torrent in a website which requestor's BitTorrent client uses to access the tracker and retrieve a list of nodes currently downloading that specific file. It can start downloading in parallel different chunks of the file from nodes that already have downloaded them. Providers of these chunks may continue retrieving the remaining chunks from other nodes; thus, a node may be simultaneously provider and requestor of different chunks of a file. If no file transfer is currently active the requestor node directs the query to the torrent publisher. Figure 2.3 shows a shapshot of BitTorrent topology with two trackers.

Each shared file may have different trackers and one tracker may coordinate the sharing of multiple files. Potential failure of a tracker cancels all current and future transfers of the files it coordinates. This scheme partially solves the Napster's single point of failure since the same nodes may continue interacting with other nodes for different files coordinated by other trackers. However, there is not an automated torrent discovery mechanism. It is a system designed for replicable persistent resources. Using web publishing and manual discovery of the appropriate torrent files for highly dynamic resources restricts the adaptivity of the network to abrupt changes such as frequent node failures.

2.4.3 Structured Networks

Aiming to a more robust network than centralised architectures, Structured P2P networks use distributed indexes of shared resources. Structured P2P networks assign a random ID to every node and a key to every resource and map each pointer tuple [key, resource] onto a specified node. The organisation of nodes is flat; that is, all nodes

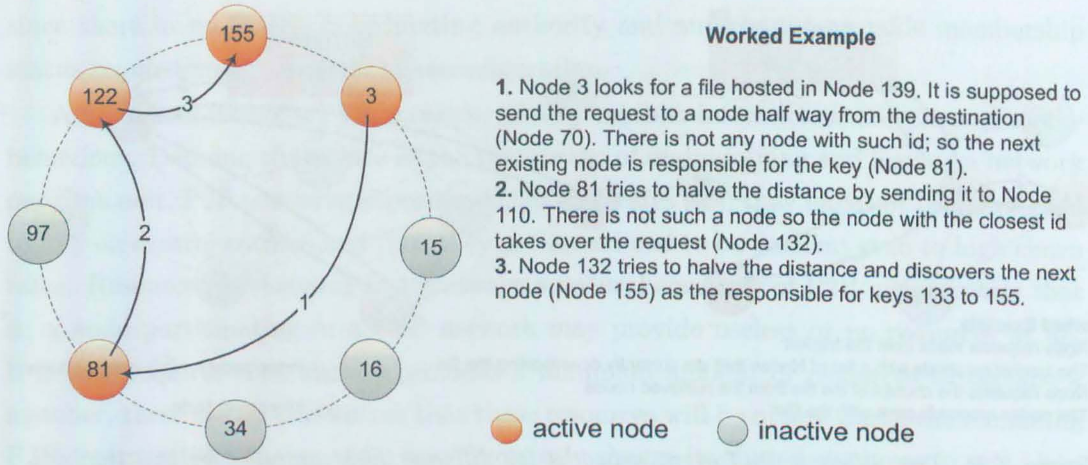

155
122
3
97
15
81
16
34
1
2
3
Worked Example
1. Node 3 looks for a file hosted in Node 139. It is supposed to send the request to a node half way from the destination (Node 70). There is not any node with such id; so the next existing node is responsible for the key (Node 81).
2. Node 81 tries to halve the distance by sending it to Node 110. There is not such a node so the node with the closest id takes over the request (Node 132).
3. Node 132 tries to halve the distance and discovers the next node (Node 155) as the responsible for keys 133 to 155.
● active node
○ inactive node

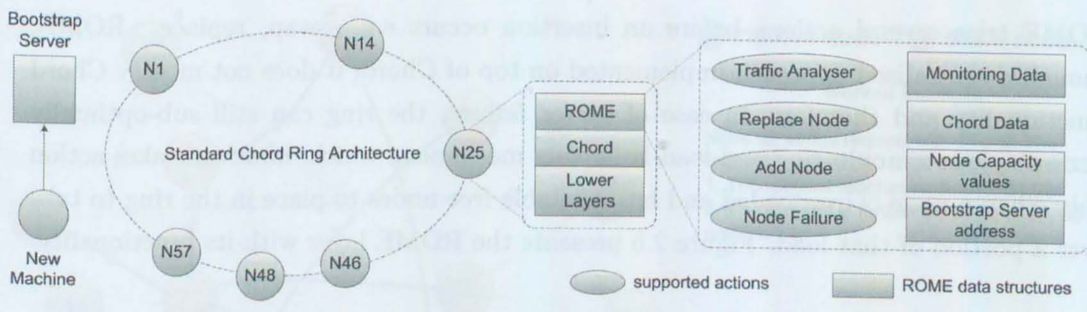
Figure 2.4: *Chord ring interactions. Node 3 tries to locate the file with key=139. node 155 hosts all the keys from 123 to 155.*

are equally important and have the same amount of responsibilities within the network. Given the keys, this type of overlay networks can perform efficient discovery but they require a copy or pointer to each resource at the node which is responsible for this resource key.

They are theoretically well-founded topologies and guarantee successful discovery if the requested resource is available in the network within a logarithmic number of, function of network size, steps. Latency becomes a considerable problem in case of large networks (huge number of nodes) since each query is routed to the next intermediate nodes as far as possible producing long-distance network traffic and delays [74]. Structured P2P topologies use nodes IDs and well-defined algorithms to establish connections ignoring their physical location; thus, a direct connection between two nodes in application layer may cause long-distance IP traffic if the two nodes are located too far apart. Low-capacity nodes can introduce further delays since they can easily become overloaded even if they don't process queries but just forward them. Query routing is based on simple keys and thus Structured P2P overlays do not support complicated queries. Although search efficiency is very important for long-term transaction discovery, there is no obvious way to apply these topologies and support low-latency partial transaction discovery on large networks.

The discovery mechanism is based on Distributed Hash Tables and achieves same performance for scarce and famous resources. The membership in P2P networks is highly intermittent and DHTs need frequent updates to retain the logarithmic-based success guarantee. However, this comes at a high cost in messages, maintenance overhead. Therefore, Structured P2P networks are not suitable for sharing reusable and with variable capacity resources since the high maintenance cost of DHTs would introduce scalability problems to the system.

Well known Structured P2P networks are the Chord [96], CAN [83], Tapestry [109], Pastry [87], Kademlia [109] and Viceroy [75]. Chord is a one-dimensional DHT-based overlay that organises nodes into a ring. The lookup function sends messages over this



Source: *Salter, J. and Antonopoulos, N. (2005), ROME: Optimising DHT-based Peer-to-Peer Networks, Proceedings of the Fifth International Network Conference (INC2005), Samos Island, Greece, 5-7 July 2005, pp. 81-88.*

Figure 2.5: *ROME Architectre*

ring with clockwise direction only. Each node has a unique identifier produced by a hash function. A shared resource is assigned a key which is hashed with the same hash function to determine the node id that will be responsible for that key. Every node has a neighbour list pointing to the next neighbour and to $log_2(n)$ different nodes (clockwise). The lookup function tries to halve the distance between currently visited and destination node. The theoretically expected average number of messages for one lookup is $\frac{log_2(n)}{2}$ and the maximum is $log_2(n)$. Figure 2.4 presents a Chord ring with its discovery mechanism in action.

Recent advances on Structured P2P Networks include modifications of existing architectures to enable complex query handling. A well used technique is the insertion of multiple layers on those architectures. Zhang and Ma in [108] propose a hierarchical (two-layered) DHT-based Chord ring for facilitating attribute and range queries on multimedia files. The upper layer is a Chord ring built based on one common attribute of those files. Each node of that ring is a virtual cluster of arbitrarily connected content providers grouped together with regards to another attribute. These clusters play the role of the lower layer and enable complex queries. Each query in this architecture, however, requires the use of the common attribute for locating the appropriate cluster before it applies a range query. Others try to enable complex query handling in Structured P2P Networks via tree-like architectures such as VBI-tree [63]. Another approach to the problem of range queries is the use of a single structured overlay not based on hash functions but rather on the actual e.g. file names of the resources they index. For instance, assuming a Chord ring they first try to locate the first node of the range using DHT-like query forwarding schemes and then sequentially hop from node to node untill the range is finished. Examples of this scheme are the following: [21], [34], [80].

Given that the discovery mechanism performance depends on the whole network size, any reduction of ring diameter would reduce the lookup cost, as well. Salter J. et al. in [88] proposed a reactive overlay (ROME) for optimizing the Chord ring size and thus the lookup overhead. They used a bootstrap server, which preserves a pool of underutilised nodes that dynamically places them in the ring based on network workload. Symmetrically, it removes nodes from the ring if they are underutilised.

ROME tries several actions before an insertion occurs e.g. swap, replace. ROME, though a centralised approach implemented on top of Chord, it does not modify Chord functionality and therefore, in case of server failure, the ring can still sub-optimally perform. It is a monitoring and load-balancing mechanism which, however, takes action only when a node is overloaded and has available free nodes to place in the ring to take over a portion of that load. Figure 2.5 presents the ROME layer with its functionality.

2.4.4 Unstructured Networks

While Structured P2P networks deploy expensive maintenance protocols to tackle high churn rates, the Unstructured P2P architectures try to minimize maintenance costs introduced by node failures/departures. Unstructured P2P overlays organise peers in a random flat or hierarchical graph which is decoupled from resource locations. No unstructured overlay may guarantee successful discovery even if the requested resource exists in the system but it supports complicated queries. In general, discovery mechanisms in Unstructured P2P networks use forwarding schemes of a query to a subset of neighbours of each node. However, network organization and/or topology may change as nodes join/leave while a query travels the overlay [97]. That is, if the behaviour of nodes is highly intermittent then the overlay configuration does not remain the same from the beginning to the end of a query propagation.

In case of hierarchical overlays, certain nodes (super-peers) take over resource indexing; all nodes discover the requested resource location via accessing a set of super-peers. Super-peers, besides providing content, have more responsibilities than normal nodes and they create a network between each other, independent from nodes, but nodes access them to use their extra services. This scheme has the same advantages as BitTorrent; it reduces the number of messages required. In contrast to BitTorrent, it typically provides a mechanism for automatic discovery of other super-peers. However, the cost of a failed super-peer has a bigger impact than that of a failed node. It may disconnect all the nodes dependent on that failed super-peer unless nodes are aware of other super-peers, too.

In flat organisations (without hierarchy) all nodes have the same responsibilities and perform the same actions in case of an incoming query. If a node can satisfy the query given just its own resources, it responds back to query originator with a set of discovered resources. Random organisations locate easily the popular resources and are highly tolerant to node failures. The resource location is not coupled to network organisation; that is, there are not any constraints as to where each resource should be placed within the network. This forces a query to search any node in the network thus resulting in many unnecessary messages.

Some well-known Unstructured P2P architectures are the Gnutella I [3], Gnutella II [4], FastTrack [71] and Freenet [33]. Freenet is designed to provide high levels of anonymity for its users. BitTorrent is a centralized Unstructured P2P architecture since there is a central entity to maintain an index of resources located in nodes which currently participate in a downloading transaction of the same file. However, trackers

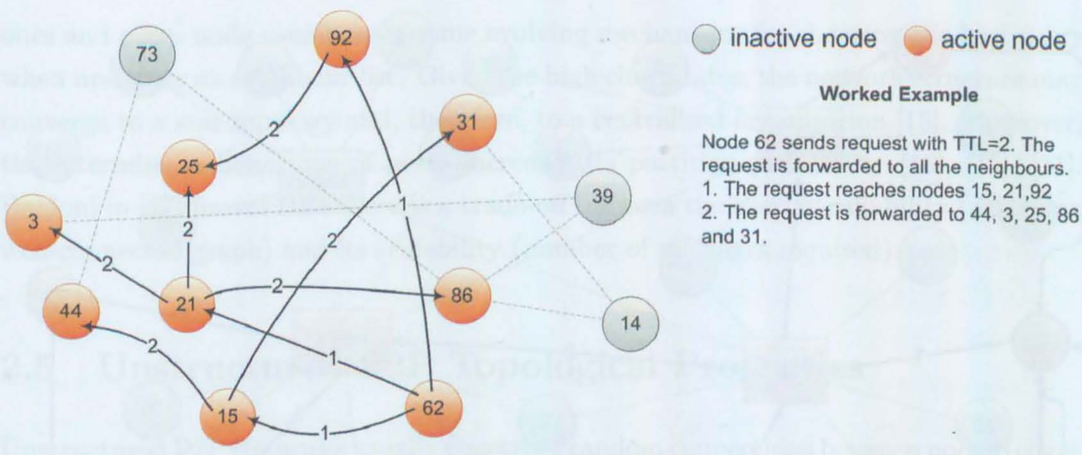

inactive node
active node
Worked Example
Node 62 sends request with TTL=2. The request is forwarded to all the neighbours.
1. The request reaches nodes 15, 21,92
2. The request is forwarded to 44, 3, 25, 86 and 31.
73
92
31
25
39
3
86
44
21
14
15
62
1
2

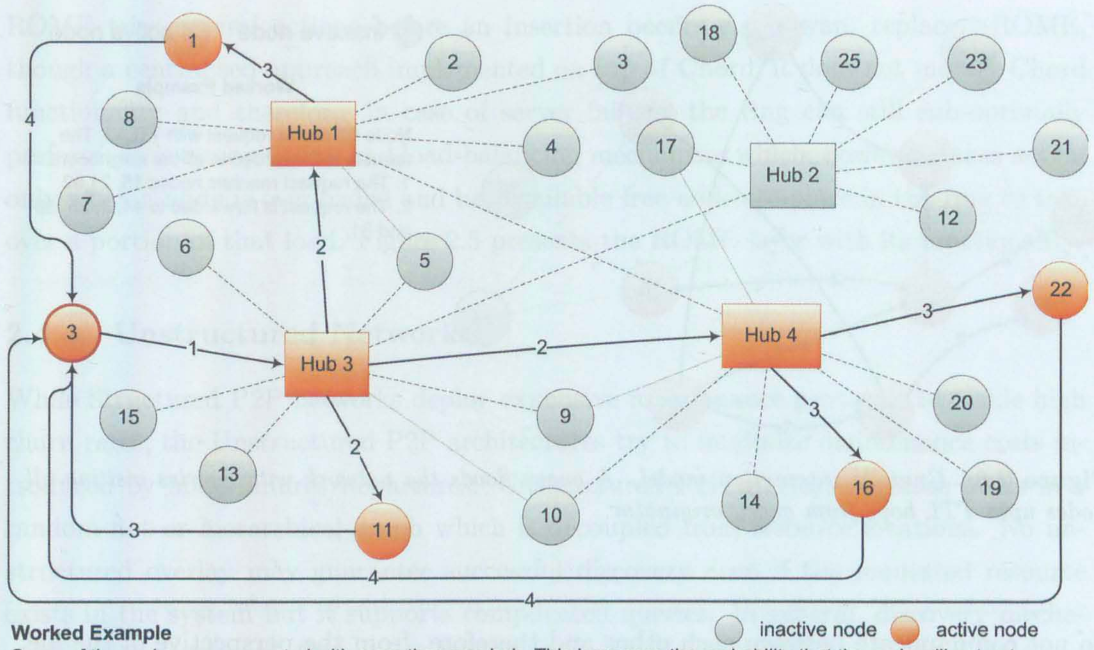
Figure 2.6: *Gnutella interaction model. A query floods the network with queries visiting all nodes upto TTL hops from query originator.*

do not communicate between each other and therefore, from the perspective of the file being downloaded, its tracker is its central coordinator and single point of failure. The following paragraphs will focus on systems (Gnutella and FastTrack) that produce the most traffic on the internet [93].

Gnutella I is a file sharing flat P2P network. Every peer has dual roles since it acts as both server and client. The topology configuration is not coupled to resource locations in the network but the connections between nodes are rather random. Each node joins the network by contacting one of the well-known peers and retrieving a list of other peers. Once connected, the requestor node forwards a query to all its neighbours. Upon receiving a query, a node processes it locally and returns back any available resource while at the same time forwards it to its own neighbours. This forwarding scheme continues from neighbour to neighbour but it stops when a horizon is reached. On every hop the query Time-to-Live counter, initially set by its originator to a specific integer number, gets reduced by one and when is zero the forwarding is abandoned. The TTL protects the network from infinite query loops and exponential increase of messages.

Forwarding a query to all neighbours may result in exponential growth of messages, given that all nodes have more than one outbound neighbour. That is, the number of nodes visited on every hop of a query increases exponentially. However, the more hops done the lower the probability to build up a tree-like searching space [92]. Therefore, the cost in messages of a flooding scheme has upper bounds even if the horizon is far away from query originator.

Gnutella II tries to reduce the number of messages and improve efficiency by adding a level of hierarchy: an overlay of randomly connected hubs. Every node (leaf) may connect to one or two hubs, usually more powerful peers with better connectivity. Each hub hashes the content of its leaves and exchanges a version of its hash table with its neighbouring hubs. Each hub may connect to 5-30 other ultra-peers and to 200-300 leaves. It processes the incoming queries on behalf of leaves; that is, it forwards the query to leaves that have the requested resource and only to its immediate neighbours



Worked Example
Some of the nodes are connected with more than one hub. This increases the probability that the node will not get disconnected if one of its hubs fails.
1. Node 3 sends request to Hub 3.
2. The hub, based on the indexes of the local nodes forwards the query to Node 11 and based on the indexes received from neighbouring hubs, forwards the query to hubs 1 and 4.
3. Hubs 1 and 4 forward the query to their appropriate local nodes and stop the forwarding. The Node 11 that belongs to the same hub as the query originator responds with its answer.
4. Remote nodes respond back with the answer.

Figure 2.7: *Gnutella II architecture. Indexing in hubs and selective forwarding saves the network a lot of messages.*

that possibly have the appropriate leaves. That is, queries from underlying nodes travel only one hop within the overlay of hubs, thus, saving the network from many messages. However, the requestor node may retrieve more hubs from one of its corresponding hubs and repeat the same process with them. There is not a mechanism for a node to change roles (leaf to hub and vice versa) according to network status. This architecture helps peers to easily locate the abundant resources.

FastTrack is a proprietary network architecture and most details about its protocols are known by research on its traffic analysis and a brief description on its dedicated website. It constructs an overlay with two classes of nodes: the supernodes which are powerful peers with good connectivity and the ordinary ones. Each node has only one parent supernode to which it uploads the index of its files and retains a short-lifetime TCP connection with it. During registration process, a node uses its supernode refresh list to check the connectivity with all other available supernodes and chooses one to establish a TCP connection with. The node hops from one supernode to another using its periodically updated supernode refresh list. Requestor nodes forward their queries to their supernodes which respond back with a list of other leaf nodes that requestors may directly contact to retrieve the file. The same queries are also forwarded to a small number of other supernodes that direct their responses directly to the query originator.

In Unstructured P2P networks, new nodes usually register with well-known existing ones and every node usually uses some evolving mechanisms for choosing the best peers when updating its neighbour list. Given the high churn rates, the network structure may converge to a star topology and, therefore, to a centralised organisation [15]. Moreover, the intermittent behaviour of peers increases its partition probability [16], [17], [11]. Baldoni in [17] proves that there is a trade-off between the overlay reliability (achieving well-connected graph) and its scalability (number of messages required).

2.5 Unstructured P2P Topological Properties

Unstructured P2P Networks usually construct random connections between nodes (edges). That is, each node is connected with a random subset of the remaining nodes. However, certain actions and decisions taken during their lifetime may gradually convert the network topology into a non-uniform one. Join, leave and rewiring actions are some of those that contribute to this phenomenon. Diestel in [37] defines the **degree** of a node as the number of its neighbours. If connections between nodes are directional then the number of incoming connections is the *in-degree* and the number of outgoing ones is the *out-degree*. Nodes with zero degree are *isolated*. A degree distribution shows the number of nodes with a specific degree $d \geq 0$. Symmetrically, in- and out-degree distributions take into account the incoming and outgoing node, respectively, connections only.

If every degree d appears on the same number of nodes then that network has uniform degree distribution and non-uniform otherwise. The Unstructured P2P Overlays that automatically construct and update their neighbour lists (unlike networks with statically defined by users neighbourhoods i.e. Instant Messaging services e.g. [18]) need to access an existing node to join the network and tend to prefer sending requests to more reliable nodes. More robust nodes are usually well-known since they have a longer and reliable contribution to the network. Therefore they are more frequently selected to send requests to and due to deployed rewiring policies their fame attracts more fame. There are several studies of unstructured networks claiming that rewiring and scaling mechanisms deployed on those networks gradually transform them into networks with small-world properties and in some cases into power-law degree distributions making famous nodes even more famous [7], [9], [44], [46], [79], [85], [86]. Finally, another factor affecting the degree of a node and thus its fame is the Quality of Service offered by that node. If neither robustness nor Quality of Service of a node are negative, then its fame keeps increasing. Several traffic analysis of random unstructured networks confirm those properties.

Based on a traffic analysis of Gnutella from Ripeanu [85], this P2P tends to create connections to the most reliable and persistent (stay alive for longer than others) nodes which are only the 20% of network size. Symmetrically, the least persistent nodes, which are the majority in the network, have less connections. Therefore, Gnutella converges to a power-law topology with a well-connected core in the centre and weakly connected branches and leaves. These famous nodes become members of a giant component in the centre of the network which works as a bridge between any two, even infamous, ones shortening the network diameter [9]. Vaucher et al. in [103] also confirm these conclusions. This topology exhibits robustness in case of random, even high, node failures; its connectivity heavily depends on those few famous nodes in its core and therefore the probability that random failures affect the majority of them and thus potentially fragment the network is low. These famous nodes bridge otherwise distant ones thus the reducing network diameter and latency of deployed search mechanisms. Both the increased robustness and efficiency of those networks allow them to freely scale; *scale-free* networks.

However, due to their fame they face high load, are easy to detect and vulnerable to targeted attacks [10], [85], [90]. Power-law properties are also observed to real-world large scale networks such as World Wide Web [8] and [27], Internet [44], citation, metabolic [23] and human sexual contacts [72] networks. Guclu and Yuksel in [57] confirm the above and propose a method to force an upper bound on the degree of a very popular node (*hard cut-off*) as a way to force fairness on load distribution among nodes investigating potential effects on network efficiency, too.

2.6 Computational Resource Sharing

Peer-to-Peer systems seem to be a promising architecture for sharing non-replicable reusable resources. Though task scheduling in Grid environments is a mature field, automated distributed resource discovery is still dependent on centralised entities or static links. However, recent research tries to enhance the scalability of Grid environments by deploying P2P networks for large-scale resource discovery over Grids.

Butt et al. [29] identifying the weaknesses of Condor Flocking, propose a self-organising Pastry [87] overlay of condor pools. Collaborative condor pools get a random ID based on which are organized into a ring and proactively advertise their idle resources to their neighbours. Their proposed system deploys an advertisement scheme; each pool sends advertisements to all its neighbouring pools which, in turn, decrease a hop counter, Time-to-Live (TTL), by one and forward it to their own and so on until TTL is zero. This P2P-based flock of Condors requires a substantial maintenance overhead for updating the proximity-aware routing tables and advertisement data since frequent changes to resources and/or pool states may quickly invalidate look-up tables and already published adverts.

P-Grid [5] uses features of P2P overlays [14], [74] to discover Grid resources. P-Grid, trying to avoid the update overhead of advertisements on DHT-based Grid overlays, uses a tree-based distributed storage system of advertisements. Resource providers locate in this tree requestors they can serve and offer themselves for use. A node is assigned a key, a path that indicates the location of the node within the tree. Each P-Grid node is responsible for resources, the hash key of which starts with a binary description of the node's tree path. One or more nodes may have the same key, in which case all of them are responsible for resources with this prefix. While other Structured P2P networks hash the indexing keys, thus limiting the searching capabilities, P-Grid enables complex queries.

The organization of this overlay has good fault-tolerance characteristics but multiple advertisements per resource. Frequent changes of the state of a resource should produce equally frequent updates to advertisement hosting nodes. This architecture introduces delays on resource joining actions since the network needs some time to periodically discover nodes in other paths.

Gupta et al. in [58] propose CompuP2P, a two-layered Chord-based market of any computing resources. Every node, either requestors or provider, calculates its service capacity based on which it joins the lower Chord ring of the architecture. The same ring may contain different resource types (e.g. both CPU cycles and storage capacity). Each node of the upper ring indexes all the nodes of same resource type and capacity. Requests for capacity first reach the upper ring to locate a subset of nodes providing the requested resources and then requestors deploy a Chord look-up process to locate those nodes in the lower ring. Though, they try to address heterogeneity of nodes, the authors implicitly admit weaknesses in high churn rates. To alleviate this problem, especially in case of CPU cycles, CompuP2P does not index the actual capacity of nodes but rather its average value within a quite long period. Moreover, the diameter of the upper layer can get as big as that of the lower one in case all nodes provide unique resource capacity in type and size. The look-up and maintenance costs for pertaining accurate finger tables on both rings can become very expensive in those cases.

Celaya et al. in [32] present a Structured CPU cycle stealing P2P Network. Nodes are organised in a B-Tree structure based on their IP-address so that sibling nodes are physically located in vicinity thus reducing latency. It is a job scheduling system which assumes well maintained resources and rare join/leave node actions, as stated by authors. Hoverlay is to be deployed on large-scale heterogeneous networks with no guarrantees on nodes behaviour.

OurGrid as analysed in [26] creates a pool of resources for every community. Each pool is connected to all other pools and broadcasts the queries it cannot serve with the local resources. It is assumed that the jobs that are to be executed do not need access to the network interface of the execution machine. There are no guarantees about the quality of service offered by the execution machines. The allocation of the resources to jobs is based on a reputation scheme that rewards peers that have a reliable contribution to the community.

A P2P-based dynamic reusable resource discovery architecture is proposed by Awan et al. in [15]. They deploy an Unstructured P2P network to share processor cycles. The main concept behind this is the submission of multiple chunks of a task with their replicas to uniformly random peers. The submission node decides how many redundant replicas of each chunk are necessary based on the network status it has so far experienced. The authors provide theoretical models to calculate replicas redundancy factors and devised a uniform walking-based sampling algorithm to distribute those tasks regardless of the network topology (uniform, power-law, etc). There is no guarrantee that the sampling walker will terminate on a free CPU. Though CPU cycles are not replicable resources, authors treat CPU sharing as a task replication-based one and, thus, their algorithms are not applicable to the context of current research.

In [28], the authors propose the "Inter-flops" (IFL) load-balancing algorithm of Active-Objects on Unstructured P2P Networks practically sharing CPU cycles. That study identifies two states of a resource, underloaded and overloaded, based on which IFL takes the appropriate decisions. If overloaded, a resourse forwards a request to a subset of its neighbouring nodes seeking underloaded ones to send its objects to; if underloaded, it proactively advertises itself to a single neighbouring node. IFL, trying to address latency problems of computational resource sharing, tends to select the best neighbours and heavily depends on locality of nodes since all requests and advertisements have a single-hop length ignoring any variations of network topology and distribution of requests which is assumed to be uniform. If certain areas of a network experience high workload, requests originated by nodes in their centres will not be treated equally; nodes on the borders would have access to a bigger plethora of resources. Real systems [8], [104], [84] tend to form power-law distributions in which case IFL would overload the famous ones. High workload situations would trap requests in queues of famous nodes increasing latency and probability of network fragmentation had they failed.

In an effort to avoid the maintenance costs of Structured P2P Networks over Grids, Hu and Klefstad in [60] present an Unstructured P2P Overlay which tries to achieve global load-balancing. That is, the maximum difference in number of assigned tasks between any two nodes needs to be kept between 0 and 2. Each node receives the task queue sizes of all its neighbours to find the biggest distance compared to its own. It then notifies back all neighbours about this difference which is added on top of their own calculated distances, thus discovering task queue size differences between any two nodes. As soon as a node detects a total difference higher than 2 it starts moving tasks to neighbour with the highest difference. This is a load-balancing algorithm which achieves global equilibrium in return to latency increase and message cost.

Bhrathi and Chervenak in [22] deploy an Unstructured P2P Network of Grid Index Services of Globus Toolkit 4. The network uses flooding to search for a requested resource and a query caching scheme to reduce the number of messages. It seems to be a preliminary research project with limited results. More focus on network topology evolution and discovery mechanism is required taking into account the special features of shared computational resources.

2.7 Unstructured Discovery Techniques Classification

There is a variety of discovery techniques that fit for Unstructured P2P Networks and can be deployed on them regardless of their organization and topology. These algorithms may be grouped in two disjoint subsets: one regarding hierarchical and another flat Unstructured P2P Networks. Both are based in the neighbour-to-neighbour forwarding scheme; that is, a node upon receiving a request, it processes and then forwards it to a subset of its neighbours. In [100], Tsoumakos D. and Roussopoulos N. classified these search methods based on the number of next destinations (size of multicast) and the

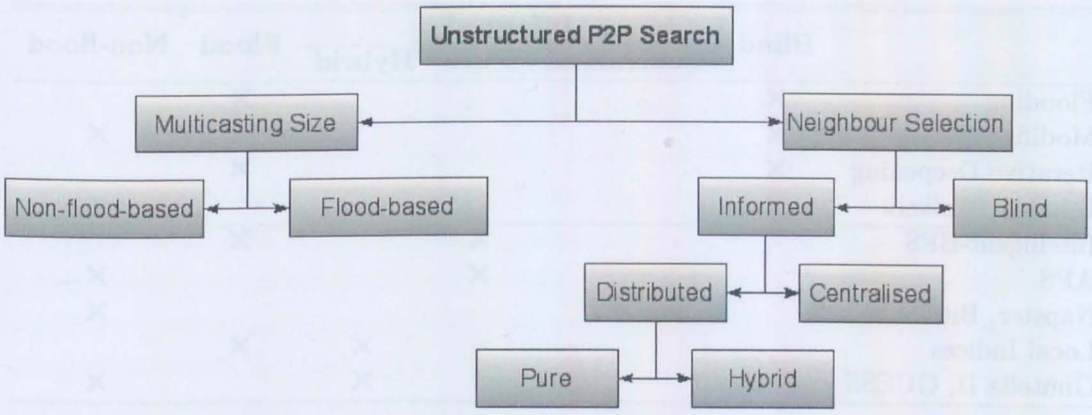


Figure 2.8: *Unstructured P2P Search Methods classification by Tsoumakos et al. [100]*

way these are selected.

In the case of *blind* search schemes, each node forwards queries to a random subset of its neighbours until a certain number of hops is reached assuming that there are more than one appropriate resources within the network. If an *informed* one is deployed, it uses some historical information about resource locations (meta-information) and forwards the query to a subset of neighbours that would maximize the search success rate. This information is either direct pointers to or some hints about resource locations and can be gathered into a single repository accessible by all (*centralized informed*) or distributed among some nodes of the network (*distributed informed*). Each node of the network, which uses a pure distributed informed method, has and manages a portion of this meta-information. If a *hybrid distributed informed* method is used this meta-information is shared among a subset of nodes that act as super-peers. A node forwards a query to all its immediate neighbours if the forwarding technique is *flood-based* or to a subset of them, otherwise (*non-flood*-based). Figure 2.8 presents a general classification of Unstructured P2P discovery mechanisms primarily derived from their work in [100].

Section 2.4.2 describes two centralized informed techniques: Napster [90] and BitTorrent [81]. Both these systems are based on central indexes from which all nodes retrieve their contemporary neighbour lists. These methods guarantee that if an answer exists somewhere in the network it is accessible by all the requests. Though they achieve high accuracy, efficiency in number of messages and good adaptability to node churn (services may be available even if an important percentile of nodes fail), their central index is a single-point-of-failure. They require a stable central entity to deal with the frequent requests in highly dynamic environments and to ensure availability of the index. Table 2.2 presents a collection of Unstructured P2P Search Methods with their classification on the categories of figure 2.8. More details on those algorithms follow in the next sections.

Flood-based techniques produce many messages but achieve good success rate, low latency and fault-resilience. The non-flood-based ones produce generally fewer messages than flood-based ones at a cost on their success. The pure distributed informed schemes manage to satisfy a high percentile of requests only after some time in operation with

	Blind	Informed			Flood	Non-flood
		Centralised	Pure	Hybrid		
Flooding	×				×	
Modified-BFS	×					×
Iterative Deepening	×				×	
Random Walkers	×					×
Intelligent-BFS			×		×	
APS			×			×
Napster, BitTorrent		×				×
Local Indices				×	×	
Gnutella II, GUESS				×		×

Table 2.2: *Classification of the most famous Unstructured P2P Methods*

number of messages comparable to blind non-flood-based ones. However, they exhibit weak adaptability to fast topology changes which can substantially reduce their success rate [101]. Finally, centralized informed schemes depend on central entities which, if failed, could shut down the whole network.

Informed methods, as presented above, are educated mechanisms which increase their efficiency as long as the resources are not continuously displaced and node connectivity stays relatively unchanged (i.e. no/minimum rewirings). A new type of informed search techniques for unstructured networks appeared with the studies of their statistical dynamics and evolution patterns. Though they do not store resource location information they may use network statistical features inferred by its topology and degree of connectivity of each node. For instance, assuming that famous nodes are more capable of providing an answer, trying to reach them before the non-popular ones during query forwarding (i.e. a query chooses to visit a high degree node before a low-degree one) may increase success rate and reduce generated messages. In this case, neighbour selection, though not educated, tends to select more suitable nodes; this is a feature of educated informed algorithms.

Therefore, informed methods can be further sub-classified into *educated* and *uneducated* ones as shown in figure 2.9. Topological properties of networks may be inferred via Neighbour Lists of nodes and thus these methods are decoupled from any stored information in them; Neighbour Lists are standard components of nodes in Unstructured P2P Networks and represent topologies in decentralised manner.

2.8 Blind Search Techniques

Flooding is a widely used blind technique that was introduced by Gnutella I [3]. It is frequently referred to as Gnutella search method, as well. The nodes that use flooding forward each incoming query to all their own neighbouring ones unless:

- the query has already been forwarded TTL consecutive times (its TTL has expired), in which case the query stops there.

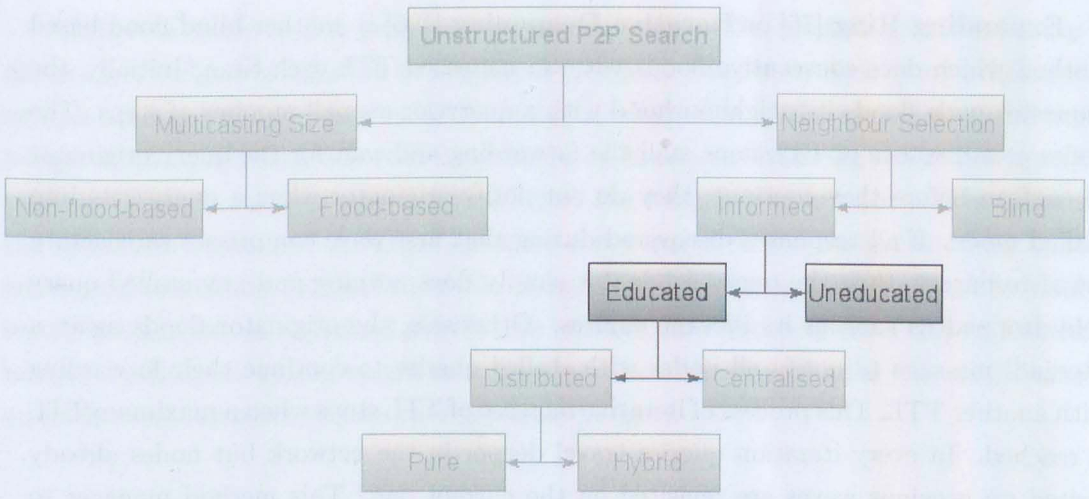


Figure 2.9: *Extended Unstructured P2P Search Methods classification including Educated and Uneducated Informed Methods*

- one of the next destination nodes is the query originator which is excluded from the query broadcasting.
- query loops are created if forwarded to neighbours that already have processed the same query. These neighbours are also excluded from that broadcasting.

Flooding is characterised for its simplicity and its high cost in number of messages per query produced in the network [101]. It can quickly access a big portion of the network consuming substantial bandwidth as the number of visited nodes per query increases exponentially per hop. The large number of messages produced puts a substantial burden on the nodes since they have to handle many incoming queries [75]. Despite this exhausting network exploration, the resources placed beyond that horizon cannot be discovered by that node [24]; TTL sets a horizon for every node. It can, however, guarantee discovery if these resources are located within that horison.

Aiming at cost (number of messages) reduction of Flooding, Kalogeraki et al [65] introduced the **Modified Breadth-First Search** technique. It is a blind non-flood-based search scheme which, instead of the whole neighbour list of each node, it uses a random strict subset of them (e.g. each node forwards every query to 2 out of its 5 neighbours). The node chooses randomly a percentage (parameter of the method) of its neighbours to forward the query to thus reducing the number of messages produced by a single query. It reduces the probability of loops as it does not explore every link but it cannot avoid the exponential increase of messages. As in Flooding, a TTL counter per query works as its propagation stopping condition. Though the overall produced messages are fewer, they are still a substantial workload and bandwidth consumption. It is a probabilistic method that does not guarantee discovery of resources located either beyond or within the TTL radius [101] and highly depends on the size of selected neighbours subset.

Expanding Ring [75] or **Iterative Deepening** [106] *is another blind flood-based* method which does consecutive floods with an increased TTL each time. Initially, the requestor node floods its neighbourhood with a query for a small number of steps. The nodes at a distance of TTL hops stall the forwarding and wait for the query originator to confirm before they continue; they do not notify originator when a query gets into stalled mode. If all responses discovered during that first wave comprise a satisfactory set of resources, then the query originator simply does nothing and any stalled query gets dropped as soon as its lifetime expires. Otherwise, the originator floods again a 'Resend' message to notify all nodes with stalled queries to continue their forwarding with another TTL. This process of iterative increase of TTL stops when a maximum TTL is reached. In every iteration queries travel deeper in the network but nodes already visited on previous waves are revisited on the current one. This method manages to almost halve the overall average number of produced messages per query and solves the problem of TTL choice of Flooding. However, this improvement comes at the cost of slight increase of discovery latency while message duplication remains an issue [101]. If the usually requested resources are the famous ones which are uniformly distributed in the network, Iterative Deepening is considered to work well [75]. 'Resend' message floods aim at reactivating stalled queries assuming that they follow same paths as the queries. However, if this mechanism is deployed on a frequently changing topology with many failures or rewirings, the probability the floods reach all nodes with stalled queries significantly decreases.

All three methods above generate high workload (messages) and message duplication. The non-flood-based blind Random Walkers method [75] efficiently resolves these two issues at the expense of considerable increase of latency and instability of its success rate. Initially, the requestor node forwards a query to random k neighbours. All the following receivers of the query forward it to a single random neighbour of theirs. Forwarding stops when the query is successful or when it has reached nodes TTL hops away from originator. A 'checking' stopping mechanism may also be used; that is, the query periodically checks with the originator whether it should stop hoping from node to node. In worst case scenario it may generate k*TTL messages but its success rate and number of discovered answers greatly vary depending on network topology. Due to the limited number of nodes it visits and the TTL stopping condition it exhibits probabilistic bahaviour and is difficult to discover scarce resources [101]. However, Gkantsidis et al. in [55] claim that Random Walkers can perform even better than Flooding in case of nodes re-issuing same queries on clustered networks assuming that nodes within their horizons do not change.

A rather special case of k-Walkers is 1-Walker, refered to as Depth-First Search (DFS) and used in Freenet [33]. DFS selects randomly one neighbour on each hop trying to forward queries as deeper in the network as possible. The query originator has to set the TTL quite big to increase its success rate. The messages spent by this mechanism are only equal to TTL but it suffers from high latency and significant fluctuations over time on its success rate.

Gkantsidis et al. in [53] propose a generalised scheme for modelling search mechanisms via a budget distribution to neighbours process. That is, *a)* each query originator assigns a budget to every replica of the query it sends out, *b)* the receivers reduce the budget of the query they received by one and *c)* distribute it among to the new query replicas. This forwarding scheme stops when the budget is consumed. This ensures the maximum number of messages spent for every query. If the budget is evenly split among the neighbours selected then this budget-based mechanism may convert to blind search mechanisms as explained above. The authors, however, propose a more informed budget distribution among neighbours based on the criticality of links (i.e. neighbours on paths with increases success rate get more budget).

2.9 Educated Informed Search Techniques

Gnutella II [4] and GUESS [36] methods are based on hybrid P2P organizations. A subset of nodes, the super-peers, keeps an index of resources shared by nodes that are connected with them (leaves). In case of Gnutella II, each node sends the query to a super-peer (hub) which then forwards it to its immediate neighbouring hubs if they seem able to answer it. The hub that receives a query matches the request to any indexed resources and if there are matches, it forwards the query to appropriate nodes. The neighbouring hubs periodically exchange their indexes so that they can send forward the query only to hubs that can help. The query does not travel further than the immediate neighbours of the query originator hub. In GUESS, however, the requestor node contacts one super-peer at a time and the super-peer forwards the query only to leaves that can respond to it based on its indexes. There is no forwarding between super-peers.

Apart from modified-BFS, Kalogeraki et al introduced Intelligent Breadth First Search in [65]. It is an informed technique based on which each node forwards the query to neighbours that are more likely to find an answer. It maintains a list of {neighbour, query} tuples when the query has been answered by or via this neighbour. The answers follow the reverse path of the query informing all intermediate nodes about that success. On any subsequent query, it tries to match, based on a heuristic, requirements of the new one with the most recent successful ones from that list. From these matches, it chooses the neighbours that have returned the most answers on similar queries. As shown in [65] and [101], Intelligent BFS is better than Flooding as it produces less messages but more than Modified-BFS. While nodes interact in a network and build up more accurate tables the number of answers discovered increase. This indicates that the learning mechanism of the method produces better results after some time. However its weaknesses are that the messages it produces cannot be less than these of Modified-BFS and that it has limited adaptability to the nodes failures and topology changes though it could be improved by using query failures along with success rate to rank the neighbours [101].

Adaptive Probabilistic Search introduced by Tsoumakos and Roussopoulos in [100] is based on a local index which reflects the probabilities each neighbour has to be chosen for the next query forwarding. Its forwarding scheme is Random Walkers; thus the query originator forwards each query to k neighbours and the intermediaries to just one. The neighbours with higher probability have higher priority to be selected. There are two approaches in modifying those probablilities tables: *optimistic* and *pessimistic*. In the optimistic approach, its current hosting node increases this probability before a query is forwarded and only decreases it when that node gets a notification about a query miss via the reverse path of the walker. In the pessimistic one, the probability is initially decreased and only increased if notified of a query success propagated over the reverse path of the walker. It has the same advantages and disadvantages as Random Walkers: good efficiency, low produced traffic but introduces significant latency and weak adaptability to topology changes. A modified version of APS, the s-APS, can dynamically switch between optimistic and pessimistic approaches. Their efficiency progressively improves in environments with low resource joins/departures rates [100].

Local Indices [106] method uses a clustering technique to organize nodes into groups. Each node has an index of resources of all nodes in a radius r. Whenever a node receives a query it tries to match the request to that index and forwards it to the discovered nodes. Otherwise, it sends the query beyond the radius r. The join action of a node involves the flooding of the network with a TTL=r with its local index of resources. It is a method that achieves high success rate and discovers many answers at the expense of many messages.

Another educated informed method is proposed in [111] by Zhuge et al. The authors describe a k-walkers based algorithm the forwarding of which, however, relies on trust values maintained in nodes for their neighbours. Initially, all neighbours are assigned a zero trust value. As soon as a path gets successful, the resource provider feeds back to its predecessor in that query path with the response. Then that predecessor increases by one the trusted value for its successful neighbour. The feedback stops there without further backtracking to the query originator. Thus, only the predecessor of responder node gets notified about a success. Query forwardings follow these values giving priority to the most promising neighbours.

2.10 Uneducated Informed Techniques

Uneducated techniques use network dynamics and topology properties derived from implicit information such as node degree. Certain unstructured topologies exhibit characteristics that enable more efficient resource discovery. Power-law networks have short average path length [9] and achieve logarithmic search efficiency [61]. As argued in [25], search techniques can be even more efficient (twice logarithmic) in scale-free networks. Though these networks have good resilience on node failures and attacks [10], a combined attack to selected high degree nodes will negatively affect the search efficiency or even fragment the network [30].

Though, blind search algorithms were initially designed assuming uniform node degree distribution of networks, several studies (as cited in section 2.5) claim that real

Unstructured P2P Networks exhibit power-law degree distributions; thus, they end up being deployed on topologies for which they do not fit. Due to rewiring and join/leave mechanisms (see section 2.5) nodes hosting famous resources, with more reliable connectivity or more resources than others tend to attract more links. Symmetrically, nodes with unpopular resources, weak connectivity or few resources tend to have small degrees and be located towards the boundaries. By definition, a vast majority of nodes in these networks have a link to a famous one and thus any query may easily reach the giant component in its centre. In contrast to unpopular resources, famous ones are easy to discover.

LV et al. in [75], claim that random walkers quickly reach high-degree nodes in power-law networks but have poor efficiency in rare resources and therefore the node degree should be ignored in query forwarding. Replication methods could alleviate the situation but in the current study this is not applicable as service capacity is a non-replicable resource. Adamic et al. in [6] proposed an algorithm based on random walkers resembling the Random Walks with Lookahead [53]. At each step, the walker visits all neighbours of the current node and if no success it hops to the one with the highest degree. This process stops after Time-to-Live (TTL) steps. Mihail et al. in [77] formalised the random-walker with lookahead discovery mechanism on power-law networks. At every step of the walker all the neighbours of the current node are checked for the requested resource and their own degree. Only the one with the highest degree then forwards the query. They claimed that the time a random walk with lookahead needs to explore the whole network is sublinear to the network size. This happens due to the giant component in its centre, a small subset of interconnected nodes attracting many links from the boundaries of the network, which eases the access to a majority of nodes as it preserves links with a big portion of the network.

Fraigniaud et al. [49] focused on developing an algorithm to exploit the power-law degree distribution properties quickly locating high-degree nodes. It is a modified Depth-First-Search which first forwards queries to neighbours with the highest degree. When a certain number of steps are completed, the query backtracks to visit the second highest degree neighbour. For every change of a node degree all its neighbours have to be notified thus increasing the maintenance cost. Backtracking introduces high latencies in search mechanisms cancelling the advantage of short average path length of scale-free networks. Adamic et al. in [6] also claimed that searching mechanisms in scale-free networks should give preference to high-degree nodes when forwarding a query. However, this assumes that the resources are replicable, non-reusable and that always hosted by the same node. This assumption is also used by the QRE algorithm [49] but is not the case with the resources on which this work focuses.

All these proposed discovery schemes assume that resource availability of a node may never drop due to its incoming degree modifications over time. Once a resource is discovered, the requestor may open a connection to that provider for future direct access to that resource assuming this resource does not change location. However, if resources migrate among different nodes those links may also become unsuccessful.

2.11 Topological System Requirements

Grids and P2P Networks are the two most decentralised and well-known paradigms of distributed computing. Given all features of service capacity, none of these two may be used as is for efficiently, reliably and inexpensively sharing this kind of resources in large scale heterogeneous environments. However, both exhibit certain very useful characteristics which may be used to compose a more appropriate architecture, Hoverlay.

Grids aggregate massive resource capacity into, sometimes complex, composite services and offer on-demand high reliability and quality of service. However, they have a number of weaknesses that reduce their adaptability in environments with volatile resources and dynamic conditions. In brief, they:

1. use static links between those services. In these cases, manual configuration is necessary.
2. allow automatic service composition via predefined well-known centralised service indexing servers.
3. require good maintenance of resources to ensure availability and promised service quality provisioning.
4. aim at financial benefits from shared resources. Resource owners and consumers are bound via contracts; human intervention is again necessary.

***Publish:** Pools to host free resources only. Each pool monitors one or more resources but each resource is monitored by one and only one pool at any time.*

Despite their weaknesses, Grids exhibit certain useful features for sharing service capacity. As with service capacity, Grid resources are reusable, non-replicable and allow exclusive only access. Gathering free resources into pools is a concept that respects resource ownership and administration policies within the same domain. There is usually a one-to-many relation between a pool and the resources it monitors/manages. At its most decentralised form, a pool may monitor only one resource (one-to-one relation) in which case connectivity with other pools is necessary. Many-to-one pools-to-resource relations may violate exclusive access to a resource, produce deadlock situations, affect system reliability or generate many unnecessary messages. In fact, if resource advertisements appear in more than one pool, there may be three situations:

- a job submission to an already busy resource without any guarantees that the new job will be executed on time (loss of system reliability).
- two jobs submitted to be sequentially executed on the same resource but the first waits for the termination of the second (deadlock situation).
- upon submission of a job, all pools hosting advertisement replicas of the same resource receive notifications, before a second job is submitted, to deactivate/remove those adverts avoiding the two previous situations. Assuming that a checkpointing scheme can take over this responsibility, heartbeats should be transmitted to all advert hosting pools. In case those advertisements migrate between pools this checkpointing scheme becomes either expensive or infeasible without an extra discovery mechanism.

For similar reasons, once a resource takes over a job (i.e. service capacity is used) it disappears from the system. This explains why resource aggregation into pools based on a *one-to-any* relation is preferable to a *many-to-any* one.

Publish: *Nodes need to re-register once they change status from busy to underloaded.*

The well-known Grid system, Condor, collects resources into pools which are used upon request. They use advertisements of resource specifications to match against job requirements and deploy a checkpointing scheme to detect resource busy/free conditions. In contrast to the reliable Grid Resources, there is no guarantee that service capacity once freed is the same as before committed since its volume depends on current conditions of its provider. Therefore, advertisements of the same resource may differ over time giving more unreliability and uncertainty. Each pool places jobs into priority queues and if no resources are available, via flocking, they migrate to other pools (only single-hop migrations are allowed). These constraints assume that one-hop job migration is enough to discover the appropriate resources; that is, Condor Flocking relies on high degree of pools or high availability or resources.

Represent: *Service capacity specification as a non-replicable advert.*

Represent: *Adverts of same resource are not the same over time.*

Discover: *Multi-hop searches on interconnected pools.*

Condor extends resource availability by using *source-* and *server*-initiative group flocking which are the reactive (job advertisements) and proactive (resource advertisements), respectively, approaches to resource discovery problem. Applying server-initiative flocking for service capacity sharing the system needs to match network policies of both submission and execution domains whereas in source-initiative approach the discovered resource migrates and has to fully adopt the network policies of its new domain. Hoverlay is to be deployed in highly heterogeneous environments where network policies may be significantly different and server-initiative flocking practically impossible to apply. Therefore, upon successful discovery, service capacity migrates to requesting environment which may use its own policies and submission procedure; that is, discovered adverts of resources are moved from providing to requesting domains (pools) ensuring system-wide uniqueness of each resource.

Commit: *Once resource is discovered, fetch its advert into requesting domain (remove it from providing and place it in requesting pool).*

Unlike Grids which aim at job execution submitting jobs to remote resources, P2P Networks focus on discovery via transferring resources to remote requestors. P2P Systems assume volatile and potentially malicious or selfish behaviour of resources, unreliable connectivity, automatic network reconfiguration and reorganisation, are application-specific, have low maintenance costs and can grow in large numbers of nodes. Their main shortcomings that make them unsuitable for service capacity sharing are:

1. P2P Networks are designed and deployed for specific applications. They currently support replicable/cache-able resource sharing (files, distributed storage, streaming), collaborations over long-life multi-access resources (office applications) and communication over static user-driven links (VoIP and instant messaging).

2. compared to centralised architectures, they suffer from longer response times to requests due to their distributed search algorithms.

3. new nodes may join a P2P Network only via accessing a well-known existing one (entry point). There are several schemes to discover those entry points; well-maintained caches of nodes membership activities or hard-coded node addresses

which are guaranteed to be always-on [67]. Though resource sharing is not directly affected by those mechanisms, its expansion depends on single points of failure. If more than one cache is used, there is a probability the network created by continuous joins/departures be fragmented.

4. P2P Networks are heterogeneous and large-scale by nature and as such suffer from high security risks. Their security guarantees depend on security levels of its members.

***Discover:** P2P interconnection of resource pools*

Robustness and scalability of P2P Networks are the two main reasons due to which they may constitute a basis for designing a service capacity sharing system. They are architectures optimised for resource discovery and that will be their role in the proposed system, Hoverlay, interconnecting resource pools in a P2P overlay. In Structured P2P Networks the manager node of a resource has the same key as its hash; every time a resource joins a network that hash function and the discovery mechanism are activated to place the resource in a specific node. Input to that hash function could only be the service capacity as, if otherwise, only qualitative or semantic searches would be possible. These networks deploy exact-matching discovery mechanisms and thus requestor needs to know the exact hash key of a requested resource. In case of service capacity, advertisements may be complicated specifications practically impossible to hash and search in Structured P2P Networks because:

- every different value of a qualitative or quantitative property of that specification would produce different hash key,
- searches based on a subset of properties are impossible as exact-matching algorithms require prior knowledge of the full resource specification,
- every time a resource joins the network new, potentially different, specification is to be registered.

If hashing of a single property is used for locating service capacity resources, a node in Structured P2P Networks indexes resources whose property values belong to the same range. In case that node fails all resources within that range disjoin the network. Given that service capacity is highly intermittent, in worst case scenario, these procedures need to be frequently executed increasing maintenance costs of such systems.

***Publish:** resources re-register with local pool*

Resource registrations dictated by structured organisations may cancel potential benefits of placing a node in its nearest pool. Every resource, once freed, may register with a different pool from the one it used before. Therefore, frequent load fluctuations cause equally often discovery mechanism invocations which would be avoided if resources registered with the local pool. To address this issue, registration could have two phases: initially the freed resource could try to register with the pool from which it moved to the local one and if unsuccessful that local one via the structured discovery mechanism locates the appropriate hosting pool. However, assuming intermittent behaviour of pools in a heterogeneous environment that first phase would frequently fail and introduce further latencies. As analysed in 2.6, there are a number of Grid architectures achieving efficient resource discovery over large-scale Structured P2P Networks which however suffer from problems derived from the nature of these overlays.

Despite their strengths, Structured P2P Overlays are not appropriate for sharing service capacity due to their weaknesses related to hashing volatile, multi-property and intermittent resources. On the contrary, Unstructured P2P Networks achieve minimal maintenance costs in return to expensive search mechanisms. In that case, pools are interconnected in an unstructured overlay with arbitrary connections and each one of them manage a set of underlying nodes/resources. This topology resembles hierarchical two-layer Unstructured P2P Networks, like Gnutella II, with pools being the super-peers. Grouping resources into pools, apart from the benefits as analysed before, reduces search latency and space as discovery mechanisms are deployed solely on pools overlay rather than underlying networks. However, it does introduce local single-point-of-failures as a pool failure would practically disconnect both its free and underlying nodes but without or slightly affecting other pools. Resourceful pools may attract many links and a possible failure could fragment the network [30]. However, the more rewirings that pool provides the more links it attracts and the faster its resources migrate. This mechanism of rewirings reduces the lifetime of its high incoming degree and the network's dependency on such nodes.

Discover: *Overlay of pools is an Unstructured P2P Network.*

Disconnected resources need to re-register with the overlay by contacting another existing pool. If not hard-coded, these pools are discovered via dedicated caches which statistically tend to provide the most stable resources. That results to preferential attachment of new nodes and pools on the overlay which gradually converges to a power-law topology. Rewiring is an action present on most self-configurable Unstructured P2P Overlays sometimes part of a distributed search algorithm or as a result of neighbours failures. In the former case, these algorithms try to improve their efficiency by replacing a neighbour with a better one while in the latter one, a failed neighbouring node gives place to another already accessed during a query propagation. Both ways tend to prefer more stable and reliable nodes; thus, famous nodes increase their incoming degree faster that the others. Though these actions may transform the overlay of pools into a power-law topology, service capacity moves between pools; the more links a famous node attracts, the faster its available service capacity is moved away from it and thus the faster its quality of service deteriorates. Resource migrations work as a countermeasure to *winner-takes-all* phenomenon which, e.g. in globally high load, may result in a load-balanced overlay as no pool will have enough time to gather many resources and significantly increase its reliability and thus its fame.

There is now a set of requirements and principles for the architecture of the proposed system. However, search of that Unstructured P2P Overlay of pools is not sufficiently addressed yet. Resource mobility and non-replicability are features not present in resources with which existing Unstructured P2P search techniques as such deal. Apart from node and pool failure, resource mobility leads to a more unstable pools overlay as links should adapt in a way that resource movements are easily tracked. Stalkers is

Discover: *Adapt links to track node movements* designed to address those situations.

2.12 Summary

This research provides an infrastructure for sharing reusable and highly dynamic resources (service capacity) between users with unreliable machines and connectivity. Grid Computing achieves resource sharing for task scheduling purposes based on centralised entities. They assume reliability of resources and try to offer high quality of services. They support, apart from scheduling, job migration and execution checkpointing which informs the central manager about its execution state helping decisions for resubmission and/or migration. However, their weaknesses include scalability and dependency on resources reliability to achieve availability.

Peer-to-Peer Networks is another well-known class of resource sharing architectures. These systems are already deployed in large scale environments whose typical users are inexperienced individuals under various administrative domains behind firewalls and with any kind of network connection settings. While Structured P2P Networks provide guarantees about discovery efficiency in deterministic number of messages using DHT indexes, they suffer from scalability in highly dynamic environments. On the contrary, Unstructured ones cannot provide these guarantees but they are resilient to high churn rates since join/leave actions of nodes have minimal effects on network configuration. However, if a well known node fails, the network may even get partitioned or some other peers isolated.

This research tries to address the problem of network workload fluctuations adapting network size to keep nodes normally loaded. Its basic concept is that underutilized nodes from a network can be used by overloaded nodes in different networks to serve their excessive workload. The solution proposed, Hoverlay and Stalkers, moves capacity (nodes) from one network to another improving the utilization of spare capacity and helping networks to deal with high workload situations. Three steps are necessary to achieve this goal: resource publishing, discovery and commitment. The first step announces availability, then appropriate nodes are located and finally they migrate to requestor network and commited by an underlying node.

Centralized systems, due to their single-point-of-failure entities, exhibit weak robustness and scalability properties as; frequent registration and discovery requests caused by intermittent shared resources could overwhelm that central entity converting it into a bottleneck. Grid Computing platforms assume well-maintained resources and seem unsuitable to share service capacity because of their priority-based request queuing scheme increasing response latency. Structured P2P Networks cannot deal with complex queries and have high maintenance costs in frequent join/leave actions of resources. However, Unstructured ones have instant join/leave procedures with minimal maintenance costs and thus that will form the base of Hoverlay.

This chapter also provides a general understanding of unstructured search algorithms and their classes. It identifies two vertical classification methods: the multicasting size (number of nodes a query is forwarded to) and the way neighbours are selected to propagate queries they receive. However, it gives an extensive description of the blind (randomly chosen neighbours) and informed (neighbours chosen based on statistical information) classes. It also proposes an extension of informed techniques classification into Educated and Uneducated Informed ones based on whether the information they use for query propagations is explicitly collected or exists hidden in network elements (i.e. node degrees or link lifetime). The descriptions of those mechanisms will provide useful material for the design of Stalkers.

Service capacity is a non-replicable reusable resource with highly intermittent behaviour. Important principles of both well-studied resource sharing paradigms, Grids and Unstructured P2P Networks, will be used to design Hoverlay. While the former can provide useful ideas for components such as the pool of resources, reasoning and feasibility for pool interconnections, job submission toolkits and security mechanisms, the latter can be the overlay that interconnects pools easing their collaboration and improving the scalability and fault-resilience attributes to the proposed architecture. The following chapters present a preliminary model which, given some extensions and improvements, results in Hoverlay and a set of appropriate search mechanisms for sharing service capacity within that environment.

Chapter 3

Global ROME: Preliminary Model

Global ROME (G-ROME) is designed to provide an interconnection of multiple independent ROME-enabled P2P networks constructing a two-layered hierarchy. ROME [88] is a capacity sharing mechanism helping Chord rings to keep a near-optimum size and reduce their average query latency by replacing, swaping, removing and inserting nodes based on the workload each area of the ring experiences. It is a system designed and developed from the same group as the one presented in this thesis. G-ROME got inspiration from that system and tried to move it one step forward by interlinking ROME servers; the aim was to improve the scalability of underlying rings. The overlay of G-ROME is an Unstructured P2P Network of ROME bootstrap servers and enables the discovery of extra not locally available capacity to cope with workload increases.

Its design follows the guidelines of section 2.11 for pools overlay topology. ROME implements the resource publication phase whereas G-ROME provides the interconnection between ROME pools. Though it uses existing Unstructured P2P search mechanisms it introduces keywords to improve search efficiency and accuracy. Servers are enhanced to deal with both ROME and G-ROME functions and with queries directly from other overlay servers and indirectly from underlying nodes. Requests for capacity from underlying nodes are handled by ROME processes which take decisions about appropriate actions. Queries are forwarded onto the overlay only when local ROME cannot satisfy the requirements of the local ring. Each layer of *ROME→G-ROME* stack fulfils a subset of requirements as set in sections 1.5 and 2.11 so that they collectively address all related design principles. Table 3.1 maps those requirements to their appropriate layer.

Inheriting all the components from ROME, a G-ROME server is the most important component of the architecture as resource publication point and member of pools unstructured overlay. It consists of:

- a **Neighbour List** (NL) to interact with other servers on the overlay. For every unsatisfied query, servers forwards it to all their neighbours.
- a **Node Pool** (NP) to store pointers to (adverts of) locally available free nodes that represent a certain amount of capacity. This is a component already present in ROME.

Requirement	Layer
1. Pools to host free resources only	ROME
2. Each pool monitors one or more resources	
3. Each resource is monitored by one and only one pool at any time	
4. Nodes need to re-register once they change status from busy to underloaded	
5. Adverts of same resource are not the same over time	
6. Resources re-register with local pool	
7. **Service capacity specification as a non-replicable advert**	**G-ROME**
8. **Multi-hop searches on interconnected pools**	
9. **Once resource is discovered, fetch its advert into requesting domain (remove it from providing and place it in requesting pool)**	
10. **P2P interconnection of resource pools**	
11. **Overlay of pools is an Unstructured P2P Network**	
12. *Search technique able to trace node movements*	*none*
13. *Adapt links to track node movements*	

Table 3.1: *Mapping of proposed system requirements to G-ROME layers*

- a **Keyword List** (KL) to describe the application deployed on an underlying network (i.e. file sharing, lookup applications or distributed processing).
- a **Keyword Exclusion List** (KEL) to describe applications which most of the nodes in Node Pool and Chord ring are not willing to serve. Each node has a set of keywords to sketch out applications it cannot serve. A collection of most frequent keywords of all nodes in the administrative domain of a server construct this latter's KEL.

G-ROME discovery mechanism uses KEL of request originating server to find not only adequate but appropriate nodes as well. It tries to match the query with free nodes whose KELs have no common keywords with those of requester server (fully compatible nodes). This enables relationships between servers supporting relevant applications. In this architecture, it is assumed that there is one-to-one mapping between a G-ROME server and an underlying network/application. In real world systems, a server may be in charge of underlying nodes involved in interactions of more than one distributed applications. For instance, a node may participate in two completely different applications (therefore networks, too) in which case may appear in two pools with different ID and different capacity. That is, a single node may physically host two or more distributed application processes but in G-ROME each such process appears as different node.

G-ROME architecture uses a keyword-based technique to initialize and update Neighbour Lists. A new server may join the network using an existing server (*registrar*) by copying the latter's Neighbour List. The new server updates its own while receiving queries, answers and registration messages from other servers based on several criteria, as described in the following sections. Both of these operations, initialization and updating of Neighbour Lists, ensure low overhead in messages for server registration

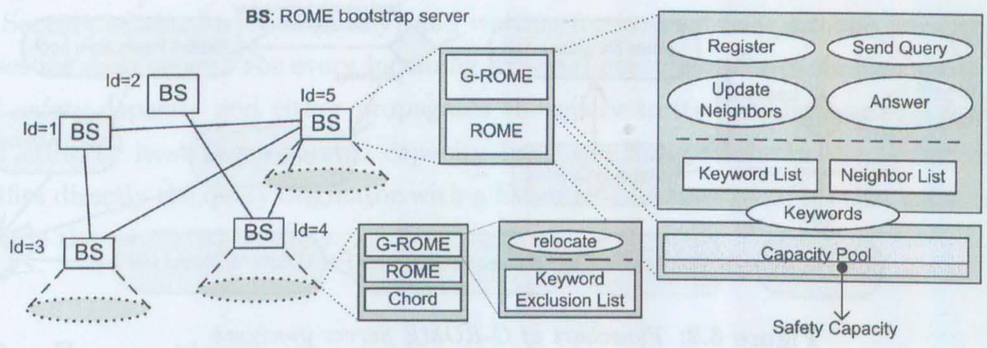


Figure 3.1: *G-ROME layered architecture: components and processes*

and maintenance in the overlay. Figure 3.1 provides a general overview of the proposed architecture.

Each bootstrap server connects to a number of other servers via its Neighbour List. Server functionality can be divided into three layers: G-ROME monitors ROME operating on top, as ROME does over Chord rings, and invokes the suitable node discovery mechanisms when necessary. Each G-ROME server supports the following operations:

1. **Register**: an existing server registers a new server or node to the overlay. A server registers with the G-ROME overlay by acquiring the Neighbour List of an existing one.

2. **Update Neighbours**: every server uses the keywords of received query or answer originator servers to update its Neighbour List.

3. **Send Query**: if a ROME server, while monitoring its Chord ring or upon receiving a query, has run out of free service capacity in its Node Pool it sends a query to all its neighbours requesting additional capacity.

4. **Answer Query**: as soon as a remote server receives a query, it answers back providing the requested appropriate capacity (if at all available) to query originator server or propagates the same query to all its neighbours.

Each server keeps a certain amount of free capacity (*safety capacity*) in its ROME pool for future use by local ring which no external requestor may use. This technique aims at reducing the number of messages transmitted in the overlay by preventing servers from sending requests whenever their rings experience slight workload fluctuations unable to satisfy with local service capacity. Both servers and nodes have to be G-ROME enabled to participate in this system. That is, servers need the G-ROME processes as described above and nodes need a G-ROME process (*relocate*) to resolve which server monitors them. *Relocate* process is invoked when a node moves to another server and its role is to kill the old ROME process that was configured to refer to the previous server and restart it parameterised with the new server address.

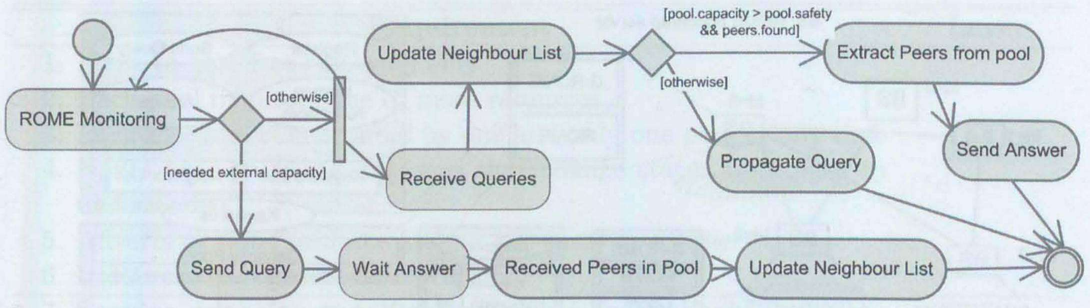


Figure 3.2: *Flowchart of G-ROME Server processes*

3.1 Architecture Overview

A G-ROME server may be in any or both of two states: a) *reply* to received queries and b) *query* for capacity modes. On external query arrival, its *reply* state is activated and all proper actions are done to either provide the requested capacity or forward the same query to its neighbours. However, internal queries trigger its *query* state and it either immediately replies back with local resources or reserves as much local capacity as possible and forwards a query on the overlay seeking for the remaining portion. If local resources in Node Pool are enough to satisfy both an internal and external query then the server can be simulataneously in both states. Safety capacity levels in Node Pool depend on the excessive workload increase rate of the underlying network. That is, safety capacity is equal to the workload increase rate times the total underlying network load. Figure 3.2 illustrates the main functionality of a server which refers to query processing (sending/propagating/answering queries).

Reserving capacity for future use by local underlying nodes practically reduces resource availability and withholds capacity from moving to servers in need. In high load situations this safety capacity reaches its maximum size as it is proportional to the requested one from underlying nodes. This adaptivity helps the system to reduce the number of queries and messages from highly loaded networks but to make maximum use of spare capacity when this is not needed locally. The level of the adaptivity depends on the history size taken into account; a small portion of past activity makes the safety capacity more responsive to workload fluctuations. Two parameters are necessary to determine the exact size of safety capacity: *a)* the percentage of requested capacity by underlying nodes and *b)* the time window size during which that requested capacity was monitored. Both these parameters are set by server administrators and accrue from calibration, policies and application type deployed on the underlying network.

If ROME monitoring identifies a node that needs extra capacity to serve part of its load but ROME Node Pool is unable to satisfy, appropriate G-ROME processes are triggered to seek that capacity in external pools by sending out to the whole NL an appropriate query. When a set of nodes that satisfy its needs is found, it places their adverts into its Node Pool and updates the Keyword Exclusion and Neighbour List with relevant details from the response originator.

Serving external queries (if any) and waiting for internal ones are two parallel processes for each server. For every incoming external query it updates its Neighbour List and safety capacity and either propagates the query to its neighbours or answers by extracting at least the requested capacity from the Node Pool. In the latter case it notifies directly the query originator with a list of IP addresses of nodes that collectively provide the requested capacity.

3.2 Semantic Cooperation

Answer refinement is necessary in the context of P2P Networks which are dedicated to specific applications. Not all discovered nodes are able to participate in an underlying Chord network. Though they may be able to provide the requested resources, they may be inappropriate or unwilling to take over a specific task. Therefore, semantics are introduced to specify special requester or provider requirements. Simple keywords are used for the semantic cooperation between servers and nodes. For simplicity and without loss of generality they are simple words or (key,value) pairs.

G-ROME uses two semantic components: Keyword List (KL) and Keyword Exclusion List (KEL). Every node has a Keyword Exclusion List to specify properties of traffic it is not willing to serve. Every server has both a Keyword List and a Keyword Exclusion List. A Node Pool can only contain nodes whose Keyword Exclusion List has no common keywords with the Keyword List of its server. This ensures that each pool strictly enlists nodes able and willing to serve traffic of that specific underlying network. For scalability purposes the server Keyword Exclusion List may contain 10% only of the most common exclusion keywords of nodes in Node Pool and underlying network (busy nodes). Both Keyword List and Keyword Exclusion List of a query originating server are included in every query forwarded to its neighbours. A server upon receiving a query, it tries to find nodes in its local pool without any common keywords in their Keyword Exclusion Lists and Keyword List of the query. This guarantees that discovered nodes will be useful in the requestor server's context; thus, compatible with the application deployed on requesting underlying network.

The server Keyword Exclusion List attached to a query is used by each visited server to update its Neighbour List. If this latter is full, one of its entries may be replaced by another or, if otherwise, appended. If the query originator server is less *semantically distant* from a visited server, this latter replaces one of its neighbours with the former. *Semantic distance* is defined as the number of common keywords between a server Keyword List and the Keyword Exclusion List of another. Symmetrically, *semantic closeness* is the number of non common keywords between a server Keyword List and Keyword Exclusion List of another. Every server, upon receiving a query or answer, compares its Keyword List with the Keyword Exclusion Lists of that message originator and its neighbours. Based on this comparison, servers choose the semantically closest neighbours, thus ensuring that they are directly connected to neighbours that will probably offer more compatible answers.

This rewiring scheme uses semantics to bring closer Chord rings that would be more willing to share capacity if available. However, both servers KL and KEL adapt to node migrations as keywords move in and out their context. For instance, a server gets notified via queries sent by one of its neighbours that this latter's KEL gets populated with keywords incompatible to its own; in this case, rewiring may be triggered and break the link between those two servers. Though a server may serve the same application, its links change based on the nodes moving in/out; two servers may lose their direct connection even if their applications are semantically related. Therefore, servers practically experience semantic cooperation with the nodes rather than the server of a remote neighbouring domain. This scheme is a dynamic quality measurement mechanism of servers as it allows them to link to neighbours that are more likely to help in the near future given their current capacity quality (semantics) and availability.

3.3 Registration Process

There are two different registration processes for a G-ROME-enabled server: a) node registration with its local server and b) server joining to the G-ROME network. A new node registers with a server if their lists, Keyword Exclusion List and Keyword List respectively, have null intersection. The server updates its Keyword Exclusion List when a new node registers with it. Once a node has registered with a server, it keeps the connection open using heartbeats so that the server can promptly identify changes in node's capacity size and understand that it has failed.

By joining G-ROME, a server initializes its Neighbour List used for sending queries to other servers. Initially, it sends a registration request to another registered server (*registrar*) which, in response, provides its own Neighbour List. It repeats the same registration messages to the newly acquired neighbours aiming at filling its list with the most semantically close neighbours. That is, out of all Neighbour Lists it receives, it keeps only servers as neigbours who have the biggest semantic similarity. To reduce the probability of overlay fragmentation, any server that receives a registration request has to register the new server with its own Neighbour List. That point onwards, it is able to initiate queries into G-ROMEand is treated as any other server.

The initial neighbours of a new server may have very few keywords in common to their lists. However, the update process, which is triggered through outgoing or incoming queries, ensures that every server tends to improve its Keyword List semantic closeness with those of its neighbours.

3.4 Query Processing

A server generates queries for service capacity discovery whenever its Node Pool cannot fully satisfy requests from its underlying ROME. Every query needs a set of fields to support *a)* appropriate propagation, *b)* Neighbour List updating and *c)* keyword-based node selection. It contains the requestor address, its Keyword List and Keyword Exclusion List; this way it describes the applications that any discovered node must support and those that the most free and busy ones of the originator server are not willing to serve.

Every query travels on G-ROME overlay by hopping between neighbouring servers using their Neighbour Lists commencing from its originator. Once a query is received, the current server inserts query originator into its Neighbour List based on the semantic neighbour list updating scheme and forks the same query to all its neighbours if unable to satisfy the request. Attaching the address of that server to the forked queries and using a Time-to-Live (TTL) counter to set a maximum number of hops a query may travel on the overlay are two ways to prevent queries from getting trapped into deadlock and infinite loops situations.

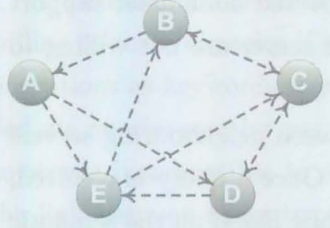
There are three conditions to be fulfilled before a response from a server is sent back to query originator: a) that server has more service capacity in its pool than the minimum required safety capacity, b) has one or more nodes that can collectively provide the whole required capacity (partial satisfaction of requests is not a valid answer) and c) the intersections between query Keyword List and Keyword Exclusion Lists of selected nodes are null. Otherwise, the TTL counter drops by one and if that is bigger than zero, the query is forwarded to neighbours of that server. Partial answers would have required the deployment of capacity reservation schemes which could in turn result in deadlocks. Query originator server can always select the first received answer because it is guaranteed to provide the required capacity and as a result the server does not have to wait for query propagation until the TTL-defined network horizon.

Each answer, apart from discovered node details (i.e. address and Keyword Exclusion List), includes the address, Keyword List and Keyword Exclusion List of the responding server. A requestor, receiving an answer, checks whether it could replace any of its existing neighbours with that responding server. This replacement takes place only if the responder is semantically closer than at least one current neighbour; the most semantically distant is replaced.

The requestor needs two more messages to complete the migration of discovered nodes. It sends back to the provider server an acceptance confirmation and a message to every fetched node to invoke its G-ROME '*relocate*' operation. This operation is active only when the node is in standby mode in a Node Pool. This ensures that G-ROME servers cannot communicate with busy nodes of their underlying networks but only with nodes in Node Pool.

3.5 Worked Scenario

A comprehensive hypothetical scenario with a network of five servers is presented below to show the functionality of G-ROME. Each server has only two neighbours to forward queries to with $TTL = 2$. Servers manage a subset of system keywords which collectively count up to ten. Table 3.2 presents the directional graph of this example network providing the configuration of those five servers including their Neighbour Lists, Keyword



Server	NL	KL	KEL	Node Pool	SF
A	D,E	a,c,h	f	$(n_1, 2, \{d, f\})$ $(n_8, 1, \{b, f\})$	8
B	A,C	b,d,i,j	a,g,h	$(n_5, 3, \{g\})$ $(n_6, 7, \{e, f\})$	4
C	B,D	d,e,g	a,b,h	$(n_7, 6, \{b, j\})$	3
D	C,E	b,c,e,g,i	d,f	$(n_2, 5, \{d\})$	7
E	B,C	a,e,d,f	c,h,j	$(n_3, 3, \{b, i\})$ $(n_4, 7, \{h, j\})$	5

Table 3.2: *G-ROME Worked Example network initial configuration: Neighbour List (NL), Keyword List (KL), Keyword Exclusion List (KEL), Node Pool and Safety Capacity (SF) of every server*

Lists, Keyword Exclusion Lists and Safety Capacity.

1. *Query generation from A*: Server A generates a query requesting for capacity of 5 units. Though its local Node Pool has two nodes, they are insufficient to satisfy the requested capacity. Since no reservation is allowed, these nodes remain free in the Node Pool and propagates a query on the overlay. The query contains the requested capacity $CAP = 5$, $TTL = 2$, the server's $KL = a, c, h$ and $KEL = f$ forwarded to neighbours E and D:

   $$Q_1 \text{ from A to D \& E: } CAP = 5,\ TTL = 2,\ KL = \{a, c, h\},\ KEL = \{f\}$$

   The index of Q denotes the number of hop that query has traveled within the overlay.

   This query propagation triggers a series of reactions to its subsequent receivers. These reactions are illustrated in figures 3.3b through 3.3d and analysed on the following three steps.

2. *Query reaches D and E*: Both servers have not received the same query before and, hence, they accept and process it. Server D contains node n_2 in its Node Pool which *a)* satisfies the requested capacity and *b)* whose KEL has no common keywords with query's KL (i.e. $KEL_{n_2} \cap KL_A = \emptyset$) but *c)* its safety threshold (7) is already above the available capacity (5). Therefore, the query originator cannot use that node and server D will forward the same query to its neighbours C and E, reducing the TTL to one since it has already travelled one hop:

   $$\boldsymbol{Q_2} \text{ from D \& E to B \& C: } CAP = 5,\ TTL = \mathbf{1},\ KL = \{a, c, h\},\ KEL = \{f\}$$

   Server D updates its neighbour list by replacing C with A since the intersection of its KL with the query originator's KEL is smaller than that with server C (if two or more of those intersections give same size sets, one of them is chosen randomly to be replaced):

   $$KL_D \cap KEL_E = \{c\},\ KL_D \cap KEL_C = \{b\},\ KL_D \cap KEL_A = \emptyset$$

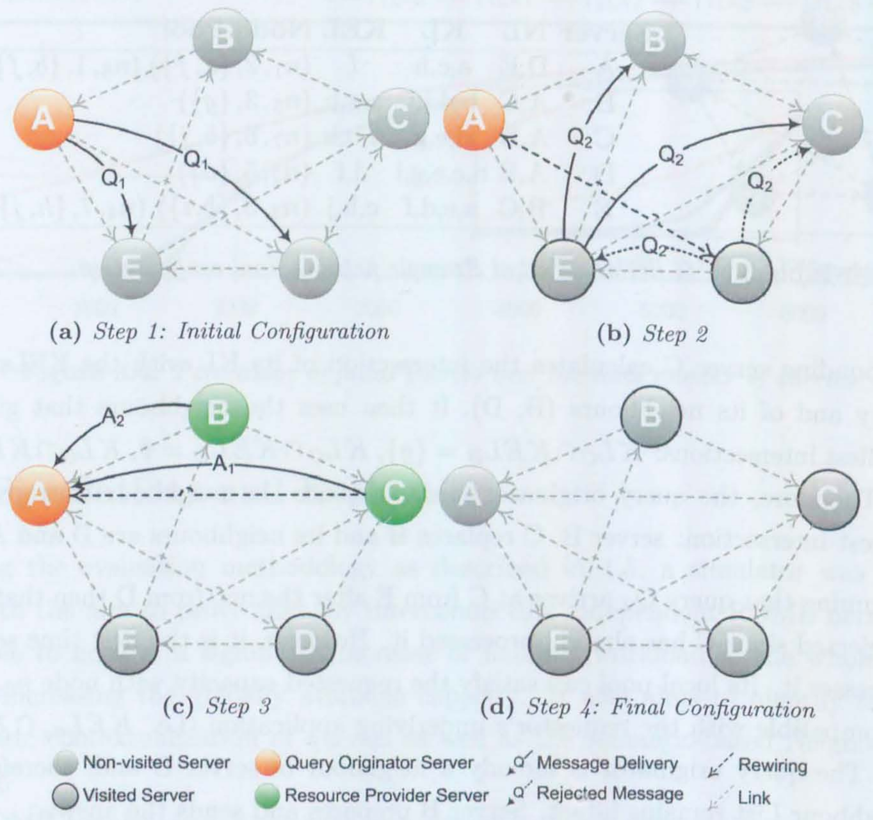
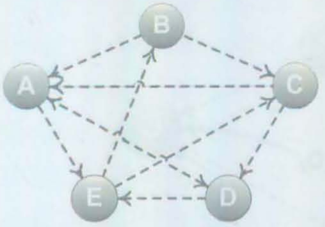


Figure 3.3: *G-ROME Working Example: query propagation, answer delivery and network reconfiguration*

As soon as the query reaches server E, its Node Pool detects one node that could satisfy the requested capacity. However, both query's KL and node's KEL contain keyword $\{h\}$ (i.e. $KEL_{n_4} \cap KL_A = \{h\}$). Therefore, this node cannot be used in the requesting underlying network application. Server E cannot serve the capacity and, thus, will forward the query (with TTL=1) to its neighbours B and C. Its KL intersections with KELs of the query and its neighbours all give one keyword. Therefore, there is no change in the Neighbour List of server E.

3. *Query reaches B and C*: Server D links with C and E; as server E has already received the query Q_1 during the previous step from server E, it rejects the incoming one. It is the first time that server C receives Q_2 and therefore accepts it. Its Node Pool has one node n_7 with more available capacity than the requested and the safety and its KEL has no common keywords with the query's KL (i.e. $KEL_{n_7} \cap KL_A = \emptyset$) . It then sends an answer with this node to query originator server, even though the available capacity in its pool becomes zero, much lower than the safety. Server C prepares and sends back to the requestor an answer with its own KL and KEL and the discovered node removing the node from its Node Pool:

$\boldsymbol{A_1}$ from C to A: $KL = \{d, e, g\}$, $KEL = \{a, b, h\}$, Nodes:$(n_7, 6, \{b, j\})$



Server	NL	KL	KEL	Node Pool	SF
A	D,E	a,c,h	f	$(n_1, 2, \{d, f\})$ $(n_8, 1, \{b, f\})$	8
B	A,C	b,d,i,j	a,g,h	$(n_5, 3, \{g\})$	4
C	A,D	d,e,g	a,b,h	$(n_7, 6, \{b, j\})$	3
D	A,E	b,c,e,g,i	d,f	$(n_2, 5, \{d\})$	7
E	B,C	a,e,d,f	c,h,j	$(n_3, 3, \{b, i\})$ $(n_4, 7, \{h, j\})$	5

Table 3.3: *G-ROME Worked Example network final configuration*

Responding server C calculates the intersection of its KL with the KELs of Q_2 query and of its neighbours (B, D). It then uses the neighbours that give the smallest intersections: $KL_C \cap KEL_B = \{g\}$, $KL_C \cap KEL_D = \emptyset$, $KL_C \cap KEL_A = \emptyset$. Therefore, the query originator has to replace the neighbour that gives the biggest intersection: server B. C replaces B and its neighbours are D and A.

Assuming that query Q_2 arrives at C from E after the one from D then that query is rejected since C has already processed it. However, it is the first time server B processes it. Its local pool can satisfy the requested capacity with node n_6, which is compatible with the requestor's underlying application (i.e. $KEL_{n_6} \cap KL_A = \emptyset$). The query originator is already a neighbour of server B and, therefore, its Neighbour List remains intact. Server B prepares and sends the answer:

$$\boldsymbol{A_2} \text{ from B to A: } KL = \{b, d, i, j\},\ KEL = \{a, g, h\},\ \text{Nodes:}(n_6, 7, \{e, f\})$$

4. *Query originator A receives both A_1 and A_2 answers*: Assuming that A_2 (from server C) arrives first, server A places the discovered node into its Node Pool. It tries to update its Neighbour List calculating the intersections of its KL with the answer's and its neighbours' KELs:

$$KL_A \cap KEL_E = \{c, h\},\ KL_A \cap KEL_D = \emptyset,\ KL_A \cap KEL_B = \{a, h\}$$

There is no change to the Neighbour List of A as B is not semantically closer compared to its existing neighbours.

Answer A_1 is rejected because the query it refers to is already answered. However, the new answer activates the Neighbour List updating scheme but no neighbour gets replaced since:

$$KL_A \cap KEL_E = \{c, h\},\ KL_A \cap KEL_D = \emptyset,\ KL_A \cap KEL_C = \{a, h\}$$

The final state of the network in this scenario is shown in Table 3.3. Note that the migrated node is not listed in A's Node Pool as it was immediately placed within its underlying Chord ring to serve the request.

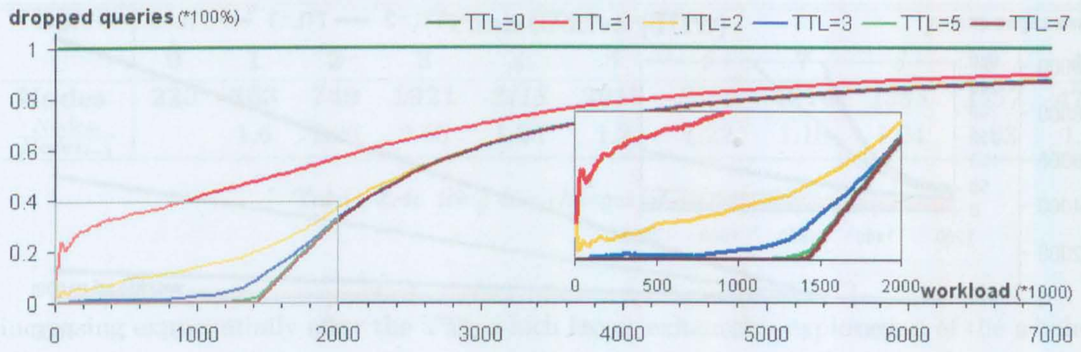


Figure 3.4: *Percentage of failed queries over the total number of queries*

3.6 Simulations and Evaluation

Following the evaluation methodology as described in 1.3, a simulator was built in C++ with the aim to prove that, by interconnecting independent ROME networks, it is possible to achieve a significant increase of handled workload in the whole system without increasing the globally available capacity. It also aims to identify the effect of semantic contextualization of queries as well as the semantic-based Neighbour List updating.

G-ROME simulator executes each experiment in three steps:

1. system-wide keyword production. That is the global set of keywords that exist in a simulated network of servers.

2. server construction and initialization with initial Node Pool, Keyword List, random Neighbour List and no ring. Each server is represented as a composable object by: Pool, Keyword List etc.

3. for every simulated TTL the simulator executes a number of iterations linearly increasing the global workload in every iteration. This workload is randomly distributed on a random subset of servers. Some servers may get a negative workload so in each iteration some rings shrink and new available capacity is produced.

During experimental execution, there are several metrics to record for every TTL and iteration: number of queries, number of failed queries, number of messages and ring size of every underlying network. It is assumed that no servers fail during the experiments. Furthermore, the server registrations take place only during their initialization. Their Neighbour Lists are initially filled with random servers and thereafter they get updatedy through the processes described above.

Below, a set of experiments demonstrate the effect of TTL on discovering the appropriate resources in terms of number of failed queries (capacity not found) and total number of generated messages. While the first one reveals the benefits of G-ROME, the second presents potential costs produced by deploying it. The last of those three experiments illustrates the ability of a single ring to handle much more workload when its

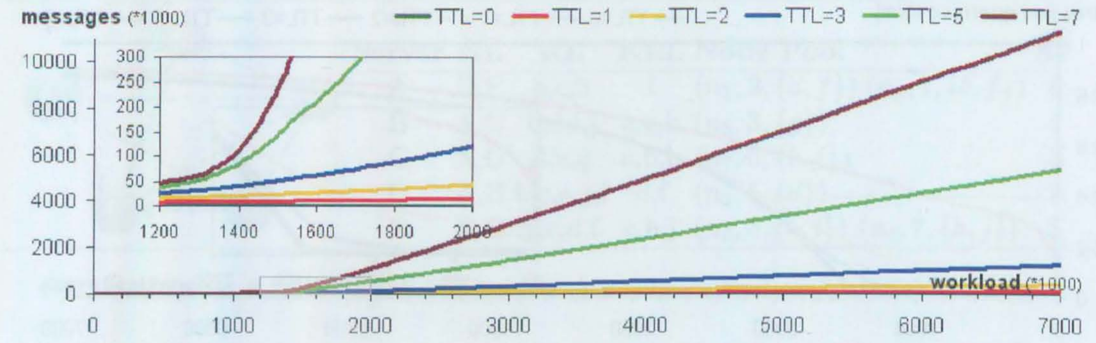


Figure 3.5: *Number of messages generated with the increase of the workload*

server participates in a G-ROME network than it would alone. The size of the network for these experiments is 1,000 servers each having a maximum initial capacity of 500 units. The size of each neighbour list is set to 3 entries and the system-wide keyword superset is 10,000 keywords of which a maximum of 10 are randomly assigned to each server and 3 to each node. The experiments were run with 1,000 iterations and a fixed global workload increase per iteration of 6000 units. On every iteration a random subset of servers are selected to assign the workload; no more than 10% of it is assigned to each one. TTLs {0, 1, 2, 3, 5, 7} were tested.

Figure 3.4 illustrates the reduction in failed queries when the separate ROME networks are interconnected and figure 3.5 presents the cost (number of messages) of these queries with the increase of TTL when interconnecting the G-ROME enabled servers. In disconnected ROME networks (TTL=0) the rings cannot handle more workload and any effort to discover more capacity would result in query failure. G-ROME enables them increase their capacity and with certain TTL and workload increase, they can find all the requested capacity every iteration. Linear increase of the system-wide workload causes exponential increase in the number of messages.

Figure 3.4 shows that initially the lower the TTL the more queries are failed. This is because of the small search depth within which the available resources are limited and further constrained by semantic-based discovery. Furthermore, the increase rate of failed queries percentage is initially lower since the semantic-based Neighbour List updating creates links to servers with enough spare nodes to serve a limited workload. As these get exhausted the failed queries increase rapidly. At any TTL, the number of failed queries is less than or equal to that of TTL=1. Increasing TTL does not always give better results since any TTL bigger than a certain point causes exhaustive exploration of the whole network; that point appears to be TTL=5. Finally, the network is flooded by huge number of messages without any benefit over non-connected independent ROME networks (TTL=0).

In Figure 3.5, the base line (TTL=0) represents a scenario with disconnected ROME servers which do not produce any messages since there are no interconnected servers. The distance between lines for different TTLs increases exponentially. As TTL increases, the number of servers that are explored increases exponentially too. This distance stops

	Time-to-Live (TTL)										
	0	**1**	**2**	**3**	*4*	*5*	*6*	**7**	*8*	*9*	**10**
Nodes	**226**	**363**	**749**	**1921**	*2418*	**2915**	*3565*	**4214**	*4385*	*4557*	**4728**
$\frac{Nodes_i}{Nodes_{i-1}}$		1.6	2.06	2.56	1.25	1.2	1.22	1.18	1.04	1.03	1.03

Table 3.4: *Ring size changes of server 788*

increasing exponentially after the TTL which forces exhaustive exploration of the whole network. Any bigger TTL would produce almost the same number of messages since query cyclic paths are avoided appending the visited server address on every query.

The actual benefits of G-ROME become more evident in Table 3.4 which illustrates the ring size increase of a single server as the TTL increases. During that experiment the simulator chose randomly to monitor the ring size of server 788. At TTL=10 the ring size is almost 21 times bigger than in TTL=0 (equivalent to a disconnected ROME server). As long as queries explore bigger parts of the network while TTL increases, more fresh capacity is discovered making the ring size augment. However, if TTL is beyond a certain point, each query explores the whole network exhausting most of its chances to find new resources and thus ring size tends to stabilise. In such situations, new resource availability depends on workload fluctuations and task completion rate of other rings. All data in bold fonts represent statistics collected by experiments whereas those in italics are intervals calculated assuming linear relations between observed data. The last line of that table gives the mathematical derivatives while number of visited nodes change as TTL increases.

G-ROME enables ring sizes to increase while there is available capacity on the interconnected servers. Nevertheless, if workload increases further than the globally available capacity, G-ROME stops having any fundamental benefit even if queries make an exhaustive search of the network. In general, a server would need bigger TTL to discover the required capacity in case of scarce as opposed to plentiful available capacity in the network. Assuming that the system-wide workload is uniformly distributed over rings as far as the available capacity is adequate, small TTLs appear to be suitable for finding the requested capacity. As the global available capacity becomes scarce slightly bigger TTLs are required. On the other hand, if workload is distributed on a small set of servers (query hotspots) large TTLs appear to be necessary to find the requested capacity in case of scarce available capacity and relatively small TTLs in case of plentiful available capacity.

3.7 G-ROME Weaknesses

Though G-ROME design follows all requirements of section 2.11, it has a number of weaknesses. It is tightly coupled to ROME processes and Chord networks as these two have taken over a substantial part of those requirements. It fulfils the aims of the current research but it is an application-specific (Chord) model. Overlay organisation of servers is a principal design requirement of this research and thus any proposed architecture should not rely on underlying applications/networks but provide necessary processes to realise its targets. Full support of heterogeneity will inevitably augment design hurdles especially in *commit* phase of remote resources. Topologies, protocols and distributed algorithms (i.e. join/leave node actions) may differ among underlying networks. Therefore, migrating nodes between networks requires an appropriate task submission mechanism so that any new node joining a network is able to register with it and participate in its traffic. These actions are interesting issues but beyond scope of this research which is to discover and deliver appropriate resources to a requestor; it then takes over their registration to its network.

As also stated in the introductory part of this chapter, G-ROME does not fulfil all requirements of search mechanism as its aim is primarily to evaluate feasibility. Therefore, existing discovery algorithms were used which however are optimised for replicable fixed-location and multi-access resources. Service capacity does not fall into this category and requires more adaptive algorithms able to trace resource movements between networks. G-ROME deploys a semantic rewiring scheme to enhance search efficiency giving priority to links between semantically close servers. This technique gradually populates Neighbour Lists with links only to the closest servers which, under certain circumstances, may result in isolated cluster formations. That is, semantic close servers develop links between each other discarding more distant ones creating well-connected clusters which, however, have no links to servers outside those clusters. In that case, the whole overlay gets fragmented into several smaller G-ROME networks reducing resource availability each server has access to; search efficiency may be affected since queries cannot escape those clusters.

Each query carries a set of server addresses which compose its propagation path. This ensures that no such path contains cycles without, however, eliminating server revisiting. As a query forks to a subset of neighbours in each server of its path, there are a number of parallel query paths originated from same requestor. Each such path is not aware of its 'sibling' paths and therefore it may revisit a number of servers already reached by a parallel similar one. Instead of appending visited server addresses at the end of a query, query ids should be cached for some time in every server.

3.8 Summary

This chapter describes the G-ROME architecture, which aims to reduce the number of failed user queries in DHT-based networks that experience high workloads. G-ROME achieves this by first creating a Gnutella-like interconnection of independent P2P networks and second by providing mechanisms that enable overloaded networks to use the underlying interconnection in order to acquire spare nodes from underutilized networks. The resulting system is simple and a number of experiments have shown that it can significantly increase a network's ability to deal with considerable increases of its workload.

Chapter 4

Hoverlay: Architecture Specification

Hoverlay is an Unstructured P2P-based management system that enables sharing of non-replicable reusable resources focusing on service capacity. It facilitates the cooperation of heterogeneous networks for improving spare (currently not used) capacity utilization in the whole system. Its overlay consists of a set of interconnected servers each of which represents nodes of an underlying network. A node can be a mobile device, workstation, cluster, supercomputer or any arbitrary interconnection of them. It aims at transferring nodes from one underlying network to another so that their capacity can be shared between networks.

Sections 1.5 and 2.11 form the frame within which the proposed architecture needs to be designed. G-ROME, which heavily relies on ROME (see section 3.7), only partially addresses the principles described there. Hoverlay uses a number of G-ROME features supporting at the same time wider heterogeneity and addressing the latter's shortcomings. Though, as all Unstructured P2P Networks, its efficiency heavily depends on deployed search algorithms, its design is not tightly coupled to any of them. It provides flexibility of deploying various ones. However, certain aspects of its design may affect search algorithm designs. The algorithm to be adopted for searching service capacity resources, Stalkers, is studied in chapter 5. Following a similar table as the 3.1 one, its requirements become clear.

Hoverlay is a two-layered architecture (servers overlay and underlying nodes/resources) resembles on certain points to G-ROME. However, unlike that, Hoverlay needs to address all those requirements without depending on ROME or Chord. Underlying nodes may be in any kind of centralised or distributed formation: P2P (Unstructured/Structured), Grid, client-server etc. As an effort to avoid overlay fragmentation, it does not use semantic contextualisation of resources or requests. However, rewiring is a technique used here, as well, not for server clustering but rather resource traceability purposes as thy migrate from server to server. Finally, message caching by every server is introduced to prevent cycles on query propagation paths.

Hoverlay is based on a set of assumptions to make its concept feasible. All nodes accept incoming connections either laying behind a firewall/NAT or not. This is an implementation-oriented problem but rather important for deploying Hoverlay in real

Requirement	Layer
1. **Pools to host free resources only**	Hoverlay
2. **Each pool monitors one or more resources**	
3. **Each resource is monitored by one and only one pool at any time**	
4. **Nodes need to re-register once they change status from busy to underloaded**	
5. **Adverts of same resource are not the same over time**	
6. **Resources re-register with local pool**	
7. **Service capacity specification as a non-replicable advert**	
8. **Multi-hop searches on interconnected pools**	
9. **Once resource is discovered, fetch its advert into requesting domain (remove it from providing and place it in requesting pool)**	
10. **P2P interconnection of resource pools**	
11. **Overlay of pools is an Unstructured P2P Network**	
12. **Adapt links to track node movements**	
13. *Search technique able to trace node movements*	*Stalkers*

Table 4.1: *Mapping of proposed system requirements to G-ROME layers*

environments. There is significant work [91], [95], [19], [50], [105] on these issues with a variety of solutions for different both hypothetical and real environment settings. Each server may only (if any) provide the whole requested capacity and no resource reservation scheme is deployed to enable partial answers. Avoiding partial answers simplifies the discovery mechanism and helps overlay to deal with resource variable availability and high failure rates of nodes and servers. Reserving unreliable nodes could easily result to deadlock situations and increase the query failure rates.

4.1 Architecture Overview

A Hoverlay server uses only a random list of other server IDs (Neighbour List) to share its own resources and discover new ones in their overlay. Whenever necessary, a local (requesting) server forwards queries originated from underlying nodes (internal queries) to its neighbours and waits for an answer. A Node Pool, embeded in every server, keeps records of available underutilized nodes and tries to satisfy an internal query using that capacity reserving as much as possible. Any extra amount of capacity (if at all), not provided by local pool, is queried to neighbouring servers. Each request has a lifetime which is the maximum time a requesting server may wait for answers from the overlay. A query terminates if its lifetime expires or an answer is received.

In case of an external query (sent by another overlay server), servers try to completely satisfy it using capacity in their local Node Pool only. A pool reserves, if there is locally sufficient capacity, at least as much as the query requirements and initiates a handshaking protocol to deliver those resources to query originator server. Otherwise (not enough capacity available), it forwards the same query to its server neighbours

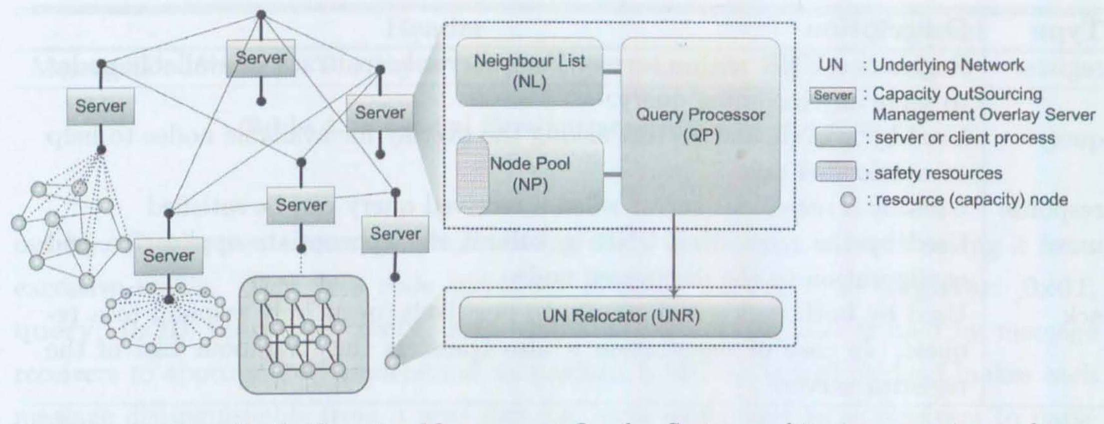


Figure 4.1: *Capacity OutSourcing Management Overlay System architecture overview and components*

without reserving any resources. Resource reservations from remote servers follow two rules: *a)* responders should fully satisfy the required capacity and *b)* each resource must be fully reserved i.e. a server cannot reserve a portion of its capacity. Though a server tries to reserve just about enough resources to address the requirements, the indivisibility feature of service capacity may force it exceed the required amount. Figure 4.1 illustrates the main Hoverlay components.

All three Hoverlay server components (Neighbour List, Node Pool and Query Processor) are for handling incoming queries and answers. The Underlying Network Relocator resides in every underlying node accomplishing its logical movement as well as monitoring its workload:

- **Neighbour List (NL)** determines the next destinations of a forwarded query. It applies the server forwarding policy (if flooding all neighbours are selected, a subset otherwise). Due to intermittent behaviour of service capacity, NLs implement blind search techniques only as otherwise information collected in case of informed techniques would quickly get out-of-date (see section 2.9). A number of different search techniques are under evaluation to identify key features that affect search efficiency and would play important role in the design of Stalkers. Following architectural requirements, there is no monitoring or updating scheme to guarantee good direct connection of servers with their neighbours; thus reducing maintenance costs of these lists and practically unbinding system efficiency from resource reliability. This may improve system applicability in large-scale networks. However, the list gets refreshed either upon receiving and answer replacing the oldest neighbour with the answer originator server or periodically (manually defined time intervals) with the external incoming query originators. These frequent updates help on keeping the overlay connected.

- **Node Pool (NP)** stores adverts of free nodes until they are reused. Internal queries may reserve any amount of capacity available whereas external ones can only reserve the capacity that fully satisfies their requirements. Node Pools keep some of the available capacity (safety capacity) for use by underlying busy nodes

Type	Description
register	Registers a UNR with a server. Each server keeps track of available nodes to serve any incoming query.
query	Used by a UNR and servers asking the overlay for available nodes to help an overloaded one.
response	Used by a responder server when a received query can be satisfied
move	Used by the requesting UNR to submit the appropriate application and configuration to the discovered node
ack	Used by both nodes and servers to positively/negatively respond to a request. In case of registration it also transfers the Neighbour List of the registrar server.

Table 4.2: *Overlay message list description*

only. No external query is satisfied if the capacity availability in a Node Pool is lower than its safety level. Safety capacity is used to serve only internal queries produced by small fluctuations of workload of underlying nodes. It prevents a large number of queries from being forwarded to the overlay. The safety capacity size is a percentage of the average requested capacity from the underlying nodes within the last few time units (time frame). This percentage and time frame are application specific and are configured by server administrators.

- **Query Processor (QP)** processes any incoming query and performs all the communication activities of a server. It caches any internal and external query for a given period of time and interacts with Node Pool to satisfy it, if possible responding back or forwarding it to neighbouring servers otherwise. In case of internal queries, it waits for the answers. As soon as an answer is received it merges the discovered capacity with that reserved in local Node Pool, if any, and acknowleges back.

- **Underlying Network Relocator (UNR)** resides in underlying network nodes and is responsible for controlling node relocations from one network to another. It is used by remote servers that nodes migrate to.

4.2 Overlay Interaction Messages

Hoverlay communication protocol defines five messages for registering an underloaded node with a server, asking overlay for underloaded capacity, responding with a list of nodes and realizing the actual migration of a node from one underlying network to another as shown in Table 4.2. Figure 4.2 demonstrates the interaction of nodes and servers in case of server and node registration, query forwarding and node migration. Each Hoverlay message has a header and a payload. The header has fixed size and consists of four fields and the payload has variable number of fields and size. Header field sizes follows those of a Gnutella message header [85] as shown in Table 4.3.

TTL represents the maximum number of steps a message may travel on servers

Header				Payload
MessageID: 16byte	TTL: 1byte	Type: 1byte	Length: 4bytes	...

Table 4.3: *General Hoverlay message header structure*

overlay. This prevents messages from visiting every node of the network saving it from excessive traffic. *Type* is a code specifying message payload type (**register**: **0x01**, **query**: **0x10**, **response**: **0x11**, **ack**: **0x00**, **move**: **0x20**) and is used by message receivers to appropriately understand its payload fields. *Length* of payload makes each message distinguishable from a next one e.g. in a buffer and helps receivers to parse variable-length payloads.

Message ID is a 16-bit message unique identifier. Any server should not process a query more than once to prevent infinite loops and unnecessary messages on the overlay. Each server, upon receiving a message, uses its Message ID and its own local cache to check whether that has already been processed. A query lives in a cache for a total of TTL-hops+1 time units, where TTL-hops is the number of remaining hops a query may travel beyond current server. A node uses the Message ID of an answer to verify that it relates to a, not yet answered, query it generated sometime in the past.

Some fields are common for more than one message types. *IP* and *Port* are essential for messages that require connection establishment between two entities to be transmitted i.e. *register*. *query*, *response* and *move* messages. Capacity specification is necessary for *register*. *query* and *response* messages.

- **Port**: (2 bytes) listening port of a message originator at which an incoming connection may be accepted
- **IP**: (4 bytes) IP address of message originator
- **NT**: (2 bytes) network throughput in kbits per time-unit that a node requests or makes available
- **PT**: (3 bytes) processor throughput - the first byte is used for the percentage of

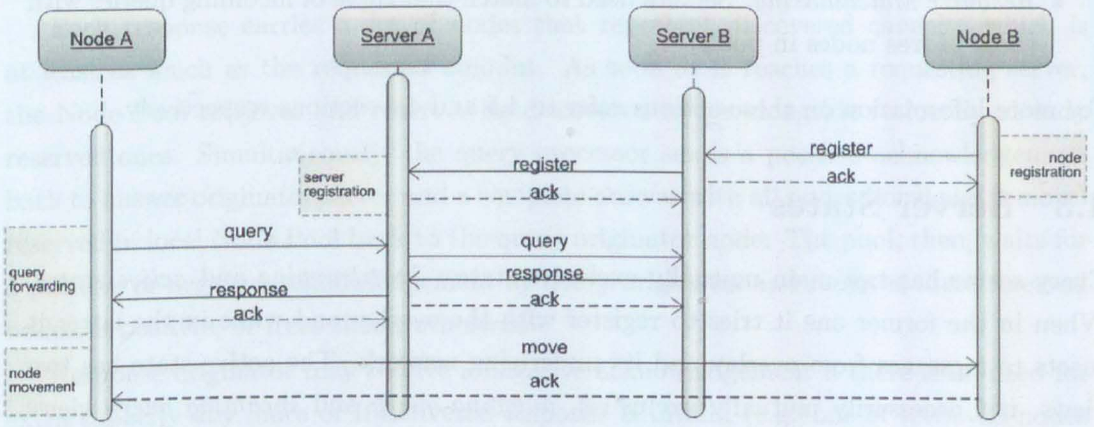


Figure 4.2: *Hoverlay communication sequence diagram*

Type	Fields
0x01	**[Port|IP|NT|PT]** — IP and Port for incoming connections and the maximum shared capabilities of the node/server to be registered with the overlay
0x10	**[Port|IP|NT|PT|S:1byte]** — IP, Port, requested minimum NT and PT of the requesting server. The response has to contain nodes each of which has a minimum NT and altogether reach PT CPU speed if S=0x00 or otherwise each one has a minimum PT and altogether the NT.
0x11	**[Port|IP|NT|PT]**$^{1 \cdots *}$ — A list of IP, Port, maximum shared NT and PT of discovered nodes
0x20	**[EL:4bytes|E|FL:4bytes|F]** — The executable E of size EL (in bytes) is necessary for a migrating node to participate in the new network. Its configuration file F has size FL (in bytes).
0x00	**[A:1byte|(Port|IP)**$^{1 \cdots *}$**]** — A=0x01 if the acknowledgement is positive and A=0x00 if negative. If used in reply to a registration request a list of Port,IP tuples are followed representing the Neighbour List of registrar server.

Table 4.4: *Hoverlay message payload description*

the CPU usage required or shared and the remaining two for its nominative CPU cycles in millions per second (MHz)

Table 4.4 shows the payload structure of each message type. Boolean (i.e. S, A) fields must have the minimum size of one byte. As in Gnutella Specification [36] fields that represent file sizes (i.e. EL, FL) use four bytes. It is assumed that an executable file and its configuration in a *move* message cannot be more than 4.2Gb. In case the executable startup requires a large dataset, it should use an application specific protocol to fetch and/or access that dataset. *Port*, *IP* and *NT* field sizes are adopted by Gnutella Specification, too. The *PT* field allows numbers up to 65536 expressing CPU speed in MHz. Hoverlay has to take two actions with regards to these *PT* and *NT* values:

- *Application monitoring*: UNR has to monitor changes of both these values and generate queries requesting the appropriate capacity (tuple of those values),
- *Resource Matchmaking*: Servers need to match that tuple of incoming queries with those of free nodes in pools.

For more information on these actions refer to 4.5 and 4.4 sections respectively.

4.3 Server States

Every server has two main mutually exclusive states: *bootstrapping* and *active* states. When in the former one it tries to register with the overlay and while in the latter it reacts to messages from overlay and its underlying network. The active state has two views, not necessarily mutually exclusive: *outgoing query* and *incoming query* views which refer to handling a query coming from an underlying node or an overlay server, respectively.

4.3.1 Bootstrapping State

A bootstrapping process is invoked if a server has empty Neighbour List or it has received no answers or external queries for some time; on any of these cases the server is practically disconnected from the overlay. Server bootstrapping while joining the overlay is equivalent to its neighbour list initialization. A new server (*unregistered*) uses an existing one (*registrar*) to retrieve a fresh Neighbour List. Both registrar and unregistered servers have to record each other as one of their neighbours, rejecting randomly another one if necessary, thus reducing the overlay fragmentation or server isolation probability. This registration process ensures low cost of the overlay bootstrapping and expansion.

As soon as a registration message reaches an existing server, taking the role of a registrar, it responds with an acknowledgement message accompanied with its own Neighbour List. The unregistered server uses it as its own temporary Neighbour List to send the same registration message to all its new neighbours which respond with their own neighbours. It appends to this list those new neighbours until its Neighbour List is full or no more acknowledgement messages are received. If no acknowledgement messages are received within a pre-set waiting time, it retries and exponentially increases that waiting time.

4.3.2 Outgoing Query State

Underlying nodes construct and send the appropriate queries to their local server whose Node Pool responds to Query Processor with an empty answer, if no nodes can be found, or, otherwise, with a list of nodes that satisfy as much of the requested capacity as possible. If that query is still not completely satisfied the Query Processor forwards it to a subset of its neighbours requesting for the remaining capacity that the local pool cannot provide and reducing its TTL by 1. Otherwise, a proper answer is sent back to requesting node. Each query is copied in cache before Query Processor forwards it and waits for answers. As soon as its lifetime expires or first answer arrives, the cache removes the query. Any answer received without its corresponding query pair being in cache is rejected and negatively acknowledged to its originator.

Each response carries a set of nodes that represent discovered capacity which is at least as much as the requested amount. As soon as it reaches a requesting server, the Node Pool retrieves and reserves its discovered nodes alongside to locally already reserved ones. Simultaneously, the query processor sends a positive acknowledgement back to answer originator server and a complete answer with all nodes (local and remote) reserved in local Node Pool back to the query originator node. The pool, then, waits for a positive or negative acknowledgement by query originator and removes these reserved nodes, if positive, or frees them, otherwise.

Response originator may receive a negative acknowledgement if there is no need for extra capacity any more or if delivered response is invalid (e.g. one of retrieved nodes has failed). A response to a requesting server may be unnecessary if the corresponding query is not in its cache; this may happen in three cases:

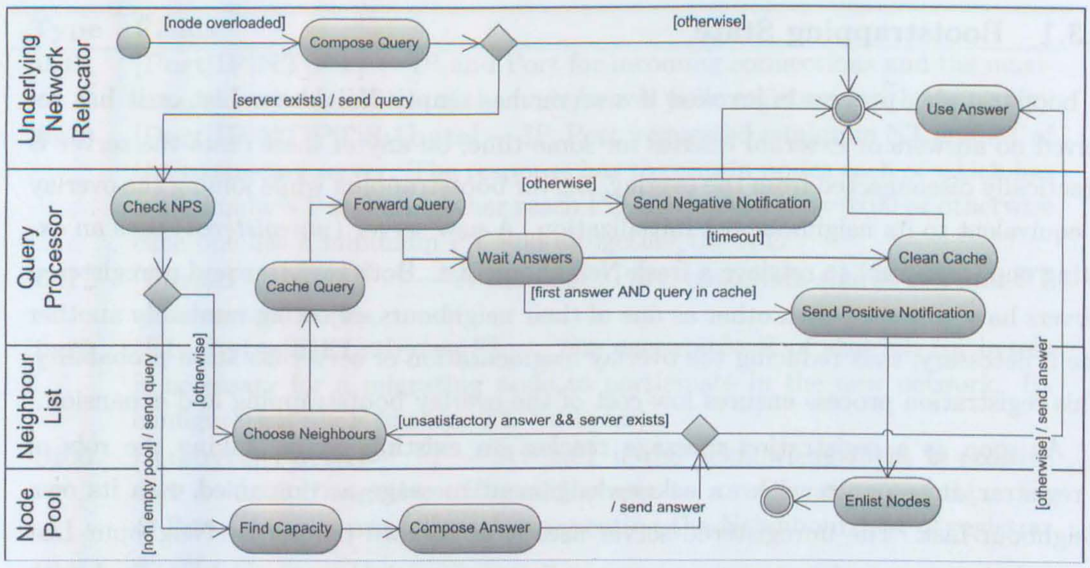


Figure 4.3: *Process flow for queries originated from an underlying network*

- requesting server fails and/or its cache is completely cleared for some reason (e.g. server failed).
- a valid response about the same query is already delivered and cache is cleared from that specific query ID.
- if new resources, sufficient to serve the request, appear in Node Pool of requesting server while waiting for responses e.g. if underlying nodes become underloaded. These are wrapped and delivered via an appropriate *answer* message to query originator node.

In case of invalid response, requesting server keeps waiting for another answer till query lifetime expires. If query fails and its originator node is still in need of capacity, it prepares a new query to local server seeking nodes to satisfy the current workload; that may include both the unsatisfied former request and any extra necessary amount or just a portion of them. Figure 4.3 presents the flow control when Hoverlay reacts to an underlying query.

4.3.3 Incoming Query State

An Hoverlay server has a different behaviour upon receiving a query originated from another from the overlay. Its primary target is to verify that it has not already processed a query with same ID by checking its cache; if a message with same ID exists in cache then the query is dropped and no action is taken or, otherwise, Query Processor caches it and transfers the control to local Node Pool.

Once Node Pool processes are triggered by an incoming external query, it tries to match all those requirements over its resource availability. That pool needs to check two things before positively answering to the query:

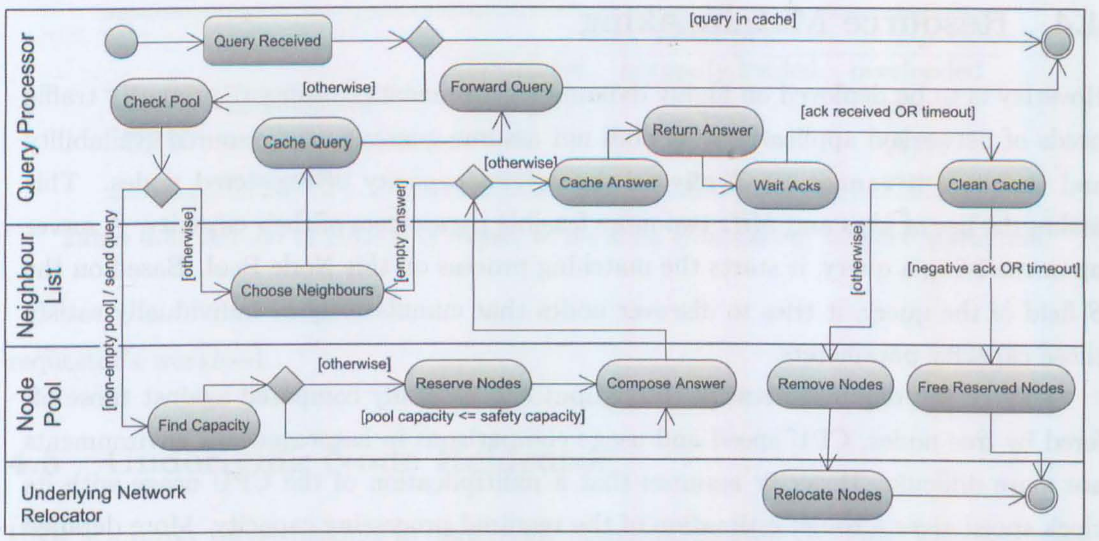


Figure 4.4: *Process flow for queries originated from other servers of the overlay*

- if it hosts more capacity than the *safety* one, and
- if that capacity is sufficient to completely serve the query requirements.

Both need a positive answer so that the pool prepares an response back to query originator server. After this resource extraction from pool its capacity availability might get lower than its safety level. This is to improve search efficiency and success giving priority to servers that have immediate needs for support; given that local pool host more capacity than its safety level, it is assumed that it has no immediate needs generated by its underlying nodes. The matched resources, if any, are reserved and returned to Query Processor who prepares the appropriate answer to deliver to query originator waiting for its acknowledgement.

That answer stays in cache for some time as the providing server needs a way to determine whether its pool should remove or free the reserved capacity. If the other end (requesting server) remains silent for that period of acknowledges back positively, these resources are removed from local pool. Though this may affect resource availability and contribute to reliability, it helps system to prevent multiple adverts and potentially parallel access attempts onto resources. Moreover, it is assumed that these resources are already unreliable and are able to detect connection loss and re-initiate registration process if necessary. In case of a negative acknowledgement from requesting server, the provider frees those resources for future use by internal/external queries. A response is cleared from the cache as soon as an acknowledgement is received or its lifetime expires.

In case Node Pool has insufficient capacity to fully serve query requirements, the Query Processor reduces its TTL counter by one and if that is bigger than zero it is forwarded to neighbouring servers from local Neighbour List. Using the same notation, Figure 4.4 illustrates the process control flow in case a query has been received from the overlay:

4.4 Resource Matchmaking

Hoverlay is to be deployed on highly dynamic environments to support primarily traffic needs of networked applications. It does not assume guaranties of resource availability and therefore it cannot 100% rely on the service capacity of registered nodes. This makes the use of *kbps* and *MHz* two more feasible parameters of their capacity. A server upon receiving a query, it starts the matching process on this Node Pool. Based on the S field of the query, it tries to discover nodes that cumulatively or individually satisfy those capacity parameters.

Though the required network throughput can be easily compared against those offered by free nodes, CPU speed and usage comparisons in heterogeneous environments are more difficult. Hoverlay assumes that a multiplication of the CPU usage with its clock speed gives a rough estimation of the required processing capacity. More detailed comparisons need information of both hardware (e.g. cache, CPU architecture, MIPS, memory speed, I/O latency) and software (e.g. language or executable, compiler version, operating system) environments of nodes. Such matchmaking is well-documented and used by Condors. However, these two systems differ in purpose: Condor focus on job completion whereas the proposed one here on traffic and enables nodes to request for extra capacity if necessary. CompuP2P [58], as a computational resource sharing paradigm deployed on dynamic P2P Networks, also uses cycles/second to represent processor capacity.

The following scenario demonstrates a simple matchmaking process on a server with two free nodes in its Pool. Though this is a rather simplistic example, section 4.10 provides other alternatives and criticism on how this resource matching can work in real-world environments. There are three nodes involved in this interaction: the requestor, and the two available ones. Their configuration appears below:

1. Node A (requestor): Its capacity is $NT = 100kbps$ and $PT = \{50\%, 2000MHz\}$ and its overload threshold is $NT = 100kbps$ and $PT = \{40\%, 2000MHz\}$. Assuming that its load is: $NT = 100kbps$ and $PT = \{60\%, 2000MHz\}$ then it seeks for resources satisfying the: $NT = 100kbps$ and $PT = \{20\%, 2000MHz\}$ with $S = 0x00$. That is, the query that reaches the server host of nodes B & C looks for a set of nodes each of which has a minimum network throughput 100kbps and collectively {40%,2000MHz} processor capacity.

2. Node B (free): Its overload threshold published in its server Pool is $NT = 20kbps$ and $PT = \{20\%, 2000MHz\}$. This node cannot match the requirements as its network throughput is well below the requested one.

3. Node C (free): Its overload threshold published in its server Pool is $NT = 200kbps$ and $PT = \{70\%, 1000MHz\}$. Its network throughput is enough to cover A's requirements. The product of its CPU usage with clock speed is bigger than that of query's; that is, $70\% * 1000MHz > 20\% * 2000MHz$.

Therefore, Node C migrates to requesting node underlying network to take on part of

		Network		
		uderloaded	normally loaded	overloaded
CPU	underloaded	*register*	—	*query*
	normally loaded	—	—	*query*
	overloaded	*query*	*query*	*query*

Table 4.5: *Actions of UNR with respect to the state of monitored network application*

requestor's workload.

4.5 Underlying Node Relocator

UNR is a proxy server residing in underlying nodes monitoring CPU and network usage of an application. The application is manually registered with UNR which measures CPU cycles spent and data volume transferred over the network connection by the application client on a per time-unit basis.

This UNR server may register and create a configuration profile for more than one application. Apart from the application installation directory path and filename, UNR profiles two tuples (min, max) representing the minimum and maximum allowed CPU and network throughput (CPU cycles in MHz and kbits per time-unit respectively) per application. These thresholds are used as metrics to trigger an action (register, query) to Hoverlay. From Hoverlay perspective, a UNR is a client which forwards appropriate messages to servers while monitoring an application.

A registration message is sent to the local server if its connection with that UNR is lost or both network and processor throughput used by the application are below their corresponding minimum thresholds (underloaded state). If any of these metrics is above its maximum threshold then the underlying node is overloaded and a message requesting for help is sent to that local Hoverlay server. Table 4.5 presents the actions a UNR can take based on the state of these two metrics. A node generates queries if either its CPU or network interface are overloaded and registration messages if both of them are underloaded.

An application is normally loaded if its consumption in CPU and network throughput does not exceed their maximum thresholds and are not both below their minimum ones. While the *max* threshold protects the application host from devoting too many resources into the network it participates, the *min* one improves its utility. The levels of processor and network bandwidth usage are determined based on the application activity the last few (user input) time units it was active. That is, the UNR records an activity history to calculate these average throughputs.

The current network throughput of an application represents the total data volume that is received and sent within current time unit whereas processor usage is the average CPU cycles consumed by that application within that time unit. A query originated by a requesting node specifies the minimum requirements in network and processor throughput. Every Hoverlay server, receiver of that query, tries to select a minimum subset of nodes. Hoverlay gives the option to overloaded nodes of choosing one criterion (high priority) that needs to be satisfied by every selected node and the other one (low priority) collectively and cumulatively by all of them.

4.6 Server and Node Failures

As analysed in system principles and requirements sections (see 1.5 and 2.11), Hoverlay is to be deployed in large-scale unreliable heterogeneous environments without any guarantees on resource availability. Therefore, both servers and underlying nodes are assumed to be hosted in unreliable machines and have equally unstable connections.

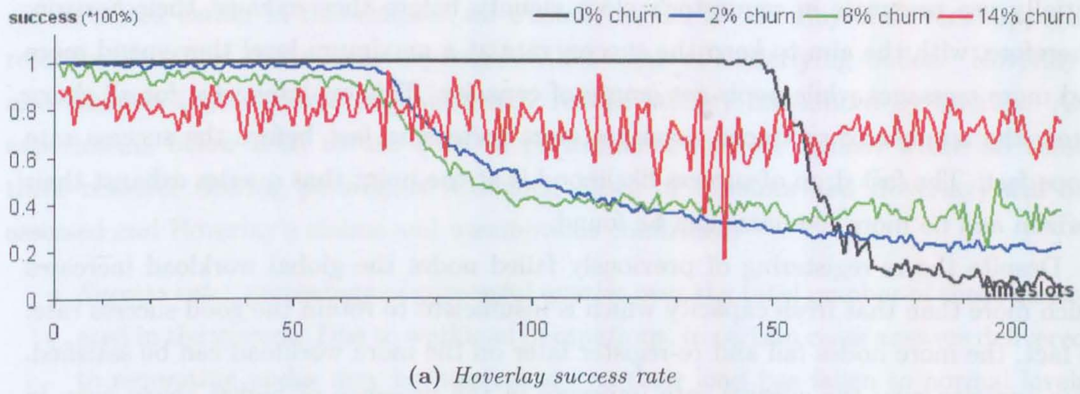
When a server fails all links from underlying nodes and its neighbours are broken. Therefore, no queries, answers or registration messages may be sent from/to that server. If no answer can be sent, neither can an acknowledgement since it is only sent after an answer. All reserved nodes in its pool become free and its cache clears as soon as the server returns to active state or rejoins the overlay. It is assumed that the acknowledgement is received immediately after a successfully sent answer.

While the local server is failed, its underlying nodes keep generating queries with a lifetime asking each time for any additional capacity they may need. Underlying nodes try to send periodically a registration message until a local server is able to respond positively. However, it is an application-specific parameter whether these nodes will seek for another server to send queries or registration messages to or simply wait for their last and failed one to become active again. If a generated but not sent query expires, its requested capacity is added to the next one. This saves the overlay from queries asking for huge amounts of capacity when the local server comes back online or another is found. If a server has permanently failed, its underlying network cannot reconnect to the network unless a server with the same DNS is activated or the nodes are reconfigured to use another server.

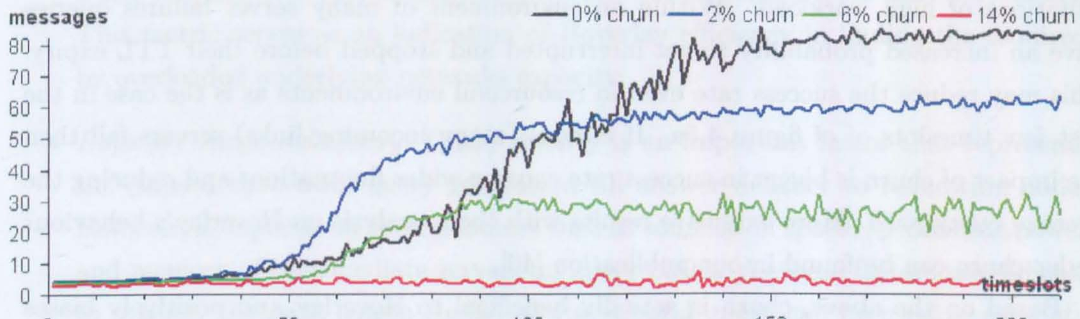
Similarly, a Node Pool may contain broken links to failed free nodes. Therefore, an answer may contain both alive and failed nodes. This is detected in the requesting node which drops that answer and original query as failed. If a requesting node is failed, the Query Processor of its local server cannot communicate to deliver e.g. a potential answer and thus, it frees any reserved node for that query.

The following experiments aim to evaluate the behaviour of Hoverlay in environments under different churn ratios. For that purpose we run simulations with the following network configuration:

- 1 000 servers and 5 0000 nodes,
- each node has on average 10 units of capacity ranging from 0 to 20 (500 000 units of global capacity),
- each server links with four other servers and the TTL of each query (starting from the underlying node) is 4.



(a) *Hoverlay success rate*



(b) *Messages per query spent by Hoverlay*

Figure 4.5: *Hoverlay behaviour under churn*

- the system-wide workload increases linearly (ranging from 0 to 500000) throughout the 210 timeslots of the experiments,
- the churn rates simulated where: 0%, 2%, 6% and 14% for both servers and underlying nodes.

For all those experiments, the system uses Flooding to discover resources. It is a widely used search mechanism which deploys no neighbour selection heuristics. Thus, the conclusions from the results cannot be biased by the search mechanism.

A fixed workload amount per timeslot ($A = 2381$) tops-up the system-wide one. However, 20% of that amount is removed from the system on every timeslot and, hence, 120% is added so that A remains fixed using randomly selected underlying nodes. Due to this system workload linearity, once a node has migrated into an underlying network it stays busy until it either fails or its workload falls below its underload threshold. Failed nodes rejoin the system as free resources after a random waiting time period. The success rate starts dropping fast as soon as the system-wide capacity gets exhausted. This increases the number of queries and as success likelihood drops these queries flood the network upto their horizon producing more and more messages. In principle, the bigger the churn rate the more fresh capacity appears in the system in every timeslot.

Figures 4.5a and 4.5b illustrate the success rate achieved by Hoverlay and messages spent under the churn ratios simulated. As the global workload increases, queries initially use resources in requestor's close vicinity before they exhaust their horizon. Therefore, with the aim to keep the success rate at a maximum level they spend more and more messages while pools get empty of capacity. This explains why, for all churn rates, the number of produced messages start increasing fast before the success rate drops fast. The fast drop of success likelihood is at the point that queries exhaust their horizon and no more resources can be found.

Despite the re-registering of previously failed nodes the global workload increases much more than that fresh capacity which is insufficient to retain the good success rate. In fact, the more nodes fail and re-register later on the more workload can be satisfied. This explains why the success rate improves in the presence of higher churn even in situations of high workload. Within an environment of many server failures queries have an increased probability to get interrupted and stopped before their TTL expiry. This may reduce the success rate even in resourceful environments as is the case in the first few timeslots of of figure 4.5a. If famous (many incoming links) servers fail then the impact of churn is bigger in success rate causing wider fluctuations and reducing the average query path. More extensive results with their analysis on Hoverlay's behaviour under churn can be found in our publication [40].

Based on the above, churn is actually beneficial to Hoverlay and positively biases its performance evaluation. Arguably then, it makes sense to omit it as a parameter in further experimentation.

4.7 Experiments and Evaluation

With a view on Hoverlay principal concepts and functionalities, the following set of experiments serves as a *proof-of-concept* and basis for a detailed evaluation. An Unstructured P2P Network of servers (pools of resources) comprises its backbone designed to support on-demand resource migration between networks. Both this architectural element (P2P Overlay of pools) and sharing technique (resource migration) comprise two main sources of differences compared to other architectures. Existing competitive resource sharing systems, appropriate to work as a benchmark appear below:

- *Condor*: a local pool (manager) to facilitate resource sharing within an individual network (centralised architecture). Such systems try to improve resource utilization within a single network by using a central resource aggregator (pool). Experimentation with *Condor* systems may provide useful material for evaluating possible costs (e.g. traffic, latency) introduced by Hoverlay as it proposes an arbitrary connection of similar systems.

- *Flock of Condors*; this category represents Condor-like systems with interconnected (via an Unstructured P2P Overlay) managers. They basically assume static links between pools and no mobility of resources. As documented in Chapter 2, current proposals for Unstructured P2P Condor Flocking, though weak, come closer to Hoverlay than any other. Their common features and behaviour will act as the principal benchmarking for all evaluation factors.

As stated earlier in this chapter, an Unstructured P2P Overlay of servers supports resource volatility minimising any registration costs of underlying nodes. Hoverlay-also assumes that resource migration may reduce latency introduced by queries. All experiments below need to use a set of performance metrics against which all these three resource sharing paradigms (Condors, Flock of Condors and Hoverlay) will be assessed and Hoverlay's claims and assumptions confirmed:

- *Success rate*: percentage of successful queries over the total number of those generated in the system. Due to workload fluctuations, in certain cases answers delivered to requesting nodes may be unnecessary as their load has fallen to normal levels while waiting for a response; such cases are not precluded from that percentage. This metric serves as an indication of Hoverlay efficiency in finding the required by overloaded underlying networks capacity.

- *Hops per Answered Query*: Query latency is an important factor that represents the elapsed time from query generation till answer delivery to requesting node. However, it depends on several factors such as connection speeds, processing power and memory of intermediate servers on query paths. As these factors are difficult to predict, the path length (hops) of a successful query can be used to estimate its latency without loss of generality assuming that all hops are temporally equal. This metric corresponds to the average path length of successful queries from their requesting nodes until first provider server. Late responses that a server may receive for an already satisfied query do not contribute to it.

- *Satisfied User Queries*: number of additional user queries (requested capacity) that overloaded networks have managed to satisfy with extra capacity discovered via a given overlay. If an underlying node gets a response with its requested capacity, that system has made it possible for these user queries to be processed successfully.

- *Messages*: number of messages (traffic) produced in the system by any operation (registrations, queries, answers and acknowledgements).

4.7.1 Simulation Practices

For these evaluation purposes, a C++ object-oriented simulator (called *Omeosis*) was developed. It can simulate time as a sequence of timeslots during which any message (query, answer, registration, acknowledgement) may travel for a single hop only and ensures their concurrent processing and propagation. It assumes that no connection introduces any extra delay; thus, any message produced during a timeslot reaches its next destination on the following timeslot. Timeslots are equivalent to iterations of the main loop. Therefore, every iteration executes three phases:

1. *Set global workload*: add or remove workload on a random subset of underlying nodes. Each node of this subset takes on a chunk of that workload (w) defined as a random integer within: $w = \{x \epsilon \aleph : 1 \leq x \leq max\,(c)\}$ where $max\,(c)$ corresponds to the maximum capacity a node can have. This chunk distribution finishes as soon

as the whole additional workload of this iteration is consumed; this determines the size of that subset. Thus, there is a non negligible probability that *a)* a node takes on two or more, *b)* multiple underlying nodes of a single server take on at least one or even *c)* no underlying node of a server take on any chunks.

2. *Generate queries*: once a message reaches a node or server another appropriate one (if necessary) is queued in its output buffer for delivery on the following timeslot. Apart from output buffer, messages are also stored in caches with the appropriate expiry time. If an underlying node is still overloaded, as soon as the waiting time of its cached query expires a new one with same requirements is generated and again cached for an exponentially increasing period (i.e. $TTL + 2^{repetitions+1}$). Therefore, message queues in output buffers follow the order of incoming ones. Exponential increase of waiting time before retry is a usual practice in request submission to networks and help them avoid bursts of requests especially during high load situations.

3. *Send produced messages*: while both phases above may be in parallel or sequentially in any order executed, this one has to follow both. Otherwise, an incoming message would probably trigger another message generation and delivery within the same timeslot: violation of the step-by-step and concurrent message propagation.

Every network component (server or underlying node) incorporates a set of modules which facilitate communication (input and output buffers), message caching, time event handling (reactions to time progress such as cache cleaning, message regeneration) and message processing. Servers use a pool which can reserve resources upon request, free resources if response is not accepted or otherwise release them. Apart from time events, underlying nodes react to workload changes, too. An increase of their load may trigger the suitable query generation module. Symmetrically, a drop to its load may force the rejecting of pending queries in its cache. All experiments used the same initial configuration of servers and nodes achieved by choosing the same parameters and feed to the random number generator used throughout. Both of server overlay size and number of underlying nodes are user inputs and remain fixed during the experiment. The simulator first creates the server overlay and carries on with the underlying nodes. As soon as it generates a server, it populates its Neighbour List with a random subset of the previously created ones; thus, their popularity follows the order they were created by resulting into a power-law network.

As with Hoverlay specification above, nodes appear in pools when their load is below a certain threshold; thus, reserved nodes may still have some load below their threshold. The global workload fluctuates based on a pattern predefined by user input; applying a positive or negative workload per timeslot on underlying nodes implements a rise or drop, respectively, of the global workload. A monitoring module records all actions triggered by any event (message deliveries, workload changes, lifetime expirations) which finally creates appropriate output files in both analytical and concise forms. These results appear below.

Testing played an important role in *Omeosis* development phase. A module of the simulator keeps track of every message delivered and records any change on any node or server status. These files were input to Matlab and AWK scripts to parse and ensure that certain tests based on a set of rules were successful after every experiment conducted including the ones presented below. These rules which greatly helped on debugging logical errors and ensure the simulator's expected functionality are the following:

1. Throughout experiments the number of servers and underlying nodes remains fixed. Nodes can either be reserved or free in local pools or busy in underlying networks. Summing up reserved, free and busy nodes should equal to the initial number of nodes set as an input parameter. Similar rule applies to their capacity, too.

2. Every node has one outgoing link to a server and no incoming ones. Servers have a fixed number of outgoing and any number of incoming ones. Summing up the incoming links of servers should equal to the outgoing links of both servers and nodes. In case of Condor-based systems, without rewiring in place, all links of servers should remain the same. However, Hoverlay introduces node migration and, hence, the simulator must ensure that once a node changes its outgoing link, the incoming ones of its provider server decrease and those of its requestor increase by one.

3. Every query involves a sequence of messages which must be kept in order. That is, query propagation on the overlay always follows a request from an underlying node to its server; resource discovery terminates a query path and produces maximum two consecutive answers which in turn trigger their acknowledgements.

4. No query may travel beyond its preset horizon and the number of messages produced by that query cannot exceed a theoretical maximum (propagation on an acyclic tree-like network).

5. The total number of messages at the end of every iteration must be equal to the sum of registration, query, answer, positive and negative acknowledgement messages.

6. Moreover, acknowledgement messages should also equal registration and answer messages as every such message is always followed by an acknowledgement.

Apart from automatic testing processes, log files record every message including their sources, destinations, delivery timeslot and their content. This analytical information was checked in two ways: *a)* random queries were selected and all their related messages (queries, answers, acknowledgements) as well as the reactions of all servers and nodes involved were manually traced, *b)* random underlying nodes and servers were also depicted and all messages they received/sent traced. This manual process was a testing procedure following all main versions of *Omeosis* and all the experiments conducted.

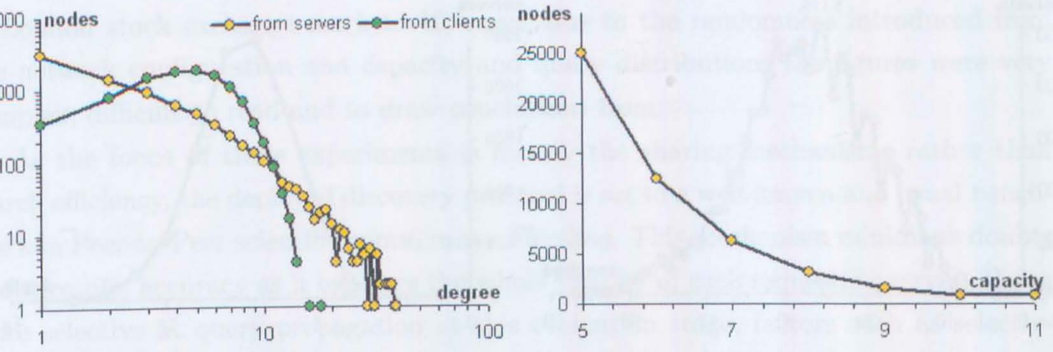
4.7.2 Experiments Configuration

The system evaluation was based on two main environments selected to test different aspects:

- *Uniform Query Distribution*: every node in the whole system has equal probability with any other to generate a query.
- *Hotspot Query Distribution*: only a small subset of nodes produce queries. All managers of these nodes are neighbouring servers and belong in the same area of the overlay. That is, all underlying nodes of servers in the vicinity of a given centre (centroid) have the same probability to produce a query; nodes of servers outside that area have zero probability to become requestors.

These environments provide enough material to evaluate Hoverlay on the metrics above. The experiments configuration is as follows:

- *Network Sizes*: 10 000 servers (i.e. independent networks) and 50 000 nodes uniformly distributed among the servers. During connections initial setup, a node links to any server with probability $\frac{1}{10\,000}$. Therefore, servers with no underlying nodes cannot generate queries or provide answers but may increase the hops per answered query and number of messages. All nodes are initially free and available in their local pools.
- *Capacity*: each node represents capacity (c) of size between 5 and 10 ($5 \leq c \leq 10$) units inclusive. The capacity density function shows the number of nodes representing a certain amount of capacity. As shown in figure 4.6b, it follows a geometric distribution: $d(c) = \left(\frac{1}{2}\right)^{c-4}$. Multiplying the number of nodes with the sum of $c * d(c)$ products gives a close estimation of system capacity: $C = 50\,000 \sum_{c=5}^{10} \frac{c}{2^{c-4}} \simeq 300\,000$ capacity units. Though the representation of resource capacity with a simple number is a bit abstract, without loss of generality, it is useful to conduct experiments, produce results and conclude focusing primarily of the contribution of this thesis with regards to the resource migration rather than the resource heterogeneity. In real world scenarios, different representations of capacity may be used (see section ?? for more details). With node capacity deviating from 5 to 10, the experiments practically assume 6 different node types; though these boundaries could be extended they would not make significant difference to the experiments.
- *Connectivity & Time-to-Live*: each server connects with a maximum 3 other random ones. Its Neighbour List initial configuration occurs during server's creation with links to other existing ones. That is, the probability a server attracting a new link from another one exponentially increases with latter's age thus resulting into a power-law incoming-degree distribution. This Neighbour List size is small enough to increase the average path length between any two servers making difficult the access of any resource from the vast majority of underlying nodes. The



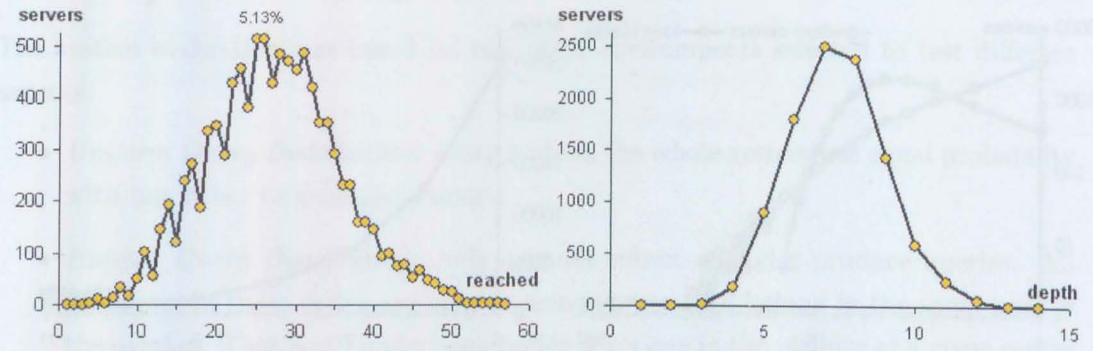
(a) *Distribution of server-to-server and nodes-to-server incoming degree* (b) *Capacity distribution over nodes*

Figure 4.6: *(a) Server popularity among other servers and nodes distribution and (b) system capacity distribution among nodes.*

TTL of every query is set fixed to 7; that is, each query may access a maximum of $\sum_{t=0}^{7} 3^t = 3280$ (= 32.8% of all) servers. Practically, this percentage is lower as the average number of servers another one may reach is 26.29; that is, only 0.26% of Hoverlay overlay size. This becomes clear in figure 4.7a which presents a density function of reachable servers (e.g. How many servers may reach 25 others in their vicinity? -answer: 513) which follows a normal distribution.

- *Workload*: given the global initially available capacity, system-wide workload should have both valleys and peaks fluctuating from 0% to even 150% of global capacity. This helps the system evaluation under several situations like workload increase/decrease, long-lasting strenuous high-load or relaxing low-load phases. These fluctuations appear in figures 4.8 and 4.14a with respect to those two evaluation scenarios mentioned above.

Figure 4.6a presents the incoming degree distribution of servers distinguishing server-to-server from nodes-to-server links. Both axes have logarithmic scales to improve readability. As explained above, server-to-server degrees follow a power-law distribution as only a small number of them are very popular; that is a result of the way new servers join their overlay. This coincides with real systems based on preferential attachment of new nodes onto older and more stable ones: e.g. Gnutella WebCaches [67]. It is a reasonable network topology for Hoverlay; it is expected to have power-law properties as strong providers will attract more links. Initial network configuration achieves a Poisson distribution of links from nodes to servers. Node migration (Hoverlay case only) may distort this distribution. While good providers (pools with plenty of resources and, thus, high node-originated incoming degree) attract more and more links they lose their resources faster. Therefore, though there must be a correlation between the two distributions presented in 4.6a, they do not necessarily coincide. Finally, figure 4.6b plots the distribution of global capacity onto nodes (number of nodes with the same capacity).



(a) *Distribution of max server horizon size* (b) *Distribution of max server horizon depth*

Figure 4.7: *Server horizon statistical properties: (a) max horizon sizes density (how many servers have e.g. 15 servers in their horizon?) and (b) max horizon depth (how many servers have horizon of depth e.g. 5 hops?)*

It follows a geometric density function complying with the idea that most of Hoverlay users offer low-capacity resources.

Despite the theoretical maximum number of reachable servers for each query (see above), Hoverlay monitoring at its initial phase shows that this number does not exceed 57 (figure 4.7a), way lower to the theoretical one. That is the maximum number of servers a query may visit via Outbound Neighbour Lists even with infinite TTL; thus, servers overlay appears to have a number of cyclic paths. Without revisiting servers, a server may access all its reachable servers with an average path length of 7.5 hops as shown in figure 4.7b. Therefore, $TTL = 7$ appears to be an appropriate query path length for this network configuration.

The small neighbour list size ensures that the overlay is a weakly connected graph and each query, even if flooding is deployed, does only explore a small portion of the network and therefore has access to a limited capacity size. This neighbourhood size becomes even more important in hotspot query distribution scenario as, preserving the same TTL, it helps the simulation of situations whereby resources may forever move out the vicinity of a hotspot area. All the experiments below were run with the same parameters apart from the input workload. The aim of the workload was to evaluate Hoverlay in as many as possible situations:

- workload peak to levels well above system-wide capacity
- workload peak to levels below system-wide capacity
- short workload valley between two peaks
- long workload valley to levels well below system-wide capacity
- long workload peak to levels well above system-wide capacity

To improve readability of the produced plots, these workload situations appear only once in the input data with one peak between every two drops or valleys and linear transition between them. These experiments were run with real data from FTSE Index of London stock exchange market. However, due to the randomness introduced from the network configuration and capacity and query distributions the figures were very complex, difficult to read and to draw conclusions from.

As the focus of these experiments is mostly the sharing mechanisms rather than search efficiency, the deployed discovery protocol is set to a well-known and usual benchmark in Peer-to-Peer scientific community: *Flooding*. This mechanism minimises doubts about results accuracy as it explores the whole vicinity of each requesting server. Being more selective at query propagation at this evaluation stage, factors such as selectivity heuristics could distort results (e.g. *k-walkers* have unstable success rate and thus results would be unclear and non-conclusive regarding benefits and costs of Hoverlay). Flooding on a static overlay ensures that queries from a server, either in a Condor-based or Hoverlay architecture, may explore the same servers; this eliminates one factor of results differentiation: deployed search technique. Further experimentation about search mechanisms follow in Chapter 5. For similar reasons, these experiments have rewiring deactivated; its evaluation appears alongside search mechanisms.

4.8 Evaluation on Uniform Query Distribution

Hoverlay evaluation with regards to uniform query distribution appears in the following subsections. That is, on every timeslot certain amount of workload (positive or negative) is distributed onto a uniformly selected random subset of nodes. Hoverlay is compared against Condors, both disconnected and Flock versions (referred to in text and graphs as *Condors* and *Flock of Condors* respectively). Condors represent a centralised approach while their Flock and Hoverlay two decentralised ones. Their explanation will follow an *observation-justification* scheme. Throughout these graphs, there are three main colours used: *black* for Condors, *blue* for Flock of Condors and *red* for Hoverlay; *grey* is mostly used to highlight differences between those two latter systems on each evaluation metric.

While global capacity remains fixed as no node joins or leaves the overlay throughout the experiments, global workload fluctuates as shown in the two-layered figure 4.8. Both

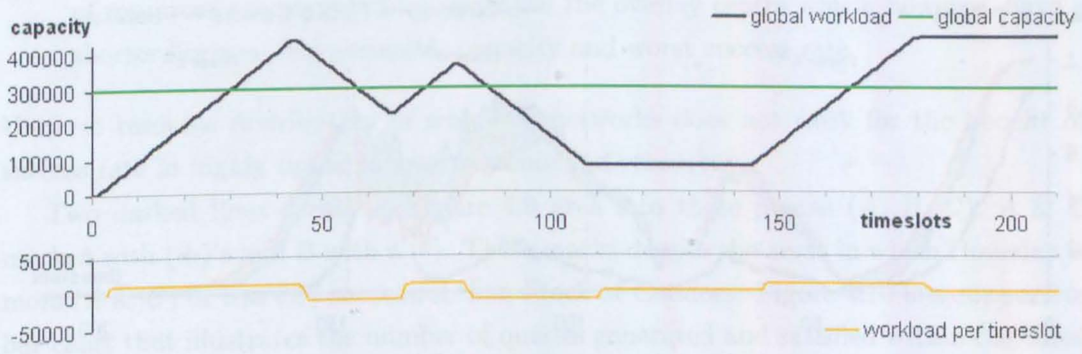


Figure 4.8: *Global Hoverlay workload and capacity: added or removed workload per timeslot (top layer) and cumulative workload and system capacity (bottom layer)*

layers share the same x-axis (timeslots) but their y-axes have same units (capacity) and different scale. Bottom layer describes the workload added or removed per timeslot whereas top one illustrates the system-wide capacity and cumulative load applied on to the system. Fixed positive or negative new load produces linear increases or decreases of global load at same intervals. After initialization phase, Hoverlay load fluctuates between $\frac{3}{2}$ and $\frac{1}{3}$ of its capacity.

4.8.1 Query Success Rate

Figure 4.9 presents Condors, Flock of Condors and Hoverlay success rates. Based on its plots, Hoverlay outperforms both disconnected Condors and their Flock with regards to the percentage of successful over the total number of generated queries. Throughout the experiment Hoverlay achieves better than Condors (up to 50%) success rate confirming that interconnecting individual networks contributes to the satisfaction of more user requests. For most of the experiment duration, when the system is normally and heavily loaded, Hoverlay manages to satisfy bigger portion of queries (on average 5% more) compared to Flock of Condors.

Any explanation of this improvement in success rate achieved with Hoverlay should derive from the fundamental differences of the two systems: resource migration. Indeed, the reasoning is two-faceted: *a)* resource reservations in Flock of Condors last longer and *b)* resource migrations are also translated to query migrations. In details:

- Service capacity is a highly dynamic resource; one of the system design requirements was that it should not rely on guarantees that migrated resources have the capacity their provider pools claim to have. A requesting node may reject discovered but unnecessary or unsuitable capacity. Resource migration eases the re-registration of this capacity with requestor's pool avoiding extra messages and latency to return it back to its provider pool. In case of Hoverlay, once capacity is discovered an answer travels from remote provider (Server A) to requestor server (Server B) which then, at the same time, acknowledges the provider and wraps that answer to forward it to the underlying node. If for any reason that capacity

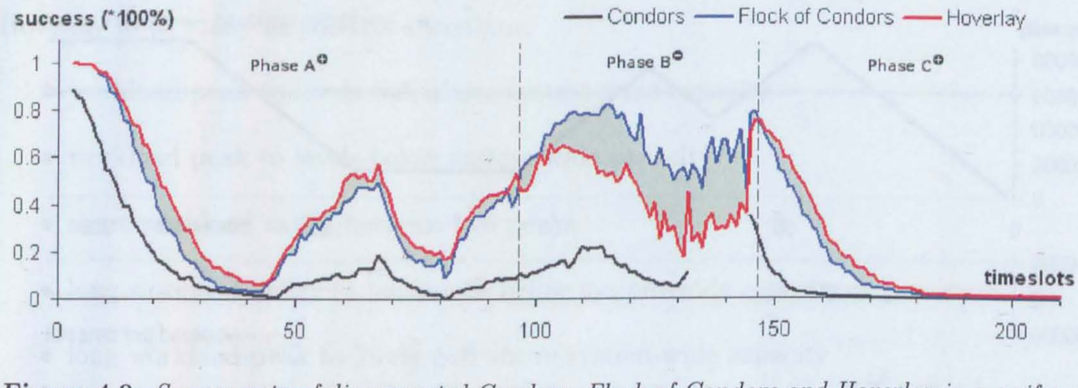


Figure 4.9: *Success rate of disconnected Condors, Flock of Condors and Hoverlay in a uniform query distribution environment*

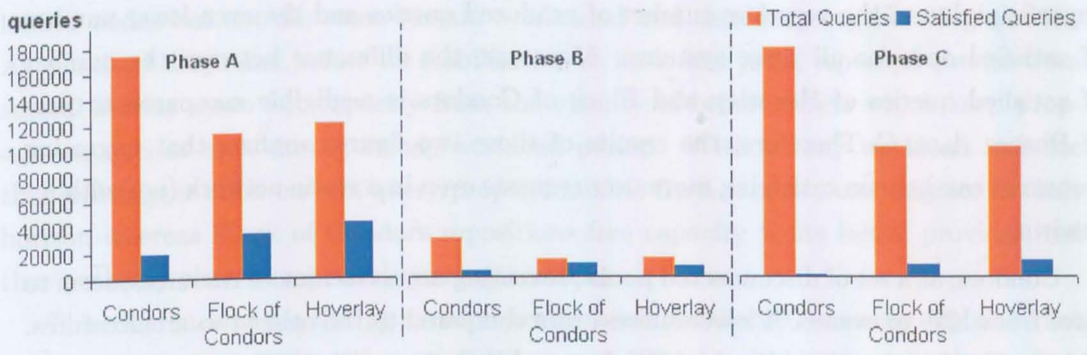


Figure 4.10: *Cumulative number of total and satisfied queries for Condors, Flock of Condors and Hoverlay*

is not used, it registers with Server B; providers get acknowledgements on the next timeslot (reservations last 2 timeslots). However in case of Flock of Condors, rejected resources need to return back to the provider with the acknowledgement to their response. Before Server B acknowledges Server A, it has to wait for the acknowledgement from the underlying node. Thus, discovered resources need to stay reserved for 4 timeslots before released. While Flock of Condors keeps resources reserved for 2 extra timeslots practically useless, Hoverlay provides them to requesting nodes serving extra load.

- The overlay topology of these experiments is a power-law one. With a uniform distribution of load on nodes, every node has the same probability to generate a query. The majority of servers are at the edges of the overlay (leaf servers) and, hence, most of the queries come from those edges. Due to this overlay topology, most of the query paths direct to high-incoming-degree servers. These queries in combination with resource migrations force capacity to move from the centre of the overlay to its edges. Further increase of global workload will most likely generate queries from those overlay edges. With an adequate TTL, queries may traverse the whole overlay. However, if resources do not migrate extra increase of their workload generate queries forwarded via always the same server. A good portion of resources are managed by servers in the overlay centre which, however, have a shorter horizon, less accessible capacity and worst success rate.

Uniform resource distribution in scale-free networks does not work for the benefit of success rate in highly dynamic environments and resources.

Two dashed lines divide the figure 4.9 area into three phases (A, B, C): A & C marked with (+)'s and B with a (-). These marks denote the areas in which Hoverlay is more (A & C) or less (B) successful than Flock of Condors. Figure 4.10 is a supportive bar chart that illustrates the number of queries generated and satisfied within the three phases of the experiment; it is a summation of total and of satisfied queries per phase. When the whole system handles low global workload (Phase B), Flock of Condors reach even 20% higher success rate than Hoverlay. As shown in figure 4.10, this deterioration is superficial due to the very low number of produced queries and the even lower number of satisfied ones for all three systems. Moreover, the difference between the number of satisfied queries of Hoverlay and Flock of Condors is negligible compared to that of Phases A or C. Therefore, the results of those two figures confirm that migrating resources can help on satisfying more user requests even in a static network (i.e. without rewiring).

Condors, as a set of disconnected pools preventing access to remote resources, seem to have from 10% to even 50% lower success rate compared to the other two architectures. For the first few timeslots, Flock of Condors and Hoverlay reach 100% success by seeking for both local and remote capacity. Some servers contain no local free capacity, even on the first few timeslots, in which case Condors cannot serve requests from their underlying nodes; hence, lower success rate than Flock of Condors or Hoverlay.

Global workload in the beginning of Phase A steadily increases and therefore no new fresh nodes appear in server pools. Gradually all resources within requestors vicinity exhaust and *a)* more queries fail, *b)* more new queries browse the overlay and *c)* more servers regenerate the unsatisfied ones. Capacity exhaustion increases the number of queries and their repetitions deteriorating the success rate of all three systems.

Symmetrically, applying negative workload on nodes makes the global cumulative one drop; some of these nodes (resources) become underloaded and available for on-demand migration or local re-commission via their pools. In some other cases despite their workload drop, they may remain overloaded but reactively adjust downwards their requested capacity. That is, responses for past queries may not be necessary and thus the discovered capacity may either stay in its new local pool or be partially used by the requesting node. The unused portion of that capacity may serve extra load of the underlying network without extra requests.

All systems during Phase B generate very few queries and satisfy even fewer, as shown in figure 4.10. Flock of Condors satisfy negligibly more queries than Hoverlay but this difference is substantial compared to the number of queries they generate. This makes the success rate of Flock of Condors by 20% better than that of Hoverlay. However, it is a misleading conclusion if not accompanied by this observation.

As shown in 4.8, from timeslot 80 till around 100, global workload drops and stabilises at almost $\frac{1}{4}$ of global capacity. On timeslot 96 (left border of phase B), system's workload approaches the $\frac{2}{3}$ of its capacity. That is the point after which Flock of Condors become more successful. As underlying nodes lose workload, some either become underloaded or normally loaded or even remain overloaded at same or lower workload levels. There are no new queries but for repeated ones. As fresh capacity becomes available, repeated queries get satisfied improving success rate for both systems.

While in case of Condor flocking unnecessary capacity returns back to its provider, Hoverlay moves it to requesting server. Within phase B, this migration is useless as that capacity may only be used on workload increase or even potentially harmful for system success rate as it may be moved away from places accessible by requesting servers. This is the case for those few repeated queries. Global workload affected most of the loaded nodes but not all; some remain overloaded and thus keep regenerating queries. As workload drops and before its stabilisation at its lowest level, repeated queries from several servers move out capacity from the vicinity of other servers which keep propagating queries even during steady-workload period. Without any workload increase that will trigger other servers query propagation, no fresh capacity can migrate in their horizon whereas Flock of Condors repositions free capacity to its initial provider and thus probably close to repeated query generators.

This clearance of requesting servers' horizon from available resources explains that success rate swap between Flock of Condors and Hoverlay architectures with former's being higher than latter's one. However, that cannot justify why their difference in phase B is well over than in the other two. That difference can be justified considering the minimal number of queries on which this percentage is based. Flock of Condors keep, in low global workload cases, the resources at their initial distribution among servers bringing a bigger impact on success rate. Overloaded nodes and their queries start increasing with the global workload. At the end of phase B a good portion of those queries are successful due to the available capacity generated during the last workload drop. Therefore, the impact of that factor gets lower and the success rate of both systems increases.

4.8.2 Average Path Length of Successful Queries

Figure 4.11 presents the average number of hops successful queries had to travel before they discover their first answer. The graphs confirm that Hoverlay manages to achieve better success rate with shorter query paths especially on workload fluctuations. It helps queries get responses from 0.5 to even 2 hops sooner than Flock of Condors do. This 2-hop improvement takes place around timeslots that global workload started decreasing: fresh capacity appears close to the regenerators of repeated queries.

The average path lengths of Flock of Condors and Hoverlay do not follow the pattern of success rate and stay well below query TTL. The power-law topology of the overlay, in the absence of rewiring, stays the same throughout the experiment. Most query

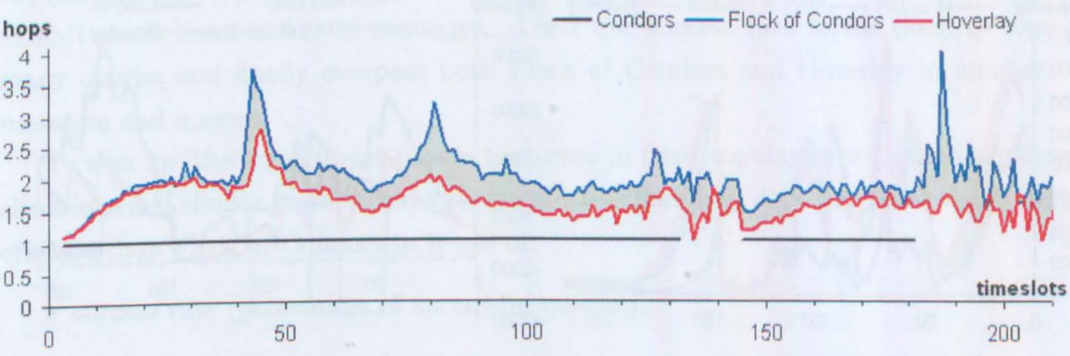


Figure 4.11: *Averge number of query hops before they discover their first answer deployed on disconnected Condors, Flock of Condors and Hoverlay in a uniform query distribution environment*

paths start from the edges of the network and finish at its centre. Thus, while workload increases resources in the centre become scarce; the biggest portion of the capacity is close to leaf servers. Long successful paths are less than short ones and, thus, their average remains below TTL mean value 4.5 (1 hop from node to server plus $\frac{TTL}{2}$).

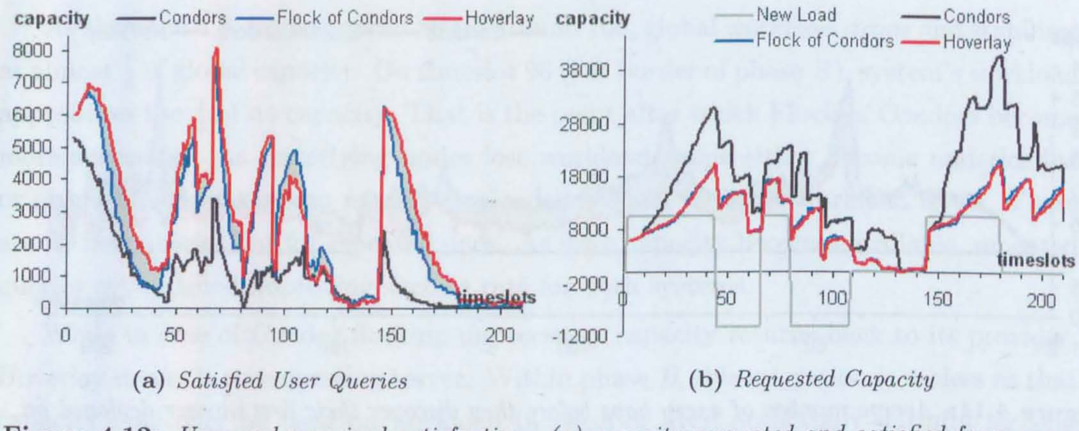
As soon as workload starts droping the effect of resource migrations becomes clearer. Fresh capacity appears in the pools of domains it migrated to, closer to requestors. On the contrary, Flock of Condors place that capacity back to its originators and thus repeated queries need to travel further to re-discover it. This explains why Flock of Condors exhibit path length bursts on the first few timeslots the workload starts decreasing. While workload keeps dropping, free capacity increases close to requestors vicinity, servers generate no new queries and more and more repeated ones are canceled. This keeps the average hop count in low levels.

In case of disconnected set of Condors, queries can only travel from underlying nodes to their local servers; hence, one hop. Until timeslot 44 successful queries of both Flock of Condors and Hoverlay exhibit almost the same hop count. Workload increases linearly for the first 44 timeslots and all migrated nodes join the requesting underlying networks. Given that both systems use flooding tested on same topologies, all servers explore their whole horizon and thus the average hop count has minimal deviation.

4.8.3 Traded Capacity and Cost in Messages

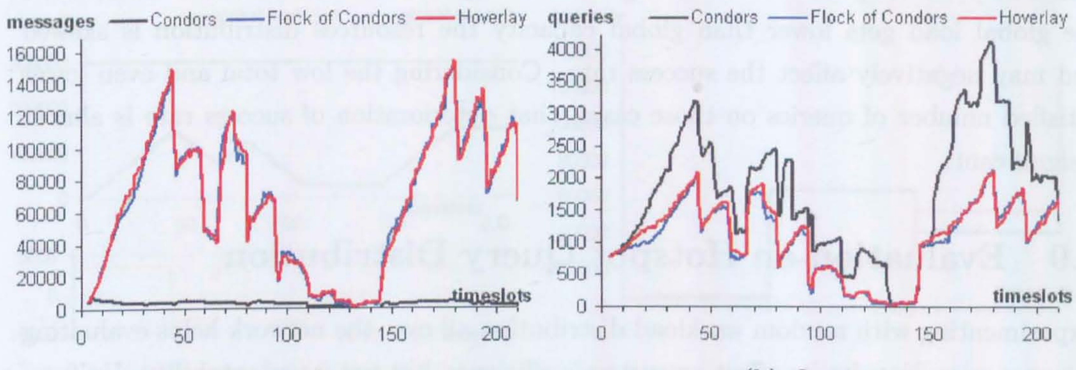
In general, Hoverlay satisfies more load than Flock of Condors (figure 4.12a) though both systems request almost the same amount (figure 4.12b). This improvement comes at almost same cost in messages (figure 4.13a). Both systems outperform Condors in terms of satisfied capacity due to the inability of the latter to access remote capacity; however, Condors have a minimal overall cost in messages as their queries can only travel one hop.

Following similar patterns as success rate, satisfied capacity of Hoverlay is more than that of Flock of Condors in phases *A* & *C*. Flock of Condors superiority in Phase *B*



(a) *Satisfied User Queries* (b) *Requested Capacity*

Figure 4.12: *User-end perceived satisfaction: (a) capacity requested and satisfied from server overlays and (b) requested capacity per timeslot.*



(a) *Total number of messages* (b) *Queries*

Figure 4.13: *Cost in messages: (a) total number of messages produces and (b) total number of generated queries.*

becomes insignificant as the requested capacity during that period is much lower than that of remaining two phases. The same reasoning as with success rate lines above makes their shape comprehensible. However, their exact shapes depend on requested capacity, too. This explains why the lines of satisfied capacity have similar drops and increases as the requested capacity but the distance between blue and red lines resembles the one of figure 4.9.

Figures 4.12b, 4.13a and 4.13b have a common characteristic: almost vertical deep decreases and high increases in plotted lines. These radical changes happen on timeslots that new load swaps from positive to negative values and vice-versa. Though within 45-65 timeslots interval the global workload drops linearly the number of messages, number of queries and requested capacity are non-zero and follow similar patterns. Similarly, the pattern of 45-65 interval tops the new positive workload of 66-79 timeslots. Differences on those patterns appear as portions of those repeated queries get satisfied or stopped as unnecessary (especially after workload drops). Finally, the lines for Flock of Condors and Hoverlay architectures, of all these three figures, practically overlap with slight differences mainly in 4.12b and 4.13b (Hoverlay produces less queries which cumulatively request less capacity). Condors, due to lack of query forwarding in the servers overlay, exhibit much less number of messages. Their low success rate forces them to repeat many queries and finally overpass both Flock of Condors and Hoverlay in number of messages and queries.

To sum up, this experiment proves that even in fixed topologies with high workload situations and similar search technique (exhaustive flooding) deployed, Hoverlay is more efficient than Flock of Condors in terms of:

- success rate (percentage of successful queries),
- satisfied capacity (portion of requested capacity that was satisfied) and
- average path length of successful queries before they hit their first answer.

This is basically the positive effect of resource migration to requesting networks and comes at practically no cost in messages. However, under certain circumstances, when the global load gets lower than global capacity the resources distribution is skewed and may negatively affect the success rate. Considering the low total and even lower satisfied number of queries on those cases, that deterioration of success rate is almost insignificant.

4.9 Evaluation on Hotspot Query Distribution

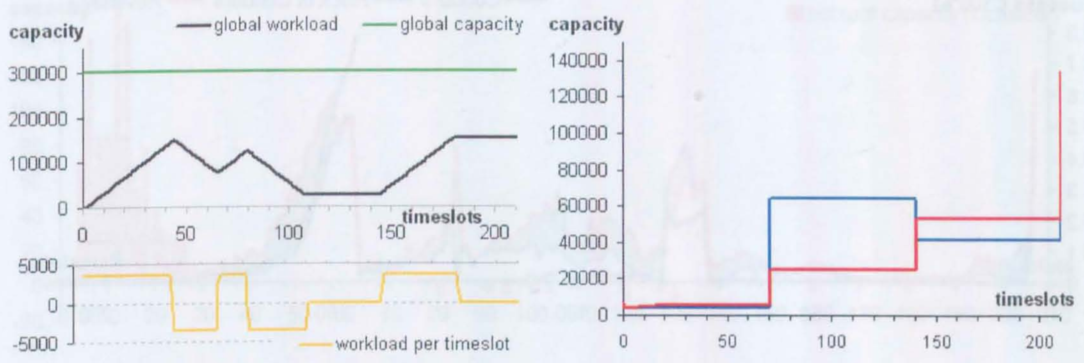
Experimenting with random workload distribution all over the network helps evaluating resource migration for its effect on system's efficiency but not its adaptability. Uniform query distribution over underlying nodes retains the incoming degree distribution of servers with respect to their underlying nodes. However, if queries come from a specific small subset of neighbouring servers (hotspot) for long periods two scenarios for further experimentation emerge regarding the behaviour of all three systems:

- workload fluctuations of underlying nodes of those servers: which system quickly addresses the requested capacity?
- hotspot shifting: which system is more successful if hotspots shift to another area of the server overlay?

On top of the common experiment configuration parameters as set in subsection 4.7.2, few more are necessary to complete hotspot initialization and shifting:

- *Number of spots*: network initialisation is exactly the same as in previous experiments but a subset of servers are picked to belong in three, not necessarily neighbouring, hotspot areas. In fact, each of these three areas consists of a centroid (server) and all other servers within its vicinity directly or indirectly accessible via incoming or outgoing links.
- *Spot radius*: the distance (number of hops) of any server within a hotspot area from its centroid cannot exceed five hops.
- *Spot lifetime*: the lifetime (timeslots) of each hotspot area. As soon as its activity terminates another one is selected. At any given moment there are three such areas (triplets) which all start and end their activity simultaneously before they pass on their role to new randomly selected ones. This lifetime is 70 timeslots; thus, within 210 timeslots three shifts of hotspot triplets take place.

Hotspot area size varies according to its connectivity. Within same radius, there may be more servers if its centroid comes closer to popular nodes. Given the power-law server-to-server incoming degree distribution, an area built around a server without incoming links (leaf-server) is smaller than another circling a hub. At no case three five-hop radius areas cover the whole network and with a TTL=7 the capacity available to requesting servers is limited. Furthermore, as shown in figures 4.7a and 4.7b, higher TTL would overload the network. While in previous experiments cumulative applied



(a) *Cumulative and fresh workload applied on hotspot triplets* (b) *Total capacity of hotspot triplets*

Figure 4.14: *Hotspot capacity and workload: (a) cumulative and new per timeslot workload and (b) capacity within hotspot triplets.*

workload reaches 150% of system-wide capacity, in the following ones accessible capacity is less and therefore the workload must have lower peaks (up to 50% of capacity) as shown in figure 4.14a.

The three relocations of hotspot triplets become clear in figure 4.14b for both Condor-based architectures and Hoverlay. It presents the cumulative capacity of hotspots on those three intervals. While that capacity is fixed per interval for Condor-based systems, Hoverlay increases this capacity at the first few timeslots after hotspot relocation via resource migration. The average maximum path length for servers is 7.5 hops; those in the interior of big hotspots cannot explore far beyond their area borders. Most of the accessible servers from a centroid belong to the same area; queries originated from that server can only travel 2 hops beyond the borders of its area.

Most of the observations from figures below share the same analysis as in previous experiments. Thus, detailed explanations will only follow observations that differentiate those figures from the ones of previous section. Two main factors contribute to these differentiations: *a)* for long intervals same servers produce large number of queries and *b)* hotspot areas change centroid every 70 timeslots.

4.9.1 Query Success Rate

Starting from query success rate of figure 4.15, throughout the experiment all systems experience, in general, low success rate but Hoverlay is the most successful (up to 20%) compared to Condor-based systems. Condors success is squeezed between x-axis and Flock of Condors line. Hoverlay performance can be explained the same way as in uniform query distribution environment in section 4.8. Both query originators migration and shorter resource reservations duration from the Hoverlay play important role.

In fact, hotspot areas are very small compared to the network size and workload is uniformly distributed among their nodes. Hotspot locations can be anywhere in the network and thus resources do not necessarily move to leaf servers; widening of requestors

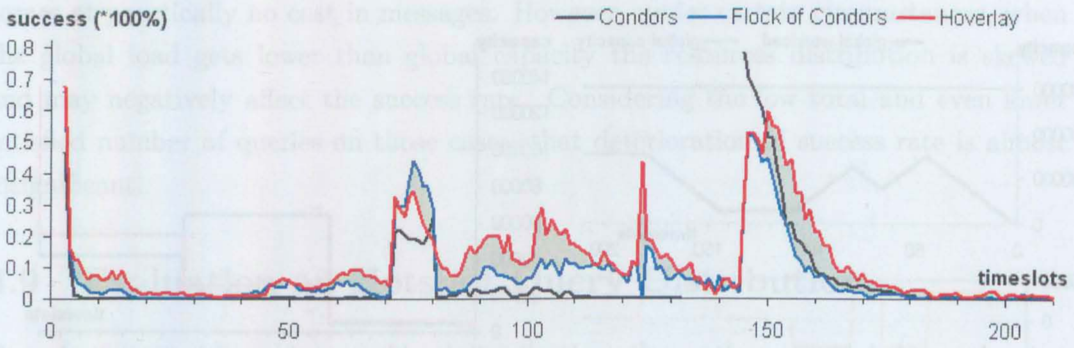


Figure 4.15: *Success rate of disconnected Condors, Flock of Condors and Hoverlay in a hotspot query distribution environment*

horizon, as claimed in 4.8, cannot stand as a reasoning in this environment. Unlike Flock of Condors, resources joining requesting underlying networks become equal members to the existing ones. Condor-based systems are task-oriented; each node is assigned a very specific job and cannot take on another as an extension of the existing remote one; the discovery mechanism needs to be triggered again. Additional load targeted to underlying nodes of a specific server cannot be assigned to remote ones currently under commission to that server.

Therefore, Hoverlay with migrations not only does it serve portion of hotspot queries but increases the capacity on which future workload may be distributed. Newly migrated nodes join the networks without getting overloaded; they can take on some more workload without overpassing their threshold. This reduces both number of queries and requested capacity and improves the success rate.

These plots show abrupt success rate increases for all three systems just after the timeslot of hotspot relocations. Moving hotspots is also translated into query originators migration; just servers within hotspot areas may generate new queries. These queries, then, browse areas full of free capacity allowing better success. Initially all systems perform well as global workload is low and available capacity plenty. However, aligning figure 4.14a and 4.14b makes clear that global workload quickly gets much bigger than available capacity and thus success rates of all systems drop. Towards timeslot 70 (hotspots relocation), success rate increases as global workload drops.

On timeslot 70, a hotspot relocation takes place to areas that cumulatively contain more available capacity. This makes success rates of all systems abruptly increase. Though there is now enough accessible and available capacity only about 50% of queries are successful. Hotspot relocation is not followed by workload; that is, workload already applied onto certain servers remains on them until satisfied even if no more is added. Therefore, past highly loaded hotspots (as is the case during 0-70 timeslots) regenerate queries alongside new ones. Around timeslot 80, though global workload starts decreasing, success rate of all systems drops quickly, too, justified by the following reasoning:

1. calculation of success rate includes every query from both past and new hotspots,

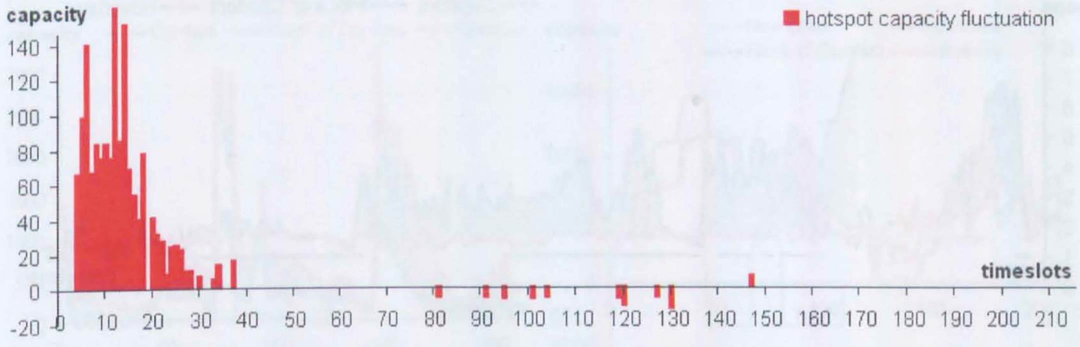


Figure 4.16: *Hotspots capacity fluctuation of Hoverlay architecture*

2. past hotspots do not generate new but only repeat old unsatisfied queries,
3. new hotspots take on new and for a short period (see figure 4.14a) workload only; they have much more capacity and less workload than past ones, thus, most of their queries get satisfied,
4. as soon as their workload starts dropping (timeslot 80 onwards), new hotspots stop producing or repeating queries,
5. queries come from old hotspots only which, however,
6. have not managed to discover more capacity despite exploring their entire vicinity.

Decreasing workload helps success rate to increase. Unlike previous experiments, Hoverlay is more successful until the end of this interval (timeslot 140). The main percentage of global workload is still on servers of past hotspots. Hence, the biggest portion of removed workload is also applied there and fresh capacity appears in their vicinity. However, in case of Flock of Condors, free capacity returns to its original host and owner. This may move capacity outside hotspot areas making difficult to reach from inner servers.

As a confirmation of the above, figure 4.16 shows hotspots capacity fluctuations (only the amount of capacity added to or removed from those areas). The first hotspot triplet (0-70 timeslots) significantly increases its capacity consuming all available resources in the vicinity (that increase reaches 40% of its cumulative initial capacity). The second triplet confirms the conclusions above as no expansion takes place in those hotspots. On the contrary, some capacity (60 capacity units) is removed as a result of repeated queries from past interval. Some servers of new hotspots are close to the previous ones and therefore capacity is moved between the two hotspots. The last hotspots expand just once when global workload starts increasing again.

4.9.2 Average Path Length of Successful Queries

Based on plots of figure 4.17, Hoverlay outperform Flock of Condors in terms of average query path length of successful queries. While the former fluctuates between 1 and 3

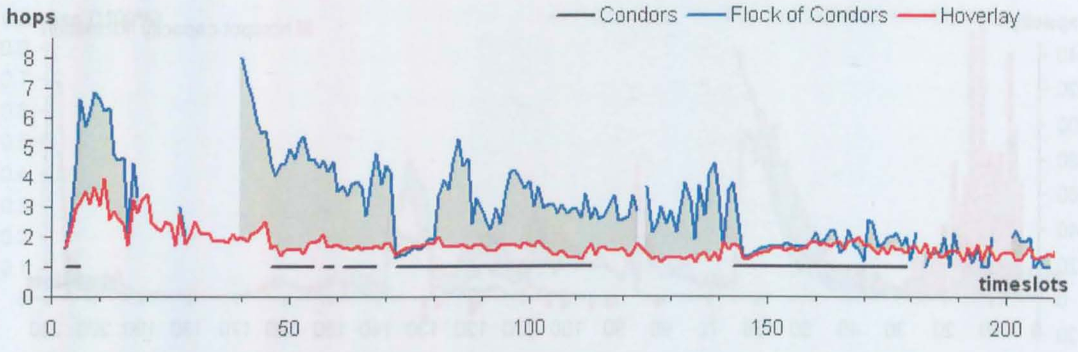


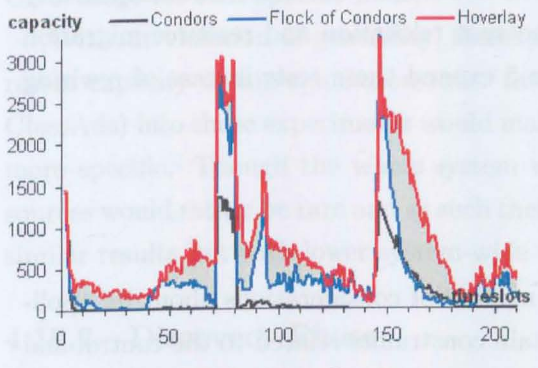
Figure 4.17: *Averge number of query hops before they discover their first answer deployed on disconnected Condors, Flock of Condors and Hoverlay in a uniform query distribution environment*

hops, the latter reaches even 8 hops with an average of 2 hops longer paths for most of the experiment.

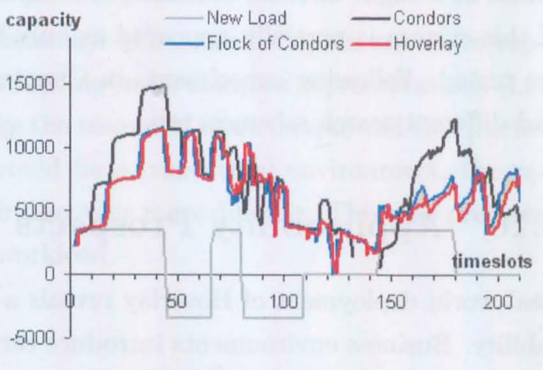
Though one would expect that cumulative hotspots capacity augments in case of Hoverlay, it does not happen as figure 4.16 reveals. These areas seem to be the most capacity-rich compared to previous ones (see figure 4.14b) and as such at least one of their centroids is close to high incoming-degree servers. The bigger and closer to hubs the area is, the biggest the probability of cyclic paths starting and terminating within same area. Flooding such a network using outgoing links deteriorates accessibility to resources outside that area. That explains why hotspots on the last interval, though highly loaded, do not significantly increase their capacity.

Initially, queries tend to travel far to discover resources as low-capacity hotspots are charged with high workload. This causes a non smooth increase of path length unlike the first scenario of Uniform Query Distribution. Resource migration helps Hoverlay reduce requested capacity per query and thus increase success in few hops. As hotspots exhaust all reachable capacity, query success drops to zero for Condor-based systems and thus no average hop count to be recorded. Within first few timeslots Hoverlay moves all discovered capacity within hotspot areas keeping query paths shorter than Condors-based systems and even gradually reducing them.

The second interval starts with a relatively small workload increase compared to capacity available within second hotspot triplet and thus both Flock of Condors and Hoverlay exhibit similar path lengths. Once workload starts dropping, all queries come from previous triplet of hotspots and fresh capacity returns to its owner: *a)* outside -if Flock of Condors- or *b)* inside -if Hoverlay- their borders; thus, the latter satisfies repeated queries faster than the former. At the final phase of this experiment, success rate of Flock of Condors approaches zero and thus the average path length is a calculation of a small sample causing a fluctuation on the graph.



(a) *Satisfied User Queries*



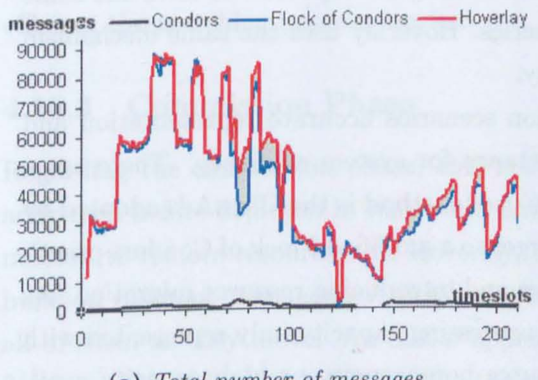
(b) *Requested Capacity*

Figure 4.18: *User-end perceived satisfaction: (a) capacity requested and satisfied from server overlays and (b) requested capacity per timeslot.*

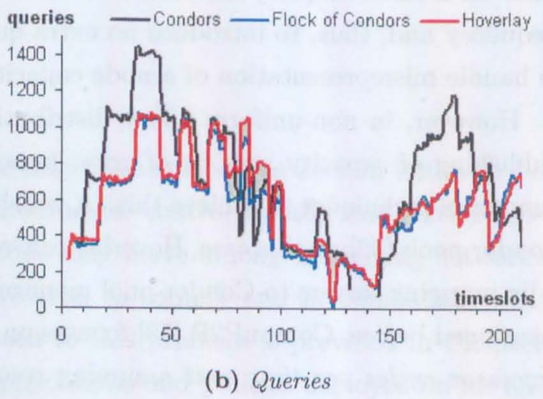
4.9.3 Satisfied Capacity and Cost in Messages

For completeness, figures 4.18a, 4.18b, 4.19a and 4.19b present the results for the remaining evaluation metrics. All lines drew in these figures follow similar patterns as those for Uniform Query Distribution experiments. In brief, Hoverlay satisfies bigger portion of requested capacity justified by the better success rate. As expected, right after the hotspot areas relocation the satisfied capacity reach its peaks as the new areas contain unused free resources. Flock of Condors and Hoverlay handles slightly more queries and experiences higher load in overlay messages requesting for the same amount of capacity. All these three graphs exhibit step-wise increases or drops as a result of repeated queries and the steady at internals additional workload.

To sum up, the experiments above evaluate Hoverlay in dynamic environments of hotspots. Though hotspot areas may change their centre, the proposed architecture achieves better performance in terms of success rate, satisfied capacity and average path length of successful queries in both high and low workload situations. Moreover, that



(a) *Total number of messages*



(b) *Queries*

Figure 4.19: *Cost in messages: (a) total number of messages produces and (b) total number of generated queries.*

comes at a slight increase in messages compared to Flock of Condors. The adaptability of this system is partially approved as only hotspot relocation and resource migration are tested. Following experiments in Chapter 5 expand these tests in case of rewiring and different search schemes, too.

4.10 Applicability Prospects

Real-world deployment of Hoverlay reveals a number of considerations about its applicability. Business environments introduce certain constraints related to the control and management of a resource. The following paragraphs present some of those important parameters concerning the proposed system's applicability. These parameters can be classified based on Hoverlay's three phases: publication, discovery and commission.

4.10.1 Publication Phase

An important aspect of the resource publication phase is its representation. As detailed in section 4.4, the current version of Hoverlay uses a rather fussy representation of the resource capacity: a tuple of network and CPU usage. Based on a number of related studies, this representation does not accurately captures the capacity of a resource. This inaccuracy derives from the nature of service capacity: stochastic variability over time and high heterogeneity of the providing machines. The proposed system implicitly relies on the reactive capacity discovery; it tolerates deviations from the advertised capacity. That is, the discovered resource may be published with a certain level of capacity which, however, may not depict the current level as it is a highly intermittent resource.

A requesting network may largely benefit from migrations of resources offering more capacity than that they were published with; this capacity may accommodate future load without triggering the discovery mechanism (can be considered as an implicit proactive mechanism). However, in some cases migrated nodes offer less capacity than that in their adverts. Instead of preventing this, Hoverlay provides the mechanism to reactively handle it by issuing a new query if the new capacity is not enough to serve the requested load. In a random query distribution, both these cases are expected to have the same frequency and, thus, to introduce no extra queries. Hoverlay uses the same mechanism to handle misrepresentation of a node capacity.

However, in non-uniform query distribution scenarios accurate representation and publishing of capacity may be of great importance for system efficiency. There are a number of techniques to achieve this. A widely used method is the ClassAds adopted by Condor pools [82]; in this case, Hoverlay converges to a modified Flock of Condors practically mapping servers to Condor pool managers and introducing resource migration. As mentioned before, CompuP2P [58] focuses on processing capacity only represented with *processor cycles per time unit* assuming resource homogeneity or high capacity availability. A method for translating the capacity of each node into units is via sampling network and CPU usage of the UNR module running on each machine. For instance, the published capacity of a node can be a (sub)multiple of the UNR client network and

CPU usage for that specific node.

Without violation of generality, the experiments above used single numbers to represent capacity within simulation runs. Introducing more complex representations (i.e. ClassAds) into those experiments would make the resources more unique and the queries more specific. Though the whole system would be a resourceful environment, the resources would rather be rare and as such their discovery more difficult. This may produce similar results but with lower system-wide workload.

4.10.2 Discovery Phase

Server misbehaviour or greedy behaviour may cause certain problems on the discovery process. For instance, a server may keep sending requests for some time gathering practically a big portion of system's capacity. As long as these resources become again available in that server's pool, Hoverlay can help others discover them and bring balance on the overlay; there is no way with the current version of Hoverlay to prevent such a phenomenon. However, Chapter 5 introduces a family of search mechanisms, Stalkers, which would disgrace this behaviour via sending many requests back to the requestor. That is, high requestors create a fame attracting more and more queries. As long as a server generates queries, it increasingly frequently appears in inbound neighbour lists. Stalkers forward queries using both outgoing and incoming links and therefore a famous requestor will soon get overloaded by queries practically bringing high maintenance and bandwidth cost for that server. This may work as a disincentive to that greedy behaviour.

In some cases, this behaviour may be the result of intentional activity with the aim of collecting and subsequently disconnecting big portion of systems resources (suicidal peer). This problem is classified to a category of security-related issues which require server authentication to ensure that each component works the way it is expected to. However, these issues become out of the scope of this thesis and are necessary to be addressed before Hoverlay is deployed on real business scenarios. Finally, resource matching is an important step of the discovery process and is highly correlated to the publishing mechanism. The more accurately a resource has been published the more efficient the matching process can be.

4.10.3 Commission Phase

Regarding the commission phase, this node migration introduces certain issues to be addressed before deployed in real world environments: control transfer and service submission to remote resources. In Hoverlay, node may move among underlying networks built on different management policies; they need to comply and be compatible with all of them as they move. An initial approach to this problem is provided in Chapter 3 using a set of keywords to describe the applications and policies deployed on an underlying network and those a node is not compatible with. Following the principles of section 3.2, queries can be more selective with regards to the appropriateness of discovered resources. This strengthens the resources uniqueness within the system practically contributing into their scarcity.

After resource migration, its UNR has to facilitate the execution of the distributed application onto that node. This practically means that the new service (job) needs to be submitted to the machine of that node. Condor system maturity on job submission toolkit can provide a suitable and realistic approach to the distributed application deployment on the fetched nodes. That scheme may offer a generalised approach but Hoverlay, for heterogeneity purposes, does not constraint the way a new node joins an underlying network. Apart from the use of Condors submission mechanism, the new node may already have the service to be deployed in which case only its parameterisation would be enough to bootstrap the node in that network.

Another issue to consider while a node migrates from one network to another is the workload left over from those served in the past. As long as its workload remains below the underload threshold, that node can move among different networks. Upon migration, the node cannot discard existing load which is, however, added to the one from the new network. After a number of migrations, the underload threshold may stay fully occupied by past workload preventing any further move. Therefore, a node may simultaneously participate to a number of networks serving very different applications but queries produced by any of those services reach the current manager server of that node. In non uniform capacity distribution environments this mechanism may affect the success rate as each server tries to optimise its neighbourhood via rewiring based on the answers it gets. If keywords or other kind of semantics are introduced, as in section 3.2, then using the current only server may disadvantage queries generated by past workload. In that case, access to multiple servers can be a technique to alleviate this problem. As mentioned in the final chapter of this thesis, this is an interesting avenue for future work.

4.11 Summary

Hoverlay is a system that enables logical movement of nodes from one network to another aiming to relieve requesting nodes which experience high workload. Remote nodes are moved into the requesting node domain to take over some of that excessive workload. It is an arbitrary network of servers (overlay) each of which represents a single underlying network. All servers use blind search techniques to discover free nodes from other networks and move them into the requesting network. It is designed to be tolerant to node and server failures since it has minimized the maintenance costs of server and node components.

Node migration and dynamic server overlay differentiate Hoverlay from Condor-based architectures which exhibit more static links between managers and nodes. A simulator tested on a set of rules was used for its evaluation aiming at conceptual characteristics of the architecture in static environments. After a number of experiments in two scenarios of uniform and hotspot query distributions, results proved that, on both scenarios, Hoverlay performs better than disconnected Condors and Flock of Condors achieving important improvements in both success rate and average successful query path length at a negligible expense in messages.

The following part of this thesis focuses on searching algorithms deployable on Hoverlay. It proposes a set of new blind search mechanisms, Stalkers, able to track resource migrations and outperform other well-known ones in dynamic environments of non-replicable reusable resources such as service capacity. Appropriate evaluation of those algorithms deployed on Hoverlay are also provided.

Chapter 5

Stalkers: Resource Migration Detection

Stalkers is a collection of three algorithms based on the same principles: ***FloodStalkers***, ***k-Stalkers*** and ***FireStalkers*** which were published in [43] and [41] under different names: *Scale-free FloodWalkers*, *Scale-free Walkers* and *Firewalks*, respectively. Their main common features are that all:

1. are based on *k-walkers*: Query recall is not a priority for Hoverlay and k-Walkers is a non-flood-based algorithm which produces low traffic compared to other search algorithms. However, its fluctuating success rate and higher latency remain challenges that Stalkers need to tackle.

2. give priority to *fresh* links: Stalkers increase their efficiency by capturing the most up-to-date configuration and topology of the network. Resources are not permanently located at some place and therefore links should, ideally, always link requestors to providers.

3. use both outgoing and incoming links: Resource migrations attach certain implicit semantics on links as a link initiator has received while its target server has provided resources. This is a way to track resource migrations.

As nodes migrate and servers experience fluctuations in the capacity they handle, appropriate rewiring could be used to bring requestors closer to providers. If providers were always the same servers, no server would benefit from connecting to other servers; however, this is not the case with Hoverlay. Rewiring frequency is an important factor; re-linking at a lower rate than the resource migration rate would cause a false capture of network topology but if it is done faster it would create excessive connection overhead that could overwhelm servers.

The following sections present an overview of design requirements for Stalkers, their actual generic design principles and the detailed description of all three aforementioned variations. This chapter finishes with their evaluation and comparisons with Flooding and k-Walkers before a summary.

5.1 Resource Discovery Requirements

Following the principles of any proposed architecture for sharing service capacity, as detailed in section 1.5, this one sets a set of requirements for the proposed search mechanism, Stalkers. Its key objective is to achieve low latency and good success rate despite the ever changing resource location and network topology.

Service capacity features raise a number of difficulties to be addressed. According to Hoverlay topology, there is a number of servers hosting free service capacity. As long as a server receives queries and responds back to requestors it attracts more links increasing its popularity. However, the more incoming links a server gathers the faster its resources migrate. Resource migrations allows requesting servers to gather many resources which when freed in its local Pool attract many links. Symmetrically, servers with little or no capacity lose incoming links. Therefore, these migrations ease the promotion or demotion of servers to hubs or leaf servers allows. As hubs lose their resources other servers become resource rich and it is their turn to attract links and become famous. Besides the factors analysed in 2.5 that make a network power-law, the appearance of hubs in Hoverlay depends on another four factors:

- *average requestors/providers density*: represents the percentage of requestors / providers within the servers overlay. If a handful of requesting servers gather a good portion of global service capacity, they are likely to be the future power-law hubs once they free these resources. Resource distribution becomes more uniform as the average query rate of servers is uniform too; that is, if all servers ask for the same amount of capacity per time unit, the resources tend to get evenly distributed among servers. If there is only a minority of good providers they tend to be the current power-law hubs especially if requestors are a majority.

- *requestors/providers density distribution*: shows how these servers are spatially distributed within the network. Splitting the Hoverlay unstructured overlay into areas of equal size, the density distribution represents the percentiles of a certain type of servers (i.e. requestors or providers) in those areas. If that ratio is the same for every area of that overlay then the density distribution of this type of servers is uniform. Otherwise, if, for instance, a minority of areas have high ratios of those servers then these servers are locally clustered. In case of Hoverlay, if most of the requesting servers are clustered in neighbouring areas of the overlay, they quickly consume any available resource within their vicinity and their queries need to travel beyond the borders their areas to access potential providers.

- *query generation rate*: refers to the average number of queries per server. It affects the resource migration frequency and thus the topology adaptation frequency. High query generation rate may shorten links lifetime and conversion from power-law to uniform topologies and vice-versa may become faster.

- *requested capacity rate*: represents the average service capacity requested per requesting server. It affects discovery success rate and number of replies but not the overlay topology as Hoverlay servers may respond with a minimum capacity equal to the requested one. However, networks with low query generation rate and few requestors but high requested capacity rate might gather plenty of resources in those servers which if freed in future may become power-law hubs.

While the first two factors relate to search efficiency the last two contribute to the Hoverlay topology changes.

Any discovery mechanism deployed on such dynamic environments needs to be adequately adaptable to achieve good success rate and low latency. Previous sections described important features of widely used algorithms. While educated informed techniques may achieve a good success rate, their adaptability is weak due to statistical data they need to retain distributed all over the network or expensive if this data needs to be promptly updated. The flood-based blind ones achieve good success rate at a high cost in messages regardless of topology changes. If non-flood-based ones are deployed, queries tend to reach high degree nodes much faster than others but have more variable success rate, higher latency and are not equally efficient at discovering rare resources. Assuming that there is no rewiring policy in place, all these mechanisms would be unable to track resource movements as links once established cannot change and every query from a requestor would follow the same path as its previous one. Given that service capacity migrates and old providers may currently have no resource left, queries could be trapped into resourceless paths. therefore, this lack of adaptability may affect search efficiency as well.

Adapt: *Rewire once a resource migrates*

Well-known uneducated informed techniques avoid statistical data but rely on degree distribution. Each query needs to hop to neighbours with the highest degree first probing all of them unless every server receives updates with the degree of its neighbours. While the first scheme introduces significant latency, the second introduces extra messages. These search mechanisms target high-degree nodes assuming that they are good providers. As long as such a node has available resources this is a valid assumption but in case of Hoverlay the more links a server gets the faster it loses its resources. Without the appropriate adaptability, queries forwarded based on these schemes get trapped to fixed paths as in case of blind and educated informed techniques. To alleviate this problem, the proposed search mechanism should be able to forward queries to providers and requestors, too. Providers are the servers that have offered an answer to the query originator and requestors the ones whose queries reached that server.

Search: *Forward queries to both providers and requestros*

There is an assumption for the design of Stalkers: *recall*; the percentage of resources discovered over all those that could be returned for that request, is not a important parameter for Hoverlay as any valid discovered resources would be used. Flood-based techniques, though they may provide better recall, are more expensive compared to non-flood-based ones. Informed techniques cannot work as the base of Stalkers. Therefore, Stalkers can only be a non-flood-based blind search mechanism. If query generation and/or requested capacity rates are high, topology changes may be fast; thus each topology settings have short lifetime. Hoverlay is a topology changing between shortlived power-law and uniform modes. Stalkers needs to exploit those power-law settings but

Search: *Base of Stalkers is a blind non-flood-based technique.*

also have a parallel mechanism to adapt as soon as these settings change.

This introduces the concept of links lifetime in that discovery mechanism. *Freshness* of links is a factor that may help queries reach resources that were recently moved. It is a piece of information already present in servers (Neighbour Lists) and require no maintenance as it gets updated once a link is created or deleted. Fresh incoming links lead to recent requestors which *a)* may soon become providers as soon as their workload drops and they free resources and *b)* can be source of other recent providers discovered via their last queries. The fresh outgoing ones drive queries to recent providers which *a)* may still have available resources and *b)* are sources of recent requestors.

Search: *Choose recent neighbours to forward queries to.*

Stalkers is a non-flood-based search algorithm that uses fresh incoming and outgoing links to forward queries to and relies on rewirings upon answer deliveries for adaptability purposes. Therefore, the choice of links is not blind but based on certain implicit criteria, *freshness*. This classifies Stalkers into the *Uneducated Informed Techniques* category.

5.2 Generic Design

Rewiring is a complement of search algorithms in Hoverlay architecture. As specified in section 5.1, new links need to be created for every resource migration from requestor to provider server. From requestor's point of view, that is an outgoing link and for a provider an incoming one. This helps both to identify fresh providers and requestors, respectively. Though query recall is not important it may be useful. If rewiring was taking place for the first and only accepted answer, a requesting server would ignore knowledge about network topology available in extra received responses. Every recent answer is an indication of fresh providers; if rejected, the probability that their originators are still providers goes up as their resources are not used and remain free in their original pool. Therefore, rewiring actions are invoked on every received answer regardless if accepted or not.

Practically, the size of both outbound and inbound Neighbour Lists are finite. The more connections a server has the bigger its burden to handle their overhead and traffic is. Therefore, the outbound list can be limited to a user-defined value whereas the size of the inbound list depends on other servers. In case these lists are full and a new link is to be created, the oldest one is replaced. If an answer originator is already in requestor's outbound list there is no update action on that list (not even for the timestamp of that link):

- If that providing server has been a provider all the time since the link was created then its resources have been migrating throughout that period; thus it is not a new provider.
- If there were intervals during which it acted as both requestor and provider there are two cases: either the answer receiver does not frequently send out queries and its network path to that provider has not significantly changed since link creation or the provider has experienced significant and frequent workload fluctuations during that period. Not updating the link timestamp in this latter case is a way of

punishment for that provider's unreliability. However, the requestor is unable to know which of those two cases holds and the former one does not severely affect the success rate since the network evolution is slow and query rate of that requestor low.

Refering to Chapter 2, high-degree nodes of a power-law network are connected to a good number of other nodes thus making them easily discoverable. Though rewiring tends to create such topologies, resource migration shortens their lifetime unless power-law hubs are able to offer fresh resources. The main concept behind Stalkers is two-folded: a query forwarded to fresh requestors can quickly locate *a)* good and famous providers via their outbound lists and *b)* recently freed local resources. Using prior knowledge collected by other nodes, one can improve its query success. This knowledge is recorded on each node by updating its neighbour list with the discovered providers. Every received answer triggers the updating process of requestor's neighbour list. Therefore, a fresh requestor may have recently discovered resources which may soon release and know a set of other potential providers. Every link's reliability and freshness are closely and positively related.

Though Hoverlay avoids partial answers to prevent deadlock situations, resources may stay reserved in their local pool from the moment they are attached to an answer till the requesting server positively or negatively acknowledges it. Therefore, the more servers a distributed discovery mechanism visits the more service capacity may be reserved, even temporarily, and thus the higher its impact on its availability is. *k-Walkers* is a blind non-flood-based technique producing low traffic and minimizing that resource reservation phenomenon. High-incoming-degree servers, if not good providers, are good pools of fresh requestors; if not fresh then they would not have high incoming degree in the first place. In power-law environments *k-Walkers* tend to reach high-degree nodes easier than others which in combination with rewiring and visiting of both outbound and inbound neighbours may stabilise their success rate and reduce latency.

Fresh providers are a good source of fresh requestors and vice-versa. Therefore, once a query reaches a provider it needs to be forked to a subset of its incoming links and when it arrives to a requestor be forwarded to a subset of outbound neighbours. Paths created only by outgoing links usually drive queries to high-degree servers (see section 2.10) assuming that a random neighbour is selected on every step. However, if that selection is biased, e.g. freshest ones in priority, there is no guarantee that path leads to power-law hubs. On the contrary, a fresh requestor has built the freshest outbound list it could have, thus, giving indirect access to high-degree servers and potential providers.

As with k-Walkers, a query originator, using Stalkers, sends out the same request to its k direct neighbours. This initial step may use, depending on the specific Stalkers variation, links from any or both outbound and inbound Neighbour Lists. Once a server receives a query that is unable to satisfy (see Chapter 4) it forwards it to a mixture of the freshest incoming and outgoing links, not necessarily both. All these three 'flavours' presented below build k independent paths which in case of:

- *FloodStalkers* are made of outgoing links only but are forked on every intermediate server to a subset of its incoming ones.

- *k-Stalkers* resemble k-Walkers but the freshest (not random one) among both incoming and outgoing is selected to hop to.
- *FireStalkers* are similar to k-Stalkers but do a broadcast on the very last step or if a very fresh link is discovered.

The three most important metrics for the evaluation of search mechanisms deployed on Hoverlay are: query success rate, average query path length (a.k.a. latency) and number of messages per query. Stalkers variations are designed to optimize one of those metrics. FloodStalkers visit many more nodes than k-Stalkers and FireStalkers aiming at high success rate. FireStalkers try to reduce the average query path lengths whereas k-Stalkers focus on reducing the number of messages. Moreover, k-Stalkers is the uneducated informed (see 2.10) version of k-Walkers. Stalkers introduce two important concepts: '*fresh*' and/or '*incoming*' links are more likely to direct queries to providers. Comparison between Flooding, k-Walkers on outgoing links and Stalkers will provide useful material for evaluating these concepts.

All descriptions of Stalkers algorithms below assume that query forwarding stops when a server is able to satisfy the query. Any parallel paths of the same query may independently keep expanding till their TTL expires, they discover appropriate resources or revisit a server that has already processed it.

5.3 FloodStalkers

FloodStalkers deploy a random k-Walkers scheme for discovering high-degree providers and one-hop broadcastings to the outbound list of a subset of their recent requestors for locating recent providers and free resources. Every server has different behaviour to queries received from incoming links compared to those from outgoing ones. In the first case, it is part of a walker and as such it needs to forward every query, if unable to satisfy, to a single random outgoing link and to w most recent inbound ones. Randomness for walkers is used to help paths reach high-degree servers as they are a good source of requestors; the higher their degree the more probable is to have fresh incoming links (a server is at its peak degree a bit after it has lost all its resources).

Branches to incoming links may locate recently discovered by those servers resources which, due to their frequent workload fluctuations, are now available again in their pools. Moreover, a recent requestor might have received a number of answers for its last queries and appropriately updated its outbound Neighbour List with originators of those responses. Each query may have resulted in resource migration from one server only and thus any unused fresh responder in that list has increased probability to still have those resources. If that recent requestor is unable to satisfy the query, it broadcasts the same query to its outbound fresh list.

FloodStalkers resemble to Lookahead Random Walkers [77] but exhibit some different features:

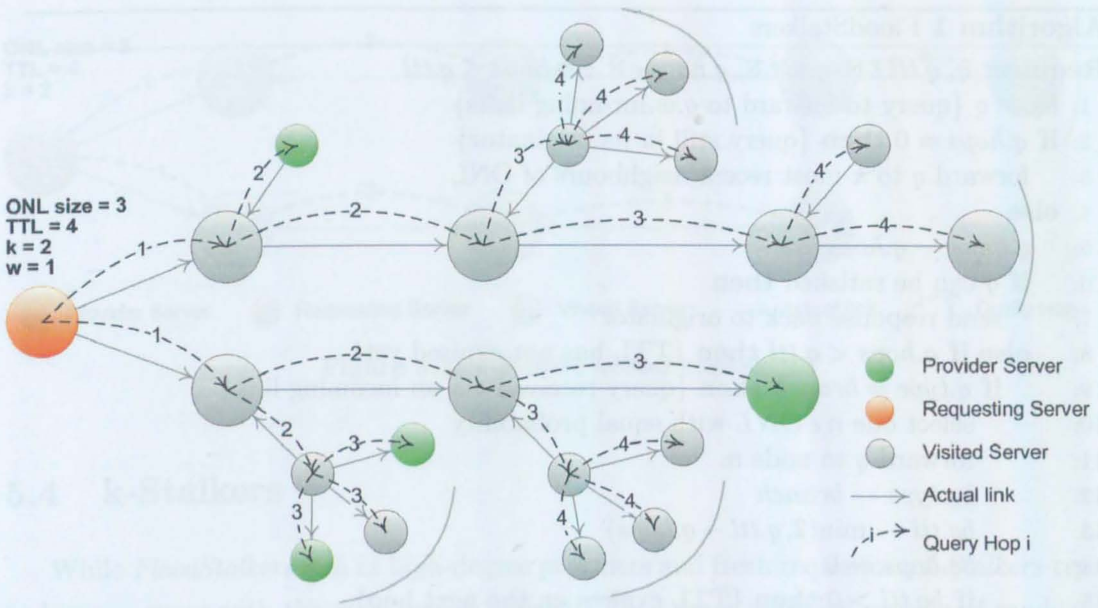


Figure 5.1: *FloodStalkers worked example*

- they use outgoing links only for walker travelling and incoming ones for the '*lookahead*' phase,
- only a fraction of those incoming links are used for branching,
- walkers are independent from their branches as walker forwarding does not depend on the degree of those branches,
- each branch terminates after a one-hop broadcasting to outbound neighbours.

A detailed view of this algorithm is available in figure 5.1 and pseudocode of Algorithm 1.

Every node in the network has a fixed-size M Outbound Neighbour List (ONL) of providers and a variable-size Inbound Neighbour List (INL) of requestors. The query originator node starts k walkers selecting k random neighbours from its ONL. Each walker travels from its originator via intermediate nodes (intermediaries) and terminates when either discovering a response provider or after a maximum of $q.ttl$ steps away from its source. The intermediaries use one random neighbour of their ONL to forward the walker to. If they are located at most $q.ttl - 2$ steps away from query originator, they use random w of the most recent inbound neighbours from INL (branch-intermediaries) to forward the same query to. The branch-intermediaries either respond or again broadcast the query with unitary TTL ($q.ttl = 1$) to all their outbound neighbours. Therefore the query can be either of type *normal*, forwarded as walker, or *branch*, forwarded via branch-intermediaries.

It is interesting to note that on $q.hops = q.ttl - 1$ the branch does not carry on the final broadcasting and that on $q.hops = q.ttl$ incoming links are not used at all. Both happen in order to prevent a query travel beyond the maximum TTL hops away

Algorithm 1 FloodStalkers

```
Require: $q$, $q.ttl \in \aleph$, $q.w \in \aleph$, $q.hops \in \aleph$ : $q.hops \leq q.ttl$
 1: $bq \leftarrow q$ {query to forward to $q.w$ incoming links}
 2: if $q.hops = 0$ then {query still in its originator}
 3:   forward $q$ to $k$ most recent neighbours of ONL
 4: else
 5:   $q.hops \leftarrow q.hops + 1$
 6:   if $q$ can be satisfied then
 7:     send response back to originator
 8:   else if $q.hops < q.ttl$ then {TTL has not expired yet}
 9:     if $q.type \neq branch$ then {query received via an incoming link}
10:       select one $n \in ONL$ with equal probability
11:       forward $q$ to node $n$
12:       $bq.type \leftarrow branch$
13:       $bq.ttl \leftarrow \min(2, q.ttl - q.hops)$
14:       $bq.hops \leftarrow 0$
15:       if $bq.ttl > 0$ then {TTL expires on the next hop}
16:         select the freshest $m_i \in INL$ where $i = 1, ..., q.w$
17:         forward $bq$ to all $m_i$
18:       end if
19:     else {query received via an outgoing link}
20:       forward $q$ to all $n \in ONL$
21:     end if
22:   end if
23: end if
```

from originator. Given the fixed size of ONL, high-degree nodes are prevented from broadcasting to a big portion of the network. Practically, an INL has a maximum number of entries which are updated on a first-in-first-out mode.

Though this algorithm adds a higher cost on any network it is deployed, compared to random walkers, it achieves a more stable success rate for two reasons: the branches of walkers *a)* contribute to resource discovery even if the walker abruptly terminates due to a server failure and *b)* in combination with that freshness-based priority system allow search paths to overpass traps of famous and resource-empty servers. k-Walkers have only k paths and visit a very small portion of the network; an abrupt termination of one of them may seriously affect the success of a query. While on one time unit failed servers may terminate quite a few query paths, on the next one they may not be in their paths at all. FloodStalkers alleviate this phenomenon by using those branches and, thus, increasing the success likelyhood. Finally, it is a more attractive technique compared to any flood-based one as its cost increases linearly (vs. exponentially) with TTL of queries.

Different varieties of FloodStalkers may emerge if e.g. a requestor server initially sends out to k of its freshest, either outbound or inbound, neighbours or random walkers are replaced by paths comprising the freshest outbound link of each intermediary. Due to that broadcasting phase, *FloodStalkers* may produce, in general, fewer than Flooding messages but they could be comparable in case of low TTL.

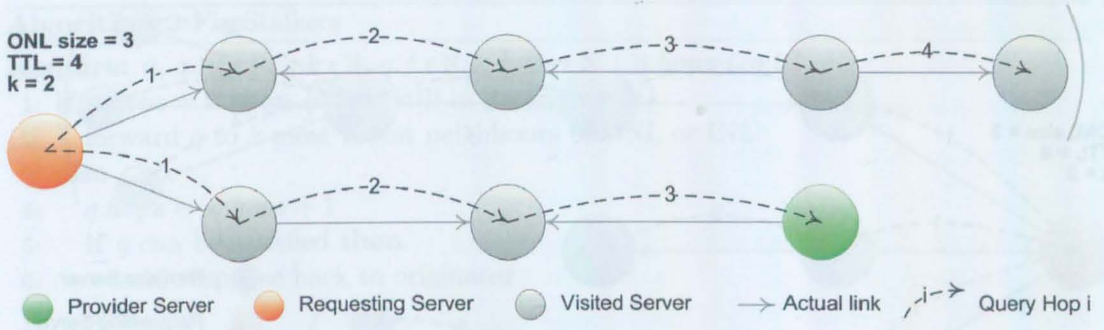


Figure 5.2: *k-Stalkers worked example*

5.4 k-Stalkers

While *FloodStalkers* aim at high-degree providers and fresh requestors, *k-Stalkers* try to locate servers with the most recent activity. They are a modified version of random k-Walkers as each server gives priority to the freshest link. As such, they introduce same workload on the overlay as k-Walkers. They do not provide, however, guarantees that with this scheme walkers head to the most recently active servers in Hoverlay.

Initially, a random subset from query originator ONL is selected to fork the query to, but all intermediaries have to choose one neighbour, the most recently added to their either ONL or INL. Each walker stops when a response is found, TTL expires or a circle has closed on a server that has already processed that query. Despite that preference to fresh links, a certain level of randomness is introduced with that random initial multicasting and random changes to network connectivity while walkers travel till they expire. Keeping same notation as in FloodStalkers, Figure 5.2 and Algorithm 2 provide further details.

Assuming that a requestor has discovered a number of providers via its latest search, a query targeting the most recent requestor hopes to discover recent providers. If, in the meantime, requestor state has changed to a provider one it would be beneficial for that query.

Algorithm 2 k-Stalkers

```
Require: $q$, $q.ttl \in \aleph$, $q.k \in \aleph$, $q.hops \in \aleph$ : $q.hops \leq q.ttl$
1: if $q.hops = 0$ then {query still in its originator}
2:     forward $q$ to $k$ most recent neighbours of ONL
3: else
4:     $q.hops \leftarrow q.hops + 1$
5:     if $q$ can be satisfied then
6:         send response back to originator
7:     else if $q.hops < q.ttl$ then {TTL has not expired yet}
8:         select the most recent $n \in ONL$
9:         select the most recent $m \in INL$
10:        forward $q$ to the most recent node between $n,m$
11:    end if
12: end if
```

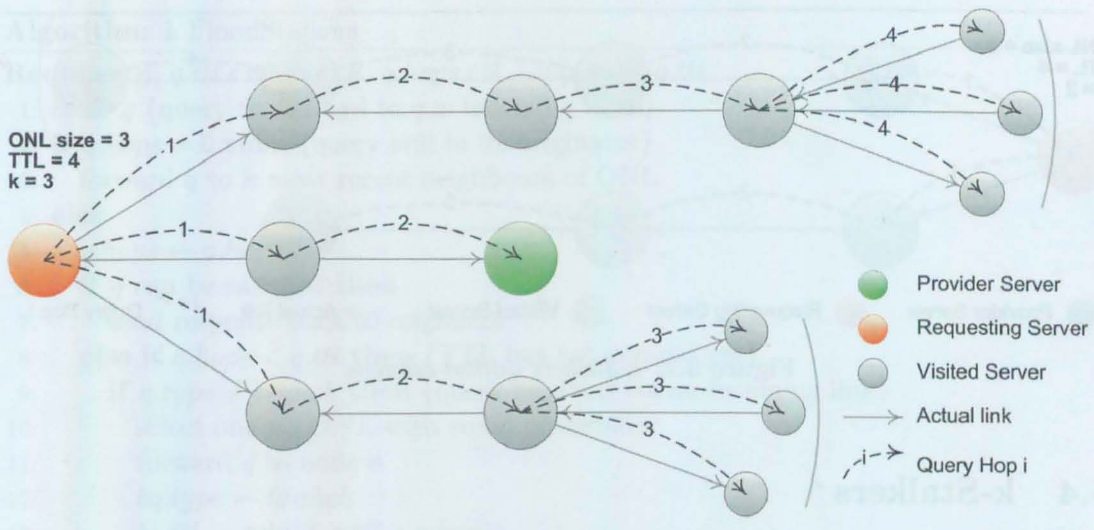


Figure 5.3: *FireStalkers worked example*

5.5 FireStalkers

FireStalkers is another Stalkers variation similar to k-Stalkers but with the differences of trying to keep their cost in messages low but increase their efficiency. Query originator servers start k-Walkers selecting the k most recent neighbours from both their ONL and INL. Each walker travels from his originator via intermediate nodes (intermediaries) and terminates with a k-multicasting on the last hop. The server located on the $q.ttl - 1$ hop of a query path forwards the same query to all its $q.k$ freshest inbound and/or oubound neighbours (final multicasting).

In principle, for every hop of that path between the first and last one (i.e. $1 < hop < q.ttl - 1$), servers select only the freshest neighbour from both INL and ONL. However, if that link was created within a time window $q.f$ then the walking path converts to one-hop k-multicasting just like the '*final multicasting*'. After these multicasting actions, query forwarding stops even if the maximum TTL is not yet reached. A link created within the last $q.f$ time units is the result of a very recent query. That query may have also triggered more rewirings helping the current server to significantly update its Neighbour List with fresh providers. The reason FireStalkers do not use broadcasting is that in case of a high-incoming-degree servers it would cause flooding of a big portion of the network. Figure 5.3 and pseudocode listing of Algorithm 3 give a more structured and comprehensive overview of FireStalkers.

5.6 Simulations and Evaluation

With appropriate extensions, Omeosis (Hoverlay simulator presented in section 4.7) may also run experiments regarding different deployed search mechanisms. That evaluation section left two parts of the proposed architecture in abeyance: *rewiring* and *Stalkers*. Rewiring on every received answer is a mechanism introduced to improve network adaptability and efficiency in the presence of resource mobility. Stalkers use two uneducated

Algorithm 3 FireStalkers

```
Require: $q$, $q.ttl \in \aleph$, $q.k \in \aleph$, $q.f \in \aleph$, $q.hops \in \aleph$ : $q.hops \leq q.ttl$
1: if $q.hops = 0$ then {query still in its originator}
2:    forward $q$ to $k$ most recent neighbours of ONL or INL
3: else
4:    $q.hops \leftarrow q.hops + 1$
5:    if $q$ can be satisfied then
6:       send response back to originator
7:    else
8:       $r \leftarrow$ the freshest outgoing or incoming link
9:       if $q.hops = q.ttl - 1$ OR $r$ created within the last $q.f$ timeunits then
10:         $q.ttl \leftarrow q.hops + 1$
11:         forward $q$ to $k$ freshest links among outbound and inbound ones
12:      else if $q.hops < q.ttl - 1$ then
13:         forward $q$ to $r$
14:      end if
15:   end if
16: end if
```

informed mechanisms (give priority to*fresh* links and visit both outbound and inbound neighbours) to track resource migrations.

Rewiring is a mechanism present in most real networks. Though there is a variety of reasons for rewiring that could change the topology of a network, the experiments below focus on the one triggered upon an answer delivery: preferential rewiring. As explained in 4.7, the overlay for these experiments is built using preferential server-to-server attachments converting the network to a power-law incoming-degree distribution. After this initialisation phase server overlay size is fixed and rewirings only take place. Good resource providers get popular quickly but as nodes move out and the server fail to answer queries this fame gradually drops and its incoming links are rewired to other servers. This allows resources to quickly gather to areas in need of capacity responding to workload fluctuations; this is a measure of Hoverlay adaptability.

Following similar experimentation practices as of section 4.7, rewiring and Stalkers evaluation rely on the same evaluation metrics and simulation parameters as in section 4.7. Thus, direct comparisons between results of all experiments are easier; two more search-specific parameters comprise the complete configuration of the following experiments:

- *Number of Walkers*: all versions of Stalkers are random walkers-based algorithms and therefore the number of walkers needs to be user input. For all the following experiments this is set: $k = 3$.
- *Freshness Time Window*: FloodStalkers and FireStalkers use a time window. The former choose $w = 2$ incoming links to forward a query to and the latter does a k-multicasting from an intermediary server if its freshest link was created within the last $f = 2$ time units.

Flooding and k-Walkers function as benchmarking for Stalkers evaluation and share same parameters (i.e. *TTL* & *k*). Flooding is an expensive but efficient technique whereas k-Walkers an inexpensive with variable relatively low success rate: they comprise the two extremes of uneducated search mechanisms.

Each search technique has a specific coloured representation in figures below. Though all are plotted as solid lines, a single colour corresponds to all plots of any metric for one search method. That is one-to-one mapping of methods to colours for all figures below:

- Flooding: *gold*
- k-Stalkers: *blue*
- FloodStalkers: *red*
- k-Walkers: *brown*
- FireStalkers: *green*

However, in case of experiments without rewiring in an effort to avoid many colours and to improve readability, greyscale only is used for all methods. While one-to-one mapping is not followed for those plots please refer to figure legends if grey ones are present.

The following sections present an analysis of the results for both rewiring and search methods gathered from experiments on random workload distribution among all underlying nodes. The contribution of rewiring is explained alongside Stalkers evaluation. Following similar analysis patterns as in section 4.7, the results presentation starts with success rate and average number of hops followed by messages spent and requested capacity.

5.7 Query Success Rate

Figures 5.4 and 5.6 compare the success rates achieved by all five simulated search mechanisms. While the former plots success rates of Flooding, k-Walkers and k-Stalkers, the latter is a figure exclusively for Stalkers. Both plot the same blue line (mapped to k-Stalkers) as a reference to ease comparisons between any subset of search techniques. With this reference line one may visually get a good estimation of the distance between any two plots of those figures.

Both Flooding and k-Walkers plot lines are well (roughly 10%) below the k-Stalkers one. FireStalkers achieve about 5% better success than k-Stalkers whereas FloodStalkers outperform all evaluated methods reaching even 60% better success rate than k-Stalkers. Therefore, ordering their ratios starting from the highest one, the following sequence confirms that FloodStalkers achieve their target (success rate): $S_{FS} > S_{FRS} > S_{KS} > S_{KW} > S_{FL}$ where S_m is the query success ratio of the mth method with $m = \{FS, FRS, KS, KW, FL\}$ representing the FloodStalkers, FireStalkers, k-Stalkers, k-Walkers and Flooding respectively.

Flooding and k-Walkers underperform Stalkers mainly because they do not manage to trace resource migrations and get trapped into high-degree but weak providers. The server overlay is initially configured as a power-law topology with many leaf servers linking to hubs in its centre (giant component). Given the uniform distribution of workload onto the nodes, the vast majority of queries originate from those leaf servers. Flooding and k-Walkers use outbound neighbours only to propagate queries and therefore they quickly reach hubs. The rewiring scheme forces those leaves to link deeper within the overlay and closer to its hubs whose resources move out as received queries increase.

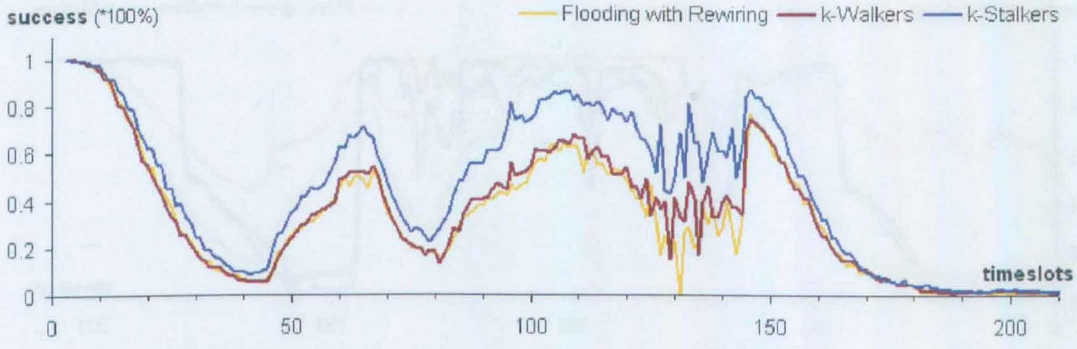


Figure 5.4: *Success rate comparison of k-Stalkers against Flooding and k-Walkers in the presence of rewiring*

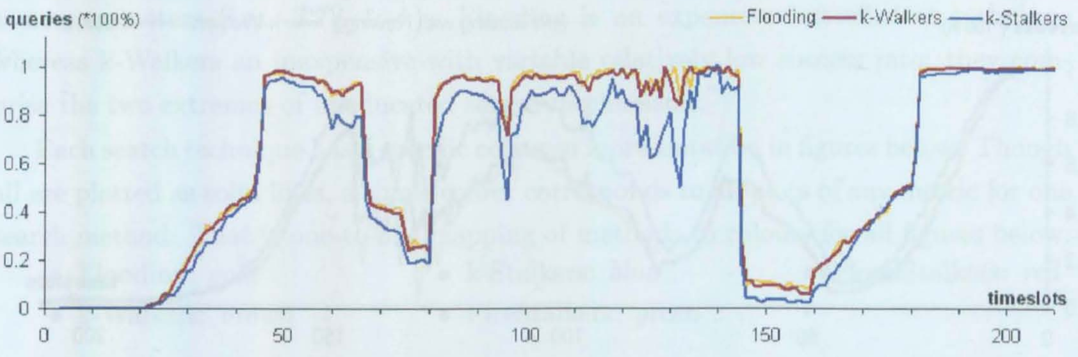
The hubs are well interconnected between each other and since they are a minority of servers their outgoing links miss the big portion of leaves to which resources might have migrated. Therefore, those two blind search schemes force resources move out of their paths and combined with rewiring leaf servers have finally access to very limited resources. This makes the success rate drop and, hence, rewirings become rarer even more affecting the success of those search mechanisms.

On the other hand, Stalkers with the use of incoming links avoid this phenomenon and locate resources even after their migrations. Rewiring ensures that the topology adapts to those migrations and the freshness-based priority scheme for query forwarding speeds up those adaptations. The giant component initially receives many queries and while providing resources, its servers accumulate a number of fresh incoming links that Stalkers use to locate resources practically avoiding the 'trap' that Flooding and k-Walkers got into. While rewiring and query propagation to inbound neighbours drive queries to the right direction, priority to freshest links helps them increase their success likelihood.

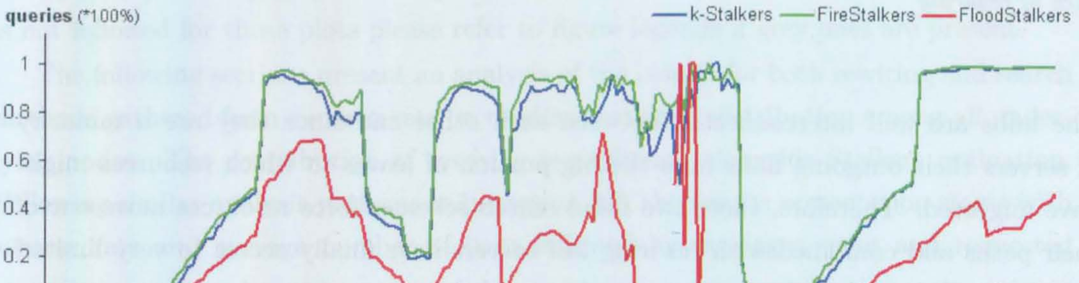
In general, all lines follow similar patterns as those of figure 4.9 with success rate increase on global workload drop and vice versa. Initially (1-44 timeslots), the requested capacity increases faster than workload and quickly pushes down the success rate of all methods. This happens because

- more and more resources get busy and even overloaded and no fresh capacity appears into node pools,
- fewer queries are satisfied causing relatively soon query repetitions,
- requested capacity increases even more with repeated queries.

Thereafter, till timeslot 65, global workload decreases and, hence, fresh capacity appears in pools, load of some nodes drops below their overload threshold and some repeated queries stop as unnecessary; thus, success rate increases. Though global workload within timeslots 109-143 gets at its lowest point, all techniques (apart from FloodStalkers) appear to have unstable success rate. During this interval only a few queries travel



(a) *k-Stalkers versus Flooding and k-Walkers*



(b) *FloodStalkers versus FireStalkers versus k-Stalkers*

Figure 5.5: *Percentage of repeated over total number of queries for Flooding, k-Walkers and Stalkers in uniform query distribution environment*

within the overlay (see figures 5.11a and 5.11b) causing that fluctuation. The percentage of repeated over those few queries is relatively high (see figures 5.5a and 5.5b) and their horizon is almost resource-free as otherwise they would have been cancelled during the workload drop of the previous phase.

From timeslot 181 onwards, the success rate drops lower than that of timeslot 44 though the system-wide workload reaches the same peak. Apart from new queries due to workload increase, there are still repeated ones from previous intervals (i.e. 109-143 as above). They remained unsatisfied even after the biggest workload drop; this increase and finally stabilisation of system-wide workload at its highest point obliterates most probabilities for discovery.

5.7.1 k-Stalkers vs. Flooding vs. k-Walkers

k-Walkers achieve roughly the same success rate as Flooding. Both these techniques are deployed on a power-law degree distribution overlay and in combination with the rewiring mechanism they increase the degree of famous servers. Given that most of the queries come from leaf servers they can both exhaustively explore their very small portion of the network within their horizon. k-Walkers becomes noticeably more successful compared to Flooding during the lowest system-wide workload levels on 109-143 timeslots interval. Flooding exhibits a higher ratio of repeated over total number of

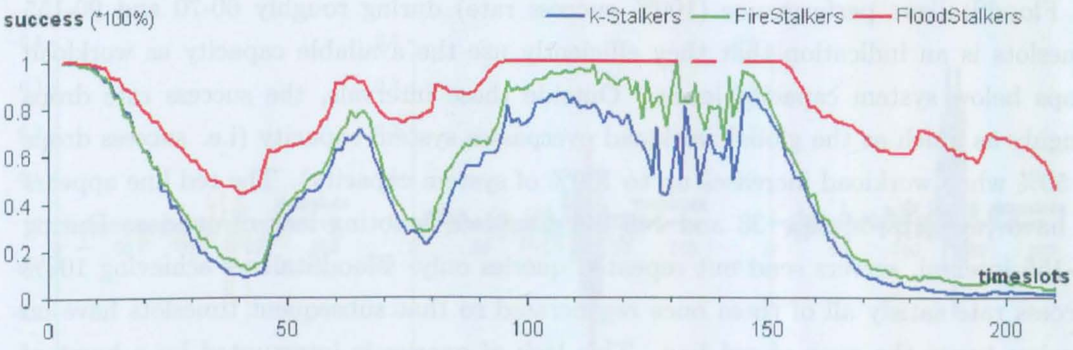


Figure 5.6: *Success rate comparison of k-Stalkers, FireStalkers and FloodStalkers in the presence of rewiring*

queries compared to k-Walkers for that interval (see figure 5.5a). For quite some time these repeated queries try without success to discover capacity and therefore remain unsatisfied affecting success rate. The biggest this ratio is, the lower the success rate.

k-Stalkers achieve an improvement in k-Walkers success rate throughout the experiment reaching a maximum of 20% due to query forwarding via both inbound and outbound neighbours. As analysed before, by following the most recent neighbour, queries manage to trace resource migrations and avoid deadlocks and loops. While system-wide workload monotonously increases this has no positive effect on the success rate as migrated resources are busy. However, queries, with the aim of tracing node migrations, finally visit servers with updated outbound neighbour lists and thus indirectly access the ones offering free resources.

While workload decreases, portion of system capacity becomes available but some nodes keep regenerating queries. In those cases, queries manage to discover resources on their pools and via inbound neighbours, too. Therefore, the benefit from inbound query propagation is access to outbound neighbours with probably free capacity in periods of workload increase and on top of that direct access to migrated free resources in periods of workload decrease. This explains why k-Stalkers improvement is more noticeable during workload drop rather than increase. The longer this load drop is, servers stop generating queries and thus their neighbour lists gradually become outdated but more and more free resources appear in their pools.

5.7.2 k-Stalkers vs. FloodStalkers vs. FireStalkers

Figure 5.6 presents two important phenomena: *a)* k-Stalkers achieve the lowest and *b)* FloodStalkers the highest success rate among Stalkers. k-Stalkers forward queries choosing the freshest link practically ignoring recently updated outbound links; apart from the link chosen to receive a query there might be other links to servers that recently offered not finally migrated resources. On the contrary, FireStalkers trust such Neighbour Lists and once a link is too fresh it is assumed that more than one will be too; hence local one-hop multicasting. This trust is beneficial for FireStalkers as they achieve on average 5% (up to 20%) better success rate compared to k-Stalkers.

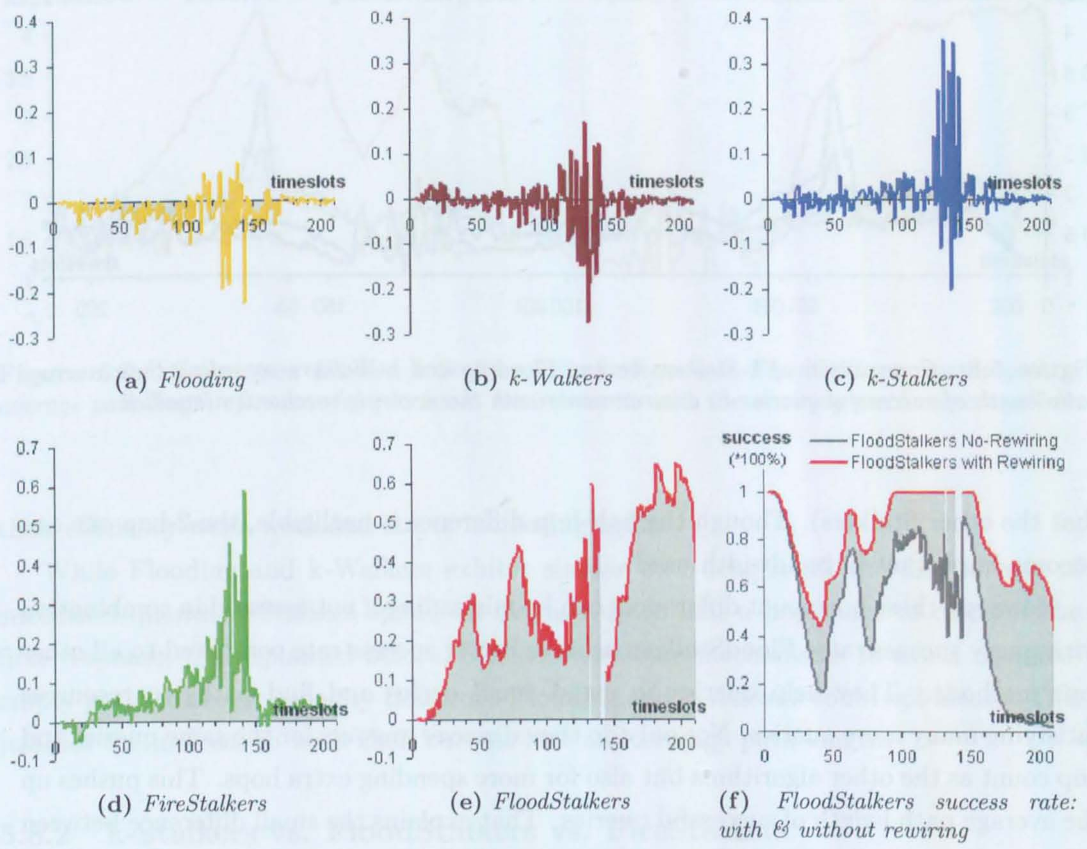
FloodStalkers performance (100% success rate) during roughly 60-70 and 90-155 timeslots is an indication that they efficiently use the available capacity as workload drops below system capacity levels. Outside those intervals, the success rate drops roughly as much as the global workload overpasses system capacity (i.e. success drops to 50% when workload increases up to 150% of system capacity). The red line appears to have two gaps on 132-133 and 140-144 timeslots denoting lack of queries. During 90-155 interval, servers send out repeated queries only. FloodStalkers achieving 100% success rate satisfy all of them once regenerated so that subsequent timeslots have no queries; hence the gaps of red line. This lack of queries is interrupted by a burst of repeated queries (red *'dash'* between those gaps).

5.7.3 Stalkers in fixed topology

After disabling the rewiring mechanism, the comparative results collected by reruning the experiments above appear in figures 5.7a through 5.7e. These plots do not illustrate the success rate of methods in the absence of rewiring but rather the subtraction of that rate from the one with rewiring presented in 5.4 and 5.6 figures. That is, their lines represent the distance between those two rates produced by the following operation $success_{m,r}(i) - success_{m,nr}(i)$ for each method m and timeslot i where r and nr stand for *rewiring* and *no rewiring* respectively. Figure 5.7f differs from the first 5 of that group in that it gives absolute values of FloodStalkers success rates used in 5.7e to calculate the operation above. In fact, its grey area is what actually that subtraction calculates.

All of these 5.7a through 5.7e plots exhibit a relatively wide fluctuation when global workload reaches its deepest *'valley'* as they rely on a small sample of queries per iteration. Most of 5.7a and, at a lesser extent, 5.7b lines are below x-axis; Flooding and k-Walkers are more successful if rewiring is disabled. They both use outgoing links only and as they discover resources rewiring mechanism forces them link to providers which, however, lose all their free resources as workload increases. Therefore, future queries travel on paths with fewer resources negatively affecting their success rate. Given that rewirings take place upon answer delivery, a drop in success rate reduces rewiring rate, too, trapping queries into paths with low success probability. Unlike k-Walkers, Flooding speeds up this phenomenon by quickly exhausting resources in those paths; hence, the distance of success rate of Flooding in rewiring-disabled from rewiring-enabled environments is more noticeable than that of k-Walkers.

Rewiring helps Stalkers, visiting incoming links, to escape paths without fresh capacity, trace migrated resources and discover fresh providers. In the absence of rewiring freshness plays no role in success rate as all links are created during network initialisation. Forwarding queries via incoming links help them avoid similar traps as those of Flooding and k-Walkers. However, they cannot detect resource migrations and therefore part of the available workload may be relocated away form their paths. This phenomenon is less noticeable in k-Stalkers as their success rate is the lowest among Stalkers. Therefore, they keep the resource migration rate low and resource distribution relatively intact.



(a) *Flooding* (b) *k-Walkers* (c) *k-Stalkers*

(d) *FireStalkers* (e) *FloodStalkers* (f) *FloodStalkers success rate: with & without rewiring*

Figure 5.7: *Comparison of search methods success rate in two environments: enabled and disabled rewiring. (a) through (e) represent the distance of each method's success rate with rewiring enabled from that with rewiring disabled*

FireStalkers and especially FloodStalkers use these mechanisms more efficiently increasing, in parallel, the distance between their success rates in those two environment modes (with versus without rewiring). Moreover, 5.7d and 5.7e figures illustrate a clear improvement for those two methods while global workload decreases. During that phase, tracing migrated nodes gets into action and facilitates resource discovery. In the absence of rewiring, queries keep exploring the same network part from which resources may have even moved away.

5.8 Average Path Length of Successful Queries

Figures 5.8 and 5.9 present a comparison of the average path lengths of successful queries for all simulated methods. While the former compares Flooding, k-Walkers and k-Stalkers the latter displays a comparison of the three Stalkers variations. k-Stalkers, being the common method for both figures, plays the role of reference line to ease comparison between lines of either figures. k-Stalkers alongside FireStalkers exhibit on average only half a hop bigger delay in resource discovery compared to Flooding and k-Walkers. FloodStalkers appear to experience longer path lengths (up to 2 hops more

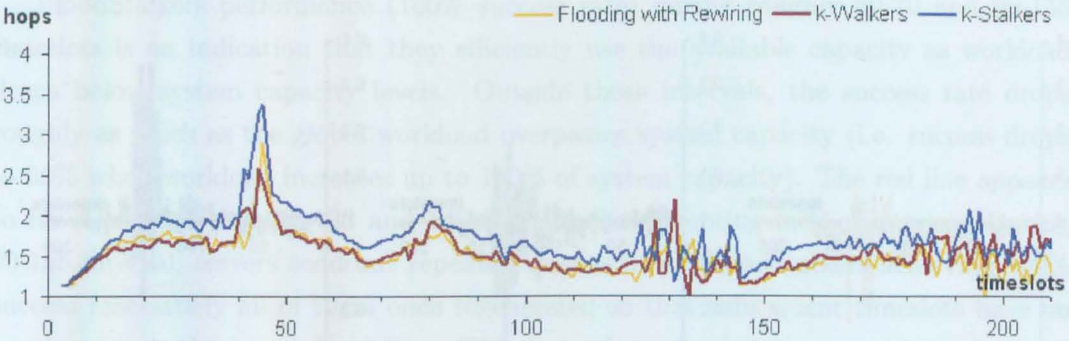


Figure 5.8: *Comparison of k-Stalkers versus Flooding and k-Walkers regarding their average path length of successful queries in environments with the rewiring mechanism enabled*

that the other Stalkers). Though the half-hop difference is negligible, the 2-hop one can become important in bandwidth used.

However, these hop count differences can be misleading if not assessed in combination with query success rate. FloodStalkers achieve better success rate compared to all other four methods. They help queries to avoid small cycles and find paths to resources satisfying many more queries. Not only do they discover answers for the same queries and hop count as the other algorithms but also for more spending extra hops. This pushes up the average path length of successful queries. That explains the small difference between Flooding/k-Walkers and k-Stalkers/FireStalkers. While Stalkers preserve their horizon wide enough to improve their success rate, Flooding and k-Walkers tend to shorten it and get locked to a small one with very few resources.

Most of the successful queries of all three methods in 5.8 discover their resources within one hop from the requesting server. Initially, as workload increases and local pools get exhausted, queries get satisfied by their immediate neighbouring ones. However, every server has the same probability to generate queries due to the uniform distribution of increasing global workload among underlying nodes and query repetitions. Therefore, they quickly spend resources in their close vicinity and queries from more distant servers do not manage to get there on time; thus, very few queries discover answers beyond their two-hop vicinity keeping the average path length low.

5.8.1 k-Stalkers vs. Flooding vs. k-Walkers

Unlike what one may expect, the average path length of successful queries decreases with global workload despite the increase of success rate. Workload drops are uniformly applied on underlying nodes and, hence, the probability a server satisfies queries from its underlying nodes with local resources is the same for all overlay servers. There are still responses from servers far away from local ones but their responses are either a minority compared to the total number of responses or even unnecessary as soon as they reach the requesting one. Thus, Flooding and k-Walkers using outgoing links only manage to increase their success probability as workload decreases via discovering mainly fresh local resources. This does not help servers adapt their Neighbour Lists and improve

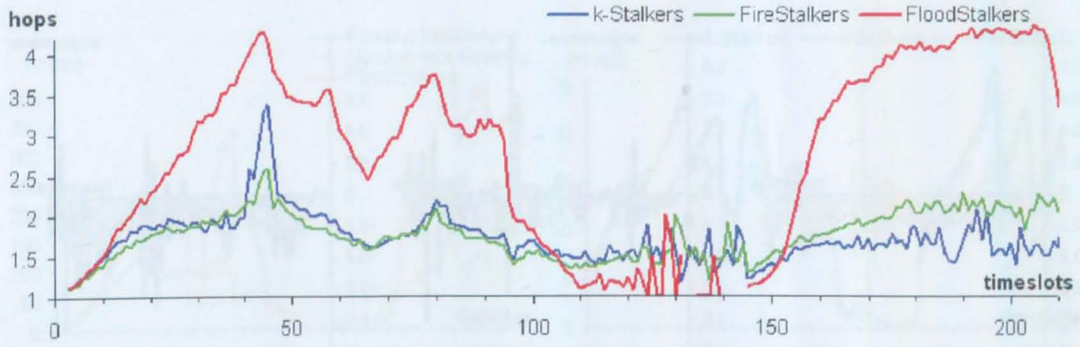


Figure 5.9: *Comparison between k-Stalkers, FireStalkers and FloodStalkers regarding their average path length of successful queries in environments with the rewiring mechanism enabled*

their efficiency when workload starts increasing again.

While Flooding and k-Walkers exhibit similar path length till the first answer of successful queries, k-Stalkers spend on average up to half a hop more to discover the first response. As explained before, Stalkers use some mechanisms to avoid resource-empty paths, discover capacity that both Flooding and k-Walkers could not locate. This justifies the increase in both their success rate and average path length.

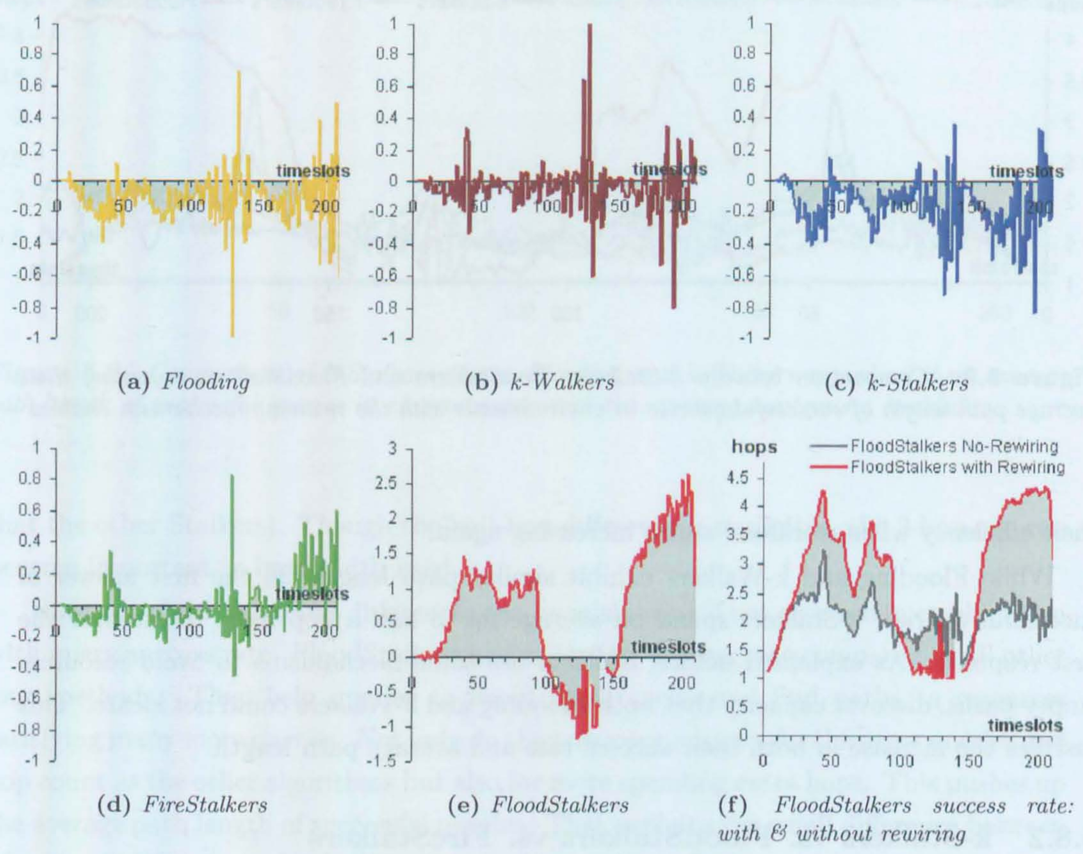
5.8.2 k-Stalkers vs. FloodStalkers vs. FireStalkers

FireStalkers is a special case as it achieves better success rate compared to k-Stalkers though their average path length is shorter. Their conditional one-hop multicasting stops query propagation assuming that queries have reached a quite promising server whose neighbours are good providers. FireStalkers appear to have bigger delays (0.5 hops) in discovering resources from timeslot 144 onwards compared to k-Stalkers. They retain better success rate throughout the experiment and thus their produce and repeat fewer queries seeking for smaller amount of capacity. Given that small resources are more frequent they manage to find resources deeper in the overlay.

All methods appear to have a peak in their average path lengths of successful queries on timeslots around 44 and 80. These are the points the additional workload per timeslot shifts from positive to negative values. For a few timeslots after those points, there are still many queries in the overlay but this workload shift places fresh capacity in pools both close to and far from requesting servers. This increases the success likelihood and the average path length. While workload keeps dropping, queries get satisfied closer and closer to their requestors as explained above; underlying nodes regenerate less queries and more and more free capacity appears in their close vicinity.

5.8.3 Stalkers in fixed topology

Experimenting with fixed server neighbour lists also produced results about the average path lengths of successful queries travelling without assistance from any rewiring mechanism. Comparing those results with the ones presented in 5.8 and 5.9, six more



(a) *Flooding* (b) *k-Walkers* (c) *k-Stalkers*

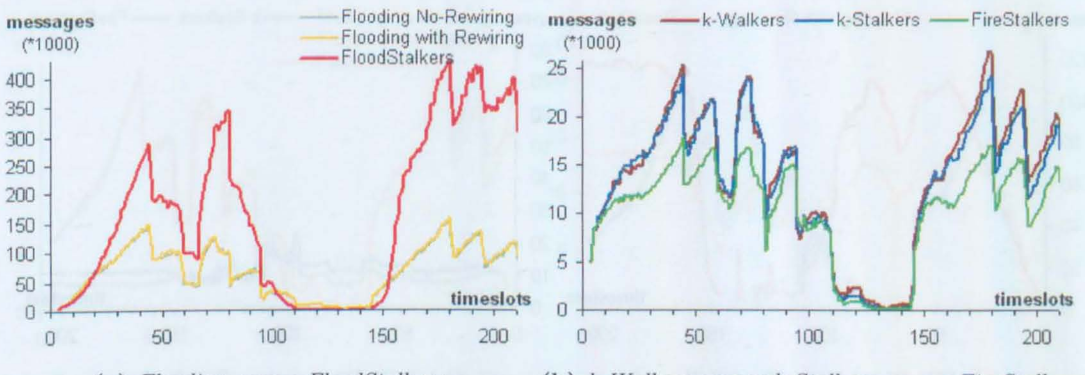
(d) *FireStalkers* (e) *FloodStalkers* (f) *FloodStalkers success rate: with & without rewiring*

Figure 5.10: *Comparison of search methods average path length in two environments: enabled and disabled rewiring. (a) through (e) represent the distance of each method's path lengths with rewiring enabled from that with rewiring disabled*

figures 5.10a through 5.10f are designed using the same principles as in 5.7a through 5.7e. These figures illustrate the difference between average path lengths of each method deployed on those two environments (active and inactive rewiring). As before, plot 5.10f presents the actual path lengths of FloodStalkers in both environments.

Flooding (5.10a) and k-Walkers (5.10b experience bigger delays if deployed to server overlay with fixed neighbour lists. Neighbour list rewirings, as explained before, in case of those two methods initially bring requesting servers closer to provider ones which however stop providing as soon as all their resources migrate; the lack of any mechanism to prevent queries getting trapped into such paths decreases their success and shortens the successful paths. When these rewirings are carefully used for the benefit of deployed methods, as is the case with Stalkers, the average length of those paths becomes even shorter. This explains why k-Stalkers reduce their success delay when rewiring mechanism is active more than k-Walkers do.

Deactivating rewiring, all links in servers' neighbour lists have the same timeslot tag. Therefore, Stalkers cannot pick the most recent neighbour as there is not such information; if selection between an incoming and outgoing link is necessary, the outgoing is used. This has no effect on Flooding and k-Walkers as by definition they use outgoing



(a) *Flooding versus FloodStalkers* (b) *k-Walkers versus k-Stalkers versus FireStalkers*

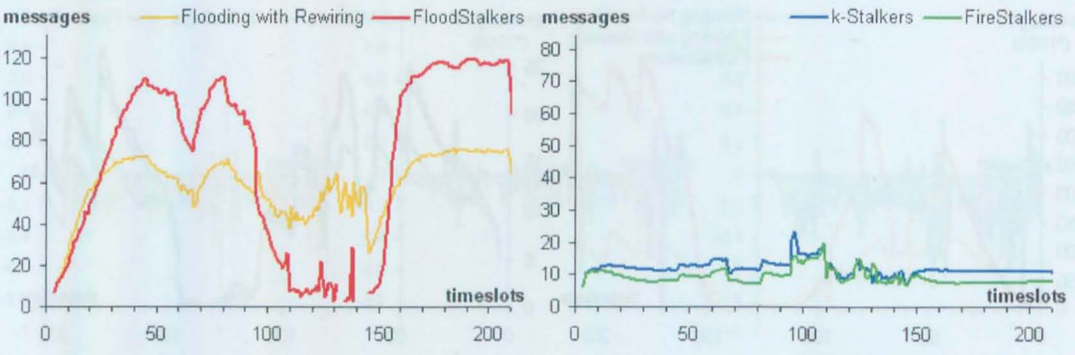
Figure 5.11: *Comparion between Stalkers, Flooding and k-Walkers regarding the number of messages they produce*

links only. Without any rewiring mechanism, the network does not adapt to resource migrations and swapping between inbound and outbound neighbours makes little difference in both success rate and delay (i.e. k-Stalkers and FireStalkers). However, both FireStalkers and FloodStalkers use multicastings exploring bigger part of their vicinity and when combined with rewirings they carefully select the source of these broadcasts increasing the success probability. Therefore, more queries get an answer from distant servers positively affecting both success rate and average path length.

5.9 Cost in Messages

Stalkers achieve significant benefits over Flooding and k-Walkers mainly related with success rate and come at a relatively low and sometimes negligible cost in the average response delay. Extending the evaluation of the proposed methods, figures 5.11a and 5.11b display the total cost in messages produced in each timeslot. A general observation is that Flooding and FloodStalkers spent at least one order of magnitude more messages than k-Stalkers, k-Walkers and FireStalkers. Among all, FireStalkers are the least expensive and FloodStalkers the most in terms of messages.

While Flooding-based queries visit every outgoing link in their vicinity, FloodStalkers-based ones visit a maximum of 21 outgoing links and 12 incoming ones. The former, avoiding incoming links, practically restricts its accessibility to outbound servers only and thus leaves big part of this horizon unexplored though the increase of messages is exponential with the Time-to-Live parameter. For the latter, messages increase linearly with the Time-to-Live but any server in the vicinity is accessible by queries. Moreover, FloodStalkers achieve better success rate and some messages are spent on answer and their positive or negative acknowledgements. However, FloodStalkers during some intervals is twice as expensive in messages as Flooding. The differences in the explorable horizon and extra messages on answers or acknowledgements are not enough to justify this cost of FloodStalkers.



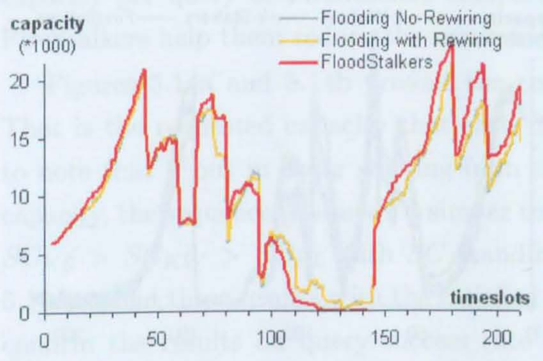
(a) *Flooding versus FloodStalkers* (b) *k-Stalkers versus FireStalkers*

Figure 5.12: *Comparison between Stalkers and Flooding regarding the average number of messages per query*

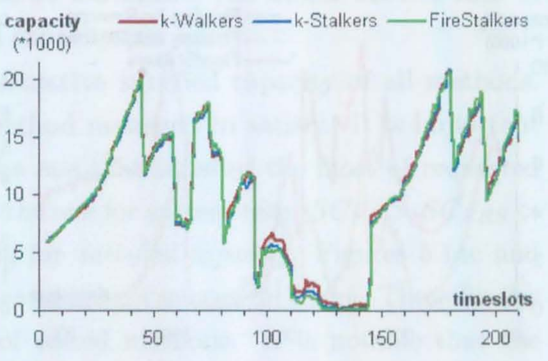
FloodStalkers produce many more queries than Flooding. Due to their better success rate more nodes join the requesting underlying networks. All these nodes while being reserved and waiting to migrate on-demand, experience their own workload fluctuations. Thus, by the time they arrive to requesting node they have a load which if added to the new one from their requestors may, or at least are close to, exceed their overload threshold. This increases the number of queries and messages traveling in the overlay. As expected, k-Walkers and k-Stalkers spend roughly the same number of messages though the former are slightly more expensive as they experience lower success rates and are forced to expire their Time-to-Live. FireStalkers, due to their conditional multicasting and especially during high workload situations with many rewirings, stop query propagation quite early achieving the lowest cost in terms of messages among all the other methods.

Figures 5.12a and 5.12b extend the evaluation of Stalkers illustrating the number of messages per query. The first figure confirms the explanation above that Flooding progresses in a restricted area of the overlay whereas FloodStalkers may access any server in originator's vicinity and have to spend some extra messages in answers and acknowledgements due to their improved success rate. Unlike Flooding, their plot shows a deep drop in number of messages per query spend during the 109-143 interval. This confirms the adaptability of Hoverlay and FloodStalkers as they achieve high success rate (100%) by spending very few messages; they first gather resources locally and then they address workload fluctuations with those local resources. Flooding, even though deployed on Hoverlay with rewiring mechanism enabled, does not gather enough resources close to requestors so that they quickly address workload fluctuations; they instead have to forward consecutive queries on the overlay.

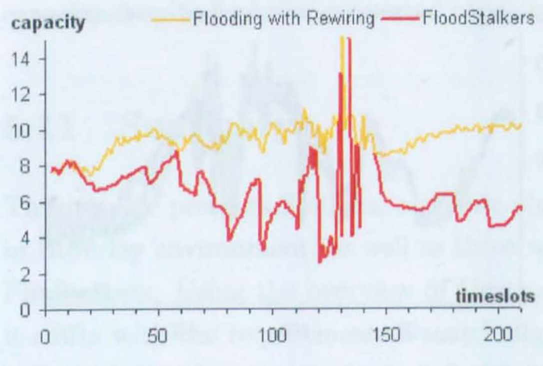
As shown in 5.12b figure, both k-Stalkers and FireStalkers produce about an order of magnitude less messages per query. This is explained by the nature of those algorithms as they spend a maximum of $k * TTL + 1$ queries, $k + 1$ answers (k from providers to originating server and one from that server to originating node) and an equal amount



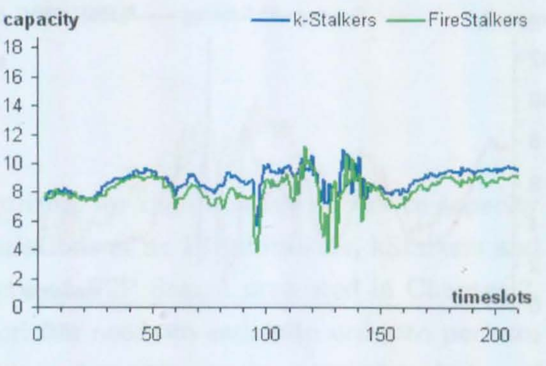
(a) *Total requested capacity of Flooding and FloodStalkers*



(b) *Total requested capacity of k-Walkers, k-Stalkers and FireStalkers*



(c) *Requested capacity per query of Flooding and FloodStalkers*



(d) *Requested capacity per query of k-Stalkers and FireStalkers*

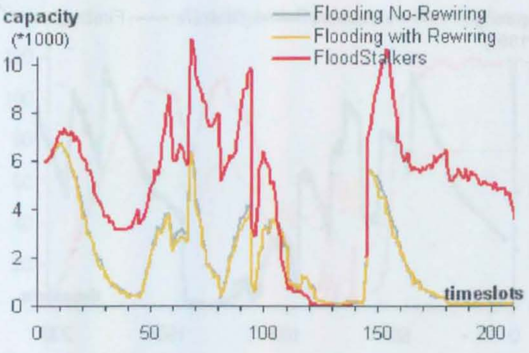
Figure 5.13: *Comparison between Stalkers, k-Walkers and Flooding regarding the requested capacity (total and per query)*

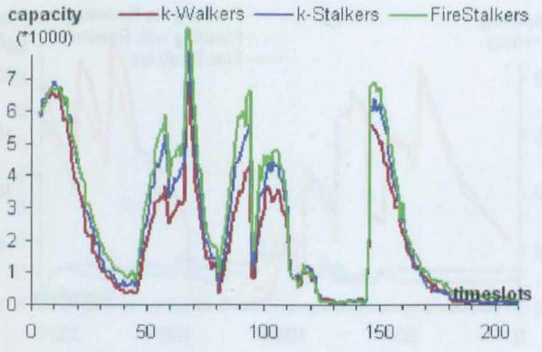
of acknowledgements; that is, $k\,(TTL + 2) + 3$ messages. FireStalkers on high load situations become even less expensive as rewirings are more frequent and more links are created within the time window f parameter of the method. This forces many queries to multicast and terminate within a few hops from their originator. While that multicasting spends k messages, it saves the network from the remaining TTL expiration. This result confirms that FireStalkers exhibit the shortest average path length of successful queries among all simulated methods.

However, in other network settings with much bigger Neighbour Lists and more walkers deployed by FireStalkers, these multicastings even if they take place quite early in query paths may produce much more messages that TTL. In these cases, FireStalkers would be more expensive than k-Stalkers but they would still retain their advantage in the query average path length.

5.10 Requested and Satisfied Capacity

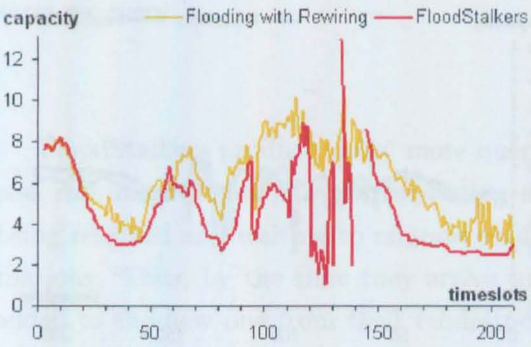
Figures 5.13a and 5.13b present the total requested capacity per timeslot for all methods. All these plots follow the global workload patterns topped up with the capacity requested

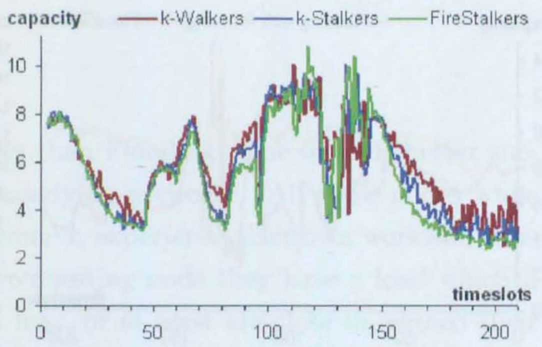




(a) *Total satisfied capacity of Flooding and FloodStalkers*

(b) *Total satisfied capacity of k-Walkers, k-Stalkers and FireStalkers*





(c) *Satified capacity per query of Flooding and FloodStalkers*

(d) *Satified capacity per query of k-Stalkers and FireStalkers*

Figure 5.14: *Comparison between Stalkers, k-Walkers and Flooding regarding the satisfied user queries (total and per query)*

by repeated queries; the periodic repetitions of past unsuccessful queries become clear with the step-like capacity drops. FloodStalkers appear to request slightly more capacity than Flooding; the latter produces fewer queries, requested capacity is distributed among them and re-requested whenever they are regenerated. FloodStalkers, especially while global workload drops, increases the number of queries via the mechanism explained before whereas Flooding would wait for geometrically increasing period to regenerate a failed query. This makes the difference between those two plots clearer towards the end of the experiment as waiting period of past queries has significantly increased.

Considering the requested capacity per query for Stalkers (plotted in 5.13c and 5.13d figures) useful conclusions can be drawn confirming the above. Due to the same mechanism that increases the number of messages and queries of FloodStalkers compared to Flooding, requested capacity per query of the former is more than that of the latter. Nodes joining a requesting underlying network take on part of the requested capacity and in addition to the load already handle may become overloaded. This forces them to generate a new query with, however, less requested capacity (the portion that exceed their overload threshold only). The higher the success rate the more dominant this phenomenon is. The same mechanism, to a lesser extent, affects the amount of requested capacity per query of FireStalkers compared to k-Stalkers; the better success rate of FireStalkers help them reduce the requested capacity per query.

Figures 5.14a and 5.14b present the cumulative satisfied capacity of all methods. That is the requested capacity that each method managed to satisfy. It is important to note that if put in order starting from the one that satisfied the most of requested capacity, the sequence produced is similar to the one for success rate: $SC_{FS} > SC_{FRS} > SC_{KS} > SC_{KW} > SC_{FL}$ with SC standing for *satisfied capacity.* Figures 5.14c and 5.14d extend these results with the satisfied capacity per successful query. These figures confirm the results on query success rate of tested methods. It is notable that the difference between the two lines of figure 5.14c is relatively much smaller than that in 5.14a. FloodStalkers manage to satisfy many more queries than Flooding and thus more capacity despite its lower requested capacity per query.

5.11 Summary

This chapter presents Stalkers, a generic algorithm for the discovery of service capacity in Hoverlay environment, as well as three variations of it: FloodStalkers, kStalkers and FireStalkers. Using the overview of Unstructured P2P Search presented in Chapter 2, it starts with the requirements a search algorithm needs to satisfy in order to perform well in such a dynamic environment. It then analyses the generic algorithm design of Stalkers and continues with the details of all three variations. The second part of the chapter is an analytical evaluation of those algorithms based on a benchmark consisting of Flooding and k-Walkers tested with the same experiment settings.

All Stalkers variations achieve better success rate compared to the benchmarks starting from 5% (k-Stalkers) and reaching even 60% improvement (FloodStalkers) with Fire-Stalkers outperforming k-Stalkers for an average 5%, too. These improvements come at a negligible extra cost in latency (on average half a hop longer path lengths of successful queries) for k-Stalkers and FireStalkers. FloodStalkers appear to introduce bigger delays (2 more hops than the other two Stalkers) but this is a rather superficial deterioration as this method manages to significantly improve its success rate by discovering capacity deeper in the network than the other can do. That is, they double their average path lengths of successful queries but achieve 6 times better success rate on the same time slots.

However, FloodStalkers are costly in terms of messages whereas FireStalkers the least expensive. The use of the freshest incoming links and rewiring help the former find paths that Flooding and k-Walkers cannot avoiding small cycles. This makes them spend more messages. On the other hand FireStalkers trust recently updated outbound neighbour lists and stop query forwarding on the first few hops with a final multicasting saving the network from many messages. Finally, FloodStalkers achieve the least requested and most satisfied capacity per timeslot and per query as a result of their good success rate.

To sum up, *a)* FloodStalkers outperform all other tested algorithms in success rate at a cost in messages fulfilling the purpose of their design, *b)* k-Stalkers spend fewer messages than k-Walkers but cannot outperform FireStalkers (in different environment settings with outbound degree higher than TTL k-Stalkers would be winners) and *c)* FireStalkers experience the shortest average path length of successful queries and thus they fulfil their design purpose, too.

While Flooding and k-Walkers perform better in terms of success rate compared to Stalkers deployed on environments without rewiring, Stalkers significantly improve their success likelihood in the presence of a rewiring mechanism due to their preference in fresh links. With regards to average path length of successful queries, rewiring helps all methods to discover resources faster. A superficial exception is FloodStalkers which, however, are the variation that makes extensive use of rewiring; lack of such a mechanism prevents query paths from tracing resource migrations that FloodStalkers would, otherwise, detect even far from query originators.

Chapter 6

Conclusions and Future Work

Network applications are dynamic distributed computing environments that frequently exhibit phenomena of underloading and overloading due to massive fluctuations in the number of user queries they produce and handle. These, sometimes abrupt, traffic variations may surpass the available capacity of their nodes and make a number of user queries be rejected. For this reason it is beneficial to provide mechanisms that allow overloaded networks to utilise free capacity of underloaded ones. In this way, systems can achieve significantly enhanced performance in terms of the amount of requested capacity they satisfy. Optimizing the node utilization and minimizing the traffic volume may be insufficient actions when excessive capacity grows well above total network capacity. Importing fresh extra capacity into an overloaded network could be another approach to the problem and is actually the main idea studied and evaluated in this document.

The current thesis discourses upon that problem and idea proposing Hoverlay as an implementation of that innovative approach. Its main conceptual innovation the logical on-demand migration of free nodes from one network to another in order to serve extra workload in terms of user queries. These "mobile" nodes may also autonomously disjoin the network they serve if they are underutilised and make themselves available for other overloaded ones. The resulting system is simple in design, deadlock-free avoiding to reserve partial answers while queries propagate through the overlay and fully supports heterogeneity as it makes no assumption regarding the topology and type of the underlying networks. Therefore, the system has good applicability in scenarios where instead of network capacity the underlying networks share processing power or even storage capacity.

Its architecture is based on Unstructured Peer-to-Peer overlays as a framework to facilitate resource (i.e. capacity) publishing, discovery and migration. Hoverlay's specification is a more detailed, generalised, enhanced and complete version of its predecessor same-purpose architecture, *G-ROME*, also presented in this document. That P2P overlay comprises with a set of arbitrarily interconnected servers on top of underlying networks. There is a one-to-one mapping between networks and servers which provide suitable services for free resource registration with their local pool and for resource provision upon request by either their own or remote via other servers underlying nodes.

Unlike Condor, resources move to request originator network rather than the opposite and once freed they register with the server of that network.

This architecture is independent from the deployed search mechanisms on its Peer-to-Peer overlay; thus, an appropriate search algorithm had to be devised to improve discovery efficiency complying with the special characteristics of service capacity. The proposed discovery mechanism is a combination of a rewiring and a query propagation strategy based on the ideas:

1. each requestor creates a link to all networks offered to provide capacity even though that was never used,
2. fresh requestors may have discovered resources which will be soon released due to workload fluctuations and
3. fresh requestors may have recently opened connections to other providing networks with free capacity based on the rewiring rule 1.

These rules shape a category of discovery mechanisms called *Stalkers* the general features of which are also detailed here. The current study also proposes three specializations of this category *k-Stalkers*, *FireStalkers* and *FloodStalkers* aiming at optimizing certain metrics. For instance, k-Stalkers reduce the number of messages required to discover resources at a cost in success rate, FireStalkers the average path length of successful queries and FloodStalkers increase the success rate of queries at a cost in messages.

Extensive simulations provided useful material for comparing Hoverlay with a competitive architecture, *Condor*. Their results have shown that the proposed architecture, under certain circumstances, performs better than the latter even with non optimized search techniques in terms of success rate, response latency and number of messages spent to discover the requested capacity. It manages to find the required spare capacity with higher probability and as a result, there is a significant increase in the number of additional successfully processed user queries that would otherwise have been dropped. Further experimentation on Stalkers and other search methods revealed the benefits of the former in environments like Hoverlay regarding the percentage of successful queries and their average path length till their first response is discovered.

6.1 Contributions & Findings

With the current dissertation, the author identifies a problem important for both industry and academia. The approach analysed and evaluated in it makes a 3-dimensional contribution defined by three axes:

- Study of the special features of reusable non-replicable resources that need to be considered if they are to be shared in a decentralised environment.
- A decentralised architecture that facilitates the publication, sharing and discovery of such resources.

- A generic search strategy with three variations for discovering those shared resources in the proposed architecture.

Despite the extensive research on replicable resources, the lack of guidelines on sharing reusable non-replicable resources on large scale heterogeneous self-configurable decentralised systems was a motivation of this study to deal with them. Thus, sharing such resources has to follow a set of rules as detailed below:

- Resource publishing: minimize the cost in messages for making available such resources and maintaining accurate information about their current condition. For instance, advertisements should be easy to update or the system able to tolerate with inconsistencies between resources and their adverts. Keeping accurate information for discovery purposes requires an expensive maintenance mechanism.

- Avoid providing guarantees for resource availability and make the system tolerant with load fluctuations caused by external factors. In decentralised systems, there is some time elapsed between the discovery and commission of a resource. During that time, both discovered and requesting resource may have changed status from underloaded to normally loaded or even overloaded and vice versa.

- Minimize the cost introduced by resource failures. Any system providing such resources should not assume soft resource departures. That is, no large-scale decentralised system dealing with reusable non-replicable resources may assume certain actions from a failing node before its departure.

Complying with these principles, Hoverlay is a main contribution of this dissertation. It is a system seated on top of different networks and enables their collaboration via exchanging resources. All resources at any given moment have a single publishing point (their local server) accessible with just one message. These servers provide upon request resources to other requesting servers which in turn do not reject responses that do not fulfil the initial request. On the contrary, they let their underlying requestor decide whether the response was satisfactory as both requestors and provided nodes may have changes status during all this migration process. Once relocated resources are freed they publish their availability on their last requesting server for future use from the local underlying network. Resources may depart at any time without any notification.

After a set of experiments and comparisons with an existing architecture, Condor, Hoverlay achieves better performance in terms of success rate and query response latency in high workload scenarios. These experiments show that:

1. The percentage of successful queries over the total number of queries increase with the decrease of system-wide workload and vice versa (finding for both systems).

2. Migrated nodes operate as equal members of requesting underlying network and reduce the overall requested per query capacity easing the success as low capacity resources are more frequent (finding for Hoverlay only).

3. Resource migration moves capacity away from the horizon of some servers. For instance, requesting servers just outside the borders of another's horizon consume resources that the latter may need in the future but will be unable to reach (finding for Hoverlay only).

4. In environments with hotspots, decreasing load in past hotspot areas frees capacity within the vicinity of requestors (finding for Hoverlay). If placed back to its owners (Flock of Condors), its rediscovery becomes more difficult (finding for Flock of Condors).

5. Hoverlay can increase dynamically the capacity of a hotspot area and thus address more user queries. (finding for Hoverlay).

6. Placing free resources back to their originators can be beneficial once hotspots change location moving closer to those originators (finding for Flock of Condors).

7. If free resources re-register with their last requesting server, local underlying network can address much quicklier their fluctuations (finding for Hoverlay). Resource migration in case of Hoverlay reduces the average path length of successful queries

Summarizing the findings above, Hoverlay achieves better success rate in high workload situations as migrated nodes operate as equal members of their new hosting networks whereas Flock of Condors perform better in low workload environments. The more the workload decreases the rarer the resource relocations and, hence, the slower the resource redistribution. Therefore, when just very few nodes generate queries and their vicinity is resource-free in Hoverlay they have low probability to discover capacity. However, returning resources back to their originators increases the probability that some of them go back within those few requestors' horizon. Furthermore, Hoverlay manages to keep the average path length of successful queries lower than Flock of Condors throughout the experiments, thus, fulfilling the requirement for low latency. These advantages of Hoverlay come almost at no extra cost in messages compared to messages produced by Flock of Condors.

Continuing with the proposed search algorithms 'Stalkers', they use two principal ideas: *a)* fresh requestors have recently discovered resources that may soon release and a more accurate and up-to-date snapshot of their vicinity and *b)* recent providers may still have some free resources. This thesis proposes three variations exploiting these ideas each of which targets different metric: *k-Stalkers* for reducing number of messages in low bandwidth networks, *FireStalkers* to improve latency keeping their cost in messages relatively low and *FloodStalkers* to optimize success rate at the expense of messages.

Their evaluation concluded that they all outperform Flooding and k-Walkers on success rate at an insignificant increase in their average path length of successful queries and in messages spent. In brief, k-Stalkers performance depends on the Neighbour List size; they achieve relatively good success rate and guaranteed maximum number of messages. FloodStalkers give the best success likelihood but spend a lot of messages and

		Connectivity (Neighbour List size)	
		Low	*High*
Bandwidth	*Low*	FireStalkers	k-Stalkers
	High	FloodStalkers	FireStalkers

Table 6.1: *Stalkers applicability in different environments*

FireStalkers are more successful and faster than k-Stalkers but can be a more expensive choice if deployed on highly-connected overlays.

Table 6.1 gives an overview of scenarios to which each of those mechanisms is suitable to be deployed. The parameters of these scenarios are the connectivity of the overlay and the bandwidth available. FireStalkers can achieve good success rate without overloading the overlay if it is deployed in low-connectivity and low-bandwidth or high-connectivity and high-bandwidth environments. High-connectivity networks with low bandwidth can afford k-Stalkers due to their guaranteed limited number of messages and relatively good success rate. Finally, FloodStalkers can perform well in high-bandwidth environments but with low connectivity in order to avoid network overloading.

Further experimentation on more static environments revealed that all these three variations perform better in the presence of rewiring mechanisms as they can make full use of the freshness of links. In static networks all links have the same lifetime and 'Stalkers' make no informed selections of neighbours. However, Flooding and k-Walkers do not manage to outperform them in success rate as Stalkers are based on incoming links, too, and still exhibit some traceability of resource migrations.

6.2 Future Work

The detailed evaluation of Hoverlay and Stalkers identified the advantages and shortcomings of these mechanisms. Furthermore, some of their aspects, being out of scope for this work, are not yet sufficiently researched. This section tries to provide an overview of possible further work to improve and study the proposed architecture and algorithms. Research on them can be extended but not limited to proactive placement of resources in carefully selected areas of the network, load-balancing of servers overlay, security related issues and applicability scenarios.

Hoverlay relies on the concept of reactive resource migration from a providing network to a requesting one in order to server an overloaded underlying node. A requesting node needs to wait till a response is discovered and nodes migrated to its own network. Moreover, based on the findings above, resource migration may negatively affect system's success rate in case of low system-wide workload levels. Therefore, a symmetrical facet of the proposed methodology is the proactive resource relocation. The main criteria of this action are the provider availability levels and requesting networks' load patterns. Providers may outsource some of their resources if they foresee they can meet their own and others requirements in near future. They may also choose to displace portion of their free capacity to avoid flash crowds as soon as they get discovered by many requestors.

Proactivity in Hoverlay can be expressed in two ways: *a)* resource placement into another server and *b)* rewiring without prior request. In the former case, servers with plenty of resources, unnecessary for the forseeable future, may choose to spread them on servers more likely to request for extra capacity or to those that are likely to receive many queries in the near future. However, the latter case is a proactive network reorganisation strategy based on which a server tries to attract more incoming links to avoid spreading resources without any definite need to do so. They are two contradictory strategies from traffic handling perspective: selfish and selfless, respectively. *Proactive placement* is a way to avoid attracting traffic or else move traffic away to other servers. A server may use *proactive rewiring* to re-route part of other servers' traffic to itself thus relieving them from routing responsibilities.

As detailed in Chapter 2, power-law networks are efficient in discovery but Hoverlay cannot create long-lasting hubs as service capacity migrates among servers. Proactive placement could be used as a way to extent the lifetime of those hubs and improve search performance of deployed mechanisms. However, it requires a small protocol for the communication between those two parties thus introducing more messages. Single-hop placement of a resource seems to be a better approach compared to an exhaustive or even expensive search of the best-fit server. That is, a server may only place a fraction of its resources to direct neighbours. Subsequent hops may reduce resource availability as part of their lifetime will be spent in continuous migrations. The node representing that resource may also suffer from isolation as it will practically be offline during its migration; addressing possible fluctuations of its workload will be delayed till its final hop is complete. Finally, in scenarios with low global workload multi-hops in the deployed proactive placement mechanism might cause resources in whole network purposelessly moving around.

On the other hand, proactive rewiring does not require any extra communication protocol as no resource exchange takes place. Given Stalkers mechanisms which distinguish and use both incoming and outgoing links, proactively opening connections to other servers pretending request delivery can be a way of forced rewiring. This method does not introduce extra messages but create cycles as new outbound servers might link back to the server which initiated those proactive rewirings once they request and fetch some of its resources. Ongoing research tests variations of those mechanisms and tries to identify the scenarios proactive placement and rewirings can be useful.

In an attempt to formalise Hoverlay functionality, Karkinsky et al. in [68] presented a Resource Allocation System (RAS) inspired by the current Hoverlay. The authors captured and modelled interesting features of an oversimplified version of Hoverlay using π-calculus and B-method. They identified three entities: servers, clients and resources. They are terms with slightly different semantics: servers have the same role as the specification above, clients represent underlying requesting nodes (overloaded resources) and resources are free nodes migrating between servers. Though they have an interesting approach, they focus on rather obvious properties and rely on assumptions that make

RAS an unrealistic system. Formal modeling can provide useful tools for assessing decentralised systems but they are not mature enough, yet, to achieve the accuracy of simulations and prototypes. Further work on could, however, be complementary to existing evaluation practices for distributed systems.

Each node of the underlying network depends on a single server to discover extra available capacity. In order to relax this limitation, the underlying nodes should be able to send queries to more than one server. This would increase the heterogeneity and fault-tolerance of the system. A simple approach would be a protocol which helps each node to populate and update a list of candidate servers. Each node uses only one server to register with if available but more than one to make requests. The way this list gets updated and which server(s) are chosen to register with or make queries to are some of the issues that need to be addressed.

Though Hoverlay focuses on resource discovery issues, it assumes a deployed mechanism on every underlying node to enable its migration and joining with the requesting network. In a heterogeneous environment, each underlying network may support different distributed applications and have different joining requirements. Therefore, each requesting node may need to submit all the appropriate data and task to migrated nodes to take on portion of its load; this requires a common job submission toolkit. Condor can provide such tools making both systems (Condor and Hoverlay) complementary rather than competitive.

Both this job submission process on relocated nodes and the actual node migration raise a number of security concerns related to access rights control of that node and other networked resources, their data integrity, possible coordinated attacks e.t.c. Problems may be caused during migration for both providing and requesting networks. For instance, if the deployed distributed application of the provider was an archiving service moving out a node maintaining a portion of that distributed archive may cause loss of information and break data integrity. Depending the archived data recovery mechanism used in that network, that loss of information may invalidate a bigger portion of date archived in that network. Furthermore, if multiple malicious nodes join a single network they may be able to compromise its trust and security mechanisms.

Malicious behaviour may appear in servers as well. Suicidal servers may fake high workload situations and gather big portion of system-wide available resources and then disconnect practically removing that capacity from the network. Symmetrically, a server may be a good provider for long thus becoming a strong hub of the overlay which, however, if failed may even fragment the topology. Another case of misbehaved servers could be the provision of already well loaded nodes and therefore force these nodes generate many queries overloading the whole overlay; this is a way to bypass the exponential waiting period of unsatisfied queries as explained Chapter 5. These can be only a handfull of security related problems that need to be addressed before Hoverlay is applied to a business environment.

Bibliography

[1] News video quality survey. Technical report, Streamcheck Research.

[2] News sites toil as visits rocket. *BBC*, July 2005.

[3] Gnutella. http://wiki.limewire.org/index.php?title=GDF, July 2008.

[4] Gnutella2. http://g2.trillinux.org/index.php?title=Main_Page, 2009.

[5] Karl Aberer. *P-Grid: A Self-Organizing Access Structure for P2P Information Systems*, volume 2172/2001 of *Lecture Notes in Computer Science*, pages 179–194. Springer Berlin / Heidelberg, 2001.

[6] Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani, and Bernardo A. Huberman. Search in power-law networks. *Physical Review E*, 64(4):046135, 2001. Copyright (C) 2008 The American Physical Society; Please report any problems to prola@aps.org.

[7] R. Akavipat, L-S. Wu, and F. Menczer. Small world peer networks in distributed web search. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers \&amp; posters*, pages 396–397, New York, NY, USA, 2004. ACM.

[8] R. Albert, H. Jeong, and A. L. Barabasi. Diameter of the World-Wide web. *Nature*, 401(6749):130–131, 1999.

[9] Reka Albert and Albert-Laszlo Barabasi. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47, 2002. Copyright (C) 2008 The American Physical Society; Please report any problems to prola@aps.org.

[10] Reka Albert, Hawoong Jeong, and Albert-Laszlo Barabasi. Error and attack tolerance of complex networks. *Nature*, 406(6794):378–382, July 2000.

[11] Andre Allavena, Alan Demers, and John E. Hopcroft. Correctness of a gossip based membership protocol. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 292–301, Las Vegas, NV, USA, 2005. ACM.

[12] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Communmunications of the ACM*, 45(11):56–61, 2002.

[13] N. Andrade, F. Brasileiro, W. Cirne, and M. Mowbray. Discouraging free riding in a peer-to-peer CPU-sharing grid. In *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 129–137, June 2004.

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

[15] Asad Awan, Ronaldo A. Ferreira, Suresh Jagannathan, and Ananth Grama. Unstructured peer-to-peer networks for sharing processor cycles. *Parallel Computing*, 32(2):115–135, February 2006.

[16] R. Baldoni, S. Bonomi, A. Rippa, L. Querzoni, S.T. Piergiovanni, and A. Virgillito. Evaluation of unstructured overlay maintenance protocols under churn. In *Distributed Computing Systems Workshops, 2006. ICDCS Workshops 2006. 26th IEEE International Conference on*, page 13, Rome, Italy, July 2006. ACM.

[17] Roberto Baldoni, Adnan Mian, Sirio Scipioni, and Sara Tucci-Piergiovanni. *Churn Resilience of Peer-to-Peer Group Membership: A Performance Analysis*, volume 3741/2005 of *Lecture Notes in Computer Science*, pages 226–237. Springer Berlin / Heidelberg, 2005.

[18] S. A. Baset and H. G. Schulzrinne. An analysis of the skype Peer-to-Peer internet telephony protocol. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–11, April 2006.

[19] B. Beckles. Current methods for negotiating firewalls for the condor system. In *Proceedings of the 4th UK e-Science All Hands Meeting 2005*, Nottingham, UK, September 2005.

[20] Tania Bedrax-weiss, Conor Mcgann, and Sailesh Ramakrishnan. Formalizing resources for planning. *In Proceedings of the ICAPS-03 Workshop on PDDL*, pages 7–14, 2003.

[21] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4):353–366, 2004.

[22] S. Bharathi and A. Chervenak. Design of a Peer-to-Peer information system using the GT4 index service. In *Grid Computing, 7th IEEE/ACM International Conference on*, pages 321–322, Barcelona, Spain, September 2006.

[23] Sven Bilke and Carsten Peterson. Topological properties of citation and metabolic networks. *Physical Review E*, 64(3):36106, September 2001.

[24] R. Blanco, N. Ahmed, D. Hadaller, L. G. A. Sung, H. Li, and M. A. Soliman. A survey of data management in peer-to-peer systems. Technical Report CS-2006-18, University of Waterloo, Waterloo, Canada, June 2006.

[25] Bela Bollobas and Oliver Riordan. The diameter of a Scale-Free random graph. *Combinatorica*, 24(1):5–34, 2004.

[26] Francisco Brasileiro, Eliane Araujo, William Voorsluys, Milena Oliveira, and Flavio Figueiredo. Bridging the high performance computing gap: the OurGrid experience. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 817–822, Rio de Janeiro, Brazil, May 2007.

[27] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309–320, June 2000.

[28] Javier Bustos-Jimenez, Denis Caromel, and Jose Piquer. *Load Balancing: Toward the Infinite Network and Beyond*, volume 4376/2007 of *Lecture Notes in Computer Science*, pages 176–191. Springer Berlin / Heidelberg, 2007.

[29] Ali R. Butt, Rongmei Zhang, and Y. Charlie Hu. A self-organizing flock of condors. *J. Parallel Distrib. Comput.*, 66(1):145–161, 2006.

[30] Duncan S. Callaway, M. E. J. Newman, Steven H. Strogatz, and Duncan J. Watts. Network robustness and fragility: Percolation on random graphs. *Physical Review Letters*, 85(25):5468, December 2000. Copyright (C) 2008 The American Physical Society; Please report any problems to prola@aps.org.

[31] L. Caviglione and F. Davoli. Peer-to-peer middleware for bandwidth allocation in sensor networks. *Communications Letters, IEEE*, 9(3):285–287, 2005.

[32] Javier Celaya and Unai Arronategui. *Scalable Architecture for Allocation of Idle CPUs in a P2P Network*, volume 4208/2006 of *Lecture Notes in Computer Science*, pages 240–249. Springer Berlin / Heidelberg, 2006.

[33] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. *Freenet: A Distributed Anonymous Information Storage and Retrieval System*, volume 2009/2001 of *Lecture Notes in Computer Science*, pages 46–66. Springer Berlin / Heidelberg, Berlin, Germany, 2001.

[34] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, pages 25–30, Paris, France, 2004. ACM.

[35] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pages 181–194, San Francisco, CA, USA, 2001.

[36] S. Daswani and A. Fisk. Gnutella UDP extension for scalable searches (GUESS) v0. 1. In *Gnutella Development Forum, Tech. Rep.* LimeWire LLC, August 2002.

[37] R. Diestel and R. Diestel. *Graph theory.* Springer New York, New York, USA, 2nd edition, 2000.

[38] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: load sharing among workstation clusters. *Future Gener. Comput. Syst.*, 12(1):53–65, 1996.

[39] X. Evers, J.F.C.M. de Jongh, R. Boontje, J. Epema, and R. Van Dantzig. Condor flocking: load sharing between pools of workstations. Technical Report 93-104, Technische Universiteit Delft, 1993.

[40] Georgios Exarchakos and Nick Antonopoulos. Resource sharing architecture for cooperative heterogeneous P2P overlays. *Journal of Network and Systems Management*, 15(3):311–334, 2007.

[41] Georgios Exarchakos and Nick Antonopoulos. Non-replicable reusable resources discovery on scale-free Peer-to-Peer networks. In *Digital Ecosystems and Technologies, 2008. DEST 2008. 2nd IEEE International Conference on*, pages 28–33, Phitsanulok, Thailand, February 2008.

[42] Georgios Exarchakos, Nick Antonopoulos, and James Salter. G-ROME: semantic-driven capacity sharing among P2P networks. *Internet Research*, 17(1):7 – 20, 2007.

[43] Georgios Exarchakos, Nick Antonopoulos, and Kan Zhang. Firewalks: Discovery mechanism for non-replicable reusable resources. In *Proceedings of the Seventh International Network Conference (INC 2008)*, page 65, Plymouth, UK, 2008. Lulu. com.

[44] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *Computer Communications Review*, 29(4):251–262, 1999.

[45] M. Feldman, C. Papadimitriou, J. Chuang, and I. Stoica. Free-riding and whitewashing in peer-to-peer systems. *Selected Areas in Communications, IEEE Journal on*, 24(5):1010–1019, May 2006.

[46] George Fletcher, Hardik Sheth, and Katy BG'Arner. *Unstructured Peer-to-Peer Networks: Topological Properties and Search Performance*, pages 14–27. 2005.

[47] Ian Foster and Carl Kesselman. Globus: a metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, 11(2):115–128, June 1997.

[48] Ian Foster and Carl Kesselman. *The Grid in a nutshell*, pages 3–13. Kluwer Academic Publishers, 2004.

[49] Pierre Fraigniaud, Philippe Gauron, and Matthieu Latapy. *Combining the Use of Clustering and Scale-Free Nature of User Exchanges into a Simple and Efficient P2P System*, volume 3648/2005 of *Lecture Notes in Computer Science*, pages 1163–1172. Springer Berlin / Heidelberg, Berlin, Germany, 2005.

[50] A. Ganguly, A. Agrawal, P. Boykin, and R. Figueiredo. WOW: self-organizing wide area overlay networks of virtual workstations. *Journal of Grid Computing*, 5(2):151–172, June 2007.

[51] W. Gentzsch. Sun grid engine: towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36, Wasington, USA, 2001.

[52] Thomer M. Gil, Frans Kaashoek, Jinyang Li, Robert Morris, and Jeremy Stribling. p2psim: a simulator for peer-to-peer (p2p) protocols. http://pdos.csail.mit.edu/p2psim/.

[53] C. Gkantsidis, M. Mihail, and A. Saberi. Hybrid search schemes for unstructured peer-to-peer networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 1526–1537, March 2005.

[54] C. Gkantsidis, J. Miller, and P. Rodriguez. Anatomy of a p2p content distribution system with network coding. *5th Int. Work. on P2P System (IPTPS)*, February 2006.

[55] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random walks in peer-to-peer networks: Algorithms and evaluation. *Performance Evaluation*, 63(3):241–263, March 2006.

[56] Andrew S. Grimshaw, Wm. A. Wulf, and CORPORATE The Legion Team. The legion vision of a worldwide virtual computer. *Communicatios of the ACM*, 40(1):39–45, 1997.

[57] Hasan Guclu and Murat Yuksel. Scale-Free overlay topologies with hard cutoffs for unstructured Peer-to-Peer networks. In *Distributed Computing Systems, 2007. ICDCS '07. 27th International Conference on*, page 32, 2007.

[58] Rohit Gupta, V. Sekhri, and A.K. Somani. CompuP2P: an architecture for internet computing using Peer-to-Peer networks. *Parallel and Distributed Systems, IEEE Transactions on*, 17(11):1306–1320, November 2006.

[59] Richard C. Holt. Some deadlock properties of computer systems. *ACM Comput. Surv.*, 4(3):179–196, 1972.

[60] J. Hu and R. Klefstad. Decentralized load balancing on unstructured Peer-2-Peer computing grids. In *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*, pages 247–250, Cambridge, MA, 2006.

[61] Ken Y. K. Hui, John C. S. Lui, and David K. Y. Yau. Small-world overlay P2P networks: construction, management and handling of dynamic flash crowds. *Computer Networks*, 50(15):2727–2746, October 2006.

[62] A. Iamnitchi, I. Foster, and D.C. Nurmi. A peer-to-peer approach to resource location in grid environments. In *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, page 419, 2002.

[63] H.V. Jagadish, Beng Chin Ooi, Quang Hieu Vu, Rong Zhang, and Aoying Zhou. VBI-Tree: a Peer-to-Peer framework for supporting Multi-Dimensional indexing schemes. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, page 34, April 2006.

[64] Mark Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. PeerSim P2P simulator. http://peersim.sourceforge.net/.

[65] Vana Kalogeraki, Dimitrios Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 300–307, McLean, Virginia, USA, November 2002. ACM.

[66] Nirav Kapadia and Jose Fortes. PUNCH: an architecture for web-enabled wide-area network-computing. *Cluster Computing*, 2(2):153–164, September 1999.

[67] Pradnya Karbhari, Mostafa Ammar, Amogh Dhamdhere, Himanshu Raj, George F. Riley, and Ellen Zegura. *Bootstrapping in Gnutella: A Measurement Study*, volume 3015/2004 of *Lecture Notes in Computer Science*, pages 22–32. Springer Berlin / Heidelberg, 2004.

[68] Damien Karkinsky, Steve Schneider, and Helen Treharne. *Combining Mobility with State*, volume 4591/2007 of *Lecture Notes in Computer Science*, pages 373–392. Springer Berlin / Heidelberg, Heidelberg, Germany, 2007.

[69] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164, 2002.

[70] Stefan M Larson, Christopher D Snow, Michael Shirts, and Vijay S Pande. *Folding@ Home and Genome@ Home: Using distributed computing to tackle previously intractable problems in computational biology.* Horizon Press, 2002.

[71] Jian Liang, Rakesh Kumar, and Keith W. Ross. The FastTrack overlay: A measurement study. *Computer Networks*, 50(6):842–858, April 2006.

[72] Fredrik Liljeros, Christofer R. Edling, Luis A. Nunes Amaral, H. Eugene Stanley, and Yvonne Aberg. The web of human sexual contacts. *Nature*, 411(6840):907–908, June 2001.

[73] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, San Jose, CA, USA, 1988.

[74] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, 7(2):72–93, 2005.

[75] 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, 2002. ACM.

[76] Carlo Mastroianni, Domenico Talia, and Oreste Verta. A super-peer model for resource discovery services in large-scale grids. *Future Generation Computer Systems*, 21(8):1235–1248, October 2005.

[77] Milena Mihail, Amin Saberi, and Prasad Tetali. Random walks with lookahead on power law random graphs. *Internet Mathematics*, 3(2):147–152, 2006.

[78] Kiran Nagaraja, S. Rollins, and M. Khambatti. From the editors: peer-to-peer community: looking beyond the legacy of napster and gnutella. *Distributed Systems Online, IEEE*, 7(3), March 2006.

[79] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[80] Nikos Ntarmos, Theoni Pitoura, and Peter Triantafillou. *Range Query Optimization Leveraging Peer Heterogeneity in DHT Data Networks*, volume 4125/2007 of *Lecture Notes in Computer Science*, pages 111–122. Springer Berlin / Heidelberg, 2007.

[81] Johan Pouwelse, Pawe Garbacki, Dick Epema, and Henk Sips. *The Bittorrent P2P File-Sharing System: Measurements and Analysis*, volume 3640/2005 of *Lecture Notes in Computer Science*, pages 205–216. Springer Berlin / Heidelberg, 2005.

[82] R. Raman, M. Livny, and M. Solomon. Matchmaking: distributed resource management for high throughput computing. pages 140–146, Jul 1998.

[83] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. *SIGCOMM Compututer Communication Review*, 31(4):161–172, October 2001.

[84] S. Redner. How popular is your paper? an empirical study of the citation distribution. *The European Physical Journal B - Condensed Matter and Complex Systems*, 4(2):131–134, August 1998.

[85] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 99–100, Linkoping, Sweden, August 2001.

[86] Matei Ripeanu and Ian Foster. *Mapping the Gnutella Network: Macroscopic Properties of Large-Scale Peer-to-Peer Systems*, volume 2429/-1/2002 of *Lecture Notes in Computer Science*, pages 85–93. Springer Berlin / Heidelberg, Berlin, Germany, 2002.

[87] Antony Rowstron and Peter Druschel. *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*, volume 2218/2001 of *Lecture Notes in Computer Science*, pages 329–350. Springer Berlin / Heidelberg, 2001.

[88] J. Salter, N. Antonopoulos, and R. Peel. ROME: optimising lookup and load-balancing in DHT-based P2P networks. In *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 05)*, pages 27–30, Las Vegas, NV, USA, June 2005.

[89] K. Samant and S. Bhattacharyya. Topology, search, and fault tolerance in unstructured P2P networks. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, page 6 pp., 2004.

[90] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Systems*, 9(2):170–184, August 2003.

[91] G. Scherp, W. Hasselbring, and J. Ploski. The WISENT grid architecture: Coping with firewalls and NAT. In *German e-Science Conference*, 2007.

[92] R. Schollmeier. Why peer-to-peer (P2P) does scale: an analysis of P2P traffic patterns. In *Peer-to-Peer Computing, 2002. (P2P 2002). Proceedings. Second International Conference on*, pages 112–119, Linkoping, Sweden, September 2002. IEEE.

[93] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. *IEEE/ACM Transactions on Networking (TON)*, 12(2):219–232, April 2004.

[94] Stephen F Smith and Marcel A Becker. An ontology for constructing scheduling systems. *Working Notes of 1997 AAAI Spring Symposium on Ontological Engineering*, pages 120–129, March 1997.

[95] Sechang Son, B. Allcock, and M. Livny. CODO: firewall traversal by cooperative on-demand opening. In *High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium on*, pages 233–242, July 2005.

[96] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Networking, IEEE/ACM Transactions on*, 11(1):17–32, February 2003.

[97] D. Stutzbach, R. Rejaie, and S. Sen. Characterizing unstructured overlay topologies in modern P2P File-Sharing systems. *Networking, IEEE/ACM Transactions on*, 16(2):267–280, April 2008.

[98] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202, Rio de Janeriro, Brazil, October 2006. ACM.

[99] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience: Research articles. *Concurrency and Computation: Practice & Expererience*, 17(2-4):323–356, February 2005.

[100] D. Tsoumakos and N. Roussopoulos. Adaptive probabilistic search for peer-to-peer networks. In *Peer-to-Peer Computing, 2003. (P2P 2003). Proceedings. Third International Conference on*, pages 102–109, Linkoping, Sweden, September 2003.

[101] Dimitrios Tsoumakos and Nick Roussopoulos. Analysis and comparison of P2P search methods. In *Proceedings of the 1st international conference on Scalable information systems*, volume 152 of *ACM International Conference Proceeding Series*, page 25, Hong Kong, 2006. ACM.

[102] Koen Vanthournout, Geert Deconinck, and Ronnie Belmans. A taxonomy for resource discovery. *Personal and Ubiquitous Computing*, 9(2):81–89, March 2005.

[103] Jean Vaucher, Gilbert Babin, Peter Kropf, and Thierry Jouve. *Experimenting with Gnutella Communities*, volume 2468/2002 of *Lecture Notes in Computer Science*, pages 255–257. Springer Berlin / Heidelberg, Heidelberg, Germany, 2002.

[104] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of /'small-world/' networks. *Nature*, 393(6684):440–442, June 1998.

[105] D.I. Wolinsky, A. Agrawal, P.O. Boykin, J.R. Davis, A. Ganguly, V. Paramygin, Y.P. Sheng, and R.J. Figueiredo. On the design of virtual machine sandboxes for distributed computing in wide-area overlays of virtual workstations. In *Virtualization Technology in Distributed Computing, 2006. VTDC 2006. First International Workshop on*, page 8, Tampa, Florida, USA, November 2006.

[106] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 5–14, Vienna, Austria, July 2002. IEEE.

[107] Xiangying Yang and Gustavo de Veciana. Performance of peer-to-peer networks: Service capacity and role of resource sharing policies. *Performance Evaluation*, 63(3):175–194, March 2006.

[108] Haiyang Zhang and Huadong Ma. An efficient hierarchical DHT-Based complex query for multimedia information. In *Multimedia and Expo, 2007 IEEE International Conference on*, pages 568–571, 2007.

[109] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41–53, 2004.

[110] D. Zhou and V. Lo. Cluster computing on the fly: resource discovery in a cycle sharing peer-to-peer system. In *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 66–73, 2004.

[111] Hai Zhuge, Xue Chen, and Xiaoping Sun. Preferential walk: towards efficient and scalable search in unstructured peer-to-peer networks. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 882–883, Chiba, Japan, 2005. ACM.