INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Resource Location in P2P Systems

João Pedro Fernandes Alveirinho

Dissertação para Obtenção do Grau de Mestre em
Engenharia de Redes de Comunicações

Júri

Presidente:	Prof. Doutor Rui Jorge Morais Tomaz Valadas
Orientador:	Prof. Doutor Luís Eduarto Teixeira Rodrigues
Vogal:	Prof. Doutor Nuno Manuel Ribeiro Preguiça

October 2010

Agradecimentos

I am grateful to Prof. Luís Rodrigues, João Leitão, João Paiva and Mário Ferreira, for the guidance, fruitful discussions and comments during the elaboration of this thesis. This work was partially supported by project “Redico” (PTDC/EIA/71752/2006).

Lisboa, October 2010
João Pedro Fernandes Alveirinho

I'd like to dedicate this thesis to my
grandfather Manuel Alveirinho.

Resumo

Os sistemas entre-pares (P2P) têm vindo a emergir como uma tecnologia para construir sistemas de larga-escala para partilha de dados e recursos. Um dos grandes desafios neste tipo de sistemas passa pela localização dos recursos. A maioria das soluções, abordam o desafio da escalabilidade através da utilização de redes sobrepostas estruturadas (ou DHT's) ou não-estruturadas. Estas soluções apresentam diferentes relações custo-benefício entre a eficiência e a flexibilidade. As DHT's são mais eficientes mas especializam-se na execução de interrogações exactas (*exact-match queries*), enquanto que as soluções não estruturadas, que tipicamente se baseiam em procuras *cegas* (não-guiadas), acarretam custos de sinalização adicionais, mas podem executar interrogações abitrariamente complexas. Esta tese analisa algumas das principais estratégias utilizadas para localização de recursos em sistemas entre-pares, identifica possíveis linhas de investigação para melhorar o estado-da-arte actual e apresenta uma nova arquitectura auto-organizável para a localização de recursos. Esta arquitectura combina soluções estruturadas e não-estruturadas para suportar uma localização de recursos flexível e eficiente. Por exemplo, esta arquitectura pode ser aproveitada para realizar alocação distribuída de recursos em infra-estruturas para computação em núvem. Os resultados experimentais obtidos através de simulação validam o desenho e mostram que a solução proposta oferece uma boa recolha (do inglês *recall*) mantendo um custo reduzido e baixa latência.

Abstract

Peer-to-peer (P2P) systems have emerged as a potential technology to build very-large distributed data and resource sharing systems. A key problem in these systems is the location of resources. Most approaches, address the scale challenge using either structured (DHTs) or unstructured overlay networks. Unfortunately, these solutions have a tradeoff between efficiency and flexibility. DHTs are more efficient but are specialized to address exact-match queries, whereas unstructured solutions, that typically rely in some sort of blind search mechanism, incur in additional overhead, but can address arbitrary complex queries. This thesis makes a survey of the main strategies to implement resource location in P2P systems, identifies some possible lines of research to improve the current state of the art and presents a novel self-organizing architecture to perform resource location. This architecure combines structured and unstructured approaches to support flexible and efficient resource location. For instance, the architecture can be used to perform distributed resource allocation in cloud computing infrastructures. Experimental results extracted through simulation validate this design, and show that the proposed solution is able to offer a good recall, with small overhead and low latency.

Palavras Chave
Keywords

Palavras Chave

Sistemas entre-pares

Redes Sobrepostas

Redes Estruturadas

Redes não Estruturadas

Localização de Recursos

Computação em Núvem

Sistemas auto-organizáveis

Keywords

Peer-to-Peer Systems

Overlay Networks

Unstructured Networks

Structured Networks

Resource Location

Cloud Computing

Self-organizing systems

Índice

List of Figures

List of Tables

Acronyms

ACM Association for Computing Machinery

DHT Distributed Hash Table

IP Internet Protocol

P2P Peer to Peer

RTT Round-Trip Time

SETI Search for Extra-Terrestrial Intelligence

UDP User Datagram Protocol

TTL Time To Live

APS Adaptative Probabilistic Search

SETS Search Enhanced by Topic Segmentation

RW Random Walk

CAN Content Addressable Network

BOT Bias the Overlay Topology

1 Introduction

Since the appearance of Napster(Flanning 1999) in 1999, P2P systems have been subject to intensive research and development efforts, both in academia and industry. Peer-to-peer file sharing systems such as Gnutella(Tsoumakos & Roussopoulos 2006), eMule(Breitkreuz 2002), Kazaa(www.kazaa.com 2000) and more recently BitTorrent(Cohen 2003) have had tremendous success. Unfortunately, these systems have been mainly used to illegally distribute copyrighted material. Still, several examples of legitimate uses of this technology also exist and can be found in, among others (http://www.coralcdn.org/ 2004; http://www.wowwiki.com/Blizzard_Downloader 2006; Korpela, Werthimer, Anderson, Cobb, & Leboisky 2001). From the technological point of view, the potential of the technology to build extremely large-scale shared repositories makes it a very interesting research topic.

One of the main challenges in P2P systems is how to efficiently support resource location. Due to scalability and dependability issues, centralized solutions are not adequate. On the other hand, exhaustive search on all peers is also a non-scalable solution. Therefore, it comes as no surprise that resource location algorithms have been intensely studied, and many different solutions have been proposed in the literature.

1.1 Motivation

As it will later detailled in the thesis, there are two main types of P2P systems: structured and unstructured systems. Typically, structured P2P systems implement a *distributed-hash-table* (DHT), such as Chord(Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001), CAN(Ratnasamy, Francis, Handley, Karp, & Schenker 2001) and Pastry(Rowstron & Druschel 2001). These systems are highly optimized to implement exact-match queries, but provide poor support for more complex inexact queries. In addition, structured systems may be expensive to maintain in highly dynamic environments. An alternative to this approach are unstructured P2P systems, that have low maintenance cost but poor support for query operations. The most basic approach to implement resource location in unstructured systems is the use of limited flooding, an extremely expensive solution that can however, easily support complex queries. In between these two extreme solutions, many algorithms have been proposed, including content-based(Totekar, Vani, & Palavalli 2008), probabilistic(Rhea & Kubiatowicz 2002), and index-based(Zhang & Hu 2005) resource location algorithms.

Our approach to tackle this problem departs from the observation that structured and unstructured overlays both have advantages and disadvantages to support resource location. On the one hand, structured solutions provide fast and efficient exact-match search but lack flexibility to efficiently support complex or inexact queries. On the other hand, systems based on unstructured overlay networks support complex queries but are usually inefficient as they typically resort to flooding mechanisms that can be extremely CPU and bandwidth consuming. Therefore, this works aims to explore new solutions that combine the usage of systems based on both unstructured and structured overlay networks in order to achieve a more efficient and flexible solution for locating resources in a P2P environment.

This solution has been named *Curiata*, a scalable and efficient resource location system that employs self-organizing techniques to integrate and combine the benefits of structured and unstructured approaches. This approach aims at supporting flexible querying, like most unstructured solutions, while retaining the speed and efficiency provided by structured (DHT-based) solutions.

1.2 Contributions

This thesis addresses the issue of resource location in large-scale P2P networks. More precisely, the thesis analyzes the current state-of-the-art solutions for resource location in large-scale P2P networks, proposes a novel hybrid architecture that combines both structured and unstructured P2P overlay networks, and evaluates the performance of the proposed solution. As a result, the thesis makes the following contributions:

- A survey and analysis of the current state-of-the-art solutions for resource location in large-scale P2P networks.

- A novel self-organizing architecture that combines structured and unstructured approaches to support flexible and efficient resource location on large-scale P2P networks.
- The identification of a set of scenarios in which the proposed solution can be applied, and the proposal of a specialization of the architecture for each of these scenarios.

1.3 Results

The results produced by this thesis can be enumerated as follows:

- A prototype implementation of the proposed architecture on the peersim simulator(Jelasity, Montresor, Jesi, & Voulgaris 2009).
- An experimental evaluation of the the prototype in various application scenarios.
- A discussion and analysis of the results obtained through simulation.

1.4 Research History

From the very beginning, the main goal of this work was to design a new architecture for resource location in large-scale P2P networks. This architecture should provide a flexible and efficient way to locate the desired resources. For the definition of the 2-layer architecture proposed in this thesis the author benefited from the discussion with other members of the GSD team at INESC-ID, namely João Leitão, João Paiva and Prof. Luís Rodrigues. Their insights and suggestions were of crucial importance. Furthermore, the proposed architecture leverages on the X-Bot(Leitao, Marques, Pereira, & Rodrigues 2009) protocol designed and developed by João Leitão, Luís Rodrigues, João Pedro Marques and José Pereira, as well as on a DHT such as Chord(Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001) (used for the evaluation process).

To validade the approach, we have attempted to apply it to scenarios with different requirements. Eventually, we have decided to use generic file sharing and resource location in the context of cloud computing as two case studies.

Parts of this work have been published in the “LADIS 2010: The 4th ACM SIGOPS/SIGACT Workshop on Large Scale Distributed Systems and Middleware, Zurich, Switzerland, July 2010” and in the “Inforum 2010, Braga, Portugal, September 2010”.

This work was performed in GSD group of INESC-ID, in the context of the “Redico” project (PTDC/EIA/71752/2006) and co-funded by the INESC-ID multiannual funding) through the PIDDAC Program funds.

1.5 Structure of the Document

The rest of the thesis is organized as follows. Sections 2 and 3 provide the context, by surveying previous research, covering P2P overlays and resource location techniques respectively. Section 4 presents the proposed architecture and its components. Section 5 describes the evaluation procedure and presents the experimental results. Finally, Section 6 concludes the report.

2 P2P Overlay Networks

As the name implies, in a P2P system all nodes cooperate in a similar manner to achieve a common goal. This report focuses on P2P systems that support distributed content sharing. However, it should be noticed that P2P systems can be used for other purposes, such as content distribution(http://www.coralcdn.org/ 2004), cycle sharing(Korpela, Werthimer, Anderson, Cobb, & Leboisky 2001), among others.

In a large scale P2P system, it is often undesirable or even impossible for each individual node to know and cooperate directly with each and every other node in the system. Instead, each peer only knows a small subset of all the system participants. The peering relations among nodes form a network; since this network is constructed on top of an underlying (usually IP-based) network (Fig. 2.1), it is called an *overlay* network.

An overlay network, as any other network, can be modeled as a graph. There are some properties of the graph that are relevant for the operation of the overlay network. Some of these properties are listed below.

The *node degree* is the number of edges that connect the node to other nodes. The graph is said to be *regular* if all nodes have the same degree. Most overlay networks are not regular, in fact, in some unstructured overlays, the node degree can vary substantially from node to node.

Another property is the *network diameter*, which is the number of edges that form the longest of all the shortest paths between any 2 nodes in the network. For instance, in Fig. 2.1 the underlying network diameter is 5.

Another property is *connectivity*. A network overlay is connected if there is a path that allows every node to reach every other node. If the overlay is connected nodes can rely on the overlay to communicate, and no peer is isolated from the rest of the system.

A property that affects both the network diameter and the connectivity of the network, is the *clustering coefficient*. The clustering coefficient of a node N is defined as:

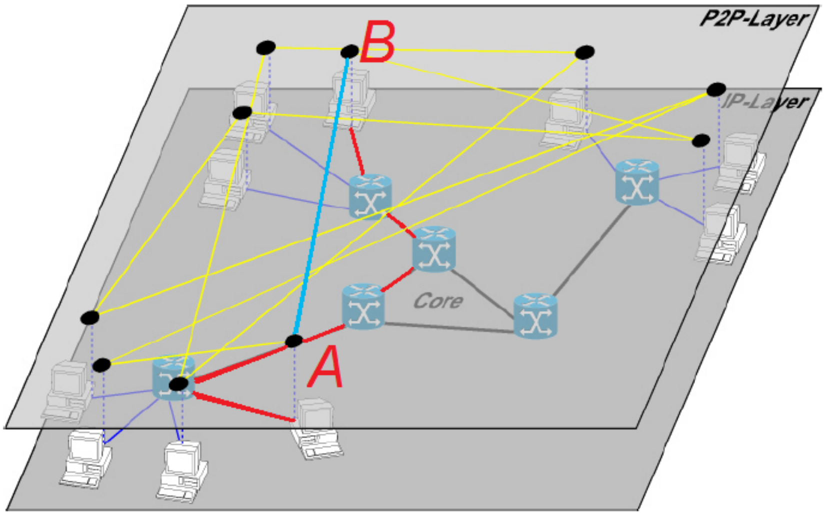


Figure 2.1: Overlay Network

$$C(N) = \frac{e(N)}{deg(N) * (deg(N) - 1)/2} \tag{2.1}$$

Where *deg(N)* is the node degree of N and *e(N)* is the number of links between 2 nodes that are also connected to N, i.e., N's neighbours.

A measure of the efficiency of routing in the overlay network is the *path stretch*: the ratio between the number of edges, of a certain path, in the underlying network and the overlay network. For instance, in Fig. 2.1 the stretch of the path between A and B is 5/1 since, the distance between A and B in the underlying network is 5 edges and the distance in the overlay level is 1 edge. It is worth noting that, when a path is used for communication, each edge crossed by a message is usually called a *hop*.

Finally, it should be noticed that overlays are not static, as nodes leave, crash and join the network. It is also possible that the overlay management protocols cause nodes to change their peers (neighbours), changing the overlay topology. A sequence of multiple join, leave, and crash events at a fast pace is a phenomena that often affect overlay networks that has been dubbed *churn*.

2.1 Unstructured and Structured Overlays

Overlay networks may be classified into two categories: *structured* and *unstructured*.

2.1.1 Unstructured Overlays

In unstructured overlay networks, few or no constraints are imposed on the network topology, which means that the neighbours of each node may be chosen at random or emerge from the way the user interacts with the application. This makes these networks simpler to build and to maintain. For instance, in order to maintain overlay connectivity it is important for each node to maintain a minimum number of links to other nodes in the network (neighbours). When one of the neighbours fails or becomes disconnected, it should be replaced by another neighbour. Since unstructured networks require few constraints to be preserved, it becomes easier to find a suitable replacement.

Even if little effort is put in maintaining an unstructured network, it has been observed that many of these networks have (by emergence or by construction) *small world* properties, i.e., networks with a small average path length and large clustering coefficient(Halim, Wu, & Yap 2008). These properties can be exploited in benefit of the operations that execute on top of the overlay. Furthermore, these networks are also naturally redundant and therefore more resilient to node failures and dynamic environments (such as churn).

Despite the characteristics listed above, unstructured networks pose challenges to the support of efficient resource location, because there is no correlation between the topology of the network and the location of the resources. The most simple manner to implement resource location on unstructured networks is through the use of flooding. However, flooding is very expensive and may cause network congestion due to the large number of redundant messages that it may generate. Therefore, flooding is not a scalable solution, unless optimizations are employed(Leito, Pereira, & Rodrigues 2008), as will be presented later in this report.

2.1.2 Structured Overlays

Structured overlay networks usually implement a distributed hash table (DHT). These overay networks, by construction, impose strong constraints on which nodes may be neighbours. These neighbours are determined by the node's *unique identifier* (*uid*). Objects stored in a DHT also own a unique identifier (typically in the same identifier space as the nodes), which determines the node responsible for storing that object. DHTs support uid-based routing, often in a number of hops logarithmic with the system's size. This functionality can trivially be used to satisfy exact-match queries. However, structured networks also have their own drawbacks. To start with, decomposing a complex query in a set of exact queries is often non-trivial or even impossible. Furthermore, in order to retain their structure, DHTs have significative associated maintenance costs. Structured overlays are also less resilient to identity forging attacks, for instance, sybil attacks(Danezis, Lesniewski-laas, Kaashoek, & Anderson 2005), and to churn.

2.2 Flat and Hierachical Overlays

Overlay networks can also be classified into *flat* and *hierarchical* overlays. Hierarchical overlays usually consist of, as described in (Buford, Yu, & Lua 2008), "two-tier overlays whereby the peers are organized into disjoint groups". The overlay routing to the target group is done using an inter-group overlay and then, an intra-group overlay is used to route to the target peer.

Hierarchical unstructured networks are often built considering that nodes have heterogeneous capacity and stability, and can be classified into two categories: *regular* peers of lower capacity (and/or more volatile) and *super-peers* of higher capacity (and/or more stable). Super-peers coordinate a group of regular peers and coordinate with other super-peers to form an inter-group overlay.

In hierarchical DHTs, each hierarchical group or set forms its own overlay(Xu, Min, & Hu 2003) and, together they form a hierarchical overlay. Hierarchical DHT overlays offer several important advantages over flat DHT-based P2P overlays(Buford, Yu, & Lua 2008), namely:

- Reduce the average number of hops in a lookup query. Fewer hops per query implies less communication overhead. Also, if the higher-level overlay topology consists of stable superpeers the network itself will become more stable.
- Reduce the query latency when the peers in the same group are topologically close. In addition, the stability and the high capacity of the higher-level *super-peers* can also help to cut down the query delay.
- Facilitate large-scale deployment by providing administrative autonomy and transparency, while enabling each participating group to choose its own overlay protocol. Intra-group overlay routing is totally transparent to the higher-level hierarchy. If there are any changes to the intragroup routing and lookup query algorithms, the change is transparent to other

groups and higher-level hierarchy. That is, any churn events within a group are local to the group, and routing tables outside the group are not affected.

2.3 Some Important Features of P2P Overlays

We now discuss a number of important features that any P2P system should own, regardless of the approach used to its construction. These features are: load balance, low maintenance overhead, scalability, and fault-tolerance.

2.3.1 Load balance

The P2P system operation should balance the communication and processing overhead among all nodes of the network. For storage, each node should be responsible for an equivalent fraction of the objects being stored in the network. When supporting resource location, it is important that all the nodes receive an equivalent fraction of the queries being made.

The hash function used in structured P2P overlays (DHTs) may help balance the load, but this is often insufficient, and additional load-balancing measures need to be considered. Some examples of strategies to achieve load balancing are:

- The use of multiple hash functions to balance the storage of objects in a structured P2P system such as Chord(Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001) has been suggested in (Byers, Considine, & Mitzenmacher 2003). The basic idea is that each object may have several identifiers, each one generated by a different hash function and, therefore, increasing the number of nodes (at most as many as the number of hash functions used) where it can be stored. The object is stored in the node with the lowest load. Other nodes that could also be responsible for the given object may also store pointers to the object's location.
- In (Rao, Lakshminarayanan, Surana, Karp, & Stoica 2003), the notion of virtual servers is explored to design load-balancing algorithms. The idea is that each node can have multiple identifiers and join a DHT in different logical locations. As a result, each node becomes responsible for noncontiguous intervals of the identifier space. To achieve load balance, identifiers may be migrated from one node to another.

- In (Rieche, Petrak, & Wehrle 2004) a new and very simple approach for balancing stored data between peers in a fashion inspired by the dissipation of heat energy in materials is presented. During thermal dissipation, a material warmer than its environment delivers energy to the surrounding area. This process continues until a balanced distribution of energy is reached in the overall system. Assuming a DHT architecture in which each interval of identifiers is stored at several nodes, 3 methods to balance the storage load are presented. Assuming f as the minimum number of nodes assigned to a specific DHT region (or interval), if more than f nodes are assigned to a specific interval, one or more of them may be moved to a different interval. If $2f$ different nodes in the same interval are overloaded, then the respective interval can be split in two so that a node only has to manage half of the objects. Finally, nodes in low data load regions can be moved to and/or merged with overloaded regions reducing the load imposed by storing data.

- Overnesia(Leitao & Rodrigues 2008) is an unstructured overlay that creates and maintains *nesos*, which are clusters of peers used for load-balancing and fault tolerance. The load of queries directed to each cluster is distributed among the members of the cluster.

2.3.2 Low Maintenance Overhead

Another important aspect of any P2P solution is the overhead required to maintain its operation. This overhead tends to be higher in structured overlay networks, since their join and leave algorithms are more complex. It is also important to consider the relationship between churn and the bandwidth consumption at a peer for overlay maintenance traffic. There are two strategies to perform maintenance in any overlay:

- *Reactive Maintenance*: In a reactive approach, maintenance is only performed in response to some external event that affects the overlay (e.g. a node joining or leaving). For instance, in a DHT-based network ,a peer handles the failure or departure of an existing neighbour (or the new joining peer added to its neighbour table) by sending a copy of its new neighbour set to other peers in the system. To save bandwidth, a peer can send only differences from the last sent information.

- *Opportunistic or Cyclic Maintenance*: In this type of approach, maintenance is performed periodically and usually involves the exchange of information of each peer with one or more neighbours. This process takes place independently of the peer detecting changes in its neighbour set.

2.3.3 Scalability

Scalability is the capacity of a system to maintain or gracefully degrade its performance as it grows (in number of users, number of objects stored, etc). Early unstructured file-sharing systems such as Gnutella(Tsoumakos & Roussopoulos 2006) suffered from scalability issues, mostly due to the need of flooding the network when performing queries. These scalability issues in unstructured networks, led some to propose DHT solutions to the wide-area file search problem. There are however several proposals to improve scalability in unstructured networks such as GIA(Chawathe, Ratnasamy, Breslau, Lanham, & Shenker 2003).

2.3.4 Fault-tolerance

This is an important feature of any large-scale P2P system, given that, as the number of members increases, the probability of node or link failures occurring also increases. It is therefore crucial for P2P systems to be able to sustain at least crash of faults. Otherwise, the whole system operation may be in jeopardy.

2.4 An Example of a Centralized P2P System: Napster

The first prominent and popular P2P file-sharing system was Napster(Flanning 1999), which was solely dedicated to sharing music files. Napster used a centralized server-based service model, as illustrated in Fig. 2.2[1], where the central server was used for indexing functions and to bootstrap the entire system. In Napster this centralized server was also responsible for executing the queries each node required. This design is simple but suffers from various problems such as:

- The central server represents a single point of failure, as the system is unable to operate without it.

[1] Fig. 2.2 taken from http://en.wikipedia.org/wiki/Peer-to-peer

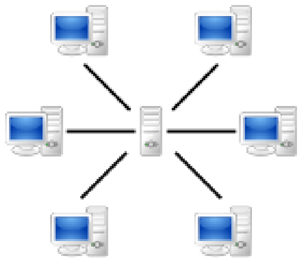


Figure 2.2: Centralized Server Model

- The central server is a bottleneck as it has to sustain all the query load in the system.

- Given that peers cannot cooperate without first contacting the central server, the network resources of the server are also a bottleneck.

2.5 Examples of Unstructured Overlay Networks

2.5.1 Gnutella

Gnutella(Tsoumakos & Roussopoulos 2006) has been proposed in 2000 as a fully decentralized P2P system based on an unstructured overlay network. Queries are supported using flooding techniques. In the initial Gnutella version (v0.4), each node would run a Gnutella client software and, on startup, it would have to find at least one other node that was already a part of the Gnutella network. Different methods can be employed this operation, including a pre-existing address list of possibly working nodes shipped with the software and web caches of known nodes (called Gnutella Web Caches). Once connected to a contact node, the client would request from the contact node a list of addresses of other nodes in the network. The client would then try to connect to those nodes, as well as to other nodes provided by the new neighbours, until it reached a certain quota. When a user wanted to perform a search, the client software would send the request to each actively connected node. In version 0.4 of the protocol, the number of actively connected nodes for a client was quite small (around 5), so each node that received the request, would then forward it to all its actively connected nodes, and they in turn forwarded the request, and so on, until the packet reached a predetermined number of "hops" from the sender (maximum 7). This number of "hops" is commonly known as TTL (Time-to-live). Unfortunately, this approach has some scalability issues, as nodes can

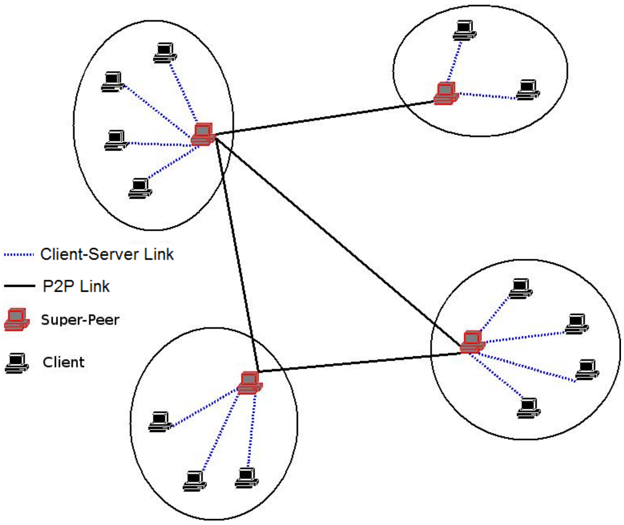


Figure 2.3: Super-Peer Architecture

easily become overloaded by simultaneous queries performed by different nodes.

Later in 2001 a new version of Gnutella was released (version 0.6), that addressed these scalability problems. This new version introduced the notion of super-peers, illustrated in Fig. 2.3[2], to which regular peers registered with the goal of reducing the signaling traffic. Flooding was now restricted to the super-peer level. In this system new super-peers are elected when: i) a super-peer leaves the network; ii) a super-peer has too many regular peers connected to it (denoted leaf-nodes); iii) a super-peer has too few leaf-nodes. The election is based on an estimate of the capacity of the peer, in terms of CPU, storage, bandwidth, and availability (uptime). Gnutella is therefore an example of a hierarchical P2P system.

2.5.2 HyParView

HyParView, which stands for Hybrid Partial View, is a gossip-based membership protocol(Leitao, Pereira, & Rodrigues 2007) that builds and maintains an unstructured overlay network. The protocol is characterized by each node maintaining two distinct views: a small and symmetric active view and a larger passive view.

The active views define an overlay that is used for cooperation among peers (the use of HyParView is illustrated with message dissemination applications). Links in this overlay are symmetric, which means that if node q is in the active view of node p then node p is also in

[2] Fig. 2.3 adapted from http://schuler.developpez.com/articles/p2p/images/super-peer.jpg

the active view of node q. When a node receives a message for the first time, it broadcasts the message to all nodes of its active view (except, obviously, to the node that has sent the message). A reactive strategy is used to maintain the active view. Nodes can be added to the active view when they join the system. Also, nodes are removed from the active view when they fail.

On the other hand, the passive view is not used to support communication. Instead, the goal of the passive view is to maintain a list of nodes that can be used to replace failed members of the active view. The passive view is maintained using a cyclic strategy. Periodically, each node performs a shuffle operation with one of its neighbours in order to update its passive view. In this shuffle operation, the node provides to its neighbour a sample of its partial views and, symmetrically, collects a sample of its neighbour's partial views. In fact, in this operation the identifiers that are exchanged belong not only to the passive view, but also to the active view. This increases the probability of having nodes that are active in the passive views and ensures that failed nodes are eventually expunged from all passive views.

This approach offers a strong resilience to node failures, even in the presence of extremely large numbers of crashes in the system. High resiliency to node failures is important to face unintentional (for instance, natural disasters) or intentional (for instance, software worms and virus) events that take down a significant portion of nodes in the system.

2.5.3 X-BOT

X-BOT(Leitao, Marques, Pereira, & Rodrigues 2009) is a protocol to bias the topology of an overlay according to some target efficiency criteria, for instance, to better match the topology of the underlying network, i.e., to reduce the average path stretch (it can be used to bias the topology for different criteria though). Based on HyParView, X-BOT relies on the combined usage of two distinct partial views; the goal of the protocol is to reduce the average link cost of the overlay network defined by the active views. For that purpose, X-BOT actively bias the neighbours in the active view using random peers extracted from the larger passive view. Moreover, the cyclic strategy used to maintain the passive view ensures that its contents are periodically updated and therefore, gives access to different potential neighbours over time to each node. Unlike HyParView, that strives to ensure the stability of the overlay, X-BOT relaxes stability to be able to continuously improve the overlay. This allows the topology of the unstructured overlay to self adapt to better match the requirements of the application executed on top of it. Periodically, each node starts an optimization round in which it attempts to switch one member of its active view for one (better) neighbour of its passive view. In the optimization protocol, a node uses its local Oracle to obtain an estimate of the link cost to some random selected peers of its passive view. Examples of oracles are:

- *Latency Oracle:* This Oracle operates by measuring round trip times (RTT) to peers. This can be performed by exchanging probe messages with Oracles located at other nodes. The Oracle must be aware of the peers which are known at the local host, and it slowly measures the RTT for each know node (this value can be directly used as the cost value).
- *Internet Service Provider Oracle:* In a setting where exchanging messages across different ISPs has an increased monetary cost, it might be useful to keep as many neighbours as possible that share the same ISP. To this end, a simple oracle can be built by maintaining information concerning the local ISP and a table of costs for each known ISP. When the Oracle becomes aware of a new peer, it simply exchanges local ISP information with the remote Oracle and asserts the cost for the link using the local cost table.
- *IP-based Oracle:* These Oracles are able to calculate neighbour proximity values, which can be used as cost, using IP aggregation information (for instance, using a match of common IP prefixes to calculate a measure of proximity between two peers).

2.6 Examples of Structured Overlay Networks

2.6.1 Chord

The first four DHT's (CAN(Ratnasamy, Francis, Handley, Karp, & Schenker 2001), Chord(Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001), Pastry(Rowstron & Druschel 2001), and Tapestry(Zhao, Huang, Stribling, Rhea, Joseph, & Kubiatowicz 2004)) were introduced at about the same time in 2001. Since then, research on DHTs has been quite active. Chord peers are organized in a flat circular (Fig. 2.4[3]) overlay network. Each node in chord, as in other DHT solutions, has its own identifier and so does each data item stored in the network. Chord provides support for just one operation: given a key, it maps the key onto a node. Data

[3] Fig. 2.4 taken from (Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001)

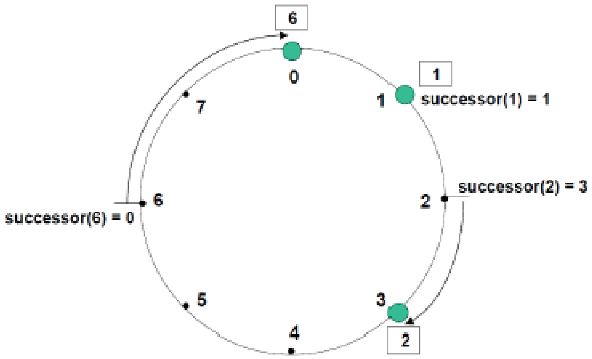


Figure 2.4: Chord Architecture

location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data item pair at the node to which the key maps. Each key/data item pair should be stored at the node in the network node that has the lowest ID which is still equal or bigger than the item's ID. Chord addresses difficult problems such as:

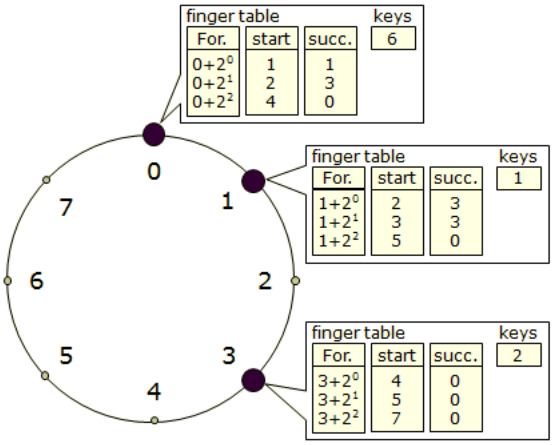
- Load balance: Chord acts as a distributed hash function, spreading keys evenly over the nodes, providing a degree of natural load balance.

- Decentralization: Chord is fully distributed and no node is more important than any other. This improves robustness and makes Chord appropriate for loosely-organized peer-to-peer applications.

- Scalability: The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible.

- Availability: Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that, barring major failures in the underlying network, the node responsible for a key can always be found. This is true even if the system is in continuous change.

In order to achieve efficient location of data items in the network, each node must mantain a reference to his successor in the ring and also a *finger-table* in which a set of routing entries are kept, to allow for larger hops in the overlay network. In this *finger-table* as can be seen in



finger table — keys: 6

For.	start	succ.
$0+2^0$	1	1
$0+2^1$	2	3
$0+2^2$	4	0

finger table — keys: 1

For.	start	succ.
$1+2^0$	2	3
$1+2^1$	3	3
$1+2^2$	5	0

finger table — keys: 2

For.	start	succ.
$3+2^0$	4	0
$3+2^1$	5	0
$3+2^2$	7	0

Figure 2.5: Finger Tables

Fig. 2.5[4]:

- Each node mantains m entries (where m is the number of bits of the identifiers used).
- The i^{th} entry in the table at a node with an identifier n contains the identity of the first node s that succeeds n by at least 2^{i-1} on the identifier circle(s=successor(n+ 2^{i-1})).
- A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node.
- The first finger of n is the immediate successor of n in the circle.

Using the *finger-table*, Chord can efficiently route any query to its destination node. To achieve this, when a node n does not know the successor of a key k, that node will search its *finger-table* for a node j whose ID is closer (biggest predecessor of k) to k and ask j for the node it knows whose ID is closest to k. By repeating this process, n learns about nodes with IDs closer and closer to k. At each step of this process, the distance to the destination node is cut down by half, thus bringing n closer and closer until the successor of k is found. In fact, the number of steps of this process is logarithmic with the number of nodes in the system.

[4] Fig. 2.5 taken from (Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001)

2.6.2 Kademlia

Another example of DHT-based systems is Kademlia(Maymounkov & Mazires 2002), which is a peer-to-peer <key,value> storage and lookup system based on the XOR metric. Each node in Kademlia has 160-bit id chosen at random or by using an hash function such as SHA-1(http://www.w3.org/PICS/DSig/SHA1_1_0.html 1995). Keys are also 160-bit identifiers and, to publish and find <key,value> pairs, Kademlia relies on a notion of distance between two identifiers (x and y) defined as the XOR (exclusive OR) of those two indentifiers (d(x,y) $=x \oplus y$). To route query messages, each node keeps contact information about other nodes. This means that, for each $0 < i < 160$, every node stores a list of <IP address; UDP port; Node ID> triples for nodes of distance between 2^i and 2^{i+1} from itself. These lists, also known as k-buckets are kept sorted by last time seen (least recently seen node at the top) mainly because studies(Saroiu, Gummadi, & Gribble 2002) have shown that the longest a node has been online, the more likely it is for that same node to remain online. Therefore, by keeping the oldest live contacts, k-buckets maximize the probability that these nodes will remain online. For small values of i, k-buckets will generally be less populated or even empty (as few suitable nodes will exist). For larger values of i, the lists can grow up to size k, where k is a system-wide replication parameter. k should be chosen so that any given k nodes are very unlikely to fail or leave within an hour of each other, thus preventing the loss of the stored data.

Whenever a node receives a message from another node, it updates the correspondent k-bucket for the sender's node ID. This update operation follows a series of rules:

- If the sending node's ID already exists in the recipient's k-bucket then, the recipient moves it to the end of the list.
- If the node is not already in the appropriate k-bucket and the bucket has fewer than k entries, then the recipient inserts the new ID at the end of the list.
- If the correspondent k-bucket is full, then the recipient pings the k-bucket's least-recently seen node. If the least-recently seen node fails to respond, it is evicted from the k-bucket and the new sender is inserted at the end of the list. Otherwise, if the least-recently seen node responds, it is moved to the end of the list, and the new sender's contact is discarded.

This update strategy diminishes the need for cyclic maintenance to be performed since it keeps the k-buckets updated as a result of network traffic. For its operation, the Kademlia protocol relies on 4 RPC's:

- PING - This RPC probes a node to see if it is online.
- STORE (ID,value) - This RPC instructs a node to store a <key,value> pair for later retrieval.
- FIND NODE(ID) - The recipient of a FIND NODE RPC should return <IP address, UDP port, Node ID> triples for the k nodes it knows about closest to the target ID. These triples may come from a single k-bucket, or from multiple k-buckets, if the closest k-bucket is not full. Either way, the RPC recipient must always return k items (unless there are fewer than k nodes in all its k-buckets combined, in which case it returns every node it knows about).
- FIND VALUE(ID) - This RPC behaves like the FIND NODE RPC returning <IP address, UDP port, Node ID> triples unless, the RPC recipient has received a STORE RPC for the key, in which case it just returns the stored value.

The most important procedure a Kademlia peer must perform is locating the k closest nodes to some given node ID. This procedure is called a *node lookup* and consists of a recursive algorithm in which:

- The lookup initiator starts by picking α nodes (where α is a system-wide concurrency parameter) from its closest non-empty k-bucket (or if that bucket has fewer than α entries, it just takes the α closest nodes it knows of).
- Then the initiator sends parallel, asynchronous FIND NODE RPC's to the α nodes it has chosen.
- The initiator then recursively sends the FIND NODE RPC to nodes it has learned about from previous RPC's until the initiator has queried, and collected responses, from the k closest nodes it has seen.

When $\alpha = 1$ the lookup algorithm resembles Chord's in terms of message cost and the latency of detecting failed nodes. However, Kademlia can route for lower latency because it has the flexibility of choosing any one of k nodes to forward a request to. Most operations are implemented using the *lookup procedure*. For instance, to store a <key,value> pair, a participant locates the k closest nodes to the key and sends them STORE RPCs. Additionally, each node re-publishes the <key,value> pairs that it has every hour in order to ensure persistence with very high probability.

2.6.3 CAN

CAN(Ratnasamy, Francis, Handley, Karp, & Schenker 2001) stands for content-addressable network, and is yet another example of a structured overlay network (DHT). Just like in Chord(Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001), CAN peers and objects have identifiers from the same virtual address space. CAN organizes this logical space as a d-dimensional cartesian space and partitions it into zones. One or more peers are responsible for each zone and every object key corresponds to a point in the space and is stored at the zone which contains that point. An example two-dimensional space with dimensions [0, 80] x [0, 80] is shown in Fig. 2.6[5]. In the figure, three peers are highlighted: x, y, and z. The extent of the zone for each peer is shown to the right of the figure. Objects are hashed to keys in the same space and stored at the peer whose zone assignment contains that key. For example, an object with a key [65, 35] would be in peer z's zone. Locating an object in CAN is reduced to routing to the node that hosts the object. In fact, routing from a source node to a destination node is equivalent to routing from one zone to another in the Cartesian space. An example of 3 possible routing pathes (a,b and c) between 2 zones are shown in Fig. 2.7[6].

2.6.4 Hieras

The usage of Hierachical DHT's has been explored in works such as Hieras(Xu, Min, & Hu 2003), Cyclone(Artigas, Lopez, Ahullo, & Skarmeta 2005) and Canon(Ganesan, Gummadi, & Garcia-Molina 2004). Hieras is a multi-layer (Fig. 2.8[7]) DHT-based P2P routing algorithm. Like in other DHT's, all the peers in a Hieras system form a structured P2P overlay network. However, Hieras contains many other P2P overlay networks (P2P rings) in different layers inside

[5] Fig. 2.6 taken from (Buford, Yu, & Lua 2008)
[6] Fig. 2.7 adapted from (Buford, Yu, & Lua 2008)
[7] Fig. 2.8 taken from (Xu, Min, & Hu 2003)

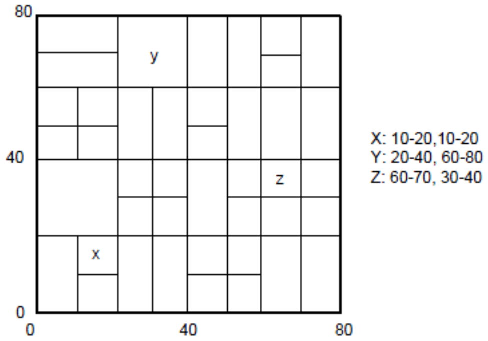


Figure 2.6: CAN 2-dimensional space with 42 peers total and 3 highlighted peers

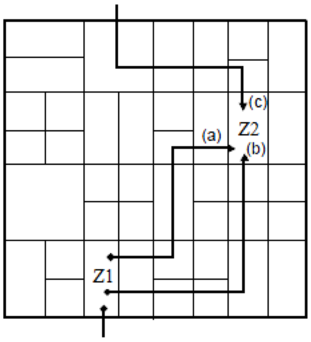


Figure 2.7: CAN 2-dimensional space: routing pathes between Zone Z1 and Zone Z2

the global P2P network. Each of these P2P rings contains a subset of all system peers. These rings are organized in such a way that the lower the layer of a ring, the smaller the average link latency between two peers inside it. In Hieras, the routing procedure starts in the lowest layer P2P ring in which the request originator is located and moves up until it eventually reaches the largest P2P ring. A large portion of the routing hops in Hieras are therefore taken in lower layer P2P rings, which have relatively smaller network link latencys. Therefore, an overall lower routing latency is achieved.

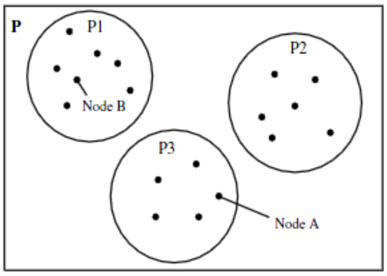
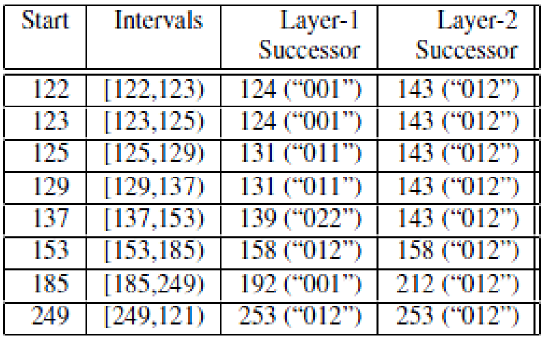


Figure 2.8: Overview of a Two-Layer HIERAS System



Start	Intervals	Layer-1 Successor	Layer-2 Successor
122	[122,123)	124 (“001”)	143 (“012”)
123	[123,125)	124 (“001”)	143 (“012”)
125	[125,129)	131 (“011”)	143 (“012”)
129	[129,137)	131 (“011”)	143 (“012”)
137	[137,153)	139 (“022”)	143 (“012”)
153	[153,185)	158 (“012”)	158 (“012”)
185	[185,249)	192 (“001”)	212 (“012”)
249	[249,121)	253 (“012”)	253 (“012”)

Figure 2.9: Node 121 (012)'s Finger Tables in a Two-Layer HIERAS System

In the Hieras's design, Chord was chosen as the underlying routing algorithm for its simplicity (it should however be easy to extend Hieras to other DHT algorithms). A few changes have to be made to Chord's *finger tables* to comprise the hierarchical structure imposed by Hieras (Fig. 2.9[8]).

The original Chord *finger table* is used as the highest layer *finger table*. In addition, each node creates *m-1* (*m* is the hierarchy depth) other *finger tables* in lower layer P2P rings it belongs to. For a node to generate a lower layer finger table, only the peers within its corresponding P2P ring can be chosen and put into this *finger table*. Fig. 2.9 shows the *finger tables* of a node with the ID 121. This node's second layer P2P ring is ”012”. In the highest layer *finger table*, the successor nodes can be chosen from all system peers. For example, the layer-1 successor node in the range [122,123] is 124 and it belongs to the layer-2 P2P ring ”001”. Whilst in the second layer, finger table successor nodes can only be chosen from peers inside the same P2P ring as node 121. For instance, the successor node in the range [122,123] is 143, which also belongs to layer-2 P2P ring ”012”.

Summary

This chapter presented the fundamentals of Peer-to-Peer Overlay Networks. We started out by identifying some of the key properties that can be used to characterize these networks and proceeded to divide P2P overlay networks into two categories: *structured* and *unstructured*. Each of these types of overlay networks present their own set of advantages and drawbacks.

[8] Fig. 2.9 taken from (Xu, Min, & Hu 2003)

Unstructured overlay networks are easier to build and usually present reduced maintenance costs. However, these networks, due to their structureless nature, do not provide efficient mechanisms for resource location. On the contrary, structured systems are harder to build and have higher associated maintenance costs, since they impose a strict structure on the participating nodes. However, by mantaining a structure these networks can easily and efficiently satisfy exact-match queries. Unfortunately, not all systems can simply rely on executing exact-match queries. For various reasons, for instance, because they rely on human interaction, some systems require the execution of complex or inexact queries. Structured overlay networks do not natively support these kinds of queries, unlike unstructured netwoks, and therefore are not appropriate solutions for those systems.

Later on, we further differentiated overlay networks by presenting hierachical overlays and introducing the notion of *super-peers*, and then proceeded to present some key features of P2P overlay networks namely: load balance, maintenance overhead, scalability and fault-tolerance. This chapter was finalized by showing some examples of each kind of overlay networks, along with presenting some relevant historical examples of famous P2P systems such as Napster and Gnutella.

3 Resource Location in P2P

According to Webster's Dictionary, to search is "to look into or over carefully or thoroughly in an effort to find or discover something". In P2P Systems, in order to find the resources one is looking for, a search strategy must be applied. The choice of which search strategy to use depends not only on the type of network involved (structured or unstructured, hierarchical or non-hierarchical,...) but also on the results expected. Therefore, it is of extreme importance to understand which are the most common strategies and in which context they may be applied to achieve desired results.

3.1 Query Types

The selection of the query algorithm may depend on how the query is expressed. The simplest form of query is an exact match query, where objects that have a single attribute (for instance, a given name) are searched. However, it is often interesting to support richer forms of search, such as keywords, range-queries, or semantic queries as described below.

Exact Match Queries - In exact-match queries, the object to be searched is specified by the value of a given attribute, for instance its name. DHTs are designed to support exact-match very effectively.

Keyword Queries - Keyword queries are a generalization of exact match. Instead of characterizing the searched objects by a single attribute value, the objects to be searched are characterized by a logical expression that combined multiple attribute values (keywords) using *and* and *or* operators. For instance, one may search for a document tagged with string *s1* but not tagged with string *s2*.

Range Queries - A range query aims at retrieving all records where some value is contained in some given interval. This type of query is commonly used in databases. An example of such type of query is: "*list all employees of a company with more than 2 and less than 6 years of experience*". Range queries are a challenging issue in the P2P search domain. They have been addressed by works such as Mercury(Bharambe, Agrawal, & Seshan 2004).

Semantic Search - A semantic search is a content-based, full-text search, whereby queries are expressed in natural language instead of simple keyword matches(Wu 2005). These types of queries aim at finding resources that are semantically similar to one described in the query itself. Semantic searches are even harder to support than range-queries: given a query, the system either has to search a large number of nodes or miss some relevant documents. Later in the document, a system that supports this type of queries (pSearch(Tang, Xu, & Dwarkadas 2003), will be briefly overviewed.

3.2 Characteristics and Performance Metrics

Peer-to-peer search algorithms may be optimized for different criteria. For instance, one might wish to get results faster at the cost of not being able to find all the possible results for a certain query. This section lists the most relevant characteristics and performance metrics of search algorithms.

Convergence - A search algorithm is said to converge if, in each step executed by the algorithm, one becomes closer to finding the the desired object. This concept is clear in systems implementing exact-match queries on top of DHTs, where each hop in the lookup operation approximates the query to the target. In this case, the average number of steps for convergence is *log(n)* hops where n is the number of nodes in the network. In an unstructured overlay where exact-match is implemented by flooding, convergence is only ensured if a TTL (Time To Live) with the diameter of the network is used (i.e., if the entire network is searched). Obviously, the cost of each step is quite different in both scenarios. Furthermore, when more complex queries are supported, the notion of convergence becomes blurred, given that it may be possible for the algorithm to return only a subset of the matching objects.

Recall Rate - The query recall rate is defined as the ratio between the number of relevant documents retrieved during the processing of the query and the total number of relevant documents that existed in the system when the query was processed. For example, assume that when a query is placed there are in the network 60 objects that match the query but only 30 of those are returned; the recall rate for this query is *0.5*. Obviously, the higher the recall the better, although there may be a tradeoff between the recall rate and the cost of the query.

Message Cost - Message cost can either be measured as the average or total number of messages necessary to execute queries, and as a result the amount of network traffic generated by such queries.

Latency - Latency is a measure of how long it takes to obtain the response to the query. Latency can be measured in absolute time, which is actually the metric of relevance to the user, but that does not depends exclusively on the search algorithm but also on the properties of the IP network that supports the overlay. A more abstract manner to measure latency is to use the number of communication steps required to execute the query.

Precision - Precision is defined as the fraction of the documents retrieved in a query that are relevant to the user's needs. Poor and inflexible query languages can lead to low levels of precision since users can experience difficulties when trying to describe the contents they are looking for. Therefore, one of the ways to improve the precision of a search mechanism is to provide rich and flexible search languages and mechanisms so that its users can properly define the data they are looking for. Another approach to improve query precision is the usage of feedback mechanisms with which systems can learn from the user's input about query results. However, sometimes there might be a trade-off between the level of precision and the recall rate of a search algorithm. This means that, by relaxing the precision one can increase the algorithm's recall rate and vice-versa.

3.3 Query Dissemination Strategies

In order to execute queries in a P2P system, different query dissemination strategies may be applied. Each strategy has its own set of advantages and disadvantages, and the choice of using one or another is not always easy to make. The organization and structure of the peers in the overlay network is just one of the factors that affects this choice. This section aims at providing an overview of commonly used query dissemination strategies in P2P Systems.

3.3.1 Network Flooding

As noted before, the basic and most straightforward approach to the search problem is network flooding. Flooding-based search was a popular approach in early unstructured P2P networks such as Gnutella(Tsoumakos & Roussopoulos 2006). In this strategy, the querying peer sends the query request to all or a subset of its neighbours. Then, each of these neighbours processes the query, returns the result if a match is found, and then forwards the query to its own neighbours. This procedure is repeated by each neighbour until a given TTL threshold is reached. This type of query dissemination mechanism, if the network is large (i.e. if each peer has a large amount of neighbours), generates a massive amount of network traffic per query. Gnutella used breadth-first search (BFS) and fixed TTL to limit the number of hops each query may take, in an attempt to reduce query bandwidth cost. Obviously, the downside of these cost mitigation strategies is the reduction of the query recall rate.

3.3.2 Iterative Deepening

Iterative deepening is a variant of flooding that aims at reducing the number of nodes that are required to process the query(Yang & Garcia-Molina 1998) until a target number of responses is obtained. The basic idea is to initiate the query procedure by performing a limited flood with a small TTL; if the desired number of responses is not achieved, a new flood with a larger TTL is initiated. This process is repeated until a maximum TTL is reached. Variants of this scheme are often called *expanded ring search*. An obvious limitation of this approach is that, when a new flood (with larger TTL) is initiated, all nodes involved in the previous flood need to re-execute the query. To prevent this behavior, when the query is re-sent, it is marked with the previous TTL, such that nodes already visited may simply forward the query without executing it. Alternatively, nodes at the border of the previous ring search may store the query for some time. In this case, the originator may just transmit a *resend* message, with the query identifier; when the resent reaches one of the border nodes, these "unfreeze" the corresponding query and forward it with the new TTL.

3.3.3 Random Walks

This strategy is an alternative to flooding, that aims at avoiding the scalability issues posed by flooding on unstructured P2P systems(Lv, Cao, Cohen, Li, & Shenker 2002). Given a query, a random walk is essentially a blind search in which, at each step, the node that receives the query processes it and then forwards it to another single randomly chosen node. This process may go on until the query is satisfied and may be terminated in two ways: TTL and *checking*. TTL means that, similarly to solutions based on flooding, each random walk terminates after a certain number of hops in the overlay network, while *checking* means that a *walker* (i.e. the query message being forwarded in the network) periodically checks with the query originator before advancing to the next node. Random walks can effectively reduce the number of redundant messages, but at the cost of increasing search latency. To reduce this delay, one may use k-way random walks (*k-walkers*) where the querying node forwards the original query message to k randomly selected neighbours instead of only one. Since the number of nodes reached by k random walkers in h hops is the same as in one random walk over kh hops, a reduction of around k times in query delay can be expected.

3.3.4 Guided Searches

Search methods can be categorized as either *blind* or *informed/guided*. In *blind* searches, nodes do not store any information regarding file locations. In *informed/guided* approaches, nodes locally store metadata that assist in the search for the queried objects. *Blind* methods usually need to consume a lot of bandwidth to achieve high performance. On the other hand, *informed* methods use their indices to achieve similar quality results and to reduce traffic overhead. The problem with most *informed* methods is the maintenance cost of the indices after peers join/leave the network or update their collections. In most cases, these events trigger floods of update messages inflating network traffic.

A Guided search is a search technique based on the construction, at each node, of indices that can "guide" the routing of the query(Crespo & Garcia-Molina 2002). Guided searches allow nodes to forward queries to neighbours that are more likely to have answers, rather than forward queries to randomly chosen neighbours or flood the network by forwarding the query to all neighbours. Routing indices indicate a promising *direction* toward the answers for queries.

These distributed indices are small (i.e., compact summary) and provide hints on the probable *best* direction toward the resource one is looking for, rather than its actual location. Indices may be built incrementally, for instance based on the results of previous queries.

3.3.5 Probabilistic Search

Probabilistic search is a name used to characterize a form of guided search based on incomplete information. These type of queries usually rely on the usage of data structures, such as *Bloom Filters*(Cai, Ge, & Wang 2008), or result caching mechanisms to determine when a certain peer or region of peers is likely to store a certain object or not. Because they rely on probability, some of these approaches may generate false positives and/or false negatives.

3.3.6 Similar Content Group-Based Search

The basic idea behind this strategy is to organize P2P nodes into groups in which peers store similar content. This is usually applied on top of unstructured P2P systems such as Gnutella(Tsoumakos & Roussopoulos 2006). The intuition behind this approach is that nodes within a certain group tend to be relevant to the same queries. As a result, this type of search strategy will guide the queries to regions of nodes that are more likely to have answers to the queries, thus allowing to configure a flooding dissemination strategy with a smaller TTL value allowing these solutions to achieve a lower operation overhead. Several works such as (Bawa, Manku, & Raghavan 2003) and (Cohen, Fiat, & Kaplan 2003) have addressed this type of strategy.

3.4 Examples of Systems Supporting Resource Location

This section illustrates how the techniques described previously have been used in different systems.

3.4.1 pSearch

The fundamental challenge that the authors of pSearch(Tang, Xu, & Dwarkadas 2003) identified as being one of the causes for the complexity of P2P resource location solutions is that,

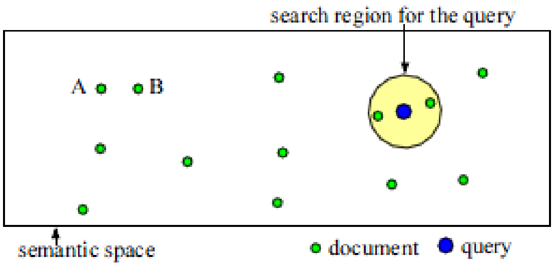


Figure 3.1: Search in a Semantic Space

with respect to semantics, documents are randomly distributed in the system. Therefore, given a query, a system has to either search a large amount of nodes or risk not being able to find relevant documents. To address this issue, the notion of semantic overlay is presented as “a logical network where contents are organized around their semantics, in such a way that the distance between two documents in the network is proportional to their dissimilarity in semantics”. pSearch is a prototype P2P information retrieval system that works by representing documents as semantic vectors and organizing them in the network around their vector representations. In pSearch, to generate the semantic space, extensions to Vector Space Model (VSM)(Berry, Drmac, & Jessup 1999) and Latent Semantic Indexing (LSI)(Deerwester, Dumais, Furnas, Landauer, & Harshman 1990) are used, and CAN(Ratnasamy, Francis, Handley, Karp, & Schenker 2001) is used to support the semantic overlay. Vector space model (VSM) represents documents and queries as *term vectors*. Each element of the vector represents the importance of a word (*term*) in the document or query. Two factors decide the importance of a term in a document: the frequency of the term in the document and the frequency of the term in other documents. If a term appears in a document with a high frequency, there is a good chance that term could be used to differentiate the document from others. However, if the term also appears in several other documents, the importance of that term is reduced. During a retrieval operation using VSM, the query vector is compared to document vectors and those closest to the query vector are considered to be similar and are returned. Latent semantic indexing was proposed to address synonymy, polysemy, and noise problems in literal matching schemes such as VSM. For instance, although *car*, *vehicle* and *automobile* are different terms, they all reference the same or very similar objects. In VSM this would make no differnce but, LSI may be able to discover that they are related in semantics and therefore generate more reliable semantic vectors. The basic idea of pSearch is to use the semantic vector(generated by LSI) of a document as the key to store the document in the CAN.

Fig. 3.1[1] shows how a semantic overlay can benefit searches. Each document is placed as a point in the (semantic) Cartesian space. Documents close in the semantic space have similar contents (e.g. documents A and B). Each query can also be positioned in this semantic space and to find documents relevant to that query it is only necessary to compare it against documents within a small region centered at the query, because the relevance of documents outside that region is relatively low. This results in an effectively smaller search space for the query.

To set-up the semantic overlay, an index for each document is stored in the CAN using the document's semantic vector as the key. Among other things, an index includes the semantic vector of a document and a reference (URL) to the document itself. The basic operation model in pSearch can be summarized in 4 steps:

- When receiving a new document (which can be submited by any node, inside or outside of the CAN) a, the Engine node, a node that is part of the CAN, generates its semantic vector Va using LSI and uses Va as the key to store the index in the CAN.
- When receiving a query q, the Engine node generates its semantic vector Vq and routes the query in the overlay using Vq as the key.
- When a query reaches its destination, it is flooded to nodes within a radius r, determined by the similarity threshold or the number of wanted documents specified by the user.
- All nodes that receive the query do a local search using LSI and report the references to the best matching documents back to the user.

3.4.2 Cubit

Cubit(Wong, Slivkins, & Sirer 2008) takes a different approach on the issues discussed in pSearch. While the focus of pSearch is on finding documents with high semantic relevance to the search keys it is unable to match misspelled search keys to documents with correctly spelled keywords.To overcome this issues, Cubit works by creating a keyword metric space that encompasses both the nodes and the objects in the system and where the distance between two points is a measure of the similarity between the strings that those points represent. The

[1] Fig. 3.1 taken from (Tang, Xu, & Dwarkadas 2003)

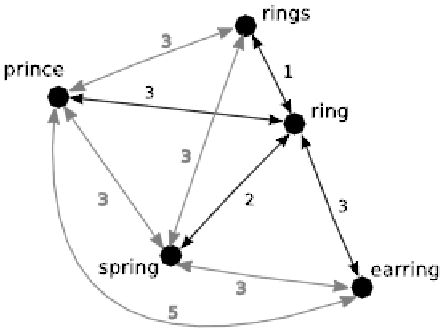


Figure 3.2: The edit distance between keywords

objective of the Cubit system is to find the k closest data items for any given search key. This is achieved by creating a keyword metric space that captures the relative similarity of keywords, assigning portions of this space to nodes in the overlay and to resolve queries by routing them through this space. The focus of Cubit is on providing approximate keyword search for multimedia content with limited content description. Keywords are derived from the content's filename and information specific to the content type, such as the comment section of torrent files or the extended video information for YouTube video clips.

An object stored in Cubit is characterized by one or more keywords. Cubit's approach to approximate matching relies on a notion of distance between keywords. Cubit mainly uses the most common notion of distance on strings, the *Levenshtein* distance, commonly known as the *edit* distance(Fig. 3.2[2]). It is equal to the minimum number of insertions, deletions, and substitutions needed to transform one string to another. However, search queries typically consist of more than one keyword (for instance, the title of a movie). Therefore, Cubit matches queries using the phrase distance (i.e. the distance between two sets of keywords) which is used to calculate the distance between a query and an object.

In Cubit, nodes are distributed in the same space as keywords. This means that each node in Cubit is assigned a unique string ID chosen from the set of keywords associated with previously inserted objects in the system. This ID determines a node's position in the keyword space and each Cubit node is responsible for storing the set of keywords for which it is the closest node.

In Cubit, a search operation is processed by a distributed protocol which navigates through nodes in the keyword space, gradually zooming in on a neighbourhood of a given (possibly

[2] Fig. 3.2 taken from (Wong, Slivkins, & Sirer 2008)

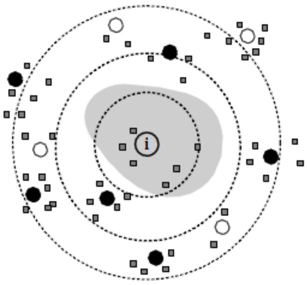


Figure 3.3: Cubit concentric rings structure

misspelled) keyword, and thus locates nodes that store possible matches. To achieve this, Cubit creates and maintains a multi-resolution overlay network on nodes such in which each node has several peers at every distance from itself. Each Cubit node organizes its peers into a set of concentric rings. In each ring, a node retains a fixed number, k, of neighbours whose distance to the host lies within the ring boundaries. This ring structure enables a Cubit node to retain a relatively large number of pointers to other nodes in its surroundings, while also providing a sufficient number of pointers to far-away peers. This ring structure is depicted in Fig. 3.3[3] in which, the solid circles represent peers in node i's neighbourhood-set, the empty circles represent other nodes, and the squares represent object keywords in the system. The shaded region depicts the subspace that is closer to i than any other node. The master record for each keyword in the shaded region is stored at node i.

The way the Cubit search protocol operates is by iteratively collecting more and more information of the target region. In Fig. 3.4[4], x is the location of the search term in the keyword space, the solid circles are node i's peers, empty circles are additional nodes in the space, and the circle around x are all nodes within an edit-distance q of x. Node i first finds the $nmin = 2$ closest nodes to x from its neighbourhood-set, and requests their $nmin$ closest nodes. As a result, two new closer nodes are discovered and subsequently sent the same query. The protocol terminates when all nodes within the circle around x, or when the $nmin$ closest nodes have been discovered. These nodes are then queried for their objects closest to x.

[3] Fig. 3.3 taken from (Wong, Slivkins, & Sirer 2008)

[4] Fig. 3.4 taken from (Wong, Slivkins, & Sirer 2008)

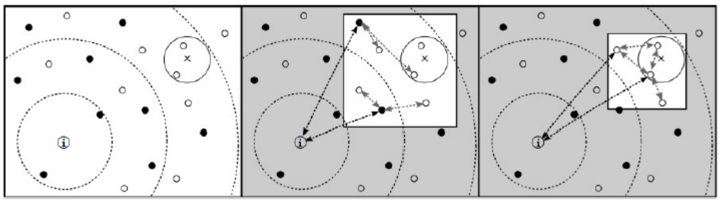


Figure 3.4: The Cubit Search Protocol

3.4.3 Adaptive Probabilistic Search (APS)

In (Tsoumakos & Roussopoulos 2003) a new search algorithm called Adaptive Probabilistic Search (APS) method is proposed. This algorithm achieves high performance at low cost. In APS, a node deploys k-walkers for object discovery, but the forwarding process is probabilistic instead of random. Peers direct these walkers using feedback from previous searches, while keeping information only about their neighbours.

The APS algorithm is defined for deployment over unstructured P2P networks and is based on some assumptions:

- Peers initiate searches for various objects that are distributed across the network according to a *replication distribution*, which dictates which objects are stored at each node.
- Popular objects get many more requests than unpopular ones.
- The search algorithms cannot dictate the placement and replication of objects in the system and are not allowed to alter the topology of the P2P overlay.
- A node is directly connected to its neighbours, and these are the only peers whose addresses the node knows about.
- Nodes can keep some soft state for each query they process. Each search is assigned an identifier, which, together with the soft state, enables peers to make the distinction between new and duplicate messages. Identifiers are also assigned to objects and nodes from a flat, non-hierarchical space.

In APS, each node keeps a local index per neighbour, consisting of an entry for each object it has requested, or forwarded a request for. The value of each entry reflects the relative probability of this node's neighbour to be chosen as the next hop in a future request for the specific object.

In the forwarding process, a node chooses its next-hop neighbour according to the probabilities given by its index values and appends its identifier in the search message, keeping soft state about that search. If two walkers from the same request cross paths (i.e., a node receives a duplicate message due to a cycle), the second walker is assumed to have terminated with a failure and the duplicate message is discarded. Index values stored at peers may be updated using one of the following strategies:

- Optimistic Approach: In this approach, when a node chooses one or k (if the node is the query originator) peers to forward the request to, it pro-actively increases the relative probability of the peer(s) it picked, assuming the walker(s) will be successful

- Pessimistic Approach: In this approach, the node decreases the relative probability of the chosen peer(s), assuming the walker(s) will fail.

Upon walker termination, if the walker is successful, nothing is done in the optimistic approach but if the walker fails, index values relative to the requested object along the walker's path must be corrected. Therefore, using information included in the search message, the last node in the path sends an *update* message to the preceding node. This node, decreases its index value for the last node to reflect the failure and the update process continues along the reverse path towards the requester. If the pessimistic approach is used, the update procedure is analogous, and nodes increase the index values along the walker's path, but the update only takes place when a walker succeeds (instead of when a walker fails).

This method uses probabilistic walkers with a learning feature that incorporates knowledge from past and present searches to enhance future performance. The learning process adaptively directs the walkers to promising zones of the network. In fact, this method has an increased recall rate in comparison to the original *blind* k-walker(Lv, Cao, Cohen, Li, & Shenker 2002). APS also does not require message exchange on node arrivals or departures. If a node detects the arrival of a new neighbour, it will associate some initial index value to that neighbour and if a neighbour disconnects, each node that has that node as a neighbour, removes the relative entries and stops considering it in future queries.

3.4.4 SETS : Search Enhanced by Topic Segmentation

SETS(Bawa, Manku, & Raghavan 2003) is an architecture for efficient search in peer-to-peer networks, building upon ideas drawn from machine learning and social network theory. The key idea behind SETS is to arrange sites (peers) in a network such that a search query probes only a small subset of sites where most of the matching documents reside. In particular, SETS partitions sites into topic segments such that sites with similar documents belong to the same segment. Each topic segment has a succinct description called the topic centroid. Sites are arranged in a segmented network that consists of two kinds of links. Short distance links connect sites within a segment. Long distance links connect pairs of sites from different segments.

When a search query is initiated, it is forwarded to other sites using a topic-driven routing protocol. First, topic centroids are used to select a small set of relevant topic segments. Next, the selected segments are probed in sequence. A probe to a particular segment proceeds in two steps: First, the query is routed along long distance links to reach a random site belonging to the target segment. Next, the short distance links are used to propagate the query to all/most/few sites within a segment. By applying this content-based search approach, SETS is able to reduce the number of nodes that are queried while mantaining a high recall rate since the nodes that are queried, are those with higher probability of holding the desired resources.

3.4.5 Gia: Making Gnutella-like systems scalable

Gia(Chawathe, Ratnasamy, Breslau, Lanham, & Shenker 2003) is an unstructured overlay based on a P2P System like Gnutella(Tsoumakos & Roussopoulos 2006). However, Gia proposes several modifications to the original Gnutella design that dynamically adapt the overlay topology and the search algorithms in order to acommodate the heterogeneity of most peer-to-peer systems. In fact, these modifications aim at improving the scalability of systems that use flooding or random walks for resource location. In order to acomplish this, Gia distinguishes overlay nodes according to a "capacity" metric and distributes load through the overlay according to the "capacity" of each node. In particular, node index state, connection degree, and permitted query rate are each regulated according each node's "capacity". The proposed enhancements can be divided into four components:

- *Dynamic Topology Adaptation* - This ensures that most nodes are within short reach of high capacity nodes. In particular, in Gia high capacity nodes, with higher availability to execute queries, have a higher connectivity degree to other nodes. This mechanism is adaptive in that nodes actively seek more neighbors to satisfy their own capacity level.

- *Active flow control* - To prevent nodes from being overloaded with query messages, nodes provide tokens to their neighbors to regulate the message rate, thus avoiding overloaded hot-spots. Therefore, a node can only send a message to its neighbor when it has received a token from that neighbour.

- *One-hop index replication* - All nodes mantain pointers to the resources shared by all of their neighbours. Due to the Dynamic Topology Adaption, this will cause high capacity nodes to keep indexes for a high percentage of the nodes in the system, since they have a higher connectivity degree. When a node receives a query message, it searches both its local index and the copies provided by its neighbours to see if there is a match. This ensures that high capacity nodes are capable of providing answers to a greater number of queries.

- *A Biased random walk search protocol* - In which peers preferably forward queries towards higher capacity neighbours that are more likely to have the answer to the query, since they have a higher degree and therefore mantain a higher number of resource indexes.

Gia's approach effectively increases the scalability of Gnutella-like systems, however this is achieved by forcing nodes to keep resource indexes of all of their one-hop neighbours and by introducing a notion of "capacity" that might not be easy to define and determine in a real P2P application.

Summary

This chapter started out by presenting some common types of queries that are used to describe the desired resources. The simplest type of queries are exact-match queries, which DHTs were designed to support very efficiently. Other types of queries include: keyword queries, range queries and semantic searches.

Later on, a few of the most relevant characteristics and metrics that can be used to evaluate a search mechanism's performance were identified. That section also introduced the concept of *convergence*. We proceeded to discuss some of the most well-known query dissemination strategies. From the simplest and most straightforward approaches, such as network flooding, to complex and intricate search techniques, like guided and probabilistic searches, a wide variety of strategies were analyzed.

This chapter ended by presenting some relevant examples of systems to support resource location. The first two examples(pSearch and Cubit), addressed the issue of semantic searches, whilst APS (Adaptative Probabilistic Search) focused on a probabilistic search mechanism. Finally, SETS aims at partitioning the P2P network into segments, so that peers with similar documents belong to the same segment.

4 Proposed Architecture

In order to improve the current state-of-the-art, a new solution needs to address two key challenges:

- Query Flexibility - The proposed solution should restrict the type of queries available as little possible. Querying mechanisms in the new solution should be flexible and allow rich and complex query languages. This would present an advantage when comparing to traditional DHT-based solutions which are typically only used for exact-match queries.

- Efficient Location of Resources - The proposed solution should be able to provide an efficient way to locate resources, in contrast with traditional unstructured solutions that typically rely on inefficient blind search mechanisms.

To answer these challenges we propose *Curiata*, a scalable and efficient resource location system that employs self-organizing techniques to integrate and combine the benefits of both structured and unstructured approaches. This approach supports flexible queries, like most unstructured solutions, while retaining the speed and efficiency provided by structured (DHT-based) solutions. The operation of *Curiata* is inspired by the organization of human societies. During the first two decades of the Roman Republic, the people were organized into units called *curia* of ethnic nature; the curia gathered into an assembly, the *comitia curiata*, for legislative, electoral, and judicial purposes, and where consuls had a special role. Similarly, in *Curiata*, peers self-organize in an unstructured overlay where nodes with similar shared resources establish neighbouring relations via a low-cost background process (the curia). Furthermore, nodes of each *curia* elect representatives to join a structured overlay (the *curiata*). Members of the structured overlay serve as contact points for other nodes with similar content. Thus, the structured layer is used to efficiently route queries towards regions of the unstructured layer which contain peers that share the type of resources being queried for. Then, the query is propagated amongst the members of the curia using unstructured techniques such as limited flooding or random walks.

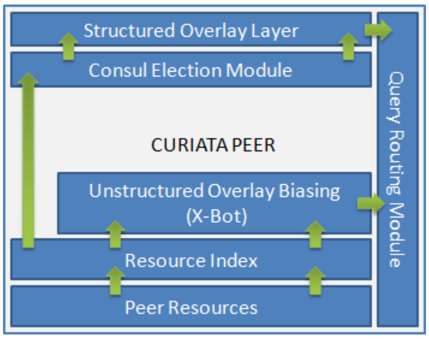


Figure 4.1: Curiata Peer

In addition to providing a fast, efficient and flexible querying infrastructure, *Curiata* also aims at achieving a high recall rate independently of the popularity (rarity) of the resources that are targeted by each query, while keeping a low overhead.

Curiata has a few generic components which will be referred to as the *Curiata* core architecture. However, some of these components can be specialized to optimize *Curiata*'s performance in each application scenario. The remainder of this chapter will:

- Introduce the basic building blocks of the *Curiata* core architecture.
- Describe and detail each module of the architecture and how they interact with each other.
- Present instantiations of the *Curiata* architecture in two distinct application scenarios.

4.1 *Curiata*: The Core Architecture

This section describes the core architecture of *Curiata* and the operation of its different components. As noted before, this architecture combines an unstructured overlay layer (the *curia*) and a structured overlay layer (*comitia curiata*). In the following, we describe the main goals of each layer and how they operate.

The components of a *Curiata* peer, depicted in Fig. 4.1, are the following: i) The *resource index*, that describes the local resources available at the peer. ii) The *biased unstructured overlay* layer, that uses a distributed self-organizing protocol to ensure that peers establish neighboring relations with other peers with similar resources. iii) The *structured overlay* layer, which is only activated when a peer is elected as a *consul*. iv) The *consul election module*, that employs a local self-organizing protocol to select which peers belong to the structured layer. v) The *query routing* module, responsible for propagating and processing queries. In addition, some of these modules have specific methods that can be implemented in various ways to accomodate the requirements for different application scenarios. Each of these methods will be identified and the specific implementations for each instantiation will be presented.

4.1.1 Resource Index

In *Curiata*, it is assumed that resources can be classified in a set of categories. The resource index keeps a local record of all categories of the resources owned (locally) by the peer as well as (if applicable) the number of resources available for each of these categories. This information is used to identify which peers have similar resources. The classification scheme is orthogonal to *Curiata*. For instance, a distributed library of computer science papers could use the ACM Computing Classification System to classify the content. A distributed repository of music could extract the categories required to classify content from the most used tags used in popular applications such as “Last.fm” (http://last.fm).

4.1.2 Unstructured Layer

The purpose of the unstructured layer is to organize all peers that have available resources in a biased unstructured overlay network. More specifically, we propose that peers run a self-organizing distributed algorithm, to adapt the overlay topology according to the resources available at each node. The used algorithm is a specialized version of X-BOT (Leitao, Marques, Pereira, & Rodrigues 2009), adapted to meet the requirements of *Curiata*.

X-BOT is used to bias the unstructured overlay topology so that each node becomes neighbour with other nodes that have similar available resources. The rationale for this strategy is to be able to efficiently process queries by limiting the relevant areas of the unstructured overlay where nodes owning resources relevant for a query are located. More precisely, the *curia* layer works as follows. Each node keeps two types of neighbors, denoted *active* and *passive*. The set of active neighbors defines the overlay that is used to propagate queries. Thus, the size of the active view d defines the degree of the node in the curia overlay. Note that *Curiata* does not require every node to have the same degree and therefore, it can accommodate nodes with different capacities. Passive neighbors are used for exploring the network and finding other peers with similar interests. The passive view is updated by a periodic and random shuffle process similar to the one described in (Voulgaris, Gavidia, & van Steen 2005). The active view is updated through the coordination procedure introduced by X-BOT, using a strategy specifically designed for *Curiata*. The the active view/unstructured layer update routine can be found in Listing. 4.1.

```
if(!active-View.has-Enough-Neighbours()) //Test if active view has enough neighbours
{
  active-view.get-More-Neighbours();     //Establish network connectivity
}
else
{
  if(!active-View.is-Fully-optimized())  //Test if the active view is optimized
                                         //Instance Dependent!
{
  active-View.optimize();                //Trade old for ‘‘better’’ neighbours
                                         //Instance Dependent!
}
return;
```

Listing 4.1: Unstructured Layer/Active View Maintenance Routine

The main methods executed during the active view update are the following:

has-Enough-Neighbours() Tests if the node has enough neighbours in its active view. If a certain quota of the active view size is not filled with neighbours, the node trys to establish neighbouring links with more peers.

get-More-Neighbours() Method that attempts to establish neighbouring links to nodes from the passive view. This method is executed to ensure network connectivity.

is-Fully-Optimized() Tests if the active view is fully optimized (i.e., if all active view requirements/constraints are met). The requirements/constraints for the active view are instance dependent and examples will be presented further ahead in the architecture instantiation section 4.2.

optimize() Method that attempts to exchange neighbours from the active view for “better” neighbours from the passive view, in order to fulfill the defined requirements/constraints for the active view.

An instantiation of these methods can be seen in Listing 4.2. However, as referred in the method descriptions, some of these methods are generic and are the same for all instantiations, and some others can be optimized to better fit the application environment in which *Curiata* is deployed.

```
//Generic Method
has-Enough-Neighbours()
{
  return (active-View.size() > (MaxActiveViewSize * ActiveViewQuota));
}

//Generic Method
get-More-Neighbours()
{
  Node node = passive-View.get-Random-Node();

  active-View.try-Connect(node);

  return;
}

//Instance Dependent
is-Fully-Optimized()
{
  // continuously try to optimize
  return false;
}

//Instance Dependent
optimize()
{
  // X-Bot default Mechanism
}
```

Listing 4.2: Method instantiation Unstructured Layer

An important advantage of the unstructured layer is that it can operate with a very low and controlled cost. Since the overlay is unstructured, there are few constraints that need to be preserved, either when new nodes join the system or when the network has to be repaired due to node departures or failures. Also, the self-organizing algorithm induces a small amount of background traffic that can be controlled by adjusting the period between consecutive biasing steps.

4.1.3 Structured Layer and Consul Election

As noted above, the unstructured layer is able to organize itself so that it promotes neighboring relations among nodes that have similar resources (*i.e.*, resources that fall into the same categories). Therefore, when searching for resources, as soon as one finds a single peer that owns resources from the desired category, one can expect to easily find other nodes with similar categories in the overlay vicinity. The purpose of the structured layer is to facilitate the first step of the resource location procedure.

For this purpose, a fraction of the nodes that belong to the unstructured overlay also join a DHT (such as Chord (Stoica, Morris, Karger, Kaashoek, & Balakrishnan 2001) or Pastry (Rowstron & Druschel 2001)). These nodes are elected and represent a category owned by nodes in each region of the unstructured overlay (or curia). These nodes are referred to as *regional consuls* or *regional contacts*. Thus, the structured overlay layer (i.e, the DHT) acts as an assembly of representatives of the different regions in the unstructured space: its purpose is to efficiently route a query to regions of the unstructured overlay where the searched resources are likely to be located.

The consul election module, represents the way nodes coordinate among themselves to elect consuls, i.e., nodes that represent a curia of peers with similar resources in the unstructured layer.

The basic idea is to have a *regional consul* for each region of peers, within a certain radius r, that share a category c. That peer is called the regional consul for category c, or simply c-consul. A c-consul joins the DHT with an identifier constructed by assigning *hash(node_id)* to the s less significant bits and *hash(c)* to the remaining (most-significant) bits. This ensures that multiple contacts for the same category in different regions have different identifiers but are placed in a consecutive region of the DHT address space.

A c-consul node uses the unstructured overlay to periodically send a beacon to nodes in its r-vicinity. Nodes that receive the beacon, abstain from competing to become a c-consul. For higher values of r there will be larger regions and a smaller number of consuls in the DHT. On the contrary, for lower values of r the opposite occurs. The value of r, the way peers are chosen to become consuls and the factors that atribute priority for consul election, can be adapted according to the specific characteristics of each application scenario. The Structured Layer maintenance routine can be found in Listing 4.3.

```
if(this.is-Consul())   // Test if node is already consul
{
  Category cat = this.get-Representing-Category();
  this.send-Beacon(cat,r);   //Send a beacon of radius ‘‘r’’
  return;
}

// Gets the set of categories for which the node can be a consul
// Instance Dependent!
Array[Category] Categories = this.get-Categories-Node-Can-Represent();

for (Category ‘‘c’’ in Categories)
{
  if(!this.has-Received-Beacon(‘‘c’’)) //Test if node has received a beacon for ‘‘c’’
  {
    this.compete-For-Election(‘‘c’’,r); //Try to become a ‘‘c’’-consul
    return;
  }
}
return;  //Terminate if all categories the node could represent already have a consul
```

Listing 4.3: Structured Layer Maintenance Routine

The main methods executed during this routine are the following:

is-Consul() Tests whether a node is already a Consul in the Structured Layer.

get-Representing-Category() Returns the category which this node is currently representing in the Structured Layer.

send-Beacon() Sends a Beacon to a radius r informing neighbouring nodes that this node is a consul for category c (c-consul).

get-Categories-Node-Can-Represent() Returns the set of categories this node can represent in the Structured Layer. This method differs for each instantiation.

has-Received-Beacon() Tests whether a node has received a Beacon for a category c within a time interval defined *a priori*.

compete-For-Election() Sends an *Election Beacon* with a radius r for this node to try becoming a Consul for a category c.

Instantiations for these methods are depicted in Listing 4.4.

```
is-Consul() // Generic Method
{ return this.belongs-To-DHT; }

get-Representing-Category() // Generic Method
{ return this.dht-Consul-Category; }

send-Beacon(Category cat, int radius) //Generic Method
{
  BeaconMessage beacon = new BeaconMessage();
  beacon.TTL = radius;
  beacon.category = cat;
  for(Neighbour ‘‘n’’ in active-View)
  { n.send(beacon); }
  return;
}

get-Categories-Node-Can-Represent() //Instance Dependent
{
  // returns all categories a node belongs to
  return node.categories-List;
}

has-Received-Beacon(Category cat) //Generic Method
{
  Array[BeaconMessage] beaconList = this.get-Received-Beacons();
  for(BeaconMessage bm in beaconList)
  {
    if(bm.category.equals(cat))
      return true;
  }
  return false;
}

compete-For-Election(Category cat,int radius) //Generic Method
{
  ElectionBeaconMessage ebm = new ElectionBeaconMessage()
  ebm.category = cat;
  ebm.TTL = radius;
  for(Neighbour ‘‘n’’ in active-View)
  { n.send(ebm);}
  return;
}
```

Listing 4.4: Method Instantiation Structured Layer

It is important to notice that, if a node that belongs to a category c, from the set of categories the node can represent in the DHT and, does not receive any beacons (within a certain time interval t), it decides to compete with other potential candidates to become a c-consul. When multiple nodes compete, a simple bully election scheme is used to select which node becomes the c-consul for that region. A node that is elected to be a c-consul, can use a consul of another category c' as a contact point for joining the DHT or, if there is none yet, it may perform a random walk in the unstructured overlay to find a node that has information concerning any regional *consul* to use as a contact point to join the DHT.

In addition, each node may only be the c-consul for a single category. A node that already is a c-consul does not compete to become the regional contact for any of the other categories that node could represent in the DHT. Also, if a node is not a *consul* and could be a *regional contact* for multiple categories, it picks one of those categories at random and competes only for that category. Only if it loses an election, will it attempt to become the *regional contact* for another category. This effectively distributes the routing load among the nodes of each region of r radius.

It is not impossible that a node becomes the only owner of two or more categories in a region. In this case, only one of these categories will be represented in the DHT. The resources owned by that node in those unique categories will not be found in queries that address exclusively those categories. However, these resources would still be found by queries that also target other resource categories in the region. This scenario is, however, extremely rare, as X-BOT effectively gathers nodes with the same category in the same region.

4.1.4 Query Routing

Curiata does not constrain the format, nor the language, of the queries that can be performed. There is only one requirement: from the query, it should be possible to extract the set $\mathcal{Q}$ of categories that match the query. For instance, assume that a query searches for a music by Aldina Duarte, one should be able to extract categories such as *world music*, *fado*, and *Portugal*[1].

Thus, the query routing module is reponsible for routing queries towards regions of the

[1] These categories were extracted from the user tags associated with this artist in Last FM.

unstructured layer which contain peers whose categories match the ones defined in the query. This involves the partipation of nodes in the unstructured layer - forwarding the query within the regions that match such categories - and nodes in the structured layer, that route queries towards those regions.

To avoid flooding the network with query messages, the idea is to disseminate and process each query with some approximate message cost k. In the current prototype the value of k is static and defined offline. However, k could be dynamically adjusted to match the estimated rarity of the searched resource (for instance, based on the results returned by previous searches).

In more detail, a query is disseminated as follows:

- First, the query is routed to the nearest member of the DHT (this is based on periodic beacons sent by regional consuls).
- Then a copy of the query is routed to each category $c \in \mathcal{Q}$ using the DHT. Each copy will be received by a c-consul for that category. To promote load balancing, for each category c, the query is routed to an identifier composed by *hash(c)*$||\{s$ random bits$\}$. This ensures that different queries are injected into the unstructured layer via different representatives of that category.
- Each c-consul starts a random walk of length $\frac{k}{|\mathcal{Q}|}$ in its vicinity. In each hop in the random walk, the query is preferably forwarded to a node that has not yet received the query (this is achieved by including, in the query message, the list of nodes to which the query has been forwarded previously). These random walks are biased and are preferably forwarded to a neighbor that owns resources of the category c associated with the random walk.
- Each node visited by the random walks (including the c-consuls) executes the query locally and checks if it satisfies the query. In affirmative case, it adds to the random walk its own identifier.
- Finally, when a random walk reaches the maximum number of hops, it returns to the source of the query all nodes matching the query (if any) that were visited by the random walk.

The total message cost of a query is $\frac{k}{|\mathcal{Q}|} \times |\mathcal{Q}|$ plus the cost of reaching a DHT member from the source (typically 1), plus the number of hops in the DHT required to reach the c-consuls which is typically low, as it is in the order of $c \cdot \ln T$ where T is the number of nodes in the DHT. As was previously discussed, one can easily configure the system to promote a low value of T.

The query dissemination process may be trivially optimized by having nodes avoiding to route the query to consuls of some categories, when the originator of the query already owns some neighbors that match some of the categories in the set $\mathcal{Q}$. In this case, the originator of the query can use such neighbors as representative of these categories and having them initiate the random walk dissemination in the unstructured overlay. This avoids the additional message cost of routing copies of the query, for those categories, through the DHT.

4.2 Instantiating the Architecture

This section will present two possible instantiations for the *Curiata* arquitecture. For both these scenarios, the *Curiata* core architecture remains unchanged. However, the implementation of methods that were previously presented as instance dependent will be further detailed. This methods have specific implementations to better ajust to the characteristics of each scenario. In the first scenario the *Curiata* architecture is used to support resource allocation in cloud computing infrastructures. The second scenario addresses resource location in Peer-to-Peer file sharing systems.

4.2.1 Resource Allocation in Cloud Infrastructures

In cloud computing infrastructures, outside users allocate resources in the cloud in order to store and execute their applications or services. However, inherent to the cloud computing paradigm is the ability to adjust or deploy, on demand, services to face dynamic changes on the workload. This brings upon the need for an infrastructure to keep track of the resource allocation in the cloud and to efficiently locate the resources required to launch/redimension a new/running service or application.

A simple solution to solve the resource location problem in cloud infrastructures would be to use some form of centralized directory. For instance, a central server that would collect, and keep up-to-date, information concerning the resources available at each server. However, available resources are dynamic, as new services are deployed, old services are decommissioned, or services are relocated to increase efficiency or reduce power-consumption. As the number of servers increases, a central directory may easily become a bottleneck. Hierarchical directories mitigate this problem, but introduce additional complexity and traffic in order to maintain the structure. Furthermore, in order to achieve high availability, replication should be employed. However such mechanisms would significantly increase the bandwidth requirements of the system, a factor especially relevant if the replicas were geographically distant. The challenges of keeping up-to-date information about a large-scale system were discussed in several papers, for instance (Renesse, Birman, & Vogels 2001; Brandt, Gentile, Mayo, Pebay, Roe, Thompson, & Wong 2009).

In this scenario, the categories used in the system are related to resources shared by nodes in a cloud environment. For instance, available CPU units, disk space, memory and architecture (32-bit or 64-bit) of each node.

Resource Index

In this scenario, every resource available at each node in the cloud, which can be allocated by clients, is classified into some category from a finite set of categories defined a priori; although the number of categories can be arbitrarily high.

For instance, one may classify nodes according to the amount of available memory as *memXS* (*1.7*Gb), *memS* (*7.5*Gb), *memM* (*15*Gb), etc. Also, nodes can be classified according to their architecture in the categories *32*bits or *64*bits. A similar approach can be used to classify other types of resources, such as disk space, number of cores, physical location, etc. A node belongs to all the categories that characterize the resources it makes available.

When characterizing resources that can take a discrete value from a possibly large range, the definition of categories simplifies the task of defining which nodes are more similar. In these cases each category represents an interval I of values for that resource. Each peer in the system that shares that resource and has a value v for it, will fall into a category c so that $v \in I$.

For instance, if a node has *5*Gb of available memory and a *32*-bit processor, it would fall into the *memXS* category, since it has more than *1.7*Gb but less than *7.5*Gb of available memory and, in terms of architecture, it would belong to the *32*bits category. The way these intervals are defined is application specific and could reflect the type of virtual machines that can be allocated in the cloud. Note that, a node with *5*Gb of available memory can allocate a virtual machine that requires *1.7*Gb (*memXS*) of memory but cannot be used to allocate a virtual machine that requires *7.5*Gb (*memS*) of available memory.

Unstructured Layer

In this particular scenario, because we consider each category within a node to have the same importance for that node (i.e, nodes could otherwise, for instance, organize themselves only based on their available memory), our X-BOT proximity metric when applied to a pair of nodes returns a distance value that depends on the number of resource categories shared by those nodes. This means that, the returned value is zero if both nodes share exactly the same set of resource categories, and increases by a fixed amount for each category owned by only one of the nodes.

Therefore, X-BOT operates by exchanging overlay links, so that for each node n it maximizes the number of neighbors of n that share the same set of resource categories.

For instance, consider nodes a,b,c and d that belong to the following sets of category respectively:

- Node $a \in$ {*memXS*; *64*-bit; diskSpaceXL}
- Node $b \in$ {*memS*; *64*-bit; diskSpaceXL}
- Node $c \in$ {*memXS*; *64*-bit; diskSpaceXL}
- Node $d \in$ {*memM*; *32*-bit; diskSpaceXL}

The distance d between the each of the nodes is the following: $d(a,b) = 1$; $d(a,c) = 0$; $d(a,d) = 2$; $d(b,c) = 1$; $d(b,d) = 2$; $d(c,d) = 2$.

This means that in order for a node's active view to be fully optimized, all neighbours that belong to a node's active view need to have a distance of *0* to that node. Therefore, the *is-Fully-Optmized()* method only returns true when all the distances in a node's active view are *0*. This method can be found in Listing 4.5.

```
is-Fully-Optimized()
{
  for(int i=0; i < active-View.size(); i++)
  {
    if(distance(this.active-View.get(i)) != 0)
    {
      return false;
    }
  }
  return true;
}
```

Listing 4.5: The is-Fully-Optimized() method (Resource Allocation in Cloud Infrastructures Instantiation)

In addition, in each optimization step, the node tries to gather peers in its active view so that their distance is as little as possible. Therefore, this leads the *optimize()* method to be defined as found in Listing 4.6. Each time this method is executed, the current node *n* trys to exchange a node *n'* in its active view with the highest distance to himself for a node *n"* from its passive with the lowest distance to himself. However, the swap is only executed if *d(n,n")<d(n,n')*.

```
optimize()
{
  int smallest_distance = MAXIMUM_DISTANCE;
  int smallest_distance_index = 0;

  for(int i=0; i < passive-View.size(); i++)
  {
    if(distance(this,passive-View.get(i)) < smallest_distance)
    {
      smallest_distance = distance(this,passive-View.get(i));
      smallest_distance_index = i;
    }
  }

  int highest_distance = 0;
  int highest_distance_index = 0;

  for(int i=0; i < active-View.size(); i++)
  {
    if(distance(this,active-View.get(i)) > highest_distance)
    {
      highest_distance = distance(this,active-View.get(i));
      highest_distance_index = i;
    }
  }
  if(smallest_distance < highest_distance)
  {
    this.swap(active-View.get(highest_distance_index),
             passive-View.get(smallest_distance_index));
  }

return;
}
```

Listing 4.6: The optimize() method (Resource Allocation in Cloud Infrastructures Instantiation)

Structured Layer and Consul Election

In this scenario, nodes can compete to become *consuls* in any category c to which they belong. Therefore, for this scenario, the *get-Categories-Node-Can-Represent()* method returns all the categories to which a node belongs. This is so because,all categories are considered to be equally important for a node and therefore the node can represent any of them in the Structured Layer.

Resource Allocation

In response to a query, the source receives a list of at most k nodes that match the query. Some of these nodes serve as *regional consuls* and others do not. However, in this particular scenario, because resources are dynamic (i.e. they may change often overtime), to promote DHT stability, resources should be preferably allocated from nodes that are not *consuls* and, only if not enough non-*consul* nodes are found, should resources be allocated from *regional consuls.*

Resource allocation itself is application dependent and orthogonal to the *Curiata* architecture. For instance, a new service may be deployed on the selected nodes or be expanded to use additional resources available in the cloud. When resources are effectively used on the nodes, the nodes update their categories accordingly.

Note that when resources are allocated, the categories owned by a node may change. In response, X-BOT will iteratively move the node to another region of the unstructured overlay. However, *Curiata* does not require the unstructured overlay to have converged in order to satisfy queries. A node that does not mach a query simple forwards the random walk to one of its better suited neighbors.

4.2.2 Resource Location in File Sharing Systems

In peer-to-peer file sharing systems users share different content with each other. From audio or video files, documents and e-books, to applications and computer games, a large variety of content is shared within these systems. Setting aside the legal issues sometimes associated with file-sharing systems, the truth is that they have had tremendous success and a definite impact on internet users around the world. One of the key challenges for these systems lies on how to efficiently locate the resources (files) that users are looking for. Therefore, file sharing systems constitute yet another scenario in which the *Curiata* architecture can be employed.

Resource Index

In P2P file sharing systems, users share different contents and, in different quantities (usually according to that user's particular likes). For instance, a user u might share a lot of rock music and a few electronics *e-books* whilst another user *u'* might share some *sci-fi* movies and *anime*.

As mentioned earlier in *Curiata*'s core architecture description, the resource index keeps a local record of all the categories of the resources owned by a peer, as well as the number of resources available for each of these categories. However, to accommodate the diversity in both shared content and quantity that characterizes users in a P2P file sharing system, the resource index component also keeps, for each category c of the resources owned by a peer, the fraction of resources of that peer that fall in that category.

This value is denoted by $frac_c$. For instance, consider a node that stores computer science papers, and half of these papers fit in the *self-stabilizing system* category. We will have $frac_{SSS} = 0.5$. Also, all the categories are sorted by *frac* values. The top t categories are used to define the neighboring relations at the unstructured layer.

Unstructured Layer

For this scenario, the curia layer divides the active view in t slices, where t is the same configuration parameter that is used to select the t top categories with larger fraction of local resources. Each of these slices is devoted to one of the t top categories. The slice for category c has a dimension $slice_c$ in the interval $\lceil smin, d \cdot \mathrm{frac}_c \rceil$. Where $smin$ is the minimum size of a slice, and is set as $\frac{d}{2t}$. For instance, consider a system where the curia layer is configured to select neighbors according to the top 5 categories ($t = 5$). Consider a peer p such that the top 5 categories $(c1, \ldots, c5)$ have the following associated fractions: $(0.5, 0.2, 0.1, 0.1, 0.1)$. If peer p had a degree $d = 20$, its active view would be sliced as follows: $(10, 4, 2, 2, 2)$.

Considering the slicing of the active view as described above, a node uses the following steps to bias its neighbors:

1. The first concern of a peer when it joins the unstructured overlay is to fill its active view, regardless of the similarity of its potential neighbors. This step aims at ensuring network connectivity.

2. After filling the active view, the next priority for the peer is to have neighbors that belong to his top categories. (i.e, each slice contains at least one neighbor of that category).

3. Then, the peer tries to exchange neighbors so that each slice is filled with neighbours that have resources of the corresponding category.

4. Finally, as soon as this last criteria is achieved the peer stops executing the self-organizing algorithm and maintains its neighbors unchanged.

This slicing of the active view leads to the implementation of the *is-Fully-Optimized()* method to be as depicted in Listing 4.7. In fact, the active view is only considered to be fully optimized if all peers in the active view belong to at least one category from the node's top categories and if each slice is filled with peers that belong to that category.

```
is-Fully-Optimized()
{
  Array[Category] Top-Categories = resource-Index.get-Top-T-Categories();

  for(int i=0; i < active-View.size(); i++)
  {
    Node neighbour = active-View.get(i);

    if(!neighbour.belongs-To-At-Least-1-Category(Top-Categories))
    {
      return false;
    }
  }

  for(Category ‘‘c’’ in Top−Categories)
  {
    if(active−View.number−Of−Neighbours−in−Category(‘‘c’’)
          < active−View.get−Slice−Size(‘‘c’’))
    {
      return false;
    }
  }
  return true;
}
```

Listing 4.7: The is-Fully-Optimized() method (Resource Location in File Sharing Systems Instantiation)

In addition, the *optimize()* method follows two steps. In the first step, the node attempts to trade neighbours that belong to neither of its categories for nodes that belong to at least one of his categories. In the second step, the node trys to gather enough nodes in each category so that the active view respects the pre-defined slice sizes. During this process, when a node finds a category *c* that lacks neighbours in the active view, if the active view still has some open slots, the node simply trys to gather one more neighbour from its passive-view that belongs to *c*. Otherwise, if the active view is completely full, the node trys to find a category *c2* that has a number of neighbours higher than its slice size and then trys to replace a neighbour from *c2* (that does not also belong to *c*) for a new neighbour from its passive view that belongs to *c*. This method is depicted in Listing 4.8.

```
optimize()
{
    Array[Category] Top−Categories = resource−Index.get−Top−T−Categories();

    for(int i=0; i < active−View.size(); i++)
    {
      Node neighbour = active−view.get(i);

      if(!neighbour.belongs−To−At−Least−1−Category(Top−Categories))
      {
        //Try to replace this neighbour for another from the passive view that belongs to
        //at least one of the top categories
        active−View.replace−Neighbour(i);
        return;
      }
    }

    for(Category ``c'' in Top-Categories
    {
      if(active-View.number-Of-Neighbours-In-Category(``c'')
           < active-View.get-Slice-Size(``c''))
        {
          if(!active-View.is-Full())
          {
            //if the active view is not yet full attempt to get
            //more neighbours belonging to ``c''
            active-View.get-More-Neighbours-From-Category(``c'');
            return;
          }
          else
          {
            for (Category ``c2'' in Top-Categories)
            {
              if(active-View.number-Of-Neighbours-In-Category(``c2'')
                   > active-View.get-Slice-Size(``c2''))
                {
                  //if a category ``c2'' has more neighbours than its slice size, and a
                  //category ``c'' has less neighbours than its slice size, try to replace a
                  //neighbour from ``c2'' for a neighbour that belongs to ``c''
                  this.swap-Neighbour(``c2'',''c'');
                  return;
                }
            }
          }
        }
      }
      return;
}
```

Listing 4.8: The optimize() method (Resource Location in File Sharing Systems Instantiation)

Note that the approach above avoids to over-bias the overlay, preventing the overlay from having an excessive clustering coefficient and the emergence of cliques that may hamper the connectivity of the overlay, excluding nodes from processing and replying to queries.

Structured Layer and Consul Election

In this scenario, nodes can only compete to become c-consuls for the category c that is their topmost category (i.e. the category with the highest fraction in a node's active view). Therefore, for this scenario, the *get-Categories-Node-Can-Represent()* method returns only the node's topmost category. This policy attempts to ensure that a consul for c has enough neighbours that belong c not to run the risk of receiving a query for c and not being able to forward it because all its neighbours that belong to c have suddenly failed. In addition, regarding the query routing procedure, when trying to locate rare resources, it could be important to start several random-walks, for each category c that the query targets, in the unstructured layes instead of only a single one. Having nodes only being able to represent their topmost category, effectively makes this feasible as they will have more neighbours to initiate these random-walks.

Summary

This chapter presented *Curiata*, a scalable and efficient resource location architecture that employs self-organizing techniques to integrate and combine the benefits of both structured and unstructured overlay networks. We have presented the basic building blocks of *Curiata*'s architecture. Furthermore, two possible instantiations for the *Curiata* architecture were identified. For each of these instantiations, we have presented the implementation of customizable methods, based on the characteristics of each target environment.

5 Evaluation

This section presents the results of the evaluation process for the *Curiata* architecture. The performance of the proposed solution was evaluated by simulation using the Peersim(Jelasity, Montresor, Jesi, & Voulgaris 2009) simulator. We will provide evaluation results for both the cloud computing infrastructure scenario (presented in 4.2.1) and also for the peer-to-peer file sharing system scenario (presented in 4.2.2).

5.1 Cloud Computing Infrastructure

This scenario was based on the pre-configured instances provided by Amazon Web Services(http://aws.amazon.com/ 2006). In the experimental setup, considered 17 distinct resource categories were considered (5 categories for different sizes of each resource type – CPU, Memory, and Disk – and two categories for the CPU architecture – 32 and 64 bits), each node belongs to 4 of these categories (one per resource type) and maintains 30 unstructured overlay neighbors ($d = 30$). Finally, the radius to which nodes announce their presence in the DHT was set to 3 hops ($r = 3$). Each physical host, in the simulated cloud computing infrastructure, owns twice the resources of the most powerful pre-configured virtual machine used by the Amazon Web Services.

Our system was evaluated for two types of queries. In the first experimental setting approximately 20.000 virtual machines were allocated with random configurations (from the ones employed by Amazon) over 10.000 physical nodes in order to promote the heterogeneity of available resources among nodes. We then executed 4.576 queries initiated from random nodes in the system, trying to locate a machine with free resources in 1, 2, 3, and 4 different categories (1.144 queries of each type were executed). The k parameter was set (associated with the cost of performing random walks) to 48, therefore 48 nodes is the maximum number of nodes that can be returned by a query.

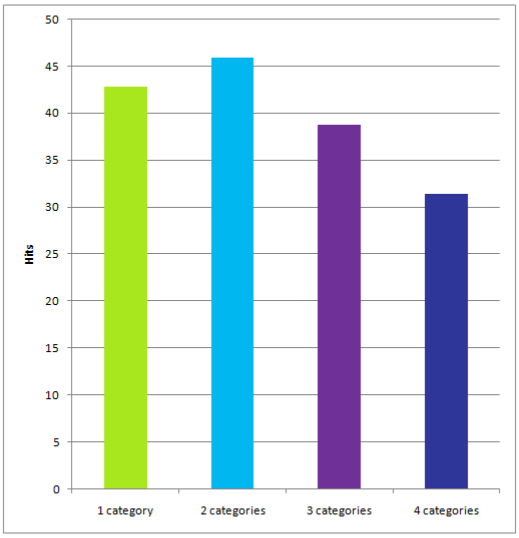


Figure 5.1: Number of Hits for Random Queries

The goal was to evaluate the number of nodes returned by each query (*i.e.* number of hits) and the cost, in number of messages, of disseminating queries targeting different number of categories. Results are reported in Figures 5.1 and 5.2 respectively. On average, queries are able to return a large set of answers. The number of nodes returned by queries mostly decreases with the increase of associated categories. This is expected, as a smaller number of nodes in the system have adequate resources to reply to such queries. However, even when performing queries for 4 resource categories, our approach is able to find more than 30 nodes that fit the requirements. Interestingly a query for a single resource category returns less elements than a query that targets 2 categories. This likely happens because, in such scenario only a single random walk is employed which can transverse, and exit, the overlay region biased for that category, being unable to locate additional valid answers afterwards.

Figure 5.2 reports the message cost for disseminating each type of query. As expected the cost in number of messages slightly increases as the number of categories rises. This happens due to the additional cost of forwarding one message through the DHT for each query category. Notice however, that the cost does not rise linearly, as the cost of performing random walks remains constant ($k = 48$). Moreover, notice that, even when performing a query for 4 categories, the additional cost (*i.e.* the number of messages above 48) is, on average, below 15 additional messages.

We have also performed experiences in an additional setting where queries target two re-

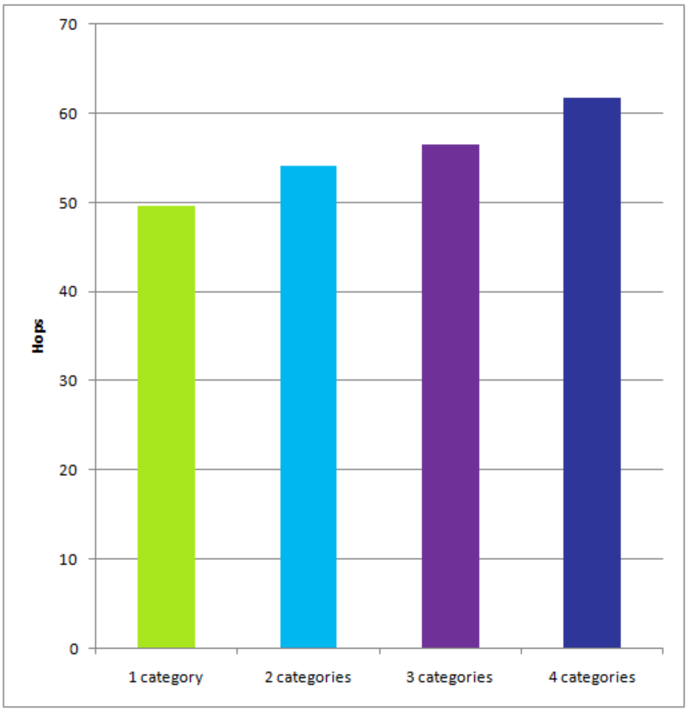


Figure 5.2: Message cost for Random Queries

source categories being each of these categories very popular individually, but where the intersection of both is rare. Notice that this is a scenario that presents a relevant challenge to our system, as the DHT will route queries to nodes belonging to each of the categories individually, starting random walks in regions of the unstructured overlay network that are biased for each category independently.

To this end, our system was configured to have half of the computational nodes belonging to category *64bits* (*i.e.* 5.000 nodes). We then varied the fraction of these nodes that also have the category *memL*. 5.000 individual queries for each tested configuration were issued. We only routed a single query in the DHT to a representative of the *64bits* category, as this is the worst entry point for queries in this scenario. We measured the number of hops required to find to first, second, and third nodes that belong to both categories.

Results are depicted in Figure 5.3. As expected, as the percentage of nodes belonging to *64bits* that also belong to *memL* diminishes, the random walk requires additional hops to find the region in the unstructured overlay where nodes belonging to both categories are clustered. This happens because the queries are injected into the unstructured layer at representatives of a single category therefore it is possible that the region of the unstructured overlay where the

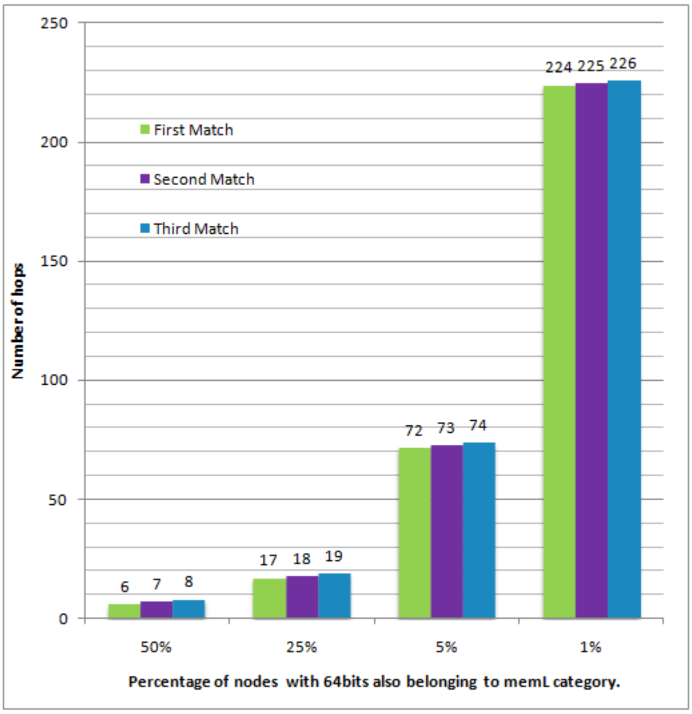


Figure 5.3: Hops for Rare Intersections

query is injected is only biased for *64bits*. Such limitation can however be easily circumvented, by allowing representatives of multiple categories to emerge. This question will be addressed as future work. Notice however that the biasing of the unstructured overlay network allows our system to find the second and third hit for the query in the hops immediately after locating the first hit.

5.2 P2P File Sharing Systems

In order, to demonstrate the importance of each module in the *Curiata* architecture, we compare the performance of the following four different sub-architectures:

Random walks on a Random Topology (RWRT): This corresponds to a system that uses only a random unstructured overlay, where no self-organizing background process is employed. In this architecture, there is no DHT and peers are not defined by the similarity of their resources.

Biased Random Walks on a Random Topology (BWRT): This corresponds to a system that uses uses a random unstructured overlay, where no biasing is applied. However, in this architecture, queries (*i.e.*, disseminated through random walks) are guided using a mechanism similar to the one used in *Curiata*.

Biased Random Walks on a Biased Topology (BWBT): This corresponds to a system that uses an unstructured overlay that is biased by the same self-organizing background process employed in *Curiata*. In this architecture the queries are also guided. However no DHT is used to route queries.

Curiata: This implements the full architecture as described in section 4.2.2. In this architecture, peers self-organize to bias their neighbors according to the similarity of their resources and *consuls* are elected and organize themselves in a DHT.

Queries

Each query is implemented using a single k-length guided random walk with $k = 128$ and $k = 256$. The length of the random walk is counted from the peer that originates the query. Thus, all hops are considered, including the (non-random) hops required to reach the nearest consul, and then to route the query in the DHT until a representative of the searched category is found.

To simplify the analysis of the results, each query in our simulation searches for a single resource. Since each resource is associated with a single category, this category is used to guide the query, both in the BWRT architecture (to guide the walk in the unstructured layer), in the BWBT architecture and in the full *Curiata* system (both in the unstructured layer and when routed in the DHT). Note that the fact that resources have unique identifiers is an artifact of the simulation. As noted before, *Curiata* is aimed at a system where users perform complex queries. Thus, a search for a specific resource simulates any complex query that is satisfied only by the resource with that identifier.

Metrics

Most of the experiments executed in this scenario use the following metrics:

- *Success rate:* The percentage of queries that found at least one copy of a given resource.
- *Recall rate:* The percentage of copies of a given resource that are found by a query (with regard to the total number of copies of that resource in the system).

Category	Nb. of peers	Unique Resources	Category	Nb. of peers	Unique Resources
A	5000	5	G	75	5
B	2500	5	H	40	5
C	1250	5	I	30	5
D	625	5	J	20	4
E	300	5	K	10	2
F	150	5			

Table 5.1: Network Distribution

- *Latency:* The number of hops required to find x copies of the resource. In most scenarios latency values to find 1, 2, and 3 copies of the resource are provided.

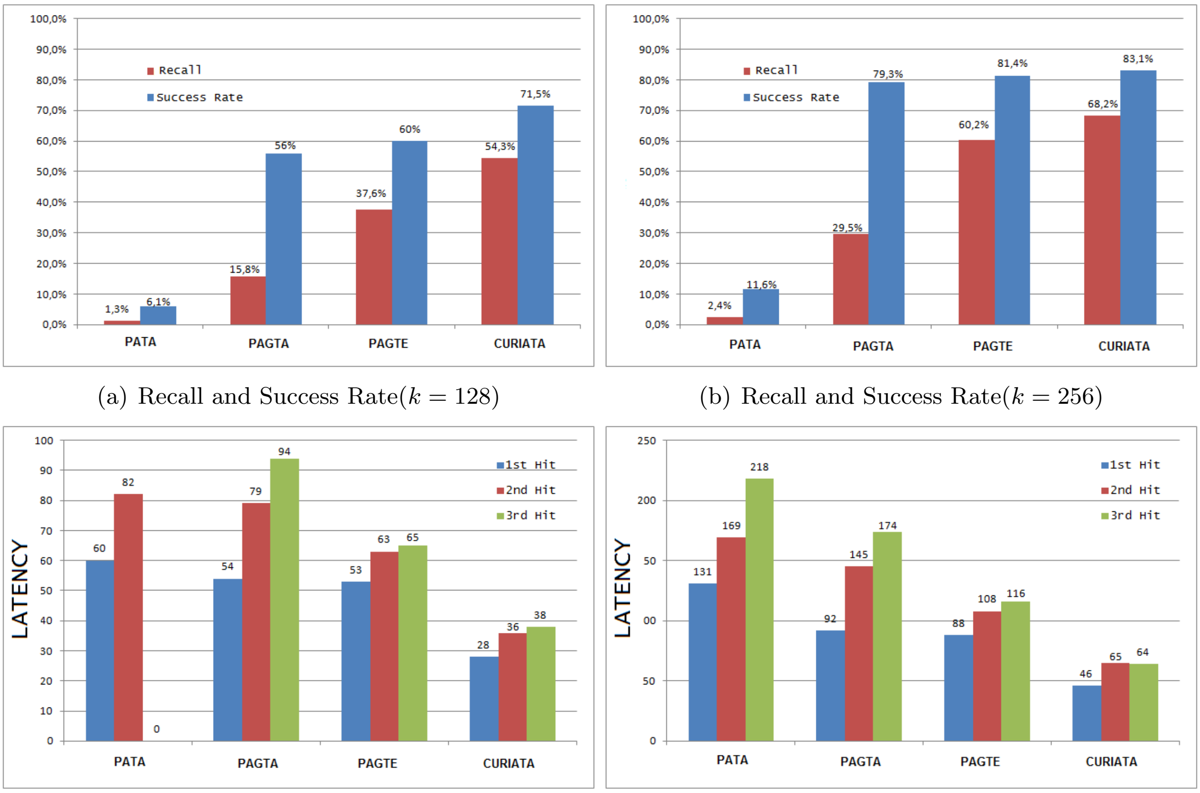
5.2.1 Experimental Setup A

For the first experiments a network of 10.000 peers was used and 11 resource categories were defined. Each node in the system is assigned 1 category out of the 11. Resources in the network are associated with a single category and have a unique identifier. Resources of each category c are randomly allocated to peers of that category, such that each resource exists in 5 distinct nodes. Nodes were configured to have 20 neighbors in the unstructured overlay network. Table 5.1 summarizes the number of peers in each considered category and the total number of unique resources associated to each category. The radius of the regions for consul election in the curia layer is set to 2 and in this scenario approximately 200 peers out of the 10.000 are elected as consuls and join the structured layer. Furthermore, each category falls over one of 3 popularity ranks as follows: *Rare Categories:* categories with less than 100 peers (Categories G to K); *Intermediate Categories:* categories with 100 to 1.000 peers (Categories D to F); *Common Categories:* categories with more than 1.000 peers (Categories A to C).

In each experiment 10.000 distinct queries were issued, starting at randomly select nodes that targeted a single existing resource. Further ahead we detail performance values for queries issued for common and rare popularity ranks.

Overall Performance Results

In this section, we present the overall results for queries in the scenario described above. Results were not discriminated by category as the goal of these results is to provide an overall overview of the system's performance.



(a) Recall and Success Rate($k = 128$)

(b) Recall and Success Rate($k = 256$)

(c) Hop Count($k = 128$)

(d) Hop Count ($k = 256$)

Figure 5.4: Overall Performance Results

As depicted in Figs 5.4(a) and Fig 5.4(b), *Curiata* outperforms the alternative architectures, both in terms of success and recall rates. The recall for RWRT almost doubles when k is increased from 128 to 256 as random walks can explore twice the number of nodes. Note however, that because the performance of *Curiata* is better than other solutions with a $k = 128$. This is a clear indication that *Curiata* can achieve good performance in terms of recall and success of queries using more conservative k values.

Figs 5.4(c) and Fig 5.4(d) present latency values. Due to the DHT initial routing of queries *Curiata* has significant lower latency times. Notice that the lack of DHT support leads to a scenario where no significant differences can be found in the number of hops required to find the first hit for a query. Additionally, the self-organizing topology of the curia combined with the DHT routing allows *Curiata* to present a much lower number of hops required for locating the second and third hit of each query.

Since the success rate is not 100%, when thee the value of k changes from 128 to 256, the hop count increases. This happens because a larger number of queries are able to locate resources with success. Note that in these results only the hop count of successful queries is taken into account (this explains why, in Fig 5.4(d), the hop count for the third hit has lower values than the second hit hop count when using a $k = 256$ using the *Curiata* system).

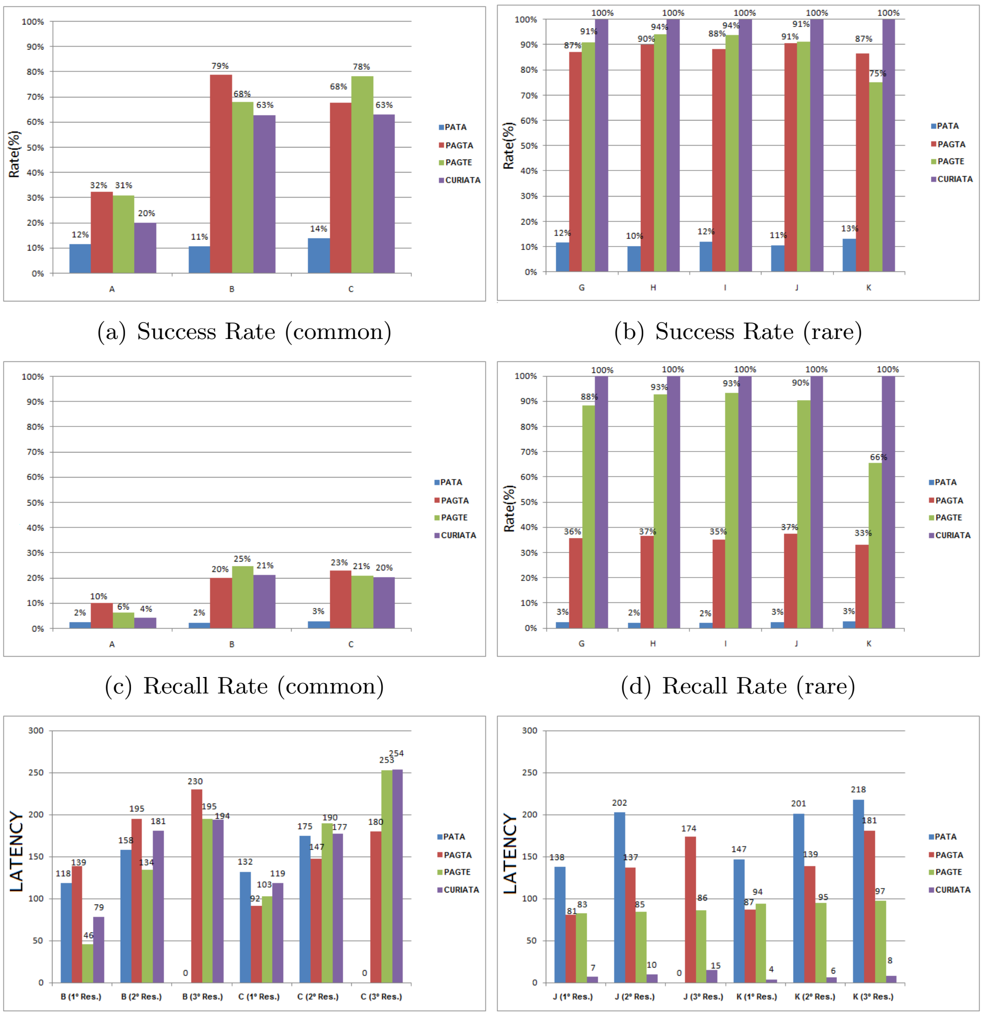
As expected, the performance of the evaluated solutions vary for queries that target items belonging to categories with different popularity (*i.e.*, with a variable number of nodes belonging to it). In order to evaluate the effects of resource category popularity, in the following we provide detailed results for different scenarios.

Performance According to Category

Fig 5.5 presents results for queries that target resources in rare categories (namely, categories G, H, I, J, and K) and common categories (categories A, B, and C). Due to space constrains only plot results of $k = 256$ were plotted, as this is the better scenario for the remaining considered solutions.

Comparative performance results show that *Curiata* brings no advantages when looking for resources in common categories. This happens because the DHT is unnecessary when the categories are very popular (given that resources from common categories are available in every region). In fact, for these categories, the recall and success rate results of *Curiata* are very similar to the ones obtained with BWBT and BWRT.

In sharp contrast, *Curiata* excels when searching resources from rare categories. For those resources, *Curiata* can achieve both a perfect recall and success rates and outperforms the remaining architectures in terms of latency. In the evaluated scenario, *Curiata* can locate every resource. Fig 5.5(f) depicts detailed results for latency only for the rarest categories J and K. Results show that *Curiata* offers a significant latency gain when compared to the other alternatives. This steams from the DHT routing that places queries in the relevant region of the self-organizing unstructured overlay. Since these regions are small, our system can easily visit all relevant nodes, effectively locating resources that are targeted by the query.



(a) Success Rate (common) (b) Success Rate (rare)

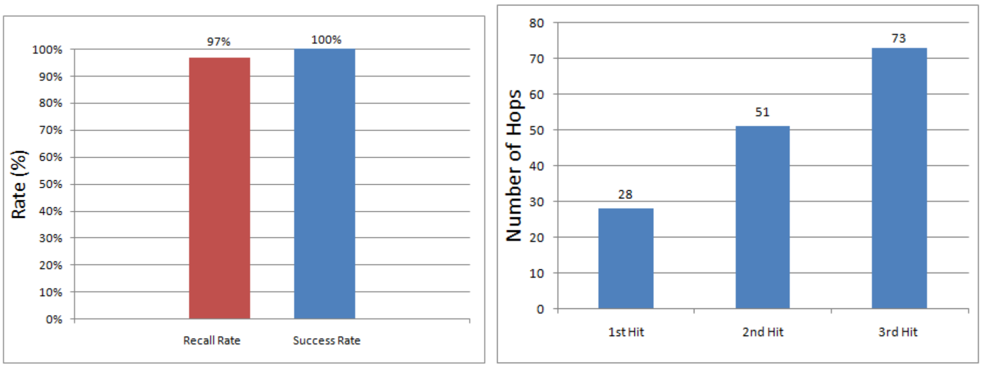
(c) Recall Rate (common) (d) Recall Rate (rare)

(e) Hop Count (common) (f) Hop Count (rare)

Figure 5.5: Performance Results of Queries for Rare and Common Categories

5.2.2 Experimental Setup B - Syntetic Dataset

Curiata was also tested on a more realistic scenario, simulating a P2P file-sharing application, like Gnutella(Tsoumakos & Roussopoulos 2006) or KaZaa(www.kazaa.com 2000). This setting used a network of 10.000 peers with an average degree of 20, ranging from 15 to 30. The degree was a function of the total number of resources maintained by each individual peer. In this experimental setup 30.000 unique resources were distributed and replicated using a zipf like distribution with an α factor of 0.6, as suggested in(Klemm, Lindemann, & Waldhorst 2004). Therefore, there was a total of 1.500.000 resources in the system. Resources were classified into 24 different categories (which were divide into 4 popularity ranks similarly to the previous scenario). Resources were distributed at random among peers in the system, so that each peer



(a) Success and Recall Rate (b) Latency

Figure 5.6: Performance for a replication rate of 0.1% ($k = 512$)

had resources from 1 to 3 categories. Queries that search for resources with a replication rate of 0.1% were then executed. These are rare resources in our setup as there are only 10 copies in the network (out of 1.500.000). In this setup a k was set to a value of 512.

Experimental results have shown (Fig. 5.6(a) that *Curiata* was able to attain a recall rate of 97% and a perfect success rate (100%). Furthermore, Fig. 5.6(b) depicts the average number of hops for the first, second, and third hit of each query. Notice that these values are relatively small, even when looking for resources that are only present in 10 nodes.

5.2.3 Experimental Setup C - The Last.fm Dataset

Last.fm is a popular Internet radio site for streaming music, founded in the United Kingdom in 2002. It has claimed over 40 million active users based in more than 200 countries. Last.fm builds a detailed profile of each user's musical taste by recording details of the songs the user listens to.

Last.fm supports user-end tagging or labeling of artists, albums, and tracks to create a site-wide folksonomy of music. Users can browse via tags, but the most important benefit is tag radio, permitting users to play music that has been tagged a certain way. This tagging can be by genre, mood, artist characteristic, or any other form of user-defined classification. However, since tagging is not moderated, it is not accurate or even consistent.

In this evaluation scenario, we retrieved data from 10.000 Last.fm user profiles and their top-50 most listened to tracks (note that new or occasional users might not have listened to 50 tracks yet). A simulation setup was prepared, where each peer shares the 50 most listened

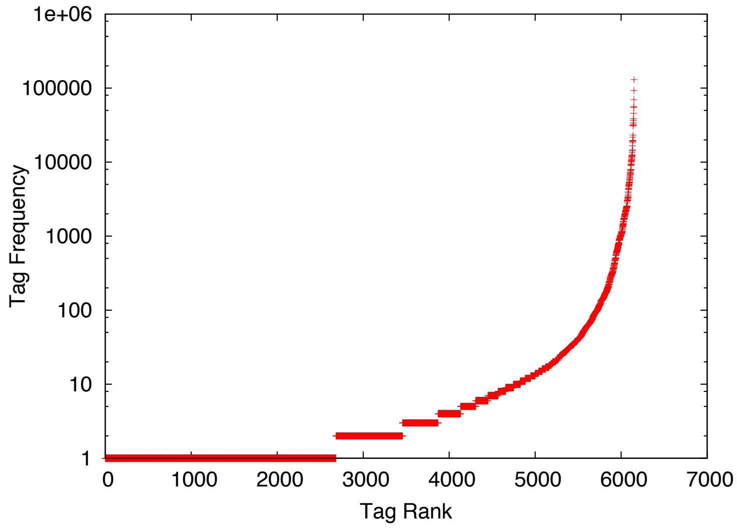


Figure 5.7: Tag Frequency for Resources

tracks of a Last.fm user, using the associated tags as *Curiata* categories. In particular, the top 3 tags most voted of each track were used to categorize them.

Analysis of the retrieved data shows that for these 10.000 users there were 197.018 unique music tracks (from a total of 492.061 tracks), and these tracks were tagged by Last.fm users with no less than 6.150 different tags (considering only the 3 most voted tags of each track). In addition, the average number of tags per user was approximately 23.9 (note that this regards only the top-50 most listened to tracks by each user). Fig. 5.7 depicts the frequency of tags in all tracks. Out of curiosity, the most common tag is “Alternative” with 129.905 tagged tracks, the second one is “Indie” with 92.830 tracks and the third one is “Hip-Hop” with 69.686.

This presents a challenging scenario for *Curiata*. Firstly, because there are some very popular tags/categories which a lot of peers belong to. This causes the search region for such categories to be very large and therefore a low value of TTL might not be enough to find the desired resources. In addition, there are in average approximately only 2.5 copies of each resource in a universe of 10.000 users, which represents a very low replication factor of each file. Furthermore, the number of categories that each user belongs to is high (23.9 in average), especially considering only the top-50 most listened to tracks are being used. This high number of categories per peer makes it too costly for each peer to establish connections for all categories, causing some categories to be ignored during the biasing process. This may cause some resources to be unreachable since they do not belong to the top categories of their respective owners. For the system simulation,

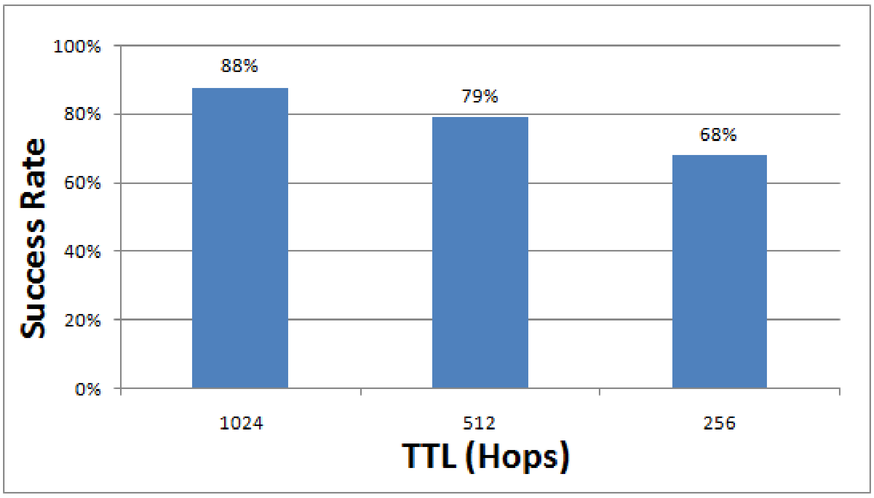


Figure 5.8: Success Rate for various TTL values

an active view size of $d = 45$ was used, peers biased their active views based on their top-15 tags/categories (the ones they had the most resources of) and can compete to become consuls for any of these 15 categories. The minimum number of neighbours for each category was set to 3 ($smin = 3$) and the radius of the consul beacons was $r = 3$. This setup led around 1.400 peers to join the structured layer (DHT) in which 864 categories were represented. In addition, 1.063 resources (out of the 492.061) had none of their categories represented on the structured layer and therefore were unreachable using *Curiata*'s search protocol (although, they could still be reached using only a biased random walk). 10.000 queries were then issued, originated in random nodes and targeting unique random resources. We experimented with different values of TTL and obtained the results for the success rate (percentage of queries that found at least 1 matching resource) of such queries. These queries targeted only a single music track and a copy of the query was routed towards each category of the resource that was represented in the DHT. Each of these copies would be disseminated with a certain value of TTL. Results for the sucess rate can be found in Fig. 5.8.

As expected, the success rate increases with the TTL value. This is mainly due to the existence of some very popular categories such as "Alternative" which 6.731 peers have in their top-15 categories or "Hip-Hop" which 4.699 peers have in their top-15 categories. Thus, the search region for these categories is huge and it is not easy to find copies of the target resources (note that the song tracks have in average 2.5 copies in the whole system) using these categories. However, when other (less popular) categories for the target resource are also represented in the structured layer, *Curiata* can use those categories, which have smaller regions in the unstructured overlay, to find matches quicker. Therefore, even with a TTL of 256, the success rate is already close to 70%. To sum up, as mentioned previously Last.fm represents a very challenging scenario for any P2P resource location system. However, although *Curiata* did not achieve a perfect success rate, the results are encouraging, given the scenario used. More specifically, it is important to notice that resources have a very low replication rate (only 2.5 copies in average), and so this scenario strengthens the idea that *Curiata* can achieve a high hit-rate for rare resources.

5.2.4 Discussion and Comparison with GIA

Ensuring good recall for rare resources has been a relevant challenge for previous work in P2P resource location (for instance see (Chawathe, Ratnasamy, Breslau, Lanham, & Shenker 2003)). *Curiata* satisfactorily addresses this scenario. It does so without imposing constraints on the type of queries, as long as it is possible to extract the set of categories from the query. This is in opposition to DHTs, which only excel for exact-match queries. *Curiata* is also effective, but not substantially better than other schemes when searching for rare resources in common categories.

As seen using the last.fm dataset (Sec. 5.2.3), this means that, when looking for a resource tagged with multiple categories, much better results are obtained when the rarest categories are used. Thus *Curiata* offers the best results when combined with a complementary monitoring scheme (in tandem with the mechanisms employed in GIA (Chawathe, Ratnasamy, Breslau, Lanham, & Shenker 2003)) that is able to assert which categories are most/less popular. Using such information, *Curiata* could route queries for rare categories via the DHT but bypass this step for common categories.

Furthermore, in scenarios with a large number of categories, such as the last.fm dataset where there were 6.150 different categories (in 10.000 peers), a small number of resources might be unreachable using *Curiata*'s efficient resource location strategy, as none of these resource's categories are represented in the Structured Layer. However, simpler query routing strategies such as a Random-Walks may be employed in order to reach such objects.

It is interesting to compare the results obtained in the syntetic dataset (Sec. 5.2.2) with results obtained with other unstructured P2P resource location strategies. For instance, GIA (Chawathe, Ratnasamy, Breslau, Lanham, & Shenker 2003) reports a latency of 15 hops to locate a resource with a replication rate of 0.1%. This value is lower than the result achieved with *Curiata*. For a replication rate of 0.5%, GIA needs a little more than 2 hops to locate the resources. In *Curiata*, for the same replication rate, our approach needs around 7 hops for resources in rare categories and 46 hops in common categories. However, *Curiata*, in fact, has a much smaller maintenance cost than GIA. First, in GIA nodes need to maintain replicated indexes of their one-hop neighbors. Secondly, nodes in *Curiata* operate with a much smaller degree than GIA (the maximum number of neighbors in our experiments is 30, whilst in GIA it is 128). Therefore, in GIA a small portion of nodes (the "high-capacity" nodes) are required to maintain indexes for a large portion of peers.

Summary

This section presented and analyzed the evaluation results for the *Curiata* architecture. First, the results for the cloud computing infrastructure scenario were presented. That scenario was based on pre-configured instances by Amazon Web Services. Then we presented results for 3 distinct datasets related to file sharing P2P systems. The most interesting scenario was perhaps the Last.fm dataset which allowed us to discover and analyze some of the issues associated with the performance of our system under realistic scenarios, such as inaccurate and inconsistent tagging.

6 Conclusions and Future Work

This thesis proposed a new solution for the problem of supporting efficient search in large-scale peer-to-peer systems. Our solution combines the usage of systems based on both unstructured and structured overlay networks in order to achieve a more efficient and flexible solution for locating resources in a P2P environment. This solution has been named *Curiata*, and aims at supporting flexible querying, like most unstructured solutions, while retaining the speed and efficiency provided by structured (DHT-based) solutions.

The architecture was applied to two different case-studies: a generic file sharing system and a resource discovery and allocation system for the cloud. The performance of the resulting systems has been extensively evaluated using simulations. The results have highlighted the benefits of our approach, namely:

- The ability to quickly and efficiently locate resources in rare categories which typically represents a challenging, sometimes troublesome, subject in P2P resource location systems.
- The ability to effectively approximate nodes that share resources belonging to the same categories and the ability to leverage on this network organization to quickly find additional results for the queries issued by participants.
- Achieving high recall and success rates even when using reduced values of TTL in queries.

And also its limitations:

- The additional overhead introduced by the structured layer when searching for resources in very common categories (containing large regions of the overlay).
- Under scenarios with very high category heterogeneity, some objects may be unreachable through efficient routing.

As future work we would like to deploy our architecture on a real peer-to-peer network. In addition, we are planning on employing the Cubit(Wong, Slivkins, & Sirer 2008) DHT in Curiata to develop a decentralized tracking and torrent search infrastructure that can cope with user errors when describing and tagging torrents and content.

Bibliography

Artigas, M. S., P. G. Lopez, J. P. Ahullo, & A. F. G. Skarmeta (2005). Cyclone: A novel design schema for hierarchical dhts. *Peer-to-Peer Computing, IEEE International Conference on 0*, 49–56.

Bawa, M., G. S. Manku, & P. Raghavan (2003). Sets: Search enhanced by topic segmentation. In *In SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pp. 306–313. ACM Press.

Berry, M. W., Z. Drmac, & E. R. Jessup (1999). Matrices, vector spaces, and information retrieval. *SIAM Rev. 41*(2), 335–362.

Bharambe, A. R., M. Agrawal, & S. Seshan (2004). Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev. 34*(4), 353–366.

Brandt, J., A. Gentile, J. Mayo, P. Pebay, D. Roe, D. Thompson, & M. Wong (2009, 23-29). Resource monitoring and management with ovis to enable hpc in cloud computing environments. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1 –8.

Breitkreuz, H. (2002). emule.

Buford, J., H. Yu, & E. K. Lua (2008). *P2P Networking and Applications.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Byers, J. W., J. Considine, & M. Mitzenmacher (2003). Simple load balancing for distributed hash tables. In *IPTPS*, pp. 80–87.

Cai, H., P. Ge, & J. Wang (2008). Applications of bloom filters in peer-to-peer systems: Issues and questions. In *NAS '08: Proceedings of the 2008 International Conference on Networking, Architecture, and Storage*, Washington, DC, USA, pp. 97–103. IEEE Computer Society.

Chawathe, Y., S. Ratnasamy, L. Breslau, N. Lanham, & S. Shenker (2003). Making gnutella-like p2p systems scalable. In *SIGCOMM '03: Proc. of the 2003 conference*, New York, NY, USA, pp. 407–418. ACM.

Cohen, B. (2003). Bittorrent.

Cohen, E., A. Fiat, & H. Kaplan (2003). Associative search in peer to peer networks: Harnessing latent semantics.

Crespo, A. & H. Garcia-Molina (2002). Routing indices for peer-to-peer systems. In *Proc. of the 22nd ICDCS'02*, pp. 23–32.

Danezis, G., C. Lesniewski-laas, M. F. Kaashoek, & R. Anderson (2005). Sybil-resistant dht routing. In *In ESORICS*, pp. 305–318. Springer.

Deerwester, S., S. T. Dumais, G. W. Furnas, T. K. Landauer, & R. Harshman (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science 41*, 391–407.

Flanning, S. (1999). Napster. http://www.napster.com.

Ganesan, P., K. Gummadi, & H. Garcia-Molina (2004). Canon in g major: designing dhts with hierarchical structure. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pp. 263–272.

Halim, F., Y. Wu, & R. H. C. Yap (2008). Small world networks as (semi)-structured overlay networks. *Self-Adaptive and Self-Organizing Systems Workshops, IEEE International Conference on 0*, 214–218.

http://www.coralcdn.org/ (2004). Coral.

http://www.w3.org/PICS/DSig/SHA1_1_0.html (1995). Sha-1.

http://www.wowwiki.com/Blizzard_Downloader (2006). Blizzarddownloader.

Jelasity, M., A. Montresor, G. P. Jesi, & S. Voulgaris (2009). The Peersim simulator. `http://peersim.sf.net`.

Klemm, E., C. Lindemann, & O. Waldhorst (2004). Relating query popularity and file replication in the gnutella peer-to-peer network.

Korpela, E., D. Werthimer, D. Anderson, J. Cobb, & M. Leboisky (2001, Jan/Feb). Seti@home-massively distributed computing for seti. *Computing in Science and Engineering 3*(1), 78–83.

Leitao, J., J. Pereira, & L. Rodrigues (2007). Hyparview: A membership protocol for reliable gossip-based broadcast. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Washington, DC, USA, pp. 419–429. IEEE Computer Society.

Leitao, J. & L. Rodrigues (2008, December). Overnesia: an overlay network for virtual superpeers. Technical Report 56, INESC-ID.

Leitao, J. C. A., J. P. S. F. M. Marques, J. O. R. N. Pereira, & L. E. T. Rodrigues (2009). X-bot: A protocol for resilient optimization of unstructured overlays. In *Proc. of the 28th IEEE International SRDS'09*, Washington, DC, USA, pp. 236–245. IEEE Computer Society.

Leito, J., J. Pereira, & L. Rodrigues (2008). On the structure of unstructured overlay networks. In *In "Fast Abstract", Supplement of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE.

Lv, Q., P. Cao, E. Cohen, K. Li, & S. Shenker (2002). Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, New York, NY, USA, pp. 84–95. ACM.

Maymounkov, P. & D. Mazires (2002). Kademlia: A peer-to-peer information system based on the xor metric. pp. 53–65.

Rao, A., K. Lakshminarayanan, S. Surana, R. Karp, & I. Stoica (2003). Load balancing in structured p2p systems.

Ratnasamy, S., P. Francis, M. Handley, R. Karp, & S. Schenker (2001, October). A scalable content-addressable network. In *Proc. of the 2001 ACM SIGCOMM Conference*, Volume 31, New York, NY, USA, pp. 161–172. ACM.

Renesse, R. V., K. P. Birman, & W. Vogels (2001). Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems 21*, 2003.

Rhea, S. & J. Kubiatowicz (2002). Probabilistic location and routing. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, Volume 3, pp. 1248–1257 vol.3.

Rieche, S., L. Petrak, & K. Wehrle (2004, Nov.). A thermal-dissipation-based approach for balancing data load in distributed hash tables. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pp. 15–23.

Rowstron, A. I. T. & P. Druschel (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, London, UK, pp. 329–350. Springer-Verlag.

Saroiu, S., P. K. Gummadi, & S. D. Gribble (2002). A measurement study of peer-to-peer file sharing systems.

Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, & H. Balakrishnan (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, pp. 149–160. ACM.

Tang, C., Z. Xu, & S. Dwarkadas (2003). Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, pp. 175–186. ACM.

Totekar, C., M. Vani, & S. Palavalli (2008, Dec.). Search improvement in unstructured p2p network considering type of content. In *Networks, 2008. ICON 2008. 16th IEEE International Conference on*, pp. 1–4.

Tsoumakos, D. & N. Roussopoulos (2003, Sept.). Adaptive probabilistic search for peer-to-peer networks. In *Peer-to-Peer Computing, 2003. (P2P 2003). Proceedings. Third International Conference on*, pp. 102–109.

Tsoumakos, D. & N. Roussopoulos (2006). Analysis and comparison of p2p search methods. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, New York, NY, USA, pp. 25. ACM.

http://aws.amazon.com/ (2006). Amazon web services. http://aws.amazon.com/.

Voulgaris, S., D. Gavidia, & M. van Steen (2005, June). Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management 13*(2), 197–217.

Wong, B., A. Slivkins, & E. G. Sirer (2008, Dec.). Approximate matching for peer-to-peer overlays with cubit. Technical report, Computing and Information Science Technical Report, Cornell University.

Wu, J. (2005). *Handbook On Theoretical And Algorithmic Aspects Of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks*. Boston, MA, USA: Auerbach Publications.

www.kazaa.com (2000). Kazaa.

Xu, Z., R. Min, & Y. Hu (2003). Hieras: A dht based hierarchical p2p routing algorithm. *Parallel Processing, International Conference on 0*, 187.

Yang, B. & H. Garcia-Molina (1998). Efficient search in peer-to-peer networks. In *Proc. 22nd International Conference on Distributed Computing Systems, 2 - 5 July, 2002, Vienna, Austria*. IEEE Computer Society.

Zhang, R. & Y. Hu (2005, March). Assisted peer-to-peer search with partial indexing. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, Volume 3, pp. 1514–1525 vol. 3.

Zhao, B. Y., L. Huang, J. Stribling, S. Rhea, A. Joseph, & J. Kubiatowicz (2004). Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*, 41–53.