Efficient Content Locating in Dynamic Peer-to-Peer Networks

by

Jun Wang

B.Sc., University of British Columbia, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University of British Columbia

May 2005

Abstract

The Peer-to-Peer (P2P) computing model has recently been recognized as a more natural and flexible approach to sharing resources. However, a fundamental issue, *content locating* (or *content search*) in P2P-based applications has not yet been successfully resolved. This thesis documents the design and implementation of a novel architecture, which combines the advantages of cluster infrastructure and *Super-Peer* overlay to address the scalability, robustness and efficiency issues in existing *unstructured* P2P systems. The first component of our architecture is called the TBCP model, which constructs a set of interconnected clusters, where each cluster forms a bounded-depth tree, and each peer node acts as a tree leaf. The duties of maintaining cluster topologies and providing search services are separated and re-balanced to address heterogeneity. The COOL model, the second component of our architecture, then constructs light-weight interconnected overlay networks, by following simple classification and mapping rules. Peers are connected in both *vertical* direction (within the cluster domain) and *horizontal* direction (within the *Super-Peer* overlay domain). We also propose two search algorithms, known as *cluster search* and *overlay search*, which works seamlessly with our new models to provide efficient and low-cost content locating services. Our experiments and analysis prove that our new architecture achieves significantly better scalability and efficiency than a basic *unstructured* model.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, Dr. Son Vuong, for his wonderful guidance and support. I would not be here without his inspiration and encouragement. I thank Dr. Charles Krasic for his helpful revisions and feedback. I would also like to express my gratitude to Xin Liu. He participated actively in this project and gave me numerous ideas and help. I thank Juan Li, Ben D'Andrea, and all NIC group members for their help and collaboration.

I gratefully acknowledge the financial support from the Computer Science Department at UBC in the forms of TA and RA as well as the University Graduate Fellowship from the Graduate Study at UBC.

Finally, I thank all my family members for their consistent support during my study.

JUN WANG

The University of British Columbia

May 2005

Chapter 1

Introduction

The Peer-to-Peer (P2P) computing model has recently been recognized as a more natural and flexible approach to sharing resources, compared to the traditional Client-Server model, which was dominant in the last 15 to 20 years. A large variety of P2P applications, ranging from simple file-sharing to more advanced computing cycle-sharing, has been implemented in numerous personal computers. This thesis describes novel overlay architecture, along with its associated search algorithms, for providing an efficient and low-cost content locating service for *unstructured* P2P systems.

1.1 Motivation

One fundamental but challenging problem in current distributed computing research is to locate resources efficiently in large-scale distributed systems (in this thesis, we use the words *resource* and *content* interchangeably). Here we define the term *locate* as looking up the precise location for a particular content, with partial or incomplete information.

Content locating on the Client-Server model is better described as a search engine service. As of today, popular search engines, such as Google [10] and Yahoo [43], appear to be the dominant and favoured solutions for web content searching and browsing. This centralized approach simplifies data collection, maintenance and analysis, as well as improves search efficiency and accuracy. Meanwhile, it preserves scalability due to the clustering of commodity machines to provide various services.

However, as the amount of resources that people need to process reaches the Petabyte level, and as more resources now come directly from the edge of regular networks (in other words, more and more content now comes from regular PCs instead of from well-connected server machines), the Client-Server model is no longer the most effective approach to sharing resources for the following reasons: (1) Numerous dedicated web servers need to be deployed and maintained to provide enough storage space, which implies a very high deployment cost. (2) It is not ideal to enforce access to dedicated web servers to share user content. (3) Central servers usually have scalability and service availability problems. (4) Web servers were not designed to encourage collaboration. The lack of collaboration and communication determines that certain types of resources, such as computing cycle, cannot be easily shared among different servers.

The Peer-to-Peer paradigm, on the other hand, dictates a fully distributed design in which participating nodes (or client PCs) autonomously construct a communication overlay on top of the existing Internet platform. As each node is assigned equivalent roles (i.e., both client and server roles), this symmetric model accumulates a vast amount of unused resources, and thereby can be easily adapted to satisfy the requirements of many distributed computing applications and is able to provide virtually unlimited scalability,

storage space and computing power, with fairly low cost. However, the P2P model introduces several new challenges to traditional searching techniques:

- Storing and accessing decentralized resources on a peer node implies operating in an environment of unstable connectivity. Resource availability tends to be much lower than it is on web servers; therefore, any search technique based on maintaining cache or index tables must update local knowledge frequently. This is infeasible in large-scale centralized search engines as maintenance traffic can easily overwhelm regular search traffic.
- Peer nodes have significant or total autonomy. In an *unstructured* P2P model, they are self-organized into a complex overlay topology. Moreover, peer nodes exhibit significant heterogeneity [32]. This complexity and heterogeneity can cause serious network congestion as well as node overloading problems, and therefore must be resolved in any effective P2P-based search algorithms.
- Information or resources are distributed unevenly and anonymously among participating machines. Any attempt to collect and store all the information on one or very few machines will cause a single point of failure problem as well as decrease system scalability.

Today, most popular Peer-to-Peer applications, such as Gnutella [11], FreeNet [8] and KaZaa [18], are built on an *unstructured* model. In this model, peer nodes discover each other and connect in an ad-hoc fashion. No central server is deployed to control either network topology or content distribution. Each node is responsible for maintaining its own resources as well as processing and replying to search requests. According to a recent study [31], this application consumes a considerable amount of Internet traffic.

The primary goal of this thesis is to explore ad-hoc infrastructures that can provide *efficient* and *low-cost* content look-up services to arbitrary *unstructured* networks. Here we define *efficiency* as the number of messages being forwarded and the number of nodes being visited to process a single request. We also define *cost* as the size and quantity of maintenance messages required to keep search *efficiency* at a high level.

Currently, search methods on *unstructured* P2P networks can be categorized as either *blind* or *informed* [38]. In a *blind* search, each peer node maintains only information about its own resources. To locate all relevant resources in such networks, flooding is usually adopted to deliver a request to each individual node in the networks. However, flooding generates a large amount of redundant messages and thereby restricts system scalability. On the other hand, with an *informed* approach, caching and indexing techniques are used to provide centralized or distributed directory services. Queries are forwarded only, based on a heuristic model, to peer nodes that are likely to hold requested information. These systems tend to suffer from high maintenance costs (for exchanging and processing local indices) as well as low predication accuracy (for unpopular content).

This thesis documents a novel architecture that attempts to provide system support to solve various problems in the above two search methods. The clustering technique (in the TBCP model) is used to reduce system complexity, while a *Super-Peer* overlay architecture (in the COOL model) is used to provide inter-cluster connectivity. But unlike a traditional Super-Peer approach, in which each powerful node controls a set of regular nodes, in our new architecture, every regular node can act as a *super peer* simultaneously, and multiple overlays are constructed to connect related *super-peers*, based on a dynamic mapping and selection rule. These two models extend both the *blind* and *informed*

searches in hierarchical and hybrid directions, as a means of finding a fairer, more balanced and more efficient search algorithm.

1.2 Thesis Contributions

This thesis provides the following contributions[1]:

- We propose a cluster-based ad-hoc P2P infrastructure (the TBCP model) to operate on *unstructured* networks. This new hierarchical level and the related concept of separating search traffic from system maintenance is intended to reduce system complexity and better balance workload on regular peer nodes.
- We design a restricted flooding algorithm, *cluster search*, which works with the TBCP model to reduce duplicated messages. With its advantage of a well-connected and tightly constructed structure, this algorithm can significantly reduce network traffic.
- We introduce the concept of category mapping, which is a classification rule that intends to associate the category domain (semantics) with the *unstructured* system domain.
- We propose a semantic-based hybrid P2P infrastructure (the COOL model), which integrates two types of overlays (*cluster overlay* and *category overlay*) and provides support to *search within context*. Load balancing is further improved by separating searching and indexing traffic.
- We combine the TBCP model and the COOL model to construct a multiple

[1] Parts of this work have also recently been accepted for publication [20].

super-peer overlay architecture, which better integrates system resources and offers a more balanced workload.

- We present an *overlay search* algorithm, which provides an efficient content locating service through our integrated architecture. It extends both the *cluster search* and *informed* search schemes, with the purpose of ensuring both search coverage and efficiency.
- We developed a general purpose P2P application framework, which separates system architecture from specific service implementation.
- We implemented a prototype system based on our P2P framework.
- We evaluated our algorithms, both mathematically and through simulations.

1.3 Thesis Organization

This thesis consists of seven chapters. Chapter 2 provides a comprehensive background to the P2P computing model as well as a detailed description of related work. In Chapter 3, we present the TBCP model and the *cluster search* algorithm. Chapter 4 provides a complete description of the COOL model, along with a discussion of category mapping and the *overlay search* algorithm. Chapter 5 discusses details regarding our application framework and prototype implementation. Chapter 6 presents both a theoretical and empirical analysis of system performance. Chapter 7 concludes the thesis and discusses potential directions for future research.

Chapter 2

Background and Related Work

This chapter introduces background information on P2P technology and previous research and techniques that are important to the present study.

2.1 Peer-to-Peer Computing

2.1.1 Background

The Internet was originally designed in a Peer-to-Peer manner. It encourages information sharing on research and development in the scientific and military fields by sending data packets between any two computers. The oldest Internet applications, such as email, Usenet news and the original Telnet, can be classified as Peer-to-Peer. Hence, the currently popular P2P computing model, since the first appearance of Napster [25] in May 1999, can be seen as "a renaissance of the original Internet model" [1].

There is no universally accepted definition for the P2P model. The Peer-to-Peer Working Group defines P2P as "sharing of computer resources and services by direct exchange between systems" [26]. Clay Shirkey, from Accelerator Group, defines P2P as "a class of applications that take advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet" and "peer-to-peer nodes must operate outside the DNS and have significant or total autonomy of central servers." However, a typical P2P application should possess some basic properties: *dual identities (client and server)*, *resource sharing* and *cooperation*. Peer nodes are usually connected to construct a deterministic or nondeterministic topology, and they cooperate with each other in providing resource sharing services: a node acts as a *client* when it is requesting resources and acts as a *server* when it is providing resources.

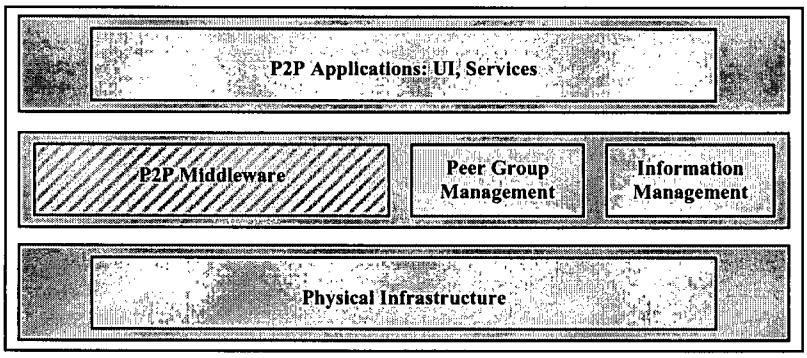

Figure 2-1 P2P Architecture

P2P architecture (Figure 2-1) generally consists of three layers: *application*, *middleware* and *network*. The Application layer provides user interaction as well as service logic or various algorithms. The middleware layer brings P2P-specific properties or features to a generic computing model. It provides a generic overlay network interface to the application layer by wrapping and hiding the physical network communication details. The two major functionalities realized in this layer are *group and resource*

management. In other words, this middleware layer is the key to distinguish the P2P architecture from other communication models, such as the Client-Server model.

In *Peer-to-Peer Computing* [24], the authors summarize the properties that the P2P computing model can provide: (1) *cost sharing* – cost can be shared and distributed to all peer nodes. (2) *scalability* and *reliability* – services are provided by many autonomous peer nodes rather than a few central servers. (3) *resource aggregation* – many types of resources, which are originally available only on local machines, can now be shared among peer nodes. (4) *increased autonomy* – resource and computation locality can be better enforced. (5) *anonymity* and *privacy* – users can prevent their information from being collected by a particular entity. (6) *ad-hoc connectivity* – a peer is not tied to any location in the system.

2.1.2 Classification

Currently most P2P architectures can be classified into three categories: *centralized systems*, *unstructured systems* and *DH-based (or structured) systems*. Each approach has a few variations in which certain features are enhanced to satisfy different application requirements.

The simplest model takes a *centralized* approach (Figure 2-2 (a)). It is first introduced by Napster [25], which is essentially a program that provides a free MP3 file-sharing service among millions of users. In terms of P2P related technology, these systems are relatively simple in the sense that centralized searching and maintenance protocols are used to keep track of a large amount of user files and to provide searching and downloading services. For example, the Napster company maintains a database of all users and some relevant information about their MP3 files, including music names, artist

names, etc. Users can access this database to look up their preferred music. The result of a user look-up query contains the exact location of the content so that an ad-hoc connection can be created between two users to transfer data. In this sense, a centralized approach is actually an extension to of the existing Client-Server model, and the only difference is that instead of storing data or files on the servers, they are provided by users and stored on client machines. Like the traditional Client-Server model, this centralized P2P model also suffers from potential scalability and single point of failure problems: When a large number of simultaneous search requests reach the server, the server might get overloaded and fail to process all the requests. The failure of the server can result in the crash of the whole system.

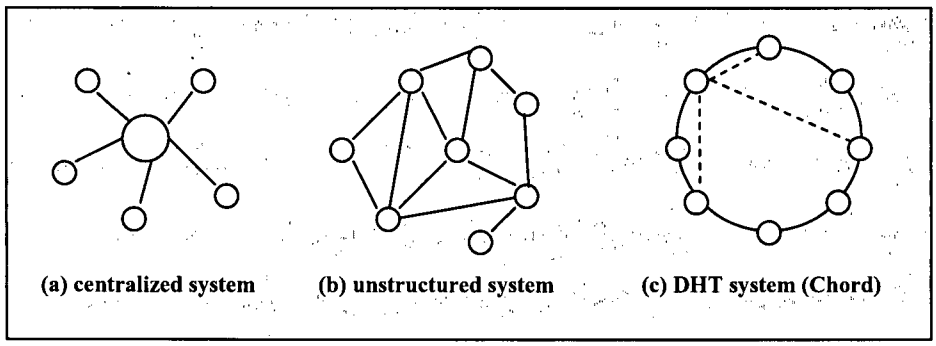

Figure 2-2 P2P Model Classification

Currently, many popular P2P applications [3,4,5] follows an *unstructured* approach (Figure 2-2 (b)), where no centralized server is deployed to coordinate system operations (servers might be deployed to provide registration or monitoring services). Thousands of users are connected in an ad-hoc fashion and construct a Power Law type network topology [7]. Global behaviour in these systems emerges from numerous local interactions [1]. Distributed searching and maintenance algorithms are developed to maintain system integrity and to provide file-sharing services. A look-up request is

usually delivered to a large number of peer nodes, and those peers that possess relevant content are responsible for contacting the requesting peer and initiating the data transfer. Due to the lack of a central server or directory services, messages usually flood over the network. This behaviour can generate a huge amount of searching and maintenance traffic and therefore limit the scalability of a purely *unstructured* P2P system. Several variations of this model have been developed to resolve this issue. The most well-known solution is called the *Super-Peer* or *Ultra-Node* approach. It is adopted in Gnutella2 [12] and KaZaa [18] systems. The idea is to build a two-level hierarchical structure in which some powerful nodes act as *Super-Peers*, which provides local directory services on behalf of a set of regular peers. *Super-Peers* are then connected to a small purely *unstructured* overlay. With this added level of indirection, queries can be processed by a fraction of the nodes in the network.

The third type of P2P model, based on *distributed hash table (DHT)* technology, was first introduced to the research community in 2001, with the near-simultaneous appearances of four different architectures: CAN [27], Chord [35], Pastry [29], and Tapestry [46]. At the center of DHT technology is the concept of distributing content onto a deterministic location in the network. In these systems, called DHT systems, content are assigned a unique key (produced, for instance, by hashing content names). Each node is responsible for a certain range of keys, and content with *similar* keys are expected to appear on the same node. For example, in the Chord system [35] (Figure 2-2 (c)), all peer nodes are connected as a single ring, where two nodes are neighbours on the ring if they manage the most similar set of keys (produced from the SHA-1 function). A file with key K is stored on the node that maintains K (this node is called a *successor* node of K). Besides the *successor* node, each peer node also maintains a *finger table*,

which is essentially a list of shortcuts pointing to strategic positions in the ring. Locating content in Chord can be seen as selecting a series of shortcuts to reach the destination node. In average case, Chord is able to locate a Key in O (log N), where N is the network size. However, DHT systems tend to maintain a tight control of system topology and object distribution; therefore, applying DHT architecture on the general-purpose file-sharing applications in the unstable Internet environment is disputable.

2.2 Related Work

Based on the above discussion of P2P model classification, we discuss and compare some popular searching algorithms specific to each category.

2.2.1 Searching over Centralized Systems

Searching over *centralized* P2P systems, due to central index servers, is like searching over Client-Server based systems. In Napster [25], a user can send a list of keywords, as a query, to a Napster server. The server then scans its local index database for a match. Because the Napster server maintains all the information available to the entire system, it is able to ensure the completeness of the search. Meanwhile, scanning local databases implies efficiency. However, due to the instability of peer nodes, index servers must frequently update their databases to reflect the most current system resource information.

2.2.2 Searching over Unstructured Systems

Flooding is the simplest searching approach used in many *unstructured* P2P systems, such as the original Gnutella [11], in which a query is forwarded *blindly* over the entire

network. Flooding is attractive because it takes advantage of rich connectivity in *unstructured* systems. Hence, it can ensure great search coverage in the dynamic Internet environment. But it suffers from significant bandwidth waste due to the delivery of duplicated messages.

Many *blind* search techniques are proposed either to restrict search coverage or to eliminate unnecessary flooding. The cost of these approaches is a low hit rate (the amount of useful results returned to the requestor) when searching unpopular content. [44] proposes several improvements on flooding, including *Iterative Deepening* and *Directed BFS*, to reduce the necessity of visiting more nodes if a desired amount of hits can be located within a small range. The random Walking [22] approach sends out K walkers to explore the network, also intended to limit the number of nodes to visit. [40] proves that this approach can generate enough results for searching popular content. [6] introduces methods like *Topology Adaption, Replication* and *Flow Control*, intended to gain more control over Gnutella topology construction and thereby shaping and restricting message flooding traffic.

Informed search methods, such as Local Indices [44], *Adaptive Probabilistic Search* [39] and *Intelligent-BFS* [16], take an approach different from a *blind* search. A query is only forwarded to nodes that are most likely to hold the content, based on local knowledge and probability models. Knowledge gained from previous query processing as well as neighbour information exchanges are used to predict the current location of a desired resource. However, maintaining and exchanging this knowledge frequently generates a large amount of network traffic. [28] proposes to use the Bloom Filter [5] to store and exchange this information, and thereby reduce network traffic.

The hierarchical *unstructured* model provides strong architectural-level support to improve search performance. In KaZaa [18] and Gnutella [12], request messages are forwarded only among *Super-Peer* nodes (i.e., within *Super-Peer overlay*). Algorithms, such as Efa [40], are designed to further restrict flooding over the Super-Peer overlay. However, these systems tend to suffer from the single point of failure issue as well as a high *free-riding* rate (as it is hard to find a fair and reliable incentive scheme not only to encourage powerful nodes to act as *Super-Peer* but also to penalize nodes that consume resources without making any contribution).

2.2.3 Searching over Structured Systems

Decentralized but *structured* systems are not designed to provide full-text search services. Content are distributed based on their key values rather than on other descriptive information, which implies that a peer node only knows shortcuts for reaching content with keys, but has no clue about locating them with keywords. Therefore, additional indexing and searching methods are required to provide search services on these systems.

A common approach, proposed by [37] and PIER [30], is to distribute links (or indices) to content onto multiple peer nodes, while the selection of these peer nodes is based on the hash values of the content's keywords. Each node then needs to maintain an index table that records all the content that contain certain keywords. Searching is then composed of two stages. In the first stage, all peer nodes that maintain relevant keywords are located. In the second stage, the index tables on these nodes are joined to discover entries that exist in all the tables. A recent study points out that in a very large P2P network, where popular content share lots of common keywords, a heavy traffic load imposed by expensive *join* operations can soon make this approach infeasible [19].

Optimization techniques are proposed either to eliminate the *join* operation completely or to decrease the amount of data to process. [36] proposes to store all relevant keywords in addition to content links on peer nodes. This approach means that the second stage of searching can be removed. However, this saving comes with the cost of distributing and storing more data, which may overwhelm search traffic load with respect to unpopular content. *Multi-level partitioning* [34] attempts to construct hierarchical node groups (on SkipNet), and the global inverted index is then partitioned among those groups. A query is processed by first intersecting tables in the same group. The results are then combined level by level until reaching the *root*.

2.2.4 Searching over Hybrid Systems

The hybrid model is designed to balance various tradeoffs among different P2P models. Injecting one model into a second type of P2P structure (through a dedicated integration rule) usually exhibits some nice properties unavailable in any basic model.

[21] and [45] propose to integrate *unstructured* and *structured* model in a parallel manner. Popular content are distributed and searched over the *unstructured* subsystem, while unpopular content are handled by the *structured* subsystem. This approach is able to overcome the issues raised when operating on each individual model. However, it is not clear how to define and distinguish popularity precisely and how to automate the process of shifting content between subsystems when their popularities change.

YAPPERS system [9] presents a design that provides a DHT-like object locating service to *unstructured* systems. A complete keyspace is first partitioned into buckets, and every node and all content is then assigned a bucket (or referred to as *color* in [9]). A content link is stored only on nodes that possess the same *color*. Each node in YAPPERS

is aware of a set of nodes within n hops. It keeps this information in a *local hash table*, which then enables query forwarding only among nodes that share the same *color* as the requested object. However, maintaining information accuracy in these *local hash tables* can be very expensive. Our approach shares some properties with this scheme.

Other systems, such as pSearch [37] and [47], introduce semantics into Peer-to-Peer searching. The *Latent Semantic Indexing (LSI)* technique is usually applied in these systems, with the purpose of adapting traditional *information retrieval* methods into the P2P domain. This system features a dual-stage query processing scheme: Queries are first forwarded to a group of nodes that are semantically close to the requested object. Then a complete search is executed among these nodes. This scheme improves both search efficiency and accuracy.

Chapter 3

Cluster Search

In this chapter, we present the design of our Tree-based Clustered P2P (TBCP) model and Cluster Search algorithm. The intention of our TBCP model is to reduce the complexity of a purely *unstructured* P2P model and to provide a solid building block for our Category Overlay infrastructure, which is presented in the next chapter. Meanwhile, our Cluster Search is a constrained flooding algorithm that works seamlessly with the TBCP model, aimed at reducing the cost of *blind* flooding.

3.1 System Design

The complexity of searching over a typical purely *unstructured* P2P system, like Gnutella, is a direct result of its rich connectivity and lack of control. As each node in the system takes equivalent roles, the decision about how to forward messages to neighbouring nodes, in most cases, is only locally optimized. The lack of monitoring and controlling

rules determines the infeasibility of achieving global optimization. A common solution to this issue is to introduce one or more levels of indirection into the system architecture.

As described in Chapter 2, a common approach to introducing indirection is to divide nodes into two categories: *peer nodes* and *super peer nodes*. Only *super peer nodes* are responsible for collecting and maintaining content information as well as answering queries. In other words, a *super peer node* has dual responsibilities: maintaining system connectivity and providing searching services.

The *Super-Peer* approach can dramatically reduce system complexity, but it introduces new challenges as well: (1) Regarding the selection of the *super peer node*, some reasonable principles must be set to evaluate the qualification of candidates. (2) Incentives must be provided to encourage *peer nodes* to compete for the *super peer node* roles. (3) Due to the unpredictable dynamics of the Internet environment, there might not always exist enough qualified *super peer nodes* at any given point of time. (4) Since a *super peer node* has dual responsibilities, its failure or voluntary disconnection can cause temporary or complete loss of connectivity, affecting all its associated *peer nodes*. It is necessary to shorten this transition period and minimize the transition cost. (5) As a result of re-construction, the rich connectivity of the original *unstructured* system is broken.

As a solution to these challenges, we propose a Tree-based Clustered P2P (TBCP) model, which distinguishes the duties of searching from maintenance. Here we define *Tree-based Cluster* as a set of nodes connected into a tree topology, which then acts as a single autonomous unit for the rest of the system. The TBCP model is also a hierarchical P2P model, but instead of maintaining system connectivity only within *super peer overlay*, it preserves the rich connectivity from the original *unstructured* system. Links in the TBCP model fall into two categories: the *Intra-Cluster Link* (for nodes connectivity

within clusters) and the *Inter-Cluster Link* (for inter-cluster connectivity). In terms of similarity, our TBCP model is a natural extension to a purely *unstructured* P2P system, and it exhibits better connectivity than the *Super-Peer* approach. Once integrated into our COOL model, the final architecture can provide the same level of efficiency as the *Super-Peer* approach, but with much more flexibility and stability.

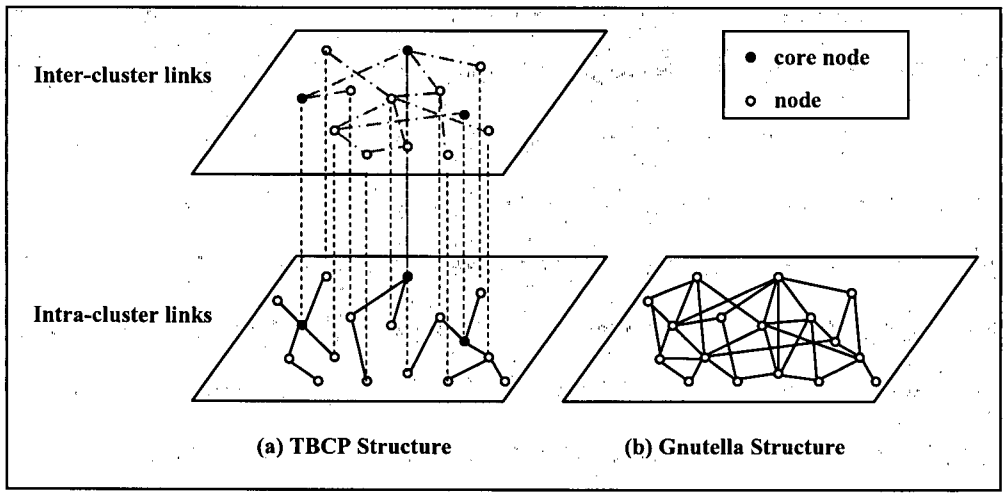

Figure 3-1 TBCP Model vs Gnutella Model

Figure 3-1 illustrates a simple TBCP architecture. The bottom panel is composed of three *tree-based clusters*. The root of each tree is called a *core node*. The only links that are included on this panel are the *intra-cluster links*, represented as tree branches. The maximum height of each tree is N. (Our simulation shows that N = 2 results in a reasonable cluster size in a typical Power Law topology network.) In the top panel, each node (including the *core node*) can have zero or more *inter-cluster* links, connecting nodes to other clusters. By preserving most links from the *unstructured* model, the TBCP system can ensure rich connectivity and thereby an enhanced fault tolerance.

Tables 3-1 and 3-2 explain some terms and data structures used in the later discussion. All three node types (*core nodes*, *master nodes* and *slave nodes)* are regular peer nodes, but they are assigned different duties with respect to maintenance. The *slave node* is located at the edge of each cluster, and its duty is simply to maintain ad-hoc links to a set of nodes. The *core node*, on the other hand, is the center of each cluster. Besides maintaining links, it also needs to collect and update cluster-specific information (e.g., *cluster member list* and *reachable cluster list*) through its communication with the *master nodes*. The *Master node* essentially has the same responsibility as *core node*, but at a different level: it needs to monitor and control only all the *slave nodes* associated with it. Searching services, however, are provided by all three types of nodes rather than by *core nodes* alone. Therefore, index maintenance is unnecessary and is eliminated completely. The implication is, unlike the *Super-Peer* approach in which each *super peer node* is assigned dual responsibilities, the TBCP model removes a significant amount of search-related traffic from the core nodes and distributes the load across all cluster member nodes. This behaviour will become more obvious after we introduce our COOL model, which eliminates the necessity of reaching every single node for complete coverage.

Terminology	**Description**
core node	root of a tree (or center of a tree-based cluster)
master node	direct child of *core node*
slave node	direct child of *master node*
intra-cluster link	link connecting parent node to child node
inter-cluster link	link between two nodes residing in different clusters

Table 3-1 Definitions Used in TBCP Model

Data Structures	Description
cluster member list	A local data structure maintained by each node independently, reflecting its knowledge about current cluster structures.
reachable cluster list	A local data structure maintained by each node independently, containing all clusters that can be reached through some nodes within the same cluster.

Table 3-2 Data Structures Used in TBCP Model

This separation of responsibilities has several advantages: 1. The *core node* and *master node* take less responsibility, which eases their selection process due to the reduction of the qualification standard. 2. Less responsibility for the *core node* also implies low replacement cost and a short transition period, which leads to high service availability. 3. The failure of the *core node* or *master node* only temporarily influences the partial intra-cluster connectivity and has small impact on searching services, which implies there is no single point of failure problem in the TBCP model. 4. Incentive rules are easier to set up because the search load is better balanced among participating nodes.

3.2 Tree-based Clustering Algorithm

Our hierarchical TBCP model is proposed for reducing system complexity, balance node heterogeneity, and improve searching availability and efficiency. To reach these targets, we present a Tree-based Clustering algorithm that creates and maintains a desirable TBCP architecture, which optimizes the following properties:

- *Similarity*: The sizes and structures of most clusters are topology dependent and vary only within a small range. Obvious outliers should be prohibited. This property helps to balance the workload across different clusters.
- *Simplicity*: Nodes within a cluster are organized into a simple height-bounded

tree, which reduces both system complexity and maintenance overhead.

- *Low Maintenance Cost*: Serious maintenance overhead should be avoided to ensure the scalability of the system.
- *Stability*: In a dynamic Internet environment, any single change to either the peer node or network link should have only limited influxes to a few other nodes rather than to the system as a whole.

3.2.1 Cluster Creation

```
join(n1, n2, c) {
/* when node n2 receives join request from node n1 */
1: if n2 is slave node, then
2:     if cluster c's size is less than cluster size limit Lcs, then goto 11.
3:     else, send rejection message to n1.
4: else if cluster c's size is less than cluster size limit Lcs, then
5:     if c's direct children is less than children limit Lnc, then
6:         /* justify n1's qualification with respect to cluster c's acceptance rules */
7:         send approved message to n1 if n1 is qualified, otherwise, send rejection.
8:     else, goto 11.
9: else, send rejection message to n1.
10: return.
11: select 3 master nodes n3, n4 and n5 from n2's cluster member list. These 3 nodes
    should have least number of children (from n2's local knowledge).
12: send information about these three nodes (including IP address and port number) to
    n1, suggesting them as next join candidates.
}
```

Figure 3-2 Cluster Creation in TBCP Model

The algorithm in Figure 3-2 describes the join process for a node *n1*, when it attempts to join cluster *c* by contacting *c's* member node *n2*. If this join request is approved (line 7), then *n2* becomes *n1*'s new parent. *n2* then forwards its own *cluster member list* to *n1*. Upon receiving this approval message, *n1* sends an acknowledgement back to *n2* to confirm the approval. However, if cluster *c* is full, then *n2* sends a rejection message to *n1* (lines 3 and 9). *n1* then needs to find another node to request for join. If cluster *c* is not full, but *n2* is either a *slave node* or is not able to accept more children, then *n2* searches its own *cluster member list* for 3 master nodes in *c* that have the least number of children at this moment. It then suggests these 3 nodes to *n1*, which then sends a join request to each of them, in turn. If all these attempts still lead to rejection, then *n1* needs to look for another node, in most cases from another cluster, to request for join. A special case occurs when *n1* is the first node in the whole system, or *n1* is not able to join any existing cluster. In this case, *n1* creates its own cluster and acts as *a core node*.

Cluster size limit *Lcs* and max children limit *Lnc* are topology-specific parameters. In chapter 6, we demonstrate, through simulation, that by setting *Lcs* = 350 and *Lnc* = 30, our clustering algorithm, operating on a typical Power Law topology network with 10,000 nodes, is able to generate clusters that possess this *similarity* property. Acceptance rules (line 6) are optional features, aimed at optimizing the performance of our clustering algorithm or satisfying application-specific requirements. For instance, a cluster can specify its lowest storage requirement, so that any node that is not able or not willing to provide a certain amount of storage space cannot be accepted as a cluster member.

3.2.2 Cluster Maintenance

Cluster maintenance in the TBCP model falls into two categories: *intra-cluster maintenance* and *inter-cluster maintenance*. Intra-cluster maintenance deals with the impact on cluster structure, from both *normal events* (node leaving) and *abnormal events* (such as node failure or a link broken). Inter-cluster maintenance aims to preserve rich connectivity among clusters. Since TBCP does not re-structure its basic *unstructured* P2P model, inter-cluster maintenance differs little from the simple *ping-pong* scheme [11].

Depending on the role that a leaving node n1 takes, a normal event has a different degree of impact on the cluster structure: (1) The impact is minimal if *n1* is either a *slave node* or it does not have children. Its disconnection affects neither the cluster topology nor any node other than its parent node *n2*. In this case, it is sufficient to notify only *n2*. (2) If *n1* is a *master node*, then its disconnection affects its direct children as well. In this case, *n1* also needs to notify them of its leaving. Upon receiving notice of this event, these children must send join requests to other *master nodes* inside the same cluster. These special join requests are guaranteed to be approved as long as the target *master node* does not exceed its children limit *Lnc*. (3) However, if n1 is a *core node*, then its disconnection has a significant impact on the cluster topology as well. A *core node candidate n3* must be selected in advance, so that the *core node* replacement can proceed smoothly and with minimum impact. Upon receiving notification from *n1*, *n3* claims itself as a new *core node* and notifies all the *master nodes* inside current cluster. These *master nodes* then update their parent node to be *n3*. If the number of *master nodes* plus the number of *n3*'s children exceeds the maximum children limit *Lnc*, then some of *n3*'s original children are forced to join other *master nodes*.

Abnormal events, such as node failures, are detected through periodical probing between each parent-child pair. Once an event is detected, a report will be directed to the parent of the failure node or to the *core node candidate* if the failure node is indeed a *core node*. This notified node is responsible for verifying the actual status of the node being reported. Once the failure status is confirmed, the same actions for handling normal events are taken to exclude the failure node from the current cluster and to re-structure the cluster topology if necessary. The simultaneous failure of both the parent and child nodes is reported to the closest ancestor of these nodes and handled separately, but if the failure nodes are the *core node* and *core node candidate*, then this event cannot be handled by the regular mechanism as it might result in cluster destruction. We propose to use a group election algorithm, such as the Bully algorithm, to select a new *core node* from the current cluster. The execution of this algorithm is expensive, but since the simultaneous failure of these two nodes is low, the maintenance cost is still acceptable.

3.2.3 Consistency Resolution

The maintenance scheme presented in the last section can preserve cluster integrity as well as inter-cluster connectivity, which then leads to great reachability – the ability to delivery a message from any node to a significant portion of clusters and thereby their member nodes. However, reachability and cluster integrity are not the only indicators of overall system performance. Other properties, such as message routing effectiveness and system load balancing, also have considerable impact. Two data structures (Table 3-1), *cluster member list* and *reachable cluster list*, are dedicated to optimizing these properties. Both lists are maintained by each node independently; therefore, a certain level of inconsistency can exist as a result of normal and abnormal events (i.e., a node

might not be aware of a particular system event). Here we define *consistency* as the relative accuracy of a node's local knowledge compared to the actual system details at any particular time.

One of the most important functionalities of the *cluster member list* is to allow a node to suggest other *master nodes* or *core node* to the requesting node at cluster creation time. Therefore, the level of consistency of each *cluster member list* has a direct and considerable impact on the quality and effectiveness of our tree-based clustering algorithm, as returning an improper join candidate list leads to unnecessary join activities and thereby significantly higher system operating costs. Inaccurate information can also cause an imbalance in terms of cluster size and the number of children, as well as a high rejection rate to join requests. The *reachable cluster list*, on the other hand, is particularly useful for reducing the complexity of the inter-cluster connections. In a well connected TBCP model, several links may exist between any two clusters. Maintaining a valid and relatively accurate *reachable cluster list* can avoid sending the same message through multiple links between the same pair of clusters.

To maintain an acceptable level of consistency while avoiding overwhelming maintenance network traffic, our TBCP model adopts a *periodical aggregation reporting* scheme accompanied by an *on-demand reporting* scheme to resolve the inconsistency. In the following description, we refer to normal and abnormal events as both events.

On-demand reporting emphasizes the necessity of immediate reporting in two scenarios: 1. If a *slave node* detects an event, it immediately reports this event to its parent node. 2. If a *master node* detects an event, and the total number of unreported events has reached the reporting threshold *Lor*, then all these unreported events are reported to the *core node*. Otherwise, this event is recorded without immediate reporting.

The periodical aggregation reporting scheme is also composed of two separate reporting procedures: (1) In every *Tmaster_report*, a *master node* needs to notify the *core node* about all unreported events since the last report. (2) In every *Tcore_report*, a *core node* needs to broadcast a complete *cluster member list* through tree branches (or *intra-cluster links*) to all member nodes within the same cluster.

The *reachable cluster list* is essentially an inter-cluster routing table. It is maintained by injecting a new parameter into a regular *ping-pong* message [11]. In an *inter-cluster* message, this parameter is the cluster ID of the sender. In an *intra-cluster* message from child to parent node, this new parameter contains a list of cluster IDs reachable (both directly and indirectly) from the child node. Upon receiving these messages, each node updates its *reachable cluster list* to ensure: (1) Each node knows a unique path to its direct reachable clusters. (2) Each node also knows a unique path to the reachable clusters from its children.

3.2.4 Algorithm Optimization

To maximize the TBCP model's performance, both connectivity and workload need to be distributed evenly throughout all the clusters. In other words, cluster *similarity* must be optimized. Aiming at constructing a well-balanced architecture, we propose the following restrictions and optimizations to further improve our basic *cluster creation* algorithm:

- *core node qualification*: Any node attempting to become a *core node* must satisfy several conditions, including computing power, dedicated storage space, bandwidth (for both upload and download), total number of connections, and a history of staying online for a relatively long period. Concrete values should be determined based on the network environment and the application.

- *cluster decomposition*: A threshold value *Min_Cluster* is applied to determine the validity of clusters. Any cluster with a size less than this value is assigned a trial period *Trial*. If its size does not reach *Min_Cluster* after this period, this cluster becomes invalid and should prepare for decomposition.
- *candidate recommendation*: During *cluster creation* period, in addition to three *master nodes*, each node can also recommend one node from another cluster, which can possibly accept this join request. This improvement effectively increases the join success rate, and thereby reduces the possibility of being forced to create new clusters.
- *cluster merge*: This is an alternative to *cluster decomposition*, with the purpose of avoiding multiple join operations. If a cluster *c1* needs to be decomposed, but it has a neighbouring cluster *c2* with a size less than *Merge_Limit*, then *c1* can be merged into *c2* by following these steps: (1) All of *c1*'s *master nodes* become *c2*'s *master nodes*. (2) *c1*'s *core node* also becomes *c2*'s *master node*. (3) If *c2*'s *core node* is unable to accept all these *master nodes*, then the unaccepted *master nodes*, along with their children, are decomposed.
- *role exchange*: If a *master node n1* has *num_c* children, which is considerably larger than the number of *master nodes* (*num_master*) in its cluster (i.e., *num_c* – *num_master* >= *Exchange_Limit*), then *n1* becomes the new *core node* and the original *core node n2* becomes a *master node*. In other words, *n1* and *n2* switch their roles inside their cluster. They also exchange their children.
- *cluster locality*: Grouping geographically close nodes into the same cluster can substantially decrease cluster maintenance and search cost, as usually less physical links are needed to forward a message between two *close* nodes. An

approach called Tiers [2], which builds a locality tree for the whole system, can be adopted in our system with minor modifications. Specifically, the K-Medoids method [17] is used to compute the *medoid* node in each cluster. These *medoid* nodes are then grouped into logical clusters (different from the TBCP's cluster), and we then compute the *medoid* nodes among them. This process continues until a single root node is computed. A node joining the system follows a top-down approach, starting from the root. It probes related *medoid* nodes at each level and selects the closest node to explore further. A well-known registration server should be deployed to maintain this locality tree.

All the parameters (threshold or limit values) used in this section are network and application specific. This thesis does not attempt to find the best values for them.

3.3 Cluster Search Algorithm

In the TBCP model, the search traffic load is distributed among all participating nodes. This obviously involves some scalability and efficiency issues. In this section, we present the *cluster search* algorithm, which is essentially a constrained routing algorithm working seamlessly with the TBCP model (the terms *searching* and *routing* are used interchangeably in this section as the query message is simply delivered through the routing algorithm).

3.3.1 Algorithm Motivation

A general *blind* flooding algorithm does not consider the richness of connectivity, and therefore generates a large amount of duplicated messages (i.e., a single query message is

forwarded, from different links, to the same node multiple times). Figure 3-3 (a) illustrates a simplified search process in a typical *unstructured* P2P system. In this particular example, node *A3* initiates a new query. This query is forwarded to node *A4* from five different nodes, which is a good example of message duplication. Moreover, considering the scale of a typical P2P system and the frequency of issuing new search requests, a simple duplication detection mechanism, such as book-keeping a list of nodes that have forwarded the same message to the current node, cannot prevent duplication. Moreover, some sources of duplication cannot be resolved by any detection-based scheme. For instance, if *A2* and *A4* receive the query message at the same time, then both of them are going to forward the message to *A1* and *B2*, which is undetectable to any simple detection-based scheme. As a result of duplication, forwarding a single query throughout this 9-node system generates a total of 26 messages.

Therefore, we need a different approach for controlling message duplication. A naïve solution is to frequently update each node's local knowledge so that at any time, each node knows a complete and relatively accurate system topology. In this case, each node can predict how a message is going to be propagated throughout the network and can thereby avoid sending messages that might cause duplication. However, this solution is not scalable, as in the Internet environment it is infeasible to assume that with a reasonable amount of maintenance traffic, local knowledge can be accurate. But, on the other hand, this approach unveils the potential for suppressing the number of duplicated messages through keeping knowledge of a small portion of the network topology. Our TBCP model has a mildly structured platform, which provides system support to achieve this short-range topology understanding.

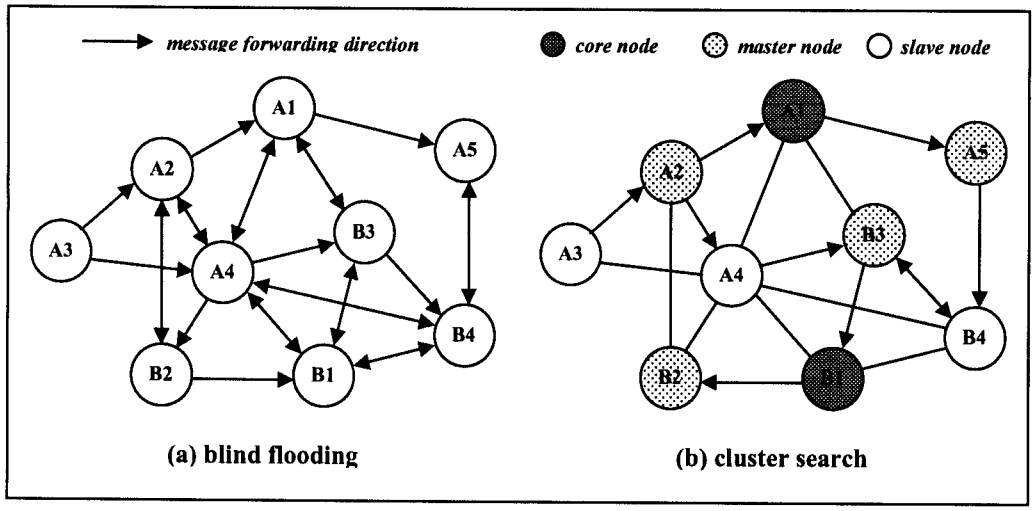

Figure 3-3 Blind Flooding vs Cluster Search

3.3.2 Cluster Search

Two duplication prediction and removal mechanisms are deployed in this algorithm: (1) Within a cluster, only *intra-cluster links* are used to forward messages; therefore, no duplication is generated within a cluster. (2) For messages sent between clusters, a *reachable cluster list* and a record of processed clusters are used to avoid sending the same message through many duplicated *inter-cluster links*. As our experiment shows, this localized algorithm can remove a large number of duplicated messages. The computation cost on each forwarding node is not high. The only real cost is the increase in message size, but we prove later that compared to the savings from suppressing duplication, our solution is definitely desirable.

Figure 3-4 presents a cluster search header format that is adopted in our Cluster Search algorithm.

Table 3-3 lists a few of the symbols used in our *cluster search* algorithm.

msg_type	n_from	TTL	flags	CTV	CP	checksum	msg_body

Figure 3-4 Cluster Search Message Header

Symbol	Description
n_curr	Current node
n_from	Node that forwards message to *n_curr*
P(n)	Parent node of *n*
Child(n)	Children node of *n*
Neighbour(n)	Nodes in reachable cluster of *n*
Cluster(n)	Cluster ID of *n*
CTV(n, msg)	Clusters to visit list for node *n* in message *msg*, i.e., a list of clusters that n needs to forward *msg* to
CP(n, msg)	Cluster processed list for node *n* in message *msg*, i.e., a list of clusters that have processed (received) this *msg* before reaching current node *n*
RCL(n)	Reachable cluster list of node *n*

Table 3-3 Symbols Used in Cluster Search Algorithm

The algorithm in Figure 3-5 is used to compute the *CTV* (or clusters to visit list). This list is computed from the *reachable cluster list* and is updated when a message is forwarded to children. *CTV* is used to avoid forwarding the same message to the same cluster (through different *inter-cluster links*).

CTV(n, msg) = $\forall$ ***c' $\in$ RCL(n_curr)***, such that ***c' is reachable from n*** AND

1. ***n_from $\neq$ null*** AND ***n_from = P(n_curr)*** AND ***c' $\in$ CTV(n_curr, msg)***
OR
2. ***n_from = null*** OR ***n_from $\neq$ P(n_curr)***

Figure 3-5 Clusters To Visit (CTV) List Computation

Figure 3-6 illustrates the routing process for the current node *n_curr* when it receives a message *msg* from node *n_from*. Assuming *TTL* is infinity, if *msg* is a new message, then *n_curr* needs to forward it to its parent *P(n_curr)*, children *Child(n_curr)*,

and neighbours *Neighbour(n_curr)*, unless some conditions are not satisfied (at line 11 and line 13). *CTV(n, msg)* is updated whenever *msg* is forwarded to *n_curr*'s children, or is reset if *msg* is forwarded to a different cluster. *CP(n, msg)*, on the other hand, is updated if *msg* is forwarded through *Inter-cluster links*.

```
clusterSearch(n_from, n_curr, msg) {
/* when n_curr receives a message from n_from, this algorithm determines how msg is
   forwarded. Assuming TTL is infinity. */
1:  if msg has been processed before, then ignore it. /* duplicated message */
2:  else
      /* forward message to parent node */
3:     if n_from = P(n_curr) and P(n_curr) exists, then
4:        forward msg to P(n_curr).

      /* forward message to children */.
5:     foreach n_child in Child(n_curr), do
6:        if n_child = n_from, then
7:           update CTV(n_child, msg).
8:           forward msg to n_child.

      /* forward message to other clusters */
9:     foreach n_neighbour in Neighbour(n_curr), do
10:       if n_neighbour ≠ n_from, then
11:          if n_from = P(n_curr) and n_neighour ∉ CTV(n_curr, msg), then
12:             continue.
13:          if n_curr ∈ CP(n_curr, msg), then
14:             continue.
15:          update CP(n_neighbour, msg) by adding Cluster(n_curr).
16:          clear CTV(n_neighbour, msg).
17:          forward msg to n_neighbour.
18:    search local database for result }
```

Figure 3-6 Cluster Search Algorithm

3.3.3 An Example

Figure 3-3 (b) shows an execution of the *cluster search* algorithm on the same sample network. For illustration purposes, we construct a TBCP model on this network, which then generates two clusters *A* and *B*. Nodes *A1* and *B1* are the *core node* for each cluster. Table 3-4 describes the *reachable cluster list* on each node. *n/a* implies that either this entry is unnecessary or a node does not know how to reach a certain cluster. The letter inside each bracket (*d* or *i*) represents *directly reachable* or *indirectly reachable*. For instance, *A5(i)* means that the cluster is indirectly reachable from node *A5*.

node	***A1***	***A2***	***A3***	***A4***	***A5***	***B1***	***B2***	***B3***	***B4***
to cluster *A*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*B3(i)*	*n/a*	*A4(d)*	*A5(d)*
to cluster *B*	*A5(i)*	*A4(i)*	*n/a*	*B3(d)*	*B4(d)*	*n/a*	*n/a*	*n/a*	*n/a*

Table 3-4 Reachable Cluster Lists in Cluster Search Example

Nodes *A2* and *A3* do not have direct inter-cluster links to cluster *B*; therefore, they do not forward the query to *B*. Node *A1* also does not forward the message to *B* because the entry inside its *reachable cluster list* suggests node *A5* as the forwarding node. In this particular example, the only two duplicated messages are sent between *A4-B2* and *B2-B4*. The former duplication is generated because node *A4* does not know that, from *A1's* point of view, cluster *B* is reachable from *A5*, not *A4* itself. The latter duplication occurs as a result of simultaneous receiving the query message by both nodes *B3* and *B4*. This duplication is not detectable or resolvable in our *cluster search* algorithm. Nodes in cluster *B* will not forward this query back to any node in cluster *A* because cluster *A* appears in the *CP (or cluster to process)* list already. As a result of the *cluster search*, only 10 messages are delivered (compared to a total of 26 messages under *blind flooding*).

Chapter 4

Category Overlay Search

Our study, as presented in the last chapter, assumes that content objects are semantic-free and that they are not organized in any form (no index is maintained). Under these assumptions, every node can potentially possess the desired content, and each node must be visited to ensure complete search coverage. In this chapter, we eliminate these restrictions and propose a novel algorithm, category overlay search (cool search), which can dramatically improve search efficiency as well as to better balance search traffic.

4.1 Motivation

In the computer world, all content is simply a series of 0's and 1's. To compare two objects is indeed to compare their binary representation. But in the real world, content has meaning, or semantics. Given a proper classification rule, any object can be classified into one or more categories. The search coverage for a particular object, therefore, can be restricted to a few categories instead of the entire content space.

This concept is widely applied in current web-based Client-Server models. Popular search engines, such as Google [10] and Yahoo [43], provide both the *blind* keywords search (regular search) and the *directory (or category)* based search. As a simple experiment, at the time of writing this thesis, a *blind* search in Yahoo with the keyword *hostage* generates 63,800,000 results, all of which contained the word *hostage*. This overwhelming result is not necessary and not helpful if the desired content is indeed any movie titled *hostage* (searching with two keywords *hostage* and *movie* can eliminate most of the hits, but this approach has already considered semantics, as conceptually *movie* should be a category rather than a keyword). Searching the same keyword with the directory-based search (*Entertainment -> Movies and Films*) generates only 8 hits, all of which are films related to *hostage*.

Therefore, our algorithm aims to reduce search costs by avoiding unnecessary search activities within irrelevant categories. One of the challenges is to find a good classification rule, which can generate categories that satisfy the following requirements: (1) The uniqueness of the object classification is maximized. In other words, an object should belong to one and only one category in most cases. (2) It is not desirable to have most objects classified into a few categories, while other categories contain only very few objects. (3) Classification should be easily understood and accepted by people. The methodology for finding a good classification rule is beyond the scope of this thesis and will not be discussed further. However, the types of content shared within a specific P2P application are usually limited, as each system has its own application domain. Therefore, it is reasonable to assume that a relatively good classification rule can be found through simple observation.

4.2 System Design

To merge semantics into an *unstructured* P2P system, we need a mapping rule to associate category domain to system domain. Here we define a mapping as an association between a category and a peer node. One extreme mapping rule is to associate every node with all the categories. This is a self-mapping rule because each node maintains its own local objects as if no classification were present. As a result, a query still needs to traverse the whole network since every node can potentially possess related content.

Another naïve solution is to split a complete system into a number of independently operating subsystems. The mapping rule in this solution is to associate each category to one subsystem. In other words, any node outside of this subsystem does not have content that can fall into the related category. Under this model, searching can be executed on only one or a few subsystems and therefore does not involve all the nodes. However, a single searching activity involves locating, joining, and leaving a particular subsystem, which obviously implies high overhead. Moreover, nodes that contain a large variety of objects must participate in multiple subsystems, which can cause severe maintenance overhead. Most important of all, searching and indexing are not separated; therefore, nodes that provide more content must experience not only more maintenance traffic but also more searching traffic. This behaviour leads to a serious system imbalance issue.

Aiming to resolve these problems, we propose a Category Overlay (COOL) model, which acts as an extension of our TBCP model by introducing semantics. The dynamic mapping rule adopted in our COOL model is to associate the category domain with the cluster domain (rather than with the system domain); therefore, the COOL model can satisfy the following conditions: (1) Each node is associated with zero or more categories

(this node is named *category agent* for these categories). (2) Each category has an *agent* node in every cluster; therefore, a complete category domain can be mapped onto every cluster. (3) *Agent* nodes for the same category are connected through logical links. (4) Category mapping is dynamic. In other words, a category can be mapped onto different nodes at different times; meanwhile, a node can act as the *agent* node for different categories at different time.

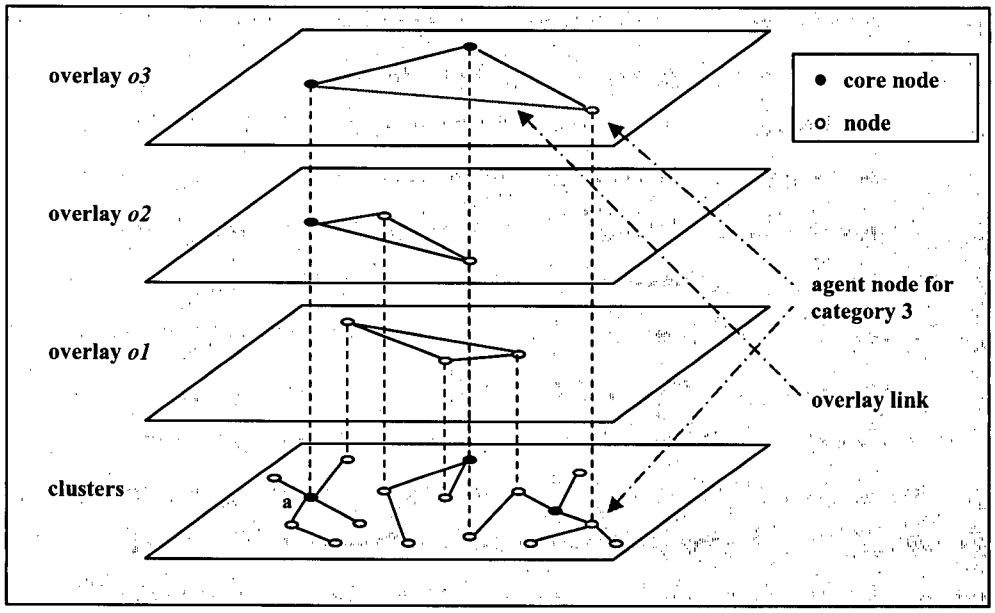

Figure 4-1 Category Overlay (COOL) Model

We define a *category overlay* (for a particular category *c*) as a virtual network composed of all *c*'s *agent* nodes and the logical links among them, with the purpose of managing and locating content belonging to *c*. Figure 4-1 illustrates a simplified COOL model with only three clusters and three categories (and therefore three category overlays *o1*, *o2* and *o3*). Node *a* (the core node of the leftmost cluster) is a common member for both overlay *o2* and *o3*. The logical links inside an overlay are *overlay links*, updated by

Inter-Cluster maintenance. Links between overlays, on the other hand, are *intra-cluster links*. They provide direct and solid support to locate any *category agent* within a cluster.

Terminology	Description
category agent	Node that is associated with a particular category
category overlay	A virtual network composed of all *agent* nodes for the same category
indexing link	Virtual link used to maintain the consistency of category related knowledge
overlay link	Virtual link that connects two agent nodes inside an overlay
search link	Virtual link used to locate *category agent* within a cluster

Table 4-1 Definitions Used in COOL Model

Data Structures	Description
Content Index Table (iTable)	For a category *c*, an *iTable* is a data structure that records all the valid tuples <*c*, *keyword list*, *Ni*> within a single cluster, where *Ni* has content that match *c* and *keyword list*.
Category Table (cTable)	For any node *N*, a *cTable* is a hash table consisting tuples <*c*, *Ni*> for every category *c*, where *Ni* is believed, by *N*, to be *c*'s *agent* node within *N*'s cluster.
Overlay List (oList)	For a category *c* and its agent node *N*, an *oList* is a list of all *c*'s *agent* nodes that are connected with *N* through *overlay links*.

Table 4-2 Data Structures Used in COOL Model

Tables 4-1 and 4-2 describe some terminology and data structures used in the Category Overlay model and referred to frequently in this thesis. In each cluster *c*, every category *cat* has an associated *Content Index Table (iTable)*, which records indices (or links) about all the content available in *c* that belongs to *cat*. The *iTable* for *cat* exists only on the *cat*'s *agent node*. The *cTable*, on the other hand, exists on every participating node. Given any category *cat* and node *n*, the *cTable* on *n* has an entry predicting which node, within *n*'s cluster, is the *cat*'s agent node. The third data structure, *oList*, also exists

exclusively on *cat*'s agent nodes in every cluster. It records all overlay links for *cat*, and therefore it serves as a routing table to our *overlay routing* algorithm.

The category Overlay model has some nice properties: (1) System integrity is preserved. Searching for content in different categories does not require joining another subsystem. (2) Locating all objects associated with a certain category only requires a complete traversal of the corresponding overlay; therefore, searching the COOL model based system generates a much smaller number of messages. (3) Searching and indexing are separated. Searching is executed within each *category overlay*, while indexing is executed within each cluster. A node providing content belonging to a category *c* does not have to be the *agent* node of *c*; therefore, in the COOL model, nodes providing a large number of content do not need to process more search requests. (4) Object locality is preserved. (5) The *category overlay* is not a fixed structure. Although different overlays are maintained independently, they are connected through the *indexing links*. Therefore, both searching and indexing traffic can be balanced. For instance, if a node *n1* experiences too much traffic, it is possible to shift partial load to another node n2 (within the same cluster) through *Category Migration*, which is described in the next section.

With these nice properties, a COOL model can be viewed as a special *Super-Peer* model, where every peer in COOL can potentially act as a *super-peer* at the same time. In other words, multiple *super-peer* overlays can exist simultaneously, and these overlays are also inter-connected. This final architecture provides not only support for efficient searches but also a more flexible and stable platform. Moreover, it prevents *free-riding* as each node, no matter how powerful it is, will be asked to act as the agent for some categories. This is reasonable because the searching load is distributed over a large set of categories and each node will therefore only be responsible for a subset of them.

4.3 Overlay Maintenance

The COOL model is a hybrid model that coordinates two types of connections (or overlays): *cluster overlay* and *category overlay*. In this section, we present maintenance mechanisms for each type of overlay, both for preserving their integrity and for coordinating their interactions. More specifically, these schemes aim to resolve the inconsistency in the *Category Table (cTable)* and *Overlay List (oList)*, respectively. The consistency level in both tables is the key factor in the performance of our COOL model.

4.3.1 Category Migration

Both *Category overlay* and *cluster overlay* are dynamic structures. A member node in an *overlay* can always be replaced by another node from the same cluster. We define this replacement activity as *category migration*. In other words, *category migration* is an activity that shifts an agent role from its original node to a new node.

For a category *c*, the original agent node is *n1* and the new *agent* node will be *n2*, then a single *category migration* involves the steps described in Figure 4-2.

Category migration is particularly useful in balancing searching and indexing traffic. However, it involves a set of expensive operations and generates inconsistency in the *cTables* of the other member nodes (within the same cluster). We identify the following four scenarios, under which *category migration* is either mandatory or desired: (1) Any node joining a cluster can request *category migration* from its parent node. (2) Any node leaving a cluster must migrate all its categories to other nodes within the same cluster. (3) When a node failure is detected, its parent node (or *core node candidate* if the

core node fails) must execute *category migration.* (4) Any node that is either overloaded or underloaded can request for *migration.*

step 1: *n1* sends a *migration* request to *n2*.

step 2: *n2* approves this request by acknowledging *n1*.

step 3: *n1* sends its *iTable* and *oList* (for category *c*) to *n2*.

step 4: *n2* records *iTable* and *oList*, and sends an acknowledgement to *n1*.

step 5: Both *n1* and *n2* update their *cTables* to reflect the changes (i.e., in both *cTables*, *c* is now mapped to *n2* instead of *n1*).

step 6: *n1* notifies *n2*'s parent node (or *core node candidate* if *n2* is *core node*) of this *migration* event. The parent nodes then update its *cTable*.

Figure 4-2 Category Migration Procedures

4.3.2 Cluster Overlay Maintenance

We define *cluster overlay* for a cluster *v* as a virtual network of all *v*'s member nodes and associated *category links (cLinks).* Each *cLink* is associated with a tuple <*c*, *n1*, *n2*>, where *c* is a category, and *n1* and *n2* are both member nodes of *v*. More precisely, each *cLink* is represented as an entry in *n1*'s *cTable* (for category *c*), which implies that, from *n1*'s point of view, *n2* is the *agent* node for *c* in cluster *v*. A *cLink* is valid if node *n2* is indeed the *agent* node for *c*. Therefore, maintaining *cLink* validity (or *cTable* consistency) is a key component in maintaining *cluster overlay* consistency.

We propose a *periodical aggregation report* scheme to maintain a certain level of consistency in the *cTable*: In every *Tc_report* period, each node *n1* prepares an *aggregation report* and shares it with a randomly selected node *n2* (within *n1*'s cluster).

This report records the latest *M category migration* events, along with their timestamps, that are known to *n1*. Upon receiving the report, *n2* updates its *cTable* to reflect these events. For instance, for a category *c*, if the *agent* node (for *c*) in *n2*'s *cTable* is *n3* and there exists an *migration* event in the report, which suggests that *n4* is the new *agent* node, then as long as this *migration* event occurs later than the last time, *n2* updates the entry related to *c*, and *n2* should change the *agent* node (for *c*) to *n4*. Both *Tc_report* and *M* are system specific parameters, determined by network size, total number of categories, and the stability of participating nodes. Chapter 6 presents simulations that demonstrate, by properly assigning values to these two parameters, a high consistency level (above 90%) can be preserved at the cost of a reasonable amount of maintenance traffic.

In addition to consistency resolution, *cluster overlay* maintenance also needs to preserve system integrity with respect to category mappings. We introduce a few simple mechanisms to bridge the gap between the category domain and the cluster domain. Specifically, we present modifications to cluster creation and maintenance algorithms in the TBCP model.

During the cluster creation, if node *n1* sends a join request to node *n2* and this request is approved, then besides the delivery of the *cluster member list* (as required in the TBCP model), *n2* also needs to forward its *cTable* to *n1*. In other words, the initial *cTable* of *n1* is a copy of *n2*'s *cTable*. *Category migration* is optional as a new node tends to be less reliable and leaves the system more frequently. If *n1* has to create a new cluster, then it also creates a default *cTable*, in which all entries point to *n1* itself. In other words, *n1* is the *agent* node for all valid categories in the system.

Normal events, such as node leaving, must perform one additional step – *category migration*. Migration follows the exact steps as presented in Figure 4-2. Abnormal events,

such as a node failure or a broken link, can be handled through one the following two approaches: *on-demand objects recollection* and *random back-up*.

The *on-demand objects recollection* is a simple approach that does not require regular or periodic maintenance. As the failure of node *n1* is detected, its parent node *n2* starts to act on behalf of *n1*. Specifically, *n2* claims itself to be the *agent* node for all the categories controlled by *n1*. However, since for each category *c*, both *n1*'s *iTable* and *oList* are missing, *n2* does not have any clue to either the content or the *category overlay* structure related to *c*. With this on-demand scheme, *n2* does not rebuild these tables unless a query for *c* reaches *n2*. To build an *iTable*, *n2* broadcasts a message to the whole cluster and asks every node for content related to *c*. To build an *oList*, *n2* follows *category overlay* maintenance mechanisms, which are discussed in the next section.

The *random back-up* approach, on the other hand, requires periodic maintenance. For each category *c* (maintained by its *agent* node *n1*), a random node *n2* is selected to back up related *iTables* and *oLists* on *n1*. (More random nodes can be picked up to store back-up copies under a very dynamic network environment.) Moreover, *n1*'s parent node *n3* must be notified of the existence of *n2*. The back-up tables and lists are updated during every Tbackup time. When the failure of *n1* is detected, *n3* needs to report this abnormal event to *n2*. Upon receiving this report, *n2* claims itself to be the new *agent* node for *c*. However, if *n3* cannot be located (e.g., *n3* also fails), then a broadcast message is propagated throughout the cluster to locate *n2*. If *n2* also cannot be located, then the *on-demand objects recollection* approach must be applied instead.

4.3.3 Category Overlay Maintenance

Category overlay's integrity and connectivity (represented as *overlay list* consistency) directly influence the performance and coverage of our overlay search algorithm: forwarding a query through invalid *overlay links* (*oLinks)* results in either searching irrelevant nodes or loss of messages. Moreover, a high inconsistency level in each node's *oLists* (or a large amount of invalid *overlay links*) can even lead to the destruction of the *category overlays*. However, as a result of *category migrations* and some abnormal events, the decreasing of *oList* consistency is unavoidable. The purpose of *category overlay* maintenance is therefore to sustain the *oList* consistency at an acceptable level.

A simple solution is to update *oList* whenever a miss (a *non-agent* node receives a query through an invalid *oLink*) occurs. In this case, the correct agent node must be searched and found to update the invalid *oLink*. This on-demand mechanism is suitable for a stable network environment in which a *category migration* event occurs only at a low frequency: As migration does not occur frequently, the *oList* can remain at a high consistency level, and the search cost can remain within an acceptable range. However, this condition does not hold in a dynamic network environment.

Therefore, we present a *periodical partial category report* to resolve *oList* inconsistency. In every *Tpc_report* period, each node *n1* prepares a report and shares it with a randomly selected node *n2* from *n1*'s *oList*. This report claims that *n1* currently controls a list of categories *Cs*. Upon receiving this report, *n2* notifies the corresponding *agent* nodes, for each category in *Cs*, about the existence of node *n1*. Meanwhile, it prepares a reply enumerating these agent nodes. *n1* then updates its own *oList* based on the reply. Table 4-3 illustrates this procedure in detail (the irrelevant entry is not shown). For simplicity, this process involves only 2 categories: *c1* and *c2*. Node *n1* and *n4* are

from cluster *c1* while *n2* and *n3* are from cluster *c2*. Initially, *n1* has an invalid *oLink* to *n2* (for category *c2*), and node *n3* has an invalid *oLink* to *n4* (for category *c2*). Both links are fixed after an exchange of the *partial category report*.

Node	*before/after report*	*cTable*	*oList*
n1	before	c1: n1 c2: n1	c1: n2, n10, ... c2: ***n2***, n20, ...
n1	after	c1: n1 c2: n1	c1: n2, ... c2: ***n3***, ...
n2	before and after	c1: n2 c2: n3	c1: n1, n30, ... no entry for c2 since n2 is not agent for c2
n3	before	c1: n2 c2: n3	no entry for c1 since n3 is not agent for c1 c2: ***n4***, n40, ...
n3	after	c1: n2 c2: n3	no entry for c1 since n3 is not agent for c1 c2: ***n1***, n40, ...

Table 4-3 A Category Overlay Maintenance Example

4.4 Overlay Search Algorithm

In this section, we present an efficient overlay search algorithm that works seamlessly with our Category Overlay (COOL) model to avoid redundant message forwarding and unnecessary computation. This algorithm consists of two parts: *Cluster Overlay Search* and *Category Overlay Search*. But for illustration purposes, some redundant codes are added to make each part a complete and independent algorithm. In addition, some codes are included with the purpose of illustrating the interaction between these two parts. An example is given at the end of this section to describe a complete searching process.

4.4.1 Cluster Overlay Search and Content Publishing

The purpose of the *cluster overlay search* is to locate the correct *agent* node for the requested category. Due to the inconsistency in *cTables*, valid *search linkes* cannot always be created by referring to *cTable*: the node from *cTable* might have already migrated the category to another node. In the worst case, if a category has been migrated many times and there does not exist a clue about which node is the correct agent, then a cluster-wide delivery (or a *cluster search*) is used to find the right *agent*. This algorithm uses a prediction technique to improve the probability of locating the right node in a few attempts, so that the necessity of an executing expensive *cluster search* is decreased.

The following table (Table 4-4) lists a few symbols and definitions used in the *cluster search* algorithm and referred to later in the discussion. Other symbols used in the algorithm are described in Table 3-3.

Symbol	Description
n_curr	Current node
n_from	Node that forwards message to *n_curr*
msg	Query message
cat	Category that is being searched
Agent(n, c)	*Agent node* for category *c*, taken directly from *n*'s *cTable*
Requestor(msg)	Node that initiates this query message
Core(n)	*Core node* in *n*'s cluster.

Table 4-4 Symbols Used in Cluster Overlay Search Algorithm

Figure 4-3 describes the process of locating the *agent* node for category *cat* when the current node *n_curr* receives a message *msg* from node *n_from*. Assuming *TTL* is infinity, if *msg* is a new message, then there exists 3 cases: (1) If *n_curr* appears to be the correct *agent* node, then it invokes the *category overlay search* (discussed in the next

section) and looks up its *iTable* for matching entries (at line 5 and 6). (2) If *n_curr* is neither the *agent* node nor the first node to process *msg* in the current cluster (*n_from* has processed it), then *n_curr* is not responsible for search in the current cluster. (3) Otherwise, *n_curr* needs to predict the correct location for *c*. It first selects the *agent* node *agent_candidate*, taken directly from *its cTable* (at line 8). As long as the entry for *c* remains consistent through *cluster overlay maintenance*, *agent_candidate* must be the right node. However, if this entry is inconsistent, then three other candidate nodes are attempted (at line 12). *Agent(agent_candidate, cat)* and *Parent(agent_candidate)* are selected because, if *c* is migrated to another node from *agent_candidate*, then both *agent_candidate* and its parent node should have more recent knowledge of *c*'s location than *n_curr*. *Core(n_curr)* is probed because it is involved frequently in category migration (or handling normal events in particular), and therefore its *cTable* is likely to have a better consistency level. Our simulation demonstrates that, even with low consistency *cTable*s, most of the cluster overlay search can be completed after these 4 attempts.

Unlike in a typical purely *unstructured* P2P system or in our TBCP model, where publishing new content is no different from storing related information in its publisher's local repository, our COOL model requires an additional cluster-wide *content publishing* stage. This publishing operation first locates the correct *agent* node for the content being published. Keywords describing the content are then transferred to that node and stored in its *iTable*. Therefore, the only difference between *content publishing* and the *cluster overlay search* is that after locating the *agent* node, instead of invoking the *category overlay search*, keyword list transferring and *iTable* updating are executed.

```
cluster_overlay_search(n_from, n_curr, cat, msg) {
/* when n_curr receives a message from n_from, this algorithm determines how msg is
   forwarded within n_curr's cluster. Assuming TTL is infinity. */
1: if msg has been processed before, then ignore it. /* duplicated message */
2: else
3:     if Agent(n_curr, cat) = n_curr, then  /* n_curr is the agent node for cat */
4:         send a positive acknowledgement to n_from.
5:         category_overlay_search(cat, msg).  /* invoke overlay search */
6:         search n_curr's iTable for result and reply to Requestor(msg).
7:     else if n_from = null or Cluster(n_from) ≠ Cluster(n_curr), then
           /* n_curr acts as the coordinator of this search */
8:         agent_candidate = Agent(n_curr, cat).
9:         forward msg to agent_candidate and wait for reply.
10:        if reply is positive, then return.  /* agent node for cat is located */
11:        else  /* no reply or negative reply */
12:            candidates = {Agent(agent_candidate e, cat), Parent(agent_candidate),
                                 Core(n_curr)}.
               /* check three candidates for agent node */
13:            foreach n_candidate in candidates, do
14:                forward msg to n_candidate and wait for reply.
15:            if receive one positive reply, then return.  /* agent node is located */
16:            else
17:                clusterSearch(n_from, n_curr, msg).  /* invoke cluster search */
18:    else  /* notifies n_from that n_curr is not the agent for cat. */
19:        send a negative acknowledgement, along with Agent(n_curr, cat) and
               Parent(n_curr) to n_from.
}
```

Figure 4-3 Cluster Overlay Search Algorithm

4.4.2 Category Overlay Search

The *overlay search* is not complete unless a query is forwarded throughout relevant *category overlays* and thereby reaches all relevant agent nodes. In the ideal situation, when all *overlay links* are valid, a query is propagated only among correct agent nodes. However, due to invalid *overlay links*, *non-agent* nodes can also receive the query. In worst case, when all *overlay links* are invalid, each cluster must perform additional operations to locate the correct *agent* node in that cluster. Moreover, if most *overlay links* are not only invalid but also broken, then search coverage cannot be assured. Aim to resolve these issues, we propose a *category overlay search* algorithm that optimizes the *overlay search* coverage while keeping the low search cost.

Table 4-5 introduces another definition used in our algorithm.

Symbol	Description
Overlay(n, cat)	Nodes in *n*'s *oList* that are connected to *n* through some *overlay links* from *cat*'s *category overlay*.

Table 4-5 Symbols Used in Category Overlay Search Algorithm

Figure 4-4 describes the procedures for forwarding a query (for category *cat*), through *category overlays,* after the current node *n_curr* receives a message *msg* from *n_from*. If *n_curr* is not the agent node for *cat*, which implies that *n_curr* cannot locate the *category overlay* for *cat*, then the *cluster overlay search* must be used to locate the correct *agent* node first. Otherwise, *n_curr* forwards msg to every node in *Overlay(n_curr, cat)*, which is based on *n_curr*'s local knowledge of *cat*'s *category overlay* (at lines 5 to 7). However, if more than half of these neighbour nodes do not respond (i.e., the related *overlay links* are broken), then a *cluster search* is invoked (at lines 9 to 11), with the purpose of making use of the rich connectivity in our TBCP model to offset the poor reachability due to *overlay* destruction.

```
category_overlay_search(n_from, n_curr, cat, msg) {
/* when n_curr receives a message from n_from, this algorithm determines how msg is
   forwarded within the category overlay for cat. Assuming TTL is infinity. */
1: if msg has been processed before, then ignore it. /* duplicated message */
2: else
3:     if Agent(n_curr, cat) = n_curr, then /* n_curr is the agent node for cat */
4:         send a positive acknowledgement to n_from.
5:         foreach n_neighbour in Overlay(n_curr, cat), do
6:             if n_neighbour ≠ n_from, then
7:                 forward msg to n_neighbour and wait for reply.
8:         search n_curr's iTable for result and reply to Requestor(msg).
9:         if number of replies > 1/2 * sizeof Overlay(n_curr, cat), then return.
10:        else
                   /* not enough nodes are forwarded */
11:            clusterSearch(n_from, n_curr, cat, msg).
12:    else
           /* msg is from an invalid overlay link
13:        send a negative acknowledgement to n_from.
14:        cluster_overlay_search(n_curr, cat, msg).
}
```

Figure 4-4 Category Overlay Search Algorithm

Lines 1, 4, 8, 13, and 14 from this algorithm are for illustration purposes only. After being integrated into the *cluster overlay* search, these lines can either be merged with existing lines (lines 1 and 6 from the previous algorithm), or be removed completely.

4.4.3 A Complete Example

A sample overlay search process (for a category *cat*) operating in a four clusters' system is described in Figure 4-5. For illustration purposes, the *overlay links* and *search links*

that are not related to cat are not shown in this figure. We assume that A3 initiates a new query *q*, and at that moment, node *A5*, *B3*, *C1*, and *D3* are *agent* nodes for *cat*. Table 4-6 also presents the values for *cTable*, *oList* and *iTable* on a few interesting nodes.

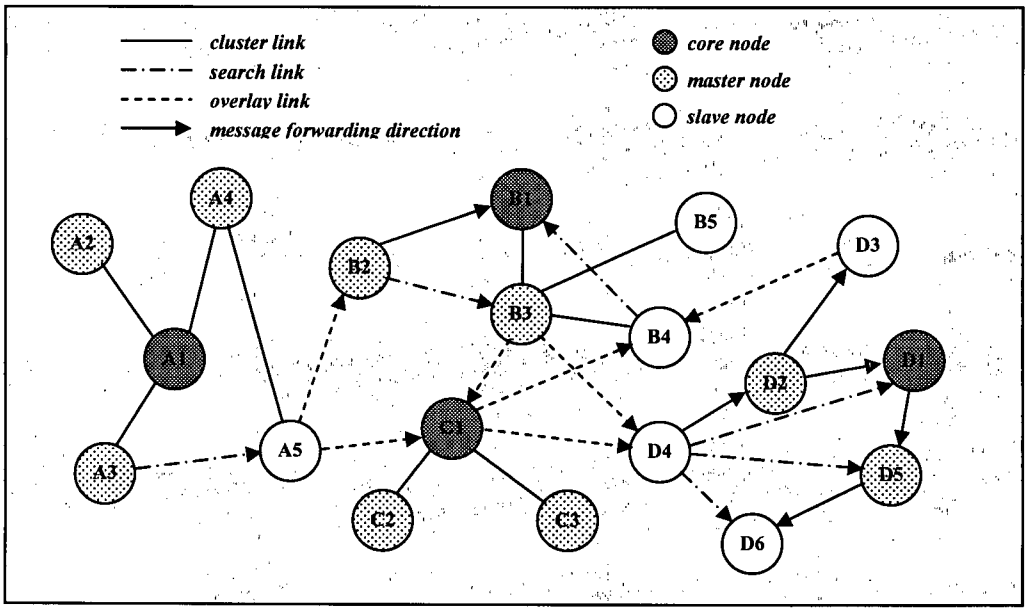

Figure 4-5 An Overlay Search Example

This particular example enumerates several interesting procedures of our *overlay search* algorithm: (1) *A3* locates the correct *agent* node *A5* directly from its *cTable*. (2) *A5* forwards *q* to cluster *B* (through an invalid *oLink* to *B2*) and *C* (through a valid *oLink* to *C1*). (3) *B2* receives *q* from *A5* and detects that *B1* is not the correct *agent* node. Then it probes *B3* (a *core node candidate*), which turns out to be the right *agent*. (4) *B3* then forwards *q* to cluster *C* (through a valid *oLink* to *C1*) and *D* (through an invalid *oLink* to *D4*). (5) *C1* receives *q* from *A5*. Since *C1* is the correct *agent* node, it then forwards *q* to cluster *B* (through an invalid *oLink* to *B4*) and *D* (through an invalid *oLink* to *D4*). (6) *D4* receives *q* from *C1*. It attempts *D6* (from *cTable*), *D5* (parent of *D6*) and *D1* (*core* node)

in order, but none of them is the *agent* node for *cat*. Then *D4* initiates a *cluster search*, which reaches the *agent* node *D3*. (7) *D3* forwards *q* to cluster *B* but not *C*, as *q* is sent from *C*. This example exhibits *overlay search* behaviour when operating on a low category consistency system. A total of 19 messages are delivered to locate all content.

Node	***cTable***	***oList***	***iTable***
A3	*cat: A5*	*n/a*	*n/a*
A5	*cat: A5*	*cat: B2, C1*	*cat: A2*
B1	*cat: B3*	*n/a*	*n/a*
B2	*cat: B1*	*n/a*	*n/a*
B3	*cat: B3*	*cat: A5, C1, D4*	*cat: B1, B5*
B4	*cat: B1*	*n/a*	*n/a*
C1	*cat: C1*	*cat: A5, B4, D4*	*cat: C2*
D1	*cat: D5*	*n/a*	*n/a*
D3	*cat: D3*	*cat: B4, C2*	*cat: D1, D5*
D4	*cat: D6*	*n/a*	*n/a*
D5	*cat: D4*	*n/a*	*n/a*
D6	*cat: D5*	*n/a*	*n/a*

Table 4-6 Data Structures in Overlay Search Example

Chapter 5

System Implementation

In this chapter, we discuss the implementation details for both the TBCP model and the COOL model, with a concentration on system architecture as well as the execution of our searching algorithms. The prototype system aims to demonstrate the feasibility of performing efficient and low-cost searches under a dynamic network environment.

5.1 System Design

As a semantic-based extension to the *tree-based clustered P2P* (TBCP) model, our *category overlay* (COOL) model requires structures and basic services provided by the TBCP model. Therefore, instead of providing two separate and independent implementations, we present a more general architecture that combines these two models. Mode switching (i.e., switching among algorithms to perform searching and publishing in different models), if necessary, is automatically detected and accomplished by the system's built-in mechanisms.

5.1.1 A General Peer-to-Peer Application Framework

To demonstrate the applicability and effectiveness of our TBCP and COOL models, it is important to apply them to various purely *unstructured* models. Therefore, it is desirable to have a general P2P application framework, in which component-based technology is used to facilitate modifications of different parts of the system, from the UI layer to the communication layer. There are some existing frameworks available. For instance, JXTA [15] (Figure 5-1), a general purpose P2P framework, focuses on providing a network computing platform and a set of open protocols to allow connected devices to communicate and collaborate in a P2P manner [14]. Its goal is to develop some basic building blocks and services to enable innovative applications for peer groups.

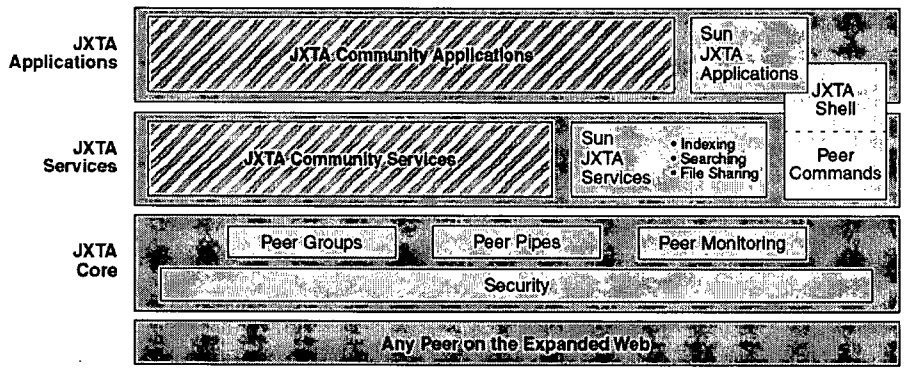

Figure 5-1 JXTA Layers [13]

However, JXTA architecture also includes specific components that provide various features such as security, ubiquity and platform independence, which are not part of our focus in this prototype system implementation. Therefore, we need a lightweight P2P application framework, which provides system level support as well as a few basic building blocks and services, to ease the task of creating a single application with different peer connection schemes.

Inspired by JXTA, we designed RTG (ready-to-go), a Peer-to-Peer framework that can be easily customized for different application domains. It is designed to simplify service algorithm replacement and extension, as well as to separate peer architecture from system services and application logic. Figure 5-2 presents the architecture of RTG.

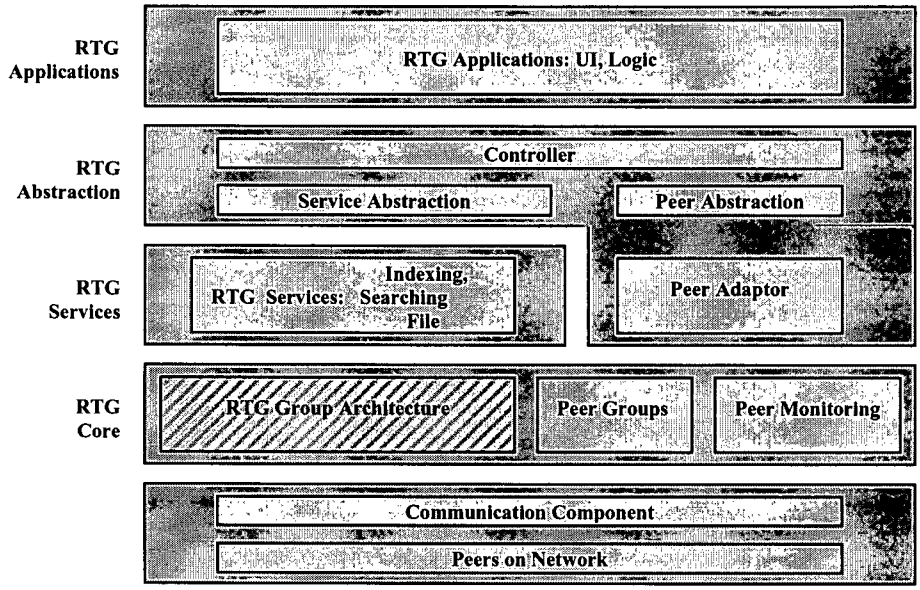

Figure 5-2 RTG (Ready-to-Go) Layers

Table 5-1 describes the basic functionalities for each layer in RTG. The *controller* in the RTG *Abstraction layer* is the key to the decoupling of application logic, services and peer architecture. More specifically, it not only controls how an application should use various services to achieve a certain goal but also bridges the gap between service components and peer architecture. Under this design, either service algorithms or peer architecture can be replaced without generating many modification requirements to other parts of the system (certain forms of P2P architecture have their own service algorithms and the decoupling concept does not apply in this case).

Layer	*Description*
Application	Application layer is where application specific components should be placed, such as user interface (UI) and business or application logic.
Abstraction (Controller)	This layer separates application domain from system architecture. Specifically, abstractions and adaptors are used to decouple business logic from any specific service or peer group implementation. A controller provides 3-way coordination (among services, peer groups and logics).
Service	Service layer contains various service modules, such as searching and indexing. Different service algorithms can be implemented here to influence system performance.
Core	This layer finalizes the actual P2P model being applied to the current system.
Communication	A Peer-to-Peer specific communication model is provided at this layer, with the purpose of removing the necessity to implement this low-level component for most P2P systems.

Table 5-1 RTG Layer Descriptions

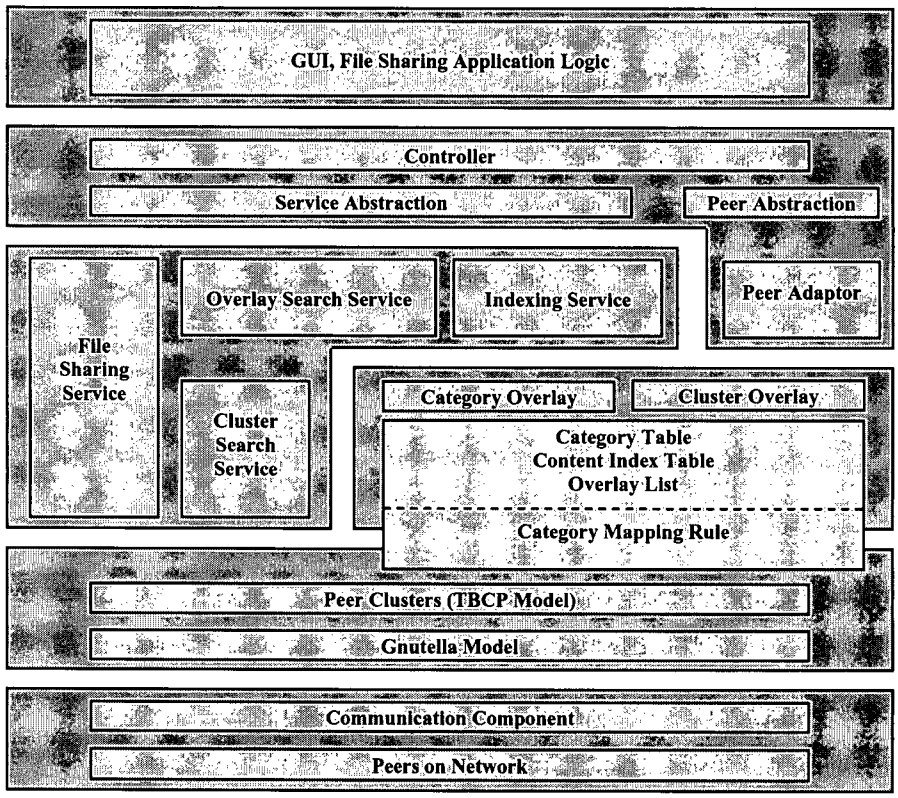

Figure 5-3 Finalized System Architecture

5.1.2 System Architecture

Based on the RTG framework, we present our finalized system architecture in Figure 5-3. A simplified Gnutella [11] model is implemented to serve as the basic *unstructured* system. TBCP then creates a hierarchical layer on top of it. Both the *file sharing service* and *cluster search service* can be implemented at this stage. If a *category mapping rule* is present, then our hybrid COOL model can also be constructed. The *overlay search service* and *indexing service* are based on both the COOL model and the *cluster search service*. The controller layers as well as an application layer are taken directly from the RTG framework.

Package	*Description*
communication	Classes in this package provide basic implementation to the *Communication Component* in Figure 5-3, including message formatting, marshalling and unmarshalling, communication channel creation, message delivery and message processing.
communication.coolsearch	Customized to provide the *Category Overlay* model specific message types and to satisfy special delivery requirements.
kernel	This package includes classes that implement the *controller component*, a generic P2P structure layer (peer abstraction) as well as a system processing model.
kernel.coolsearch	It provides the COOL model specific extensions to generic classes in kernel package, such as peer node identification, peer adaptor and message processing scheme.
property	This package stores and maintains generic system parameters
property.coolsearch	Stores and maintains the COOL Model specific parameters, such as content classification rule.
resource	Resource package provides an abstract layer for resource storage, maintenance and sharing. Implementation is given for a generic resource type.
resource.coolsearch	Implementation for a COOL model based resource
service	Service abstraction layer.
service.coolsearch	This package contains classes that implement various service algorithms. Other classes in this package implement creation and maintenance algorithms for the TBCP and COOL models.
ui.coolsearch	GUI implementation
util.coolsearch	Utility package that contains general purpose data structure implementation, general constants and other utility functions.

Table 5-2 Packages for Prototype Implementation

5.2 Implementation Details

5.2.1 Core Classes

The following core classes provide important functionalities for the prototype system. Each entry is composed of a full class name, a direct super class or interface name (to the right of the class name) and a description. The next section describes the communication and cooperation among these classes to realize various system functionalities.

communication.MessageImpl communication.Message

Providing generic marshalling and unmarshalling services.

communication.coolsearch.CoolSearchClient communication.Client

Providing a *communication layer* interface for peers at core layer to deliver messages.

communication.coolsearch.CoolSearchUnblockedRequest java.lang.Thread

Providing unblocked message delivery interface to peer nodes at the *core layer*.

kernel.LocalController

Implementing core functionalities in the *controller* component at the *abstraction layer*.

kernel.LocalNode kernel.AbstractNode

A *Peer Abstraction* representing a node object on the current peer, providing a common interface to the *controller* for functionalities and services available to a local peer.

kernel.RemoteNode kernel kernel.AbstractNode

A *Peer Abstraction* representing a node object on a remote peer, providing marshalling and unmarshalling services to the *controller*.

kernel.coolsearch.CoolSearchFactory kernel.factory

Factory design pattern to release the tight coupling relationship among specific implementations.

kernel.coolsearch.CoolSearchIncomingMsgServant kernel.ImcomingMsgServant

Dispatching various incoming requests to different parts of the system.

resource.ResourceDBManager

Managing the local content database.

resource.coolsearch.CoolSearchGeneralResource resource.GeneralResource

Representing a generic COOL model resource type, providing resource specific marshalling and unmarshalling services.

service.coolsearch.CategoryManager

Maintaining the *category overlays* and providing the COOL model-specific maintaining and controlling functionalities.

service.coolsearch.ClusterService

Implementing the *cluster search* algorithm.

service.coolsearch.ContentIndex

Providing the *content index table (iTable)* management and indexing service.

service.coolsearch.CoolSearchStrategy service.Service

A *Service Abstraction* to decouple the *controller* from any specific service implementation. Combining with a *Peer Abstraction*, they provide the common interfaces to facilitate the 3-way communication among the *controller*, *peers* and various *services*.

service.coolsearch.GroupManager

Implementing the clustering and cluster maintenance algorithms in the TBCP model.

service.coolsearch.RetrieveService

Providing file-sharing service.

service.coolsearch.SearchService

Implementing the *cluster overlay* and *category overlay* search algorithms.

ui.coolsearch.CoolSearchUI javax.swing.JFrame

Graphic user interface.

5.2.2 System Processing Flow

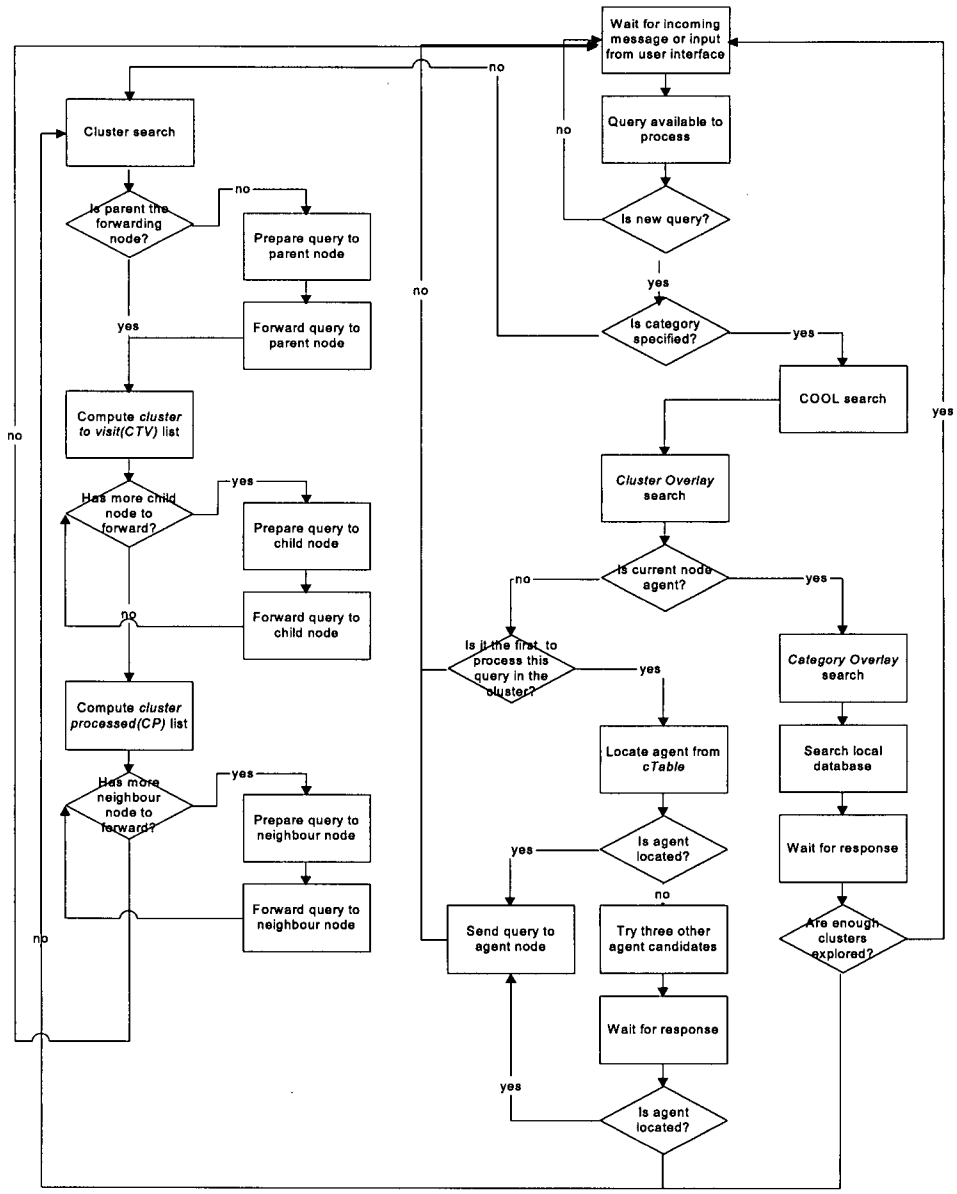

Figure 5-4 Control Flowchart for Complete Overlay Search Algorithm

This section describes a complete search control flow and several important execution flows implemented in our prototype system. The control flowchart (Figure 5-4) illustrates the details of each search algorithm (*cluster* and *overlay search*) and their collaboration.

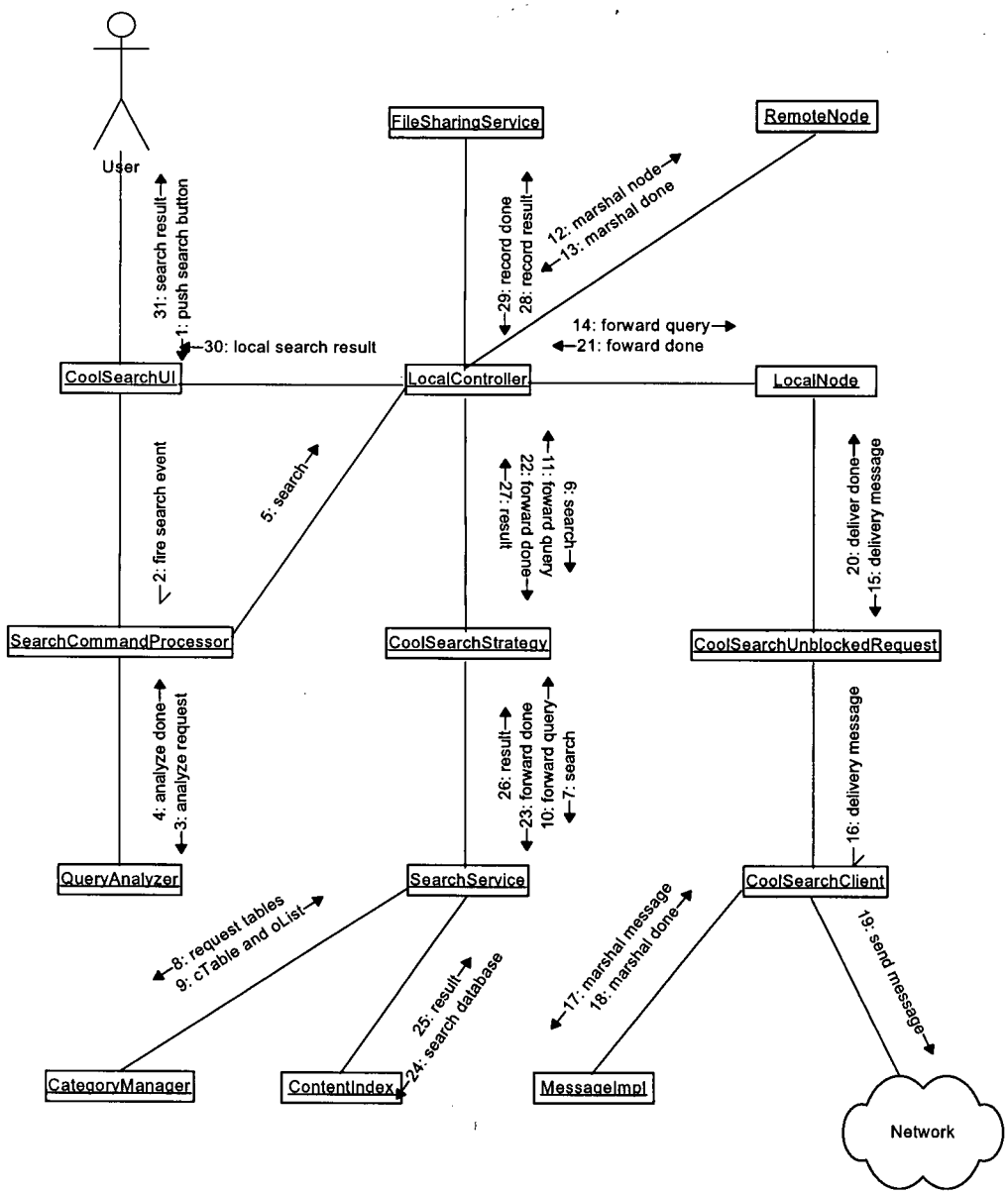

Figure 5-5 Collaboration Diagram for Query Initiation

Figures 5-5 to 5-7, on the other hand, present detailed collaboration diagrams on three typical searching related scenarios, with the purpose of illustrating the cooperation among the core classes to realize major functionalities.

The above diagram (Figure 5-5) illustrates class collaboration under the scenario when a user initiates a new query with a certain category and a list of keywords, including how an *overlay search* is executed as well as how to return the results located in the local database.

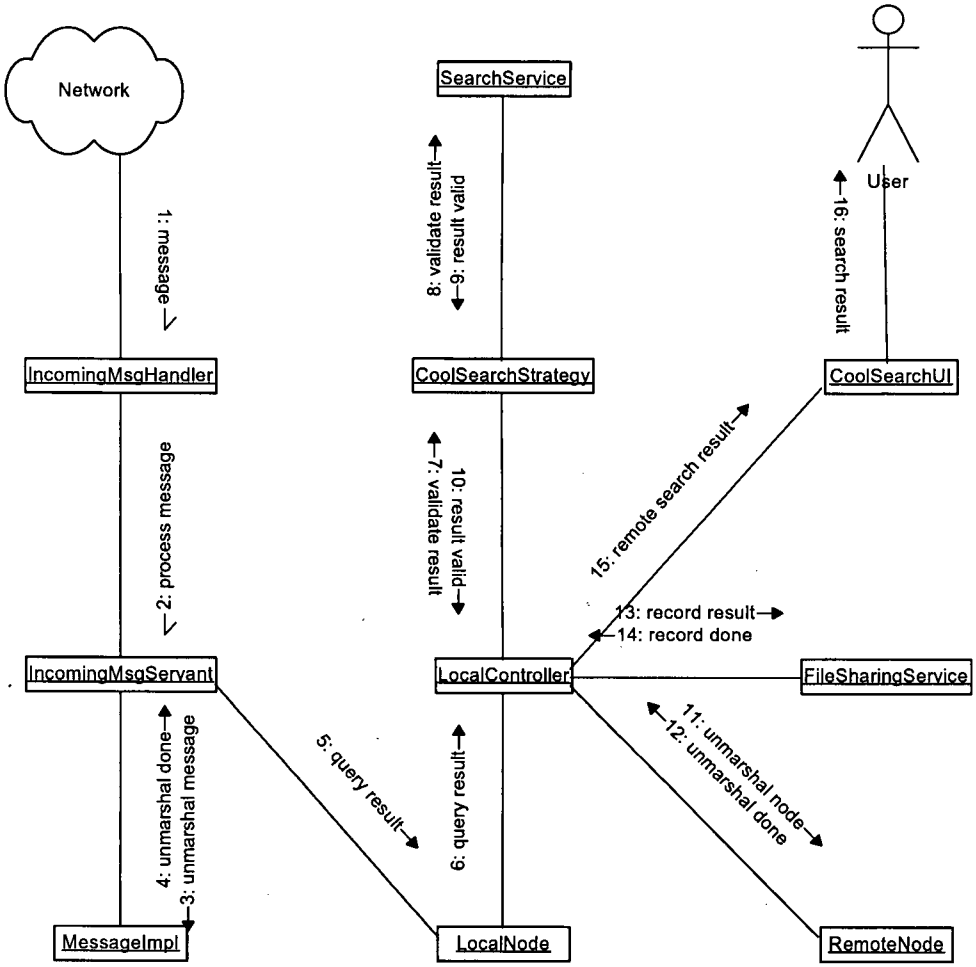

Figure 5-6 Collaboration Diagram for Result Processing

The next collaboration diagram (Figure 5-6) describes the operations performed when a result message arrives at the current node, including results validation, processing and presentation to users.

When a query message is forwarded to the current node (the agent node for the category being searched), Figure 5-7 presents how this query is processed and forwarded to other nodes in the same *category overlay*, as well as how results in the local database are found and returned to remote users.

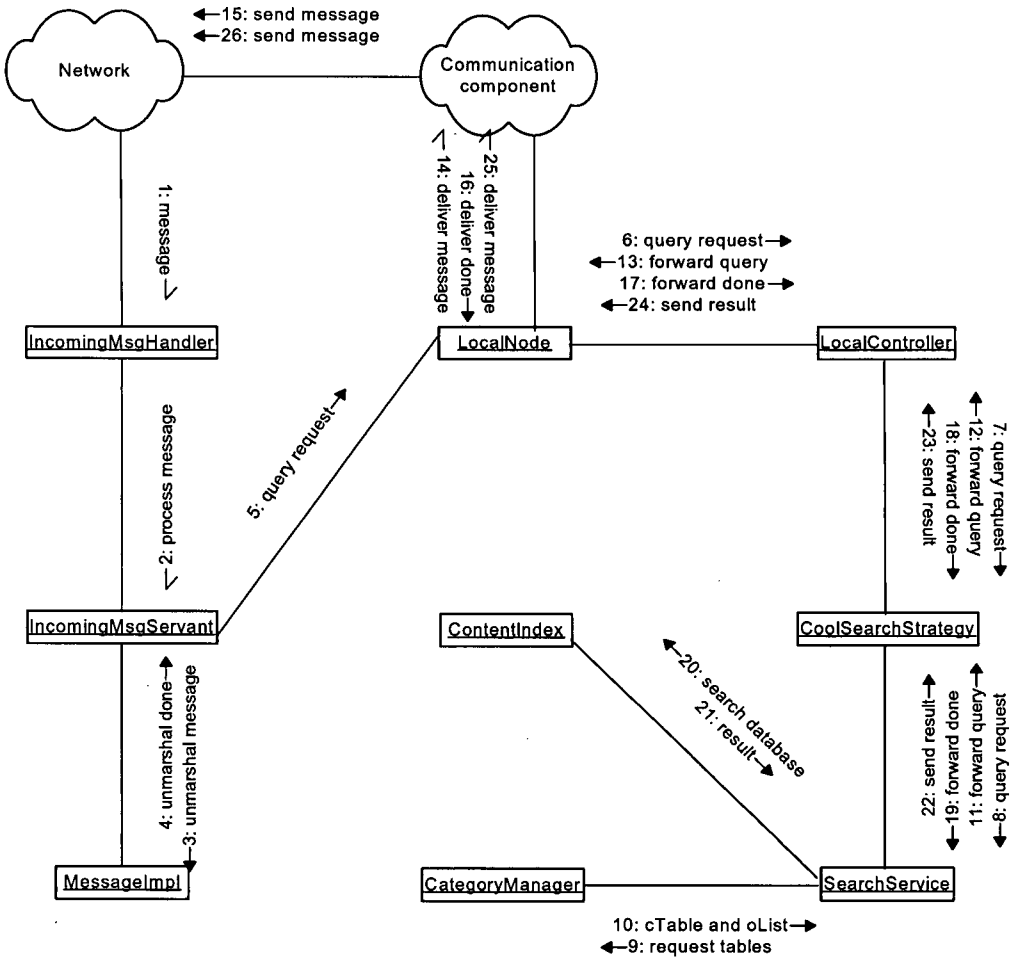

Figure 5-6 Collaboration Diagram for Query Processing and Forwarding

Chapter 6

Evaluation

6.1 Theoretical Analysis

This section presents a mathematical analysis intended to identify and characterize situations in which applying the *cluster search* on the TBCP model or the *overlay search* on the COOL model can lead to significant overall system performance improvement. Both approaches are compared to the basic flooding algorithm on a general *unstructured* P2P system, as this is the most common approach used in the literature to make comparisons with other efficient search algorithms.

First we compare the costs of the Gnutella search (*blind flooding*) and the *cluster search*. We compute both the bandwidth savings from the *cluster search* and the extra maintenance costs from the TBCP model. We then identify the scenarios in which the bandwidth savings are greater than those extra costs.

The bandwidth consumed by the Gnutella search can be defined as:

$$C_{flooding} = \overline{S}_{flooding} * \left(1 + \sum_{i=1}^{N}(D_i - 1)\right) = \overline{S}_{flooding} * \left(1 + N * (\overline{D}_{gnutella} - 1)\right) \qquad (6\text{-}1)$$

$C_{flooding}$: the total bandwidth consumption of a complete flooding

$\overline{S}_{flooding}$: average flooding message size

N : number of nodes in the network

D_i : degree of node i

$\overline{D}_{gnutella}$: average node degree of a Gnutella system

Here, we assume that upon receiving a query message, the current node will forward this message to all its neighbours, except the one from which the message comes. Therefore, two messages for the same query will go through any single link.

The bandwidth consumption of the *cluster search* can be defined as:

$$C_{cluster} = \overline{S}_{cluster} * \left(1 + \sum_{i=1}^{G}((DC_i - 1) * L_i) + \sum_{i=1}^{G}(CS_i * W_i)\right)$$

$$= \overline{S}_{cluster} * \left(1 + \frac{N}{\overline{CS}_{TBCP}} * \left(\overline{DC}_{TBCP} - 1\right) * l + \left(N - \frac{N}{\overline{CS}_{TBCP}}\right) * w\right) \qquad (6\text{-}2)$$

$C_{cluster}$: the total bandwidth consumption of a complete cluster search

$\overline{S}_{cluster}$: average *cluster search* message size

G : number of clusters

DC_i : degree of cluster i (number of clusters to which i has direct connections)

$\overline{DC}_{TBCP}$: average cluster degree of a TBCP model based system

CS_i : cluster size of cluster i

$\overline{CS}_{TBCP}$: average cluster size of TBCP model

L_i : number of duplicated *inter-cluster* links (from cluster i) used to deliver messages

W_i : rate of duplication within cluster i

A typical *cluster search* consists of two parts, the *intra-cluster* and *inter-cluster* searches. The first summation in equation 6-2 computes the number of messages required to do the *inter-cluster* search, while the second summation computes for the *inter-cluster* search. L_i represents the number of duplicated links between each pair of clusters. In the worst case, when all duplicated *inter-cluster* links are used to forward messages, the total cost of the first summation is equal to the cost of *blind flooding* through these links. W_i is the ratio between the number of messages forwarded through cluster i's *intra-cluster* links and the total number of *intra-cluster* links. Because each cluster is a well organized structure and because message forwarding, within a cluster, can only go through certain paths, therefore,

$$1 \le W_i \le 2$$

From these observations, we can see that even in the worst case, the cost of *blind flooding* is still an upper bound compared to the cost of *cluster search*. But with the *CTV (cluster to visit)* list and the *CP (cluster processed)* list, the average cost for the *cluster search* is much less. Parameters l and w are used to compute this average search cost.

Now we compare the maintenance costs. For Gnutella, maintenance is a series of ping-pong messages [11] between each pair of connected nodes:

$$\begin{aligned} M_{gnutella} &= \overline{S}_{gnutella} * P_{gnutella}\left(1+\sum_{i=1}^{N}(D_i-1)\right) \\ &= \overline{S}_{gnutella} * P_{gnutella}\left(1+N*(\overline{D}_{gnutella}-1)\right) \end{aligned} \tag{6-3}$$

$M_{gnutella}$: maintenance cost of Gnutella model

$\overline{S}_{gnutella}$: average Gnutella maintenance message size

$P_{gnutella}$: probability of sending a probing message to another node

Here, we calculate the maintenance cost for one second; therefore, for each node, $P_{gnutella}$ is the probability of sending a report message in each second and $1/P_{gnutella}$ is the expected time of initiating one round of maintenance.

We define the cost of TBCP maintenance as:

$$M_{TBCP} = M_{gnutella} + \overline{S}_{master} * N_{master} * P_{master} + \overline{S}_{core} * (N - N_{core}) * P_{core} \qquad (6\text{-}4)$$

M_{TBCP} : maintenance cost of TBCP model

$\overline{S}_{master}$: average TBCP *master report* message size

N_{master} : total number of *master nodes*

P_{master} : probability of sending a *master report* in each second

$\overline{S}_{core}$: average TBCP *core report* message size

N_{core} : total number of *core nodes*

P_{core} : probability of sending a *core report* in each second

The last two parts in equation 6-4 represent the total maintenance costs for the *master reporting* mechanism and the *core reporting* mechanism, respectively. $M_{gnutella}$ is an upper bound for M_{TBCP}, and their cost difference is determined by parameters P_{master} and P_{core}. Regarding the TBCP model, these two parameters only affect the effectiveness and quality of handling the cluster creation as well as other normal and abnormal events.

Combining the above four equations, we can identify the scenarios in which the TBCP model and the *cluster search* are definitely favoured:

$$(C_{flooding} - C_{cluster}) * F_{search} \geq M_{TBCP} - M_{gnutella}$$

$$F_{search} \geq \frac{M_{TBCP} - M_{gnutella}}{C_{flooding} - C_{cluster}} \qquad (6\text{-}5)$$

Therefore, as long as the frequency of searching (F_{search}) is greater than a certain value (i.e., condition 6-5 is satisfied), applying the *cluster search* can decrease resource consumption. To better illustrate this result, we use simulation results (Figure 6-15) on the *Internet* topology to quantify the above condition.

We select a network with 10,000 nodes (the average node degree is 4) and construct a TBCP model on top of it. The cluster creation algorithm generates 35 clusters, 921 master nodes and 9044 slave nodes. Other statistics are presented in Table 6-1.

Parameter	N	$\overline{D}_{gnutella}$	$\overline{S}_{flooding}$	$\overline{S}_{cluster}$	$\overline{S}_{gnutella}$	$\overline{S}_{master}$	$\overline{S}_{core}$
Value	10,000	4	131	161	276	110	3509

Table 6-1 Flooding vs Cluster Search

Assuming messages are sent with TCP/IP protocol on top of IEEE 802.3 (Ethernet), the size of each type of messages is then computed by adding the size of actual message body to the TCP/IP and Ethernet frame header as well as the Ethernet frame trailer, a total of 66 bytes. A general Gnutella Descriptor header is 23-byte long plus a 2-byte long query header [11]. We also assume an average query size is 5 keywords, and the average length of each keyword is 8 ASCII characters, which gives us a 40-byte long query body (the selection is arbitrary and does not have a significant effect on our results). Therefore, an average Gnutella query message in this simulation is 131 bytes. A typical *cluster search* query message, on the other hand, contains additional spaces for the *CTV* list and the *CP* list. We restrict each list to contain 5 entries. The average size of a *cluster search* message is 131 + (5 * 6 + 5 * 6) / 2 = 161 bytes.

As for maintenance, a pair of Ping-Pong messages has a total of 89 * 2 + 14 = 192 bytes. We set the frequency of the occurrence of normal and abnormal events at 1.6% per minute [33], and both P_{master} and P_{core} at 1/600. Therefore, as each master node is assessed, every 600 seconds, only 1.6 events happen to its direct children. The average size for a TBCP's *master report* body is therefore (6 + 6 + 1) * 1.6 = 21 bytes, where the two 6 stand for node identity and timestamp respectively, 1 for event type and 1.6 for the average number of events occurred in 600 seconds. The average size for a TBCP's *core report* body is 285 * 2 * 6 = 3,420 bytes, where 285 is the average cluster size, and the other values have the same meaning as in the last computation. At this maintenance rate, the bandwidth usage for a *core report* is only 3,509 / 600 ≈ 47 bps. Therefore, the *core report* maintenance traffic for core node (30 master nodes) is 47 * 30 ≈ 1,410 bps.

A complete *cluster search* in this simulation requires a total of 11,654 messages. By substituting these values into equation 6-5, we get $F_{search} \approx 0.03$, which suggests that as long as there is a query submitted to the system every 33 seconds, our *cluster search* is more desirable than flooding. In other words, if every living node initiates a new query every 4 days, then the *cluster search* along with the TBCP model is more favourable.

Now we compare the costs of the Gnutella search with the *overlay search*. The bandwidth consumed by the *overlay search* is defined as:

$$C_{category} = \overline{S}_{category} * \left(1 + \frac{N}{\overline{CS}_{COOL}} * \left(\overline{DC}_{COOL} - 1\right) * \left(1 + x_1 * P_{direct} + 4 * x_2 * P_{indirect}\right) + C_{cluster} * P_{cluster}\right) \quad (6\text{-}6)$$

$C_{category}$: the total bandwidth consumption of a complete *overlay search*

$\overline{S}_{category}$: average *overlay search* message size

$\overline{CS}_{COOL}$: average cluster size

$\overline{DC}_{COOL}$: average cluster degree

P_{direct} : probability of locating the agent node by sending only one message

$P_{indirect}$: probability of locating the agent node by sending four messages

$P_{cluster}$: probability of performing the cluster search

As shown in Figures 4-3 and 4-4, an *overlay search* can be categorized into one of three cases: (1) the *agent* node can be located by looking directly at *cTable*. (2) the *agent* node can be located by probing parents, *core* node, etc., and in this case, the cost of a search is three messages. (3) the *agent* node cannot be located through a conventional approach and the *cluster search* is then used instead. All these three cases (Equation 6-6) are associated with a probability value. The *overlay search* is only performed among *agent* nodes, and therefore the cost of a search is much lower than *flooding*.

As for the maintenance cost, the COOL model has the following maintenance requirements:

$$M_{COOL} = M_{cluster} + N * P_{intrareport} * \overline{S}_{intrareport} + N * P_{interreport} * \overline{S}_{interreport} \qquad (6\text{-}7)$$

M_{COOL} : maintenance cost of COOL model

$P_{intrareport}$: probability of sending an intra-cluster report

$\overline{S}_{intrareport}$: average message size of intra-cluster aggregation report

$P_{interreport}$: probability of sending an inter-cluster report

$\overline{S}_{interreport}$: average message size of inter-cluster category report

To identify the condition for which the *overlay search* is desirable, we use equations 6-1, 6-3, 6-6, and 6-7 to compute the ratio:

$$\left(C_{flooding} - C_{category}\right) * F'_{search} \geq M_{COOL} - M_{gnutella}$$

$$F'_{search} \geq \frac{M_{COOL} - M_{gnutella}}{C_{flooding} - C_{category}} \qquad (6\text{-}8)$$

The COOL model, along with the *overlay search*, can create huge network resource savings if equation 6-8 holds. A quantitative analysis can also be carried out by using the simulation results presented in next section. We select the same network, as presented in above study, and construct COOL model on top of it. We assume that each node maintains 1,000 indices (it leads to 10 TB data if the average content size is 1 MB), each index has 10 keywords, and each keyword is 6-byte long in average. We set the category migration rate to be 1.6 % per minute (same as the network dynamics) and both overlay maintenance rates to be once per 3 minutes. With these statistics, the *cTable* consistency is at 80% (see Figure 6-7) and the average maintenance traffic, including cluster maintenance (for the TBCP model), overlay maintenance (for the COOL model) and the category migration traffic, is 231 bps for *slave node* and 1,891 bps for *core node*. Moreover, for each query, 1132 messages in average are forwarded to locate all the content within the system.

By substituting these values into above equations, we get $F_{search} \approx 0.07$, which implies that as long as there is a query submitted to the system every 15 seconds, our *overlay search* is more desirable than flooding. Moreover, applying the same study to compare the *cluster search* and *overlay search*, we observe that if the query submission rate is greater than once per 8 seconds, then the *overlay search* has better performance (in terms of the addition of search and maintenance traffic) than the *cluster search*. In other words, the *overlay search* reduces overall network traffic if each node submits a query every 2 days (compared to *flooding*) or every 1 day (compared to *cluster search*).

Figure 6-1 demonstrates the above analysis by computing the network bandwidth consumption, for search only as well as for the total of search and maintenance costs, at different search frequency levels. This figure clearly shows the break-even points (e.g.,

33 seconds, 15 seconds, and 8 seconds) as well as the increasing trends for overall costs of various models. At the low frequency level, the COOL model and *overlay search* consumes even more bandwidth than the basic Gnutella model and flooding, due to its high maintenance overhead. But as the frequency increases, the overall cost increases much more slowly than both flooding and the *cluster search.*

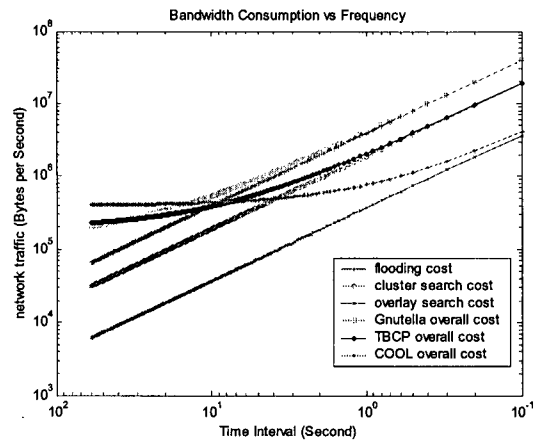

Figure 6-1 Bandwith Consumption (Log Scaled) vs Search Frequency (Log Scaled)

6.2 Empirical Analysis

6.2.1 Simulator and Network Topology

To get further insight into the behaviour and performance of our new models and search algorithms, a reasonably sized network should be set up to run our prototype implementation. However, a quantitative analysis of a large-scale P2P system in a real environment remains a difficult problem due to the lack of a central authority as well as the related difficulties of measurement and real behaviour reproduction. Due to limited system and network resources, popular approaches, including setting up a small clustered

environment or deploying applications on Planet Lab, are more suitable for demonstrating some benefits of a particular implementation, rather than the scalability or stability of a new algorithm. Considering these restrictions, we decided to use simulations to further explore the behaviour of the *cluster search* and *overlay search* algorithms.

For the purpose of this simulation, we developed an event-based simulator in Java. This simulator uses an nondeterministic model to generate normal and abnormal system events and to execute all the basic functionalities and algorithms defined in this thesis, including cluster creation, cluster maintenance, cluster consistency resolution, *cluster search*, category mapping, overlay creation, overlay maintenance, overlay consistency resolution and *overlay search*. There are four major parts in this simulator, each of which serves, separately or combined, a specific simulation purpose:

- *cluster creation*: demonstrates the effectiveness of using our cluster creation algorithm to construct the TBCP model that possesses the *similarity* property.
- *object distribution*: generates content objects for randomly selected categories and distributes them, by assigning various popularities, throughout the network.
- *Consistency resolution:* demonstrates the correctness and effectiveness of our consistency resolution mechanisms. Our simulator can create and maintain various levels of inconsistency in both the TBCP and COOL models, which makes it possible for us to monitor the impact of inconsistency on a search.
- *Search:* computes the number of messages forwarded or the number of nodes visited during a single round of search, for the purpose of illustrating the performance of our search algorithms. For each search, users of our simulator are able to adjust and configure search coverage, such as to visit only a portion

of nodes within a cluster, or to only locate a desired number of objects.

To compare the performance improvement of our search algorithms operating on different network topologies with typical P2P statistics, most of our simulations were executed on three network topologies, including two router-level topologies, the Waxman model [41] and the Barábasi-Albert model [3], and one AS (Autonomous System) level topology, the Inet-3.0 model [42].

The Waxman model is one of the earliest network topology models, featuring a simple random topology generation algorithm. In our simulation, we use Brite v2.1 [23] tool to generate different sized random networks with a common average node degree 10.

Both the Barábasi-Albert (BA) model and the Inet-3.0 model features a Power Law type node degree distribution. In 1999, Faloutsos *et al.* [7] found several power laws relating to the topology of the Internet. One of these power laws was $f_d \propto d^o$, which indicates that the frequency, f_d, of an outdegree, d, is proportional to the outdegree to the power of a constant o. The BA model is based on a related observation that the probability $P(k)$ for a vertex in the network interacts with k other vertices decays as a power law, following $P(k) \sim k^{-\gamma}$. We also use Brite v2.1 to generate network topologies based on the router-level BA model, with an average outdegree 10. The Inet-3.0 aims to reproduce a modern AS-level Internet topology. It adopts a new exponential growth law $F(d) = e^c d^{at+b}$ to calculate degree distribution. We use this topology generator to generate networks in which 30% of the total nodes are degree-one nodes. We refer to this model as the *Internet* model in our later discussion.

6.2.2 Clustering Algorithm Analysis

The theoretical analysis in the previous section shows that average cluster size is a very important parameter for determining both search efficiency and maintenance cost. However, a more important factor hidden in the above analysis is how clusters differ from each other. As explained in Chapter 3, one of the most important objectives of our TBCP model's cluster creation algorithm is to produce clusters that are "similar" in size and structure. Cluster similarity has a potential impact on *cluster degree*, *local knowledge consistency* and other performance-related parameters. Moreover, cluster similarity is the key to achieving a workload balance in our COOL model, because if clusters have similar size and structure, then the overall search workload can be evenly distributed into clusters, which can further distribute the load onto individual nodes. Large clusters might experience low local knowledge consistency, which directly increases search cost. On the other hand, in small clusters, participating nodes are frequently involved in query forwarding and therefore also experience a high search cost.

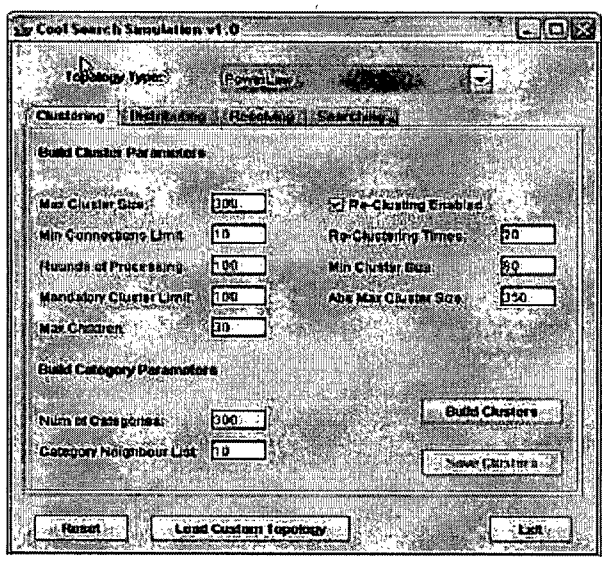

Figure 6-2 Parameter Settings in Cluster Creation Simulation

This simulation intends to demonstrate that our cluster creation algorithm is able to optimize cluster similarity. We run this simulation on all three topologies described in Section 6.2.1. For each topology, we generate a 10,000-node network and perform clustering algorithm 100 times. Parameter settings are configured to optimize clustering on different network topologies. Figure 6-2 shows the parameter settings for the *Internet* topology. Both *Max Cluster Size* and *Abs Max Cluster Size* are only a stage-based cluster size limit. The final cluster size may exceed both values. In this particular setting, any node with more than 100 connections must serve as the *core node*. Meanwhile, any cluster that has less than 80 nodes must merge into other clusters. The results are presented in Figures 6-3 to 6-5. Most of the clusters on all three topologies have a similar size. The existence of small clusters is due to the fact that a large amount of nodes have very limited connections and thereby cannot attempt to join more clusters.

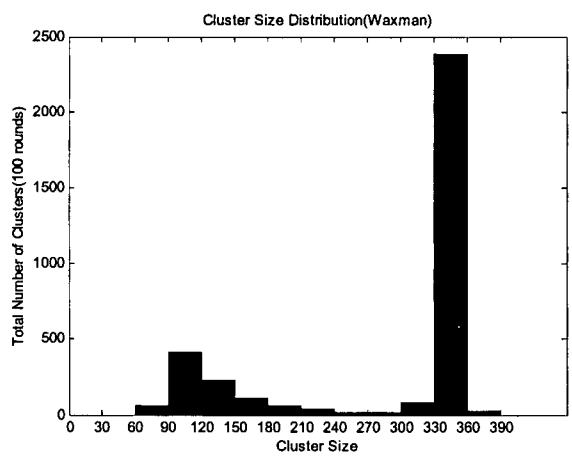

Figure 6-3 Cluster Size Distribution on Waxman Model (100 times)

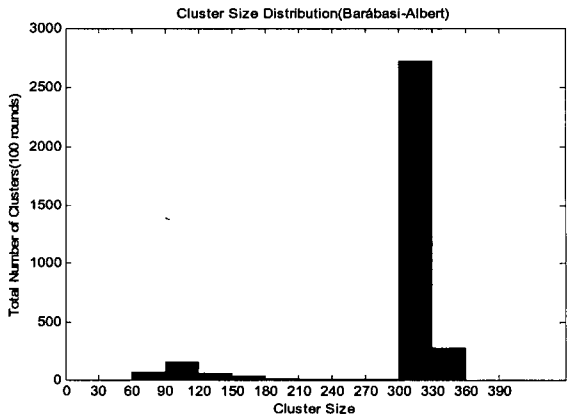

Figure 6-4 Cluster Size Distribution on Barábasi-Albert Model (100 times)

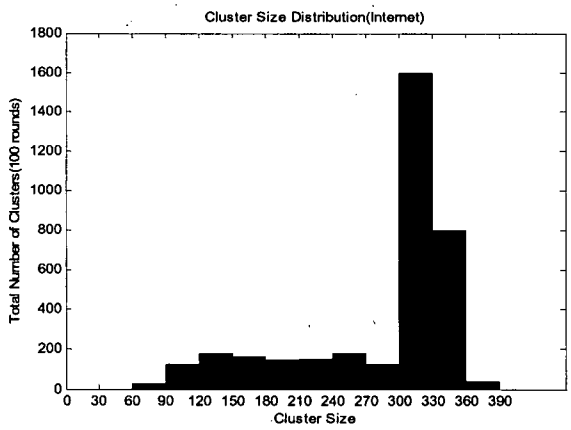

Figure 6-5 Cluster Size Distribution on Internet Model (100 times)

6.2.3 Consistency Resolution Analysis

Consistency in both the *category table (cTable)* and the *overlay list (oList)* is the most important factor for determining the effectiveness of our *overlay search* algorithm. High consistency in the *cTable* implies a high probability of locating the right *agent node* without requiring the *cluster search* (or broadcasting). Meanwhile, high consistency in

the *oList* can also decrease the necessity of performing the *cluster search* (see algorithm in Figure 4-4). In equation 6-6, these probabilities are represented as a high P_{direct} and $P_{indirect}$, as well as a low $P_{cluster}$ value. However, the cost of achieving high consistency is to execute maintenance more frequently. This simulation intends to demonstrate that by using our *periodical aggregation report* scheme, a relatively high level of consistency in the *cTable* can be sustained. Since the cause of inconsistency in the *cTable* is *category migration*, we decided to invoke *category migration* explicitly, at a frequent rate, rather than reproduce the scenarios in which *migration* should occur. To avoid over optimism, we perform *migration* at the rate that would not occur in a real-world environment. The simulation is performed on an *Internet* topology network with 10,000 nodes. On top of it, we construct the COOL model, which generates a total of 35 clusters. We also generate 300 categories and map them onto every cluster. Figure 6-6 presents the parameter settings in the following simulation. In every minute, 160 normal system events (join and leaving) occur in the whole network, and every node should perform a *category migration*. We vary the maintenance rate and run the simulation for a period of 120,000 seconds. Figure 6-7 is the simulation result. Each spot on this figure reflects a sampling once every 600 seconds. Four maintenance rates, 60, 120, 180 and 240, are selected, which imply how often, in terms of seconds, each node will send a maintenance report. Assuming each report contains a complete *cTable*, the size of a report is only 1,800 bytes. If the maintenance rate is 240, then the maintenance cost is only 60 bps. This result demonstrates that high consistency (above 70%) in the *cTable* can be achieved only by requiring a low-cost maintenance. Our simulation results in the next section show that a 70% consistency rate is still able to decrease the search cost dramatically.

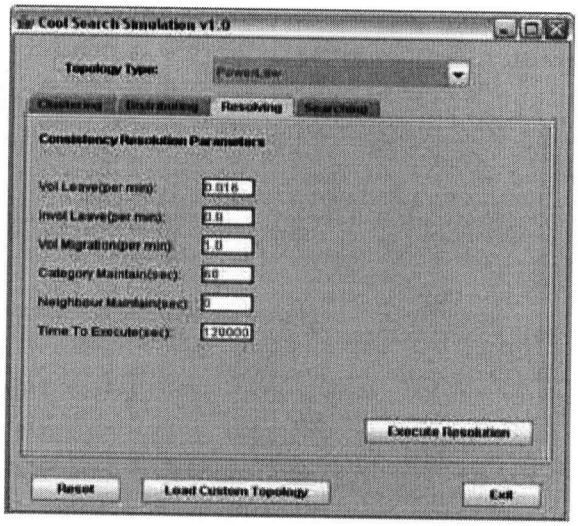

Figure 6-6 Parameter Settings in Consistency Resolution Analysis

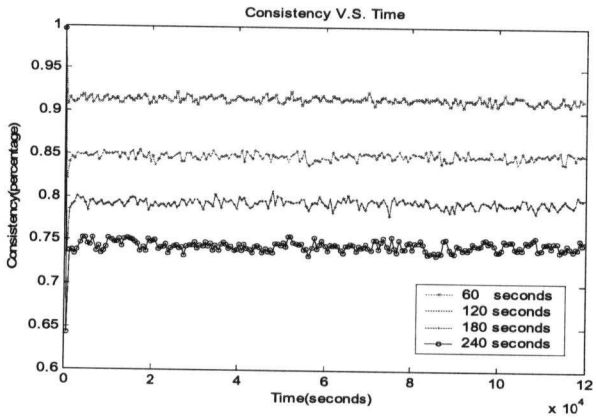

Figure 6-7 Consistency Resolution on Internet Model

6.2.4 Searching Algorithm Analysis

A major criterion for evaluating the effectiveness of any P2P search algorithm is the redundant or unnecessary workload generated on both client machines and in the network. Minimizing this overhead is a primary goal of this study. To further explore the performance of our *cluster search* and *overlay search* algorithms, we define the

processing overhead as redundant nodes required to process the same query, and define the network overhead as duplicated messages in the network.

Figures 6-8 to 6-10 compare the network overhead for different search algorithms, including flooding, cluster search, and overlay search with *cTable* consistency ranging from 100% to 70% on the same network. Figures 6-11 to 6-13 compare the processing overhead for the same set of algorithms as above. We set TTL as infinity to ensure that all the objects are located. To further eliminate the impact from coincidence, we set the popularity of each object at a minimum of 10%. We then count the number of messages sent throughout the network. Each simulation is performed on three different network topologies. For each topology, we vary network size and repeat the tests 1,000 times. The average number is spotted in the above figures. The results reveal that the *cluster search* is able to reduce huge network overhead (on all three topologies) when compared with flooding. But among the three different algorithms, the *overlay search* (with different consistency levels) is undoubtedly the most effective approach for reducing both network and processing overhead.

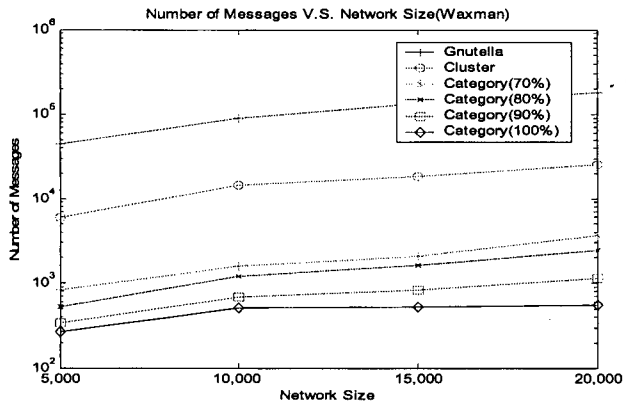

Figure 6-8 Network Overhead (Log Scaled) on Waxman Model

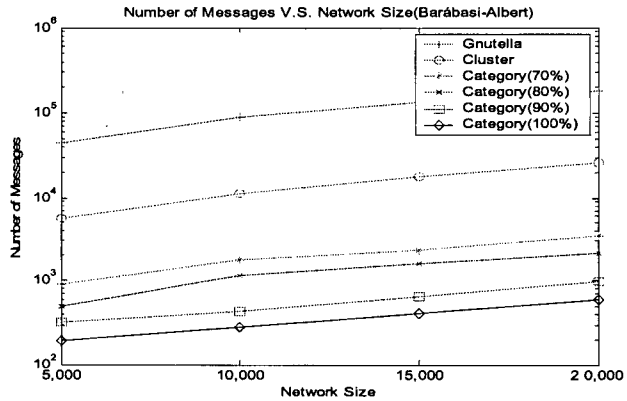

Figure 6-9 Network Overhead (Log Scaled) on Barábasi-Albert Model

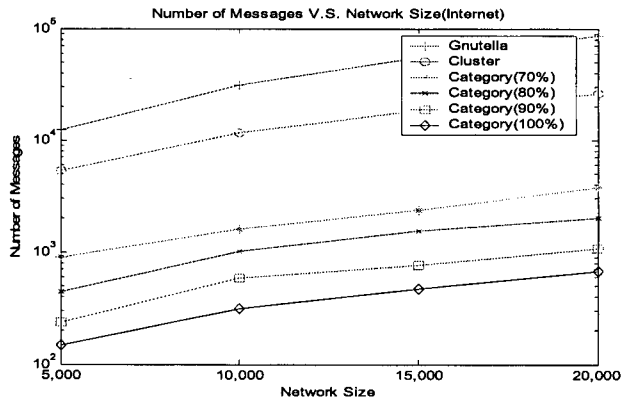

Figure 6-10 Network Overhead (Log Scaled) on Internet Model

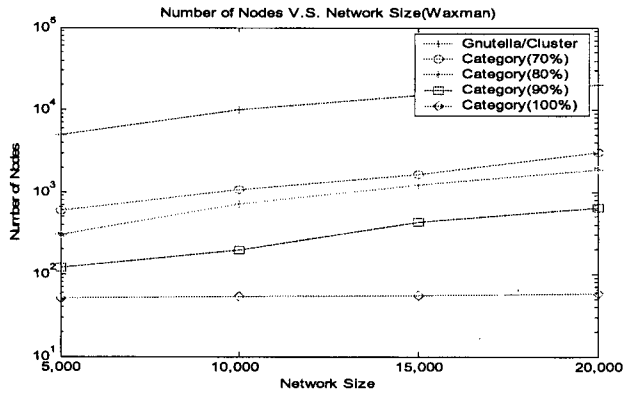

Figure 6-11 Processing Overhead (Log Scaled) on Waxman Model

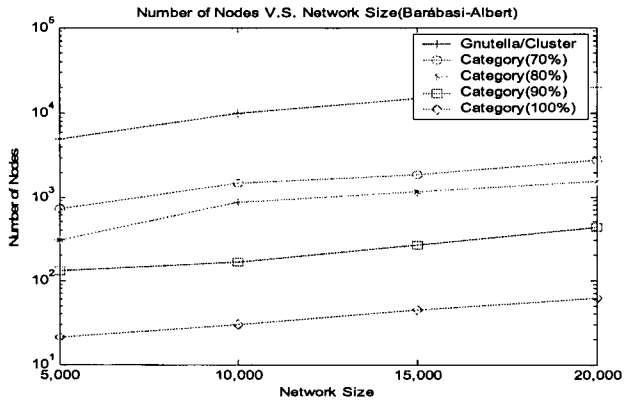

Figure 6-12 Processing Overhead (Log Scaled) on Barábasi-Albert Model

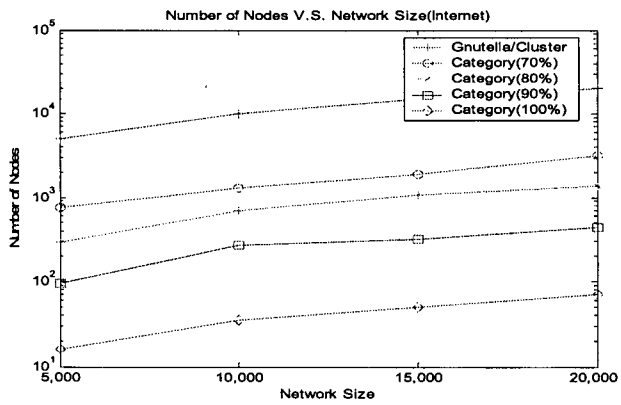

Figure 6-13 Processing Overhead (Log Scaled) on Internet Model

Flooding uses TTL (time-to-live) to control the number of hops that a single message should go through before it is discarded. This is a very effective scheme for decreasing the number of duplicated messages when searching popular content, as enough hits can usually be located in small search coverage. The same idea applies to both *cluster search* and *overlay search*. Figure 6-14 illustrates the increase of network overhead with respect to the increase of TTL. This simulation is performed 1,000 times

on a 10,000-node *Internet* network. Average values are used to generate this figure. The result shows that the TTL control scheme works better on both *cluster search* and *overlay search*, because the mild increase in duplicated messages when applying these two approaches gives TTL more control over search coverage.

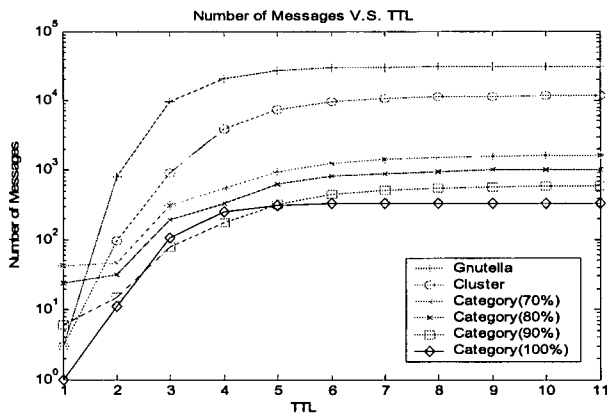

Figure 6-14 Network Overhead (Log Scaled) vs TTL

As the TTL control mechanism indicates, it is not always necessary to search the complete network if a large amount of hits can be located in a smaller range. Therefore, another important factor for evaluating the quality of search algorithms is to compare their overhead when only a portion of the objects need to be located. This simulation compares the network overhead when different algorithms are used to meet various success rates (i.e., the portion of objects needing to be located). The simulation ends as soon as the desired portion of the objects is located. At this point, nodes that have not processed the query will immediately discard the query message. This will improve the relevance of the results by shifting the control away from the TTL. Figure 6-15 shows the simulation result. The search is also performed 1,000 times on an *Internet* network with 10,000 nodes. Objects are distributed onto 10%, 1% or 0.1% randomly selected nodes, representing high, mid and low popularities, respectively. The result confirms that with

cluster search and *overlay search*, fewer messages are required to locate the desired number of objects, and the trends of increase in the number of duplicated messages are smoother than in flooding.

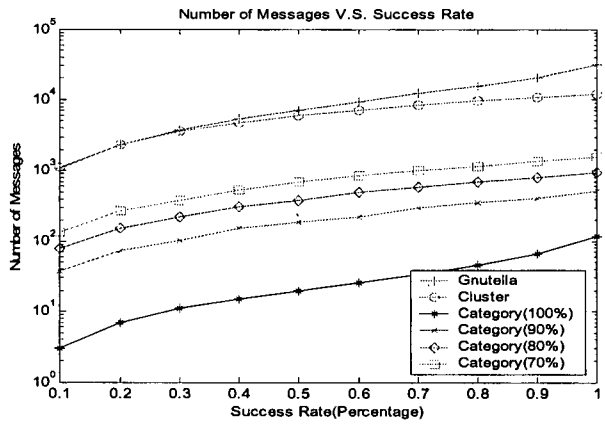

Figure 6-15 Network Overhead (Log Scaled) vs Success Rate

Our final simulation (Figures 6-16 and 6-17) helps us understand the relationship between system overhead and consistency in the COOL model. The goal is to identify a point, which associates acceptable system overhead with a relatively low consistency level (thereby low maintenance cost). This simulation is also performed 1,000 times on a 10,000-node *Internet* type network. Both maximum and mean values are plotted, with respect to various consistency levels. Figure 6-16 reflects the change of network overhead while 6-17 reflects the change of processing overhead. Under both scenarios, the TTL is set to infinity and all objects must be located. Combining the results in both figures as well as the result from section 6.2.3, we conclude that it is feasible to reduce a significant amount of system overhead (with respect to search) while keeping the maintenance cost low.

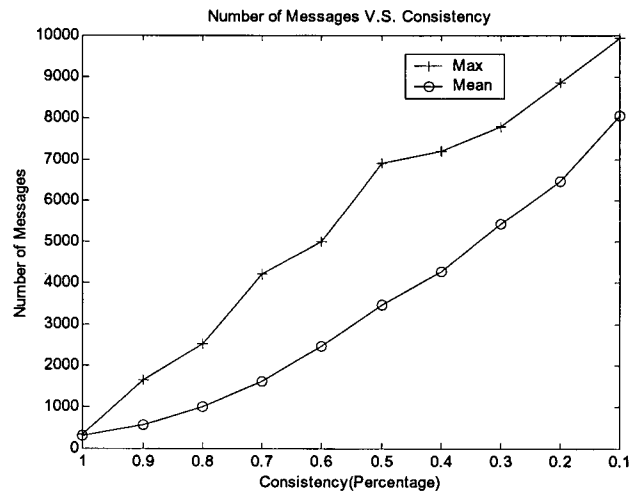

Figure 6-16 Network Overhead vs Consistency

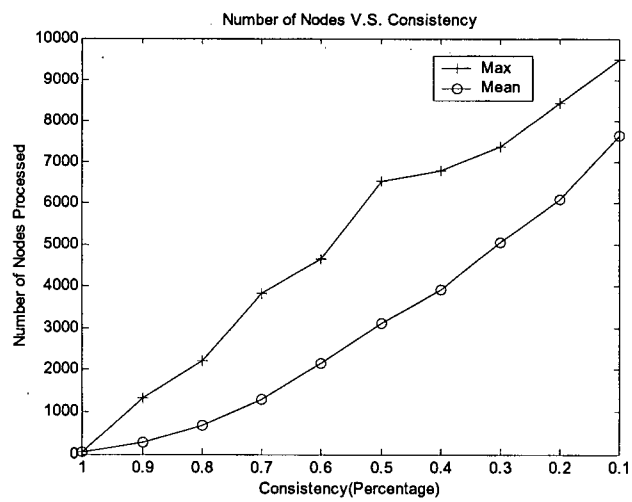

Figure 6-17 Processing Overhead vs Consistency

Chapter 7

Conclusions

7.1 Summary

This thesis investigates efficient search algorithms on *unstructured* P2P systems, with respect to scalability, robustness and maintenance cost. We propose a novel architecture which combines two components, the TBCP model and the COOL model, with the purpose of providing system-level supports to eliminate the necessity of general flooding.

The TBCP model constructs a hierarchical extension to a basic *unstructured* network, by grouping peers into a set of inter-connected clusters. Peers inside each cluster are connected to a bound-depth tree. Rich inter-cluster connectivity is sustained, as all basic peer connections are neither re-directed nor discarded. We design a restricted flooding algorithm, *cluster search*, to operate on the TBCP model. The COOL model, on the other hand, is a hybrid extension of the TBCP

model. It constructs two types of virtual networks, *cluster overlay* and *category overlay*, on top of the cluster domain, with the purpose of integrating a semantic mapping rule into a general purpose P2P application. The *category overlay* can be seen as a *Super-Peer* overlay, while the cluster overlay connects and integrates multiple *Super-Peer* overlays. This design matches the same level of search efficiency as the *Super-Peer* approach, and meanwhile, it prevents the single point of failure as well as avoids a massive number of *free-riders*. Moreover, workload is more balanced among participating peers. We design a category (or semantic) based search algorithm: *overlay search*. *Overlay* search decreases the system overhead by restricting the search traffic within a small set of well-balanced peers. Our analysis proves that the costs for maintaining these two models are small compared to the huge benefits gained from reducing search overhead.

A prototype system with COOL architecture is implemented in Java. We design a general-purpose P2P application framework, RTG, to facilitate the implementation procedure. Our simulation results demonstrate that these two models and search algorithms yield significant performance and scalability benefits in a very large-scale and dynamic Internet environment.

7.2 Future work

There are four major areas that call for further investigation. First, a semantic-based inter-cluster search can be integrated into our *category overlay* search algorithm. Current design emphasizes taking advantage of the rich connectivity to ensure search coverage within each *category overlay*. But the fact that content on

the same overlay tend to have close semantics and thereby small keyword spaces reveals the possibility of applying more intelligent inter-cluster search algorithms.

Second, peer-to-peer specific information retrieval techniques, including searching and ranking, could be introduced to further improve search performance, results accuracy as well as user experience. For example, the *overlay search* could be executed in an organized and controlled manner, such that the search activity stops as soon as enough highest ranked results are located.

Third, a more general and flexible category mapping rule need to be studied further. For instance, instead of being rèstricted to a flat and fixed category domain, as applied in the current design, a dynamically configurable and adaptable scheme can be used as a replacement. One approach is to construct a dynamic category hierarchy. Semantically closed categories fall into a more general category. A search is executed in a bottom-up manner, until enough desired hits are located.

Finally, some "killing" applications need to be identified to exhibit the real beauty of our search models and algorithms. One application, which is currently under development, is a large-scale Peer-to-Peer Video-On-Demand system, with VCR-like functionalities support, such as *fast forwarding*, *forwarding jump* and *rewinding*. Our COOL model provides system-level support for movie segment distribution, organization and locating.

Bibliography

[1] Aberer, K., Hauswirth, M. *Peer-to-Peer information systems: concepts and models, state-of-the-art and future systems*. In Proceedings of ICDE, 2002.

[2] Banerjee, S., Kommareddy, C., Bhattacharjee, B. *Scalable Peer Finding on the Internet*. In Proceedings of Global Internet Symposium, Globecom, 2002.

[3] Barábasi, A., Albert, R. BA. *Emergence of Scaling in Random Networks*, Science, 286:509-512, 1999.

[4] BitTorrent Website: *http://www.bittorrent.com*

[5] Bloom, B. *Space/time Trade-offs in Hash Coding with Allowable Errors*. Communications of ACM, vol. 13, no. 7, pp. 422-426, July 1970.

[6] Chawathe, Y., Ratnasamy. S., Breslau, L. Lanham, N., Shenker, S. *Making Gnutella-like P2P Systems Scalable.* In Proceedings of ACM SIGCOMM, 2003.

[7] Faloutsos, M., Faloutsos, P., Faloutsos, C. *On Power-Law Relationships of the Internet Topology*. In Proceedings of ACM SIGCOMM, pages 251–262, 1999.

[8] FreeNet Website: *http://freenet.sourceforge.net*

[9] Ganesan, P., Sun, Q., Molina, H. *YAPPERS: A Peer-to-Peer Lookup Service Over Arbitrary Topology*. In Proceedings of IEEE INFOCOM, 2003.

[10] Google Website: *http://www.google.com*

[11] Gnutella Website: *http://www.guntella.com*

[12] Gnutella2 Website: *http://www.gnutella2.com*

[13] JXTA Technical Document. *Project JXTA: An Open, Innovative Collaboration*

[14] JXTA v.2.3.x: *Java™ Programmer's Guide*, April, 2005

[15] JXTA Website: *http://www.jxta.org*

[16] Kalogeraki, V., Gunopulos, D., Zeinalipour-Yazti, D. *A Local Search Mechanism for Peer-to-Peer Networks*, In Proceedings of the Eleventh International Conference on Information and Knowledge Management, 2002.

[17] Kaufman, L., Rousseeuw, P. J. *Finding Groups in Data: an Introduction to Cluster Analysis*, John Wiley & Sons, 1990.

[18] KaZaa Website: *http://www.kazaa.com*

[19] Li, J. Y., Loo, B. T., Hellerstein, J., Kaashoek, M., Karger, D., Morris, R. *On the Feasibility of Peer-to-Peer Web Indexing and Search*. In Proceedings of IPTPS, 2003.

[20] Liu, X., Wang, J., Vuong, Son. *A Category Overlay Infrastructure for Peer-to-Peer Content Search*. In Proceedings of 7th Workshop on Advances in Parallel and Distributed Computational Models, April, 2005

[21] Loo, B. T., Huebsch, R., Stoica, I., Hellerstein, M. *The Case for a Hybrid P2P Search Infrastructure*. In Proceedings of IPTPS '04.

[22] Lv, Q., Cao, P., Cohen, E., Li, K., Shenker, S. *Search and Replication in Unstructured Peer-to-Peer Networks*. In Proceedings of ICS, 2002.

[23] Medina, A., Lakhina, A., Matta, I., Byers, J. *Brite: Universal Topology Generation from a User's Perspective*. University of Boston Technical Report BUCS-TR-2001-003, April, 2001.

[24] Milojicic, D. S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., Xu, Z. *Peer-to-Peer computing*. Technical Report HPL-2002-57, HP Laboratories, Palo Alto.

[25] Napster Website: *http://www.napster.com*

[26] Peer-to-Peer Working Group Website: *http://www.P2Pwg.org*

[27] Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S. *A Scalable Content-Addressable Network*. In Proceedings of ACM SIGCOMM, August 2001.

[28] Rhea, S. C., Kubiatowicz, J. *Probabilistic Location and Routing*. In Proceedings of INFOCOM, 2002.

[29] Rowstron, A., Druschel, P. *Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems*. In Proceedings of the International Conference on Distributed Systems Platforms, Middleware, November 2001.

[30] Ryan, H., Joseph, M., Lanham, N., Loo, B. T., Shenker, S., Stoica, I. *Querying Internet with PIER*. In Proceedings of VLDB, 2003.

[31] Sandvine Incorporated. *The impact of file sharing on service provider networks. An industry white paper*. December 2002.

[32] Saroiu, S., Gummadi, P. K., Gribble, S. D. *A Measurement Study of Peer-to-Peer File Sharing Systems*. In Proceedings of Multimedia Computing and Networking, 2002.

[33] Saroiu, S., Gummadi, P. K., Gribble, S. D. *Measuring and analyzing the characteristics of Napster and Gnutella hosts*. Multimedia Systems Journal, vol. 8, issue 3, November, 2002.

[34] Shi, S. M., Yang, G.. W., Wang, D. X., Yu, J., Qu, S. G., Chen, M. *Making Peer-to-Peer Searching Feasible Using Multi-level Partitioning*. In Proceedings of IPTPS, 2004.

[35] Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, In Proceedings of ACM SIGCOMM, August 2001.

[36] Tang, C., Dwarkadas, S. *Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval*. In Proceedings of USENIX NSDI, 2004.

[37] Tang, C., Xu, Z., Dwarkadas, S. *Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks*. In Proceedings of ACM SIGCOMM, 2003.

[38] Tsoumakos, D., Roussopoulos, N. *A Comparison of Peer-to-Peer Search Methods*. In Proceedings of WebDB 2003.

[39] Tsoumakos, D., Roussopoulos, N. *Adaptive Probablistic Search for Peer-to-Peer Networks*. In Proceedings of 3rd IEEE International Conference on Peer-to-Peer Computing, 2003.

[40] Vuong, S., Li, J. ECSP: an efficient cluster based P2P architecture. In Proceedings of the International Conference on Internet Computing, Jun 2003.

[41] Waxman, B. M. *Routing of multipoint connections*. IEEE Journal on Selected Areas in Communications, 6(9):1671.1622, 1988.

[42] Winick, J., Jamin, S. *Inet 3.0: Internet Topology Generator*. University of Michigan Technical Report CSE-TR-456-02, 2002.

[43] Yahoo Website: *http://www.yahoo.com*

[44] Yang, B., Molina, H. *Improving Search in Peer-to-Peer Networks*. In Proceedings of ICDCS, 2002.

[45] Zaharia, M., Keshav, S. *Design, Analysis and Simulation of a System for Free-text Search in Peer-to-Peer Networks*.

[46] Zhao, B. Y., Kubiatowicz, D., Joseph, A. D. *Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing*. Technical Report, UCB/CSD-01-1141, April 2000.

[47] Zhu, Y. W. Wang, H. H. Hu, Y. M. *Integrating Semantics-Based Access Mechanism with P2P File Systems*. In Proceedings of P2P 2003.