ABSTRACT

Title of dissertation:	SEARCH, REPLICATION AND GROUPING FOR UNSTRUCTURED P2P NETWORKS
	Dimitrios Tsoumakos, Doctor of Philosophy, 2006
Dissertation directed by:	Professor Nicholas Roussopoulos Department of Computer Science

In my dissertation, I present a suite of protocols that assist in efficient content location and distribution in unstructured Peer-to-Peer overlays. The basis of these schemes is their ability to learn from past interactions, increasing their performance with time.

Peer-to-Peer (P2P) networks are gaining increasing attention from both the scientific and the large Internet user community. Popular applications utilizing this new technology offer many attractive features to a growing number of users. P2P systems have two basic functions: Content search and dissemination. Search (or lookup) protocols define how participants locate remotely maintained resources. In data dissemination, users transmit or receive content from single or multiple sites in the network.

P2P applications traditionally operate under purely decentralized and highly dynamic environments. *Unstructured* systems represent a particularly interesting class of P2P networks. Peers form an overlay in an ad-hoc manner, without any guarantees relative to lookup performance or content availability. Resources are locally maintained, while participants have limited knowledge, usually confined to their immediate neighborhood in the overlay.

My work aims at providing effective and bandwidth-efficient searching and data sharing. A suite of algorithms which provide peers in *unstructured* P2P overlays with the state necessary in order to efficiently locate, disseminate and replicate objects is presented. The *Adaptive Probabilistic Search (APS)* scheme utilizes directed walkers to forward queries on a hop-by-hop basis. Peers store success probabilities for each of their neighbors in order to efficiently route towards object holders. *AGNO* performs implicit grouping of peers according to the demand incentive and utilizes state maintained by *APS* in order to route messages from content holders towards interested peers, without requiring any subscription process. Finally, the *Adaptive Probabilistic REplication (APRE)* scheme expands on the state that *AGNO* builds in order to replicate content inside query intensive areas according to demand.

SEARCH, REPLICATION AND GROUPING FOR UNSTRUCTURED P2P NETWORKS

by

Dimitrios Tsoumakos

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Committee:
Professor Nicholas Roussopoulos, Chair/Advisor
Professor Lise Getoor
Professor Amol Deshpande
Professor Louiqa Raschid
Professor Virgil Gligor

© Copyright by
Dimitrios Tsoumakos
2006

DEDICATION

To my beloved father, Nikiforos.

ACKNOWLEDGMENTS

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will value forever.

First and foremost I would like to thank my parents. My father and mother have always stood by, supported and actively encouraged me to follow my dreams throughout life. Without their moral, psychological and material support, I would not have been able to achieve my goals. I owe them my deepest gratitude and love.

I offer my wholehearted thanks to my advisor, Professor Nick Roussopoulos for trusting my abilities and giving me the opportunity to study in one of the best graduate schools. He provided me with the freedom to follow my research interests and the support under his research program. His technical and moral advice followed me throughout the duration of my stay here. It has been a pleasure to work with and learn from such an extraordinary individual.

I would also like to thank the members of my committee for the valuable advice they provided during and after my proposal. Dr. Lise Getoor and Dr. Jim Reggia offered some very useful pointers about Reinforcement Learning and Dr. Bobby Bhattacharjee offered his expert opinion on a variety of occasions. Special thanks are due to all my course professors, with whom I still have excellent relations, that helped me enhance my understanding on many diverse notions in computer science. I also like to thank Dr. Timos Sellis, my advisor in the National Technical Institute of Athens, who both guided my decision to pursue my PhD and co-advised me on the *GrouPeer* project.

My colleagues and close friends have enriched my graduate life in many ways and deserve a special mention. Dr. Antonios Deligiannakis, who besides being a close friend and roommate of 5 years, offered valuable advice on research and technical issues. Konstantinos Bitsakos, my neighbor and colleague of years, who proved a rare friend and collaborator. The same is true for Emmanuel Hourdakis, Athanasios Chryssis, Dr. Yiannis Sismanis, Polyvios Pratikakis, Dr. Alexandros Lambrinidis, Konstantinos Spiliopoulos, Vassilios Botopoulos and all my friends here in College Park. I thank them all for their friendship and support.

Deepest thanks are due to my dear friends in Greece: Georgios Gkoumas, Georgios Andritsopoulos, and the rest of my friends who have constantly supported me in both good and bad periods of my stay here. Among them is my very good friend and colleague, Verena Kantere. Our collaboration was fruitful and utterly satisfying for me. I thank her for both for her cooperation as well as for her support all those years.

Finally, I would like to acknowledge the financial support from the U.S. Army Research Laboratory and the U.S. Army Research Office for all the projects discussed herein, as well as the continuous efforts of our our project leader, Dr. Barras. Without their support, all this would have been near-impossible.

Special thanks are due to the administrative staff of CS and ISR who have accommodated all my needs effortlessly. The same is true for the technical support offered by the CS and UMIACS staffs.

It is impossible to remember all, and I apologize to those I've inadvertently left out. Lastly, thank you all and thank God!

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 The Notion of Peer-to-Peer and its Internet Origins

Peer-to-Peer computing (hence P2P) represents the notion of sharing resources available at the edges of the Internet [1]. The P2P paradigm dictates a fully-distributed, cooperative network design, where nodes collectively form a system without any supervision. Most importantly, they operate in a symmetric manner, running the same protocols and communicating freely and equally with each other.

The Internet started out as a system operating under the aforementioned basic P2P properties [2]. The original ARPANET was a network among equal interconnected computers. Every site could contact and accept connections from *every* other site in this network. Examples of such applications are *USENET* [3] and *DNS* [4].

USENET is a distributed worldwide system that allows users to post, read and exchange messages by directly connecting with each other. An important characteristic of USENET is the lack of any requirement for a central administration or controlling host to manage the network. DNS combines principles of P2P with a hierarchical organization in order to achieve efficient file-sharing. Instead of replicating and managing a single *hosts.txt* file, DNS allows the delegation of responsibility through the use of name servers. These hosts operate as both clients and servers, making, answering and forwarding DNS requests.

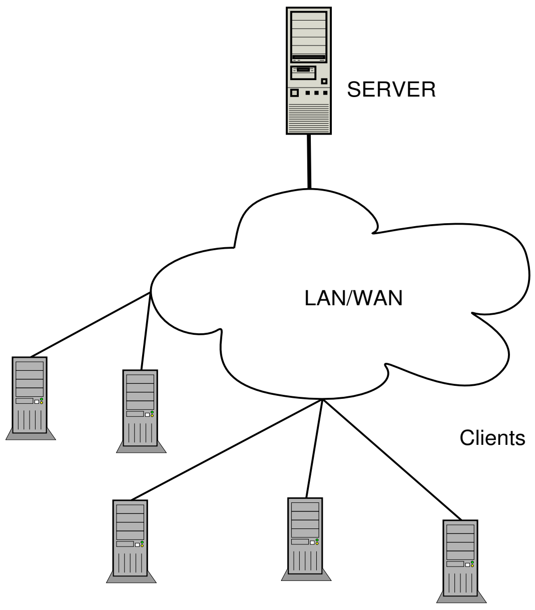


Figure 1.1: Client-Server architecture

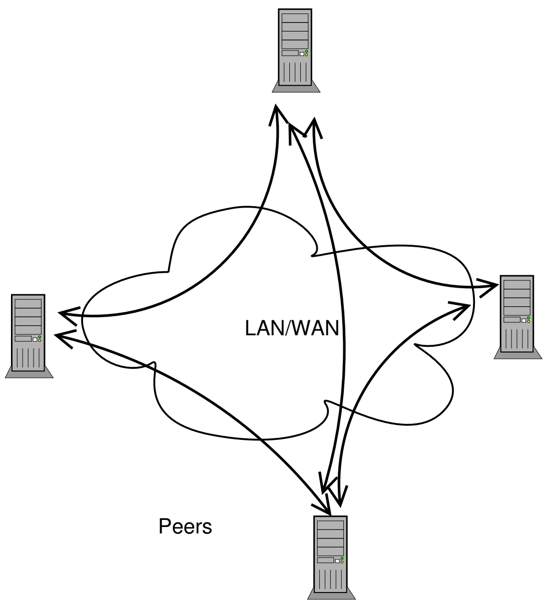


Figure 1.2: Peer-to-Peer architecture

With the explosion of the Internet, its nature gradually shifted from being symmetric and cooperative towards an asymmetric and disjoint environment. Web browsing, the dominant application during Internet's surge in popularity, is based on the client/server architecture: Client machines send requests to a small number of powerful, well-known sites running special software, retrieve answers and display them locally (see Figure 1.1). The immense popularity also brought the need for security and control. Large sub-networks got behind firewalls, denying the majority of their hosts direct access to or from the outside world. The same effect is produced by the numerous NAT boxes which are used to provide a single point of contact between the Internet and the numerous local networks. Finally, the Internet Service Providers, realizing the dominant trend is to request and not to disseminate data, engineered and provided asymmetric bandwidth services which inherently changed the ability to equally share content.

With the emergence of file-sharing P2P applications (especially [5,6]), users started massively sharing multimedia resources freely and equally, without any central control (see Figure 1.2). A large number of systems and architectures that utilize this technology have emerged since ([7,8], etc.). Its advantages (although application-dependent in many cases) include robustness in failures, extensive resource-sharing, self-organization, load balancing, data persistence, anonymity, etc.

According to very conservative estimates [1], there exist more than 10×10^9 MHz of CPU power and 10,000 TB of storage not utilized at the edges of the Internet. According to a different report [9], bandwidth consumption attributed to popular file-sharing applications amounts to a considerable fraction (up to 60%) of the total Internet traffic. These two reports identify two different challenges: First, there is a vast amount of “untapped” potential over the Internet. On the other hand, current resource-sharing applications are responsible for huge amounts of data transmissions over the network. P2P technology can play a key role in our efforts to tackle both issues.

1.2 Categorization of Peer-to-Peer Overlays

An *overlay* network is a computer network built on top of one or more existing ones (often the Internet itself). Its connectivity usually differs from the underlying physical connectivity. Nodes can be thought of as being connected by logical links, each of which corresponds to a path of one or more physical links (see Figure 1.6). Thanks to NAT, firewalls and private IP address, the implementation of applications for end-to-end communication over the IP network eventually requires some sort of overlay structure. Several P2P overlays have been proposed by both academia and industry in the last few

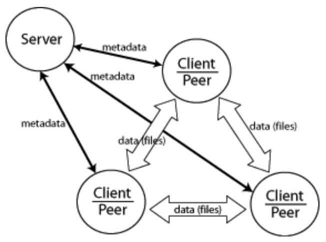


Figure 1.3: Centralized P2P network

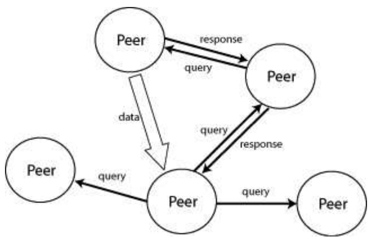


Figure 1.4: Pure P2P network

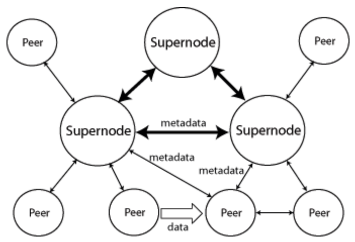


Figure 1.5: Hybrid P2P architecture

years. Their primary functionality is to provide a routing substrate between nodes identified by a mechanism other than their IP addresses.

We can roughly classify P2P architectures into two categories: *Centralized* approaches utilize a central directory for object location, ID assignment, etc (see Figure 1.3). *Decentralized* approaches abandon this solution to employ a distributed directory structure. *Pure* decentralized systems exhibit a fully distributed behavior with all peers equally making, answering and forwarding requests (Figure 1.4). In *hybrid* systems, nodes are categorized as *leaf-nodes* or *super-peers* (also referred to as *supernodes* or *ultrapeers*, see Figure 1.5). *Super-peers* are responsible for returning results to the queries posed by their neighboring leaf-nodes. They usually achieve that by indexing the repositories of all their leaf-nodes and communicating with a number of different super-peers.

Another taxonomy classifies P2P networks into *structured* and *unstructured*, according to the degree of control over the topology and routing infrastructure they provide. *Structured* networks provide strict rules for file placement and object discovery, while *unstructured* approaches offer arbitrary network topology, file placement and search.

Several researchers have proposed the *Distributed Hash Tables (DHTs)* as a means of organizing a P2P overlay (e.g., [10–13]). In these systems, files and node-IDs are asso-

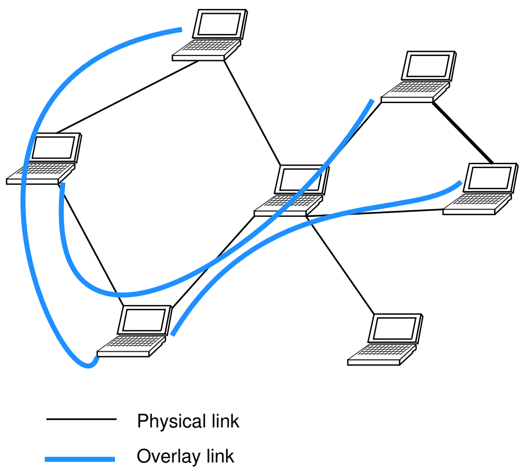


Figure 1.6: Schematic description of an overlay network

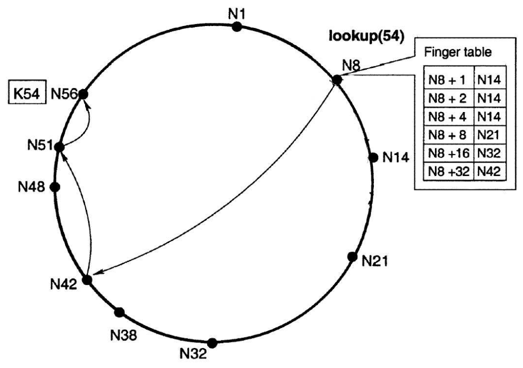


Figure 1.7: An identifier circle (ring) with 10 nodes in a DHT. Node 8 issues a *lookup(54)* command and this is routed to the host node (56)

ciated with a key produced by hashing filenames or addresses. Each node is responsible for a range of keys in this namespace: Object locations are stored at the node(s) whose ID(s) is(are) numerically closest to the given key. The basic operation in these DHT systems is to implement `lookup(key)`, which returns the identity of the node storing the object with that key. When a `lookup(key)` is issued, the message is routed through the overlay network, each time bringing the request to nodes numerically closer to the `key`, until the node responsible for it is reached (see Figure 1.7). DHTs provide a very efficient ($O(logn)$, with n equal to the size of the network) routing mechanism. This comes at a cost of maintaining state about a number of overlay nodes that assist in routing.

Today, some of the popular P2P applications operate on *unstructured* networks. In contrast to DHTs, peers connect in an ad-hoc fashion, the location of the documents is not controlled by the system and no guarantees for the success or the complexity of a search are offered to the users. More important, peers obtain only local knowledge of a network where nodes enter and leave frequently. For such systems, searching for an object is traditionally implemented by either broadcast-based schemes [6], or randomized walks [14]. Queries utilizing exact-match object-IDs or keywords are propagated inside the overlay on a hop-by-hop basis. Each time a peer receives a request, it evaluates it against its local repository and (if necessary) forwards it to a number of its overlay *neighbors* (i.e., nodes directly connected to it).

1.3 A General Model for an Unstructured P2P Overlay

In many realistic scenarios, the topology cannot be controlled and thus DHTs cannot be used (e.g., ad-hoc networks or current large-scale unstructured overlays). In our work, we focus on *pure decentralized unstructured* P2P systems. Such systems have been shown to attract large user populations and be of great impact to the network community [9].

We now describe our system model for search and content-sharing in unstructured P2P networks. We assume a pure P2P model, with no imposed hierarchy over the set of participating peers. All of them may equally serve and make requests for various objects. Peers and documents are assumed to have unique identifiers, with object IDs used to specify the query target. Ignoring physical connectivity and topology from our talk, we assume that peers are aware of their one-hop neighbors in the overlay. Neighboring nodes are connected with direct logical links and can contact each other with one overlay message. Throughout its lifetime, a node periodically checks the availability and status of its neighbors. The system can generally exhibit a dynamic behavior, with peers entering and leaving at will and also updating their local repositories. We should also note that we do not expect the overlay structure to be static, since nodes are not guaranteed to connect to the same neighbors each time they return from an off-line state.

Peers overcome some of these deficiencies by keeping *soft state*, i.e., auxiliary information stored at a node, erased after a short amount of time and the loss of which will not keep the node from functioning. For example, peers temporarily store the unique ID of each query they process, enabling them to make the distinction between new queries and duplicate ones.

Each peer retains a local collection of documents (or objects), while it makes requests for those it wishes to obtain. The documents are stored at various nodes across the network, without the system dictating a relationship between content and its location (unlike DHTs). Objects are assumed to be of varying popularity, which affects the respective number of replicas and received requests. Objects are distributed over the network according to the *replication distribution*, which dictates the number and identity of objects stored at each node. Each peer makes requests according to a *query distribution*, which controls how many requests are made for each object (e.g., popular objects get many more requests than unpopular ones). A search is *successful* if it discovers at least one replica of the requested object. The ratio of successful to total searches made is called the *success rate* (or *accuracy*). A search can result to multiple discoveries (or *hits*), which are replicas of the same object stored at distinct nodes. A global *time-to-live (TTL)* parameter represents the maximum hop-distance a query can travel before it gets discarded.

Figure 1.8 shows an sample overlay to demonstrate the concepts of this model. Our system consists of 10 nodes with IDs A–L. Node E holds objects *i* and *k*. Node A initiates a search for object *k*, indicated by the arrows. Searches are propagated among peers

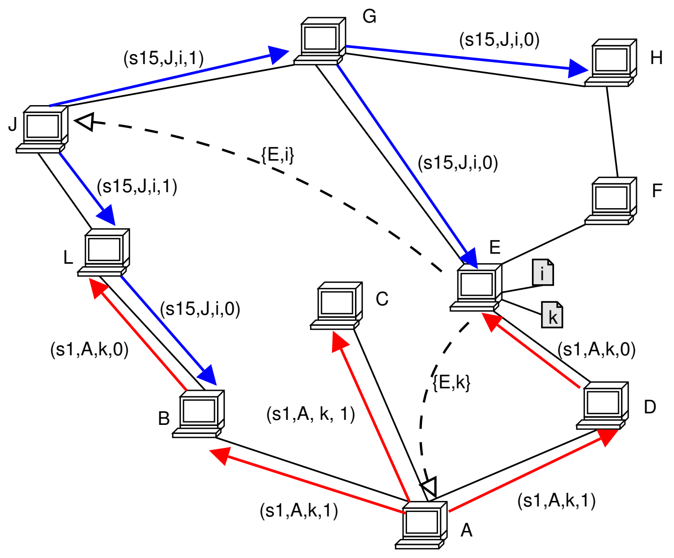


Figure 1.8: Pictorial description of our framework. Links represent logical connections in the overlay. Two searches take place, one from node A (for object *k*) and the other from node J for object *i*

usually in a hop-by-hop fashion. Search messages have their *TTL* value set to 2, so that no node more than 2 hops away from A can receive it. Each search message contains a unique identifier (e.g., *s*1), the initiator's ID, the requested object and the remaining hops, reduced by one at each node. Peer E replies directly to A, notifying it that it obtains the desired object (dotted arrow). A similar search, identified as *s*15, takes place from J. Node E now notifies J that it can share object *i*.

1.4 Our Contribution

In our previous discussion, we argued about the potential and impact of the P2P paradigm in the modern Internet. A variety of sources attest to the importance that P2P has received over the last few years. On one hand, we have economic incentives that follow the success and popularity of sharing content available at the edges of the Internet. On the other hand, we cannot but notice the profound economic, social and practical impact that such applications have: Their operation often challenges pre-defined notions of copyright, trust, accountability and security. We have witnessed a barrage of legal disputes, which represent the materialization of the clash between powerful economic interests and the users' desire to freely share. While such issues evolve and hold great significance, we cannot overlook some equally important practical issues. The unsupervised, ad-hoc character of P2P systems puts inherent strains on their ability to operate both efficiently and at low-cost. Valuable resources such as bandwidth, processing power, and connection time must be carefully distributed according to demand and the system's resources.

In this dissertation, we exclusively deal with the practical/technical challenges in P2P networking. Our goal is to provide functional, adaptive and bandwidth-efficient algorithms for unstructured Peer-to-Peer networks. Our main contribution is to describe an efficient search algorithm in order to locate content. Its most notable characteristic is the utilization of a learning feature that enables an increase in accuracy as more requests are generated in the network. Extending this, we also present our protocol for content dissemination to groups of peers in a cost-effective manner. We conclude by presenting a scheme which, building on our previous algorithms, achieves adaptive replication in order to perform efficient content sharing in high-demand scenarios. Hence, our contribution can be divided in 3 major parts:

1. Adaptive Probabilistic Search for Unstructured P2P Networks:

We propose a new search algorithm that achieves efficient lookups with low bandwidth consumption, the *Adaptive Probabilistic Search* method (*APS*) [15]. In *APS*, a node deploys *k* walkers for object discovery, but the forwarding process is probabilistic instead of random. Peers effectively direct walkers using feedback from previous searches, while keeping information only about their neighbors. As we show in this work, *APS* exhibits many plausible characteristics, such as:

- High accuracy
- Low bandwidth consumption
- Robust and adaptive behavior in rapidly-changing environments

These features come as a result of our algorithm's *learning* character, which enables peers to share, refine and adjust their search knowledge with time. Furthermore, *APS* induces zero overhead over the network at join/leave/update operations. We present a formulation of our method by defining it as a *Reinforcement Learning* problem. This formulation explains many of our empirical observations.

Concurrently with this work, Appendix A presents a detailed comparison of contemporary search algorithms for unstructured overlays. Our work in [16, 17] describes a detailed categorization, description and performance evaluation of current approaches. Our focus lies on the behavior of these algorithms for each of the following metrics:

- Efficiency in object discovery (*accuracy* and number of *hits*)
- Bandwidth consumption
- Adaptation to changing topologies and workloads

To evaluate our analysis, we simulate eight of those methods (alongside *APS*) and present a direct quantitative comparison of their performance. We identify the relative advantages and disadvantages of each method as well as the conditions under which they can be most or least effective. We believe this is an important contribution that can provide a better understanding of the various search mechanisms and assist in choosing an algorithm that best fits a particular application.

2. Adaptive Group Notification for Unstructured P2P Networks:

In this part, we propose a novel approach to content dissemination based on the demand incentive. The goal of our *Adaptive Group Notification* (*AGNO*) [18] protocol is to enable peers to disseminate important updates/notifications that relate to shared objects in the overlay. *AGNO* combines the utilization of state accumulated during the *APS* search process together with a set number of probabilistically stored requester addresses to contact groups of nodes defined implicitly through lookups. Our method builds its knowledge by only monitoring the independently conducted searches and avoids the cost of explicit multicast group formation.

3. Adaptive Probabilistic Replication for Unstructured P2P Networks:

This part of our work describes *APRE* (*Adaptive Probabilistic REplication*) [19]. It represents the third member of our suite of algorithms that build on probabilistic soft state in order to provide higher-order functionality. Our goal is to design and implement a replication protocol that will provide efficient sharing of objects (servers operating under low load), scalability and bandwidth-efficiency. *APRE* is a distributed scheme that automatically adjusts the replication ratio of every shared item according to the current demand for it. By utilizing inexpensive routing indices during searches, loaded servers are able to identify "hot" areas inside the unstructured overlay with a customizable push phase. Chosen nodes receive copies thus sharing part of the load. Under-utilized servers become freed and can host other content. The rationale behind *APRE* is the tight coupling between replication and the lookup protocol which controls how searches get disseminated in the overlay. By combining the Adaptive Probabilistic Search (*APS*) state with *APRE*, we are able to identify in real-time “hot” or “cold” paths and avoid the need of advertising constantly created replicas. We show that this method proves very efficient in minimizing the number of overloaded peers and achieving a robust and well-balanced distribution in a variety of settings.

The remainder of this thesis is organized as follows: Chapter 2 presents our work on searching in P2P overlays, presenting the Adaptive Probabilistic Search. Chapter 3 describes the group notification scheme, which implicitly groups peers according to their demand for an object. In Chapter 4 we present the last member from this family of algorithms, a replication scheme that adaptively expands or contracts the replicas of any given object based on local demand computation. Our presentation ends with the related work (Chapter 5) and our conclusions (Chapter 6). Appendix A presents the performance evaluation of a number of search methods presented in Chapter 5. Finally, Appendix B describes *GrouPeer*, a system that adopts our goal of efficient content sharing through learning and grouping in the area of relational peer-databases.

Chapter 2

Searching in Unstructured P2P Overlays: The APS Method

2.1 Overview

Searching for information has been a fundamental tool in society's continuous effort for progress. We are witnesses to a series of breakthroughs in technology which, in turn, fundamentally alter the way humans communicate with each other. With the rise and popularity of the Internet, immense amounts of information have become available to an increasing number of people. To search and process this ocean of information has become an absolute necessity. As an example, a nationwide survey of Internet users in 2004 [20] shows how important search engines have become: Over 85% of the Internet users were reported to search daily for content, ranking this activity second only to email. Steadily, these "consumers" become producers, adding their own content, in a self-reinforcing process.

The primary goal of P2P systems is to allow large peer populations to interconnect and share content. In these systems, each peer individually decides on its availability, conformity to protocols and identity of objects to share. Due to the decentralization and heterogeneity of these environments, it is vital that efficient lookup schemes are provided to their users. The lack of a centralized directory or global knowledge forces searches to take place in a distributed manner, with peers directing queries to a greater part of the system. This, combined with the large popularity and enormous volumes of data being exchanged, necessitates bandwidth-efficient P2P searches. Finally, it is important to note that popular P2P networks display a highly dynamic behavior, with most users connecting for small periods of time and then leaving the system [21], locally managing their object repositories. Any algorithm that fails to scale along this pattern, inevitably puts excessive burden on network traffic.

A search process includes aspects such as the query-forwarding method, the set of nodes that receive query-related messages, the form of these messages, local processing, stored indices and their maintenance, etc. We associate the *performance* of an algorithm with its success rate and number of hits, while its *cost* relates to the number of messages produced, either directly during the search or indirectly during index updates, object relocations, etc.

We can categorize search schemes according to the query forwarding method into *flood-based* (utilizing the standard flooding scheme or one of its variations, e.g., [22]), non-*flood-based* (e.g., hop by hop [23], direct contact [24]) or combinations of the two (e.g., [25]).

According to the type of information used, there exist two general strategies to search for an object: *Blind* and *informed* searches. *Blind* schemes try to propagate the query to a sufficient number of nodes in order to satisfy the request. Current methods waste a lot of bandwidth to achieve high accuracy. Every search requires contacting many nodes within a distance called *time-to-live* (TTL), generating huge overhead to the network. This approach aims at finding the maximum number of results within an area of the network with the originating node being at the center and the radius being a TTL-related parameter.

Several search protocols have been proposed with an intention to reduce the overhead of the original flooding scheme. In the *Random Walks* algorithm [14], the requesting node sends out *k* query messages to an equal number of randomly chosen neighbors. Each of these queries follows its own path, having intermediate nodes forward it to a randomly chosen neighbor at each step. These queries are known as *walkers*. While this approach manages to reduce messages by more than an order of magnitude, it exhibits low accuracy due to its random nature and inability to adapt to different query loads.

Informed approaches, on the other hand, utilize stored or created information in order to locate various content in the overlay. The semantics of the used information range from simple forwarding hints to exact object locations. The placement of this information may also vary: In *centralized* approaches (e.g., [5]), a central directory known to all peers exists. Distributed approaches can also be subdivided into *pure* and *hybrid*. In *purely distributed* approaches (e.g., [15, 22, 26]), all participating peers maintain some portion of the information. In *hybrid* schemes, certain nodes assume the role of a *super-peer* and the rest become *leaf-nodes*. Each super-peer acts as a proxy for its leaf-nodes by indexing all their documents and serving their requests. *GUESS* [27] is an example from this category. Peers are ranked as *ultrapeers* or *leaf-nodes*. A search is conducted with the requester's ultrapeer iteratively contacting different (not necessarily neighboring) ultrapeers and having them ask all their leaf-nodes, until a number of objects are retrieved.

The semantics of the stored indices in informed approaches can be used for another categorization. Indices might relate to exact object locations (e.g., [25]), probability of discovery through a link (e.g., [15]), number of objects through a link (e.g., [26]), or other metrics (e.g., [28]). Informed methods use their indices in order to achieve high accuracy (by choosing “good” neighbors to forward the query to) and to reduce overhead. The shortcoming of most informed methods is the maintenance cost of the indices while peers join/leave the network or update their collections. In most cases, these events trigger *floods* of update messages, increasing network traffic.

In this thesis, we propose a new search algorithm that achieves high performance at low cost, the *Adaptive Probabilistic Search* method (*APS*). In *APS*, a node deploys *k* walkers which are probabilistically directed using index values that each peer stores regarding its neighbors. The indices are updated along paths of *every* walk according to the outcome of the search. This enables searches to become more accurate as more requests are generated in the network, while each peer stores only a small part of this knowledge. Because of the nature of these indices, *APS* induces zero overhead over the network at join/leave/update operations. As we show in this work, *APS* achieves high accuracy and maintains a low message consumption in both static and dynamically changing environments. In the remainder of this Chapter we will make the following contributions:

1. Define the *APS* algorithm for search in unstructured P2P networks. We describe the main idea, the indexing scheme, the search and update procedures and analyze its performance.

2. Present two improved versions of the algorithm which exhibit significant gains in message reduction and the number of objects discovered near the requesters respectively.

3. Formulate our problem as a Reinforcement Learning problem and show that con-

vergence of the index values to optimal ones can be achieved.

4. Perform extensive simulations and compare *APS* with the *Random Walks* and *GUESS* methods over different environments. Our algorithm achieves excellent results in the success rate, number of discovered objects, message consumption and adaptation to changing topologies.

2.2 The APS Method

2.2.1 Algorithm Description

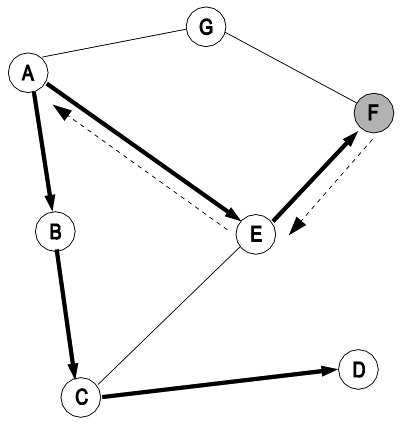
In *APS*, each node keeps a local index consisting of one entry per neighbor for each object it has requested, or forwarded a request for. The value kept for each index entry reflects the relative probability of this node's neighbor to be chosen as the next hop in a future request for the specific object. Searching is based on the simultaneous deployment of k walkers and probabilistic forwarding: A node forwards to k (if it initiates a search) or one (if it is an intermediate node) of its neighbors not randomly, but using the probabilities computed by the stored index values.

The search message is defined by the tuple: `(requesterID, objectID, searchID, TTL, v[TTL])`. The requester node includes its identity, the identity of the object in search, the unique ID of the search, its scope (`TTL`) and an initially empty array of `TTL` values. Entry `v(d)` identifies the node visited by this walker after `d` forwarding steps.

The requester chooses k out of its N neighbors (if $k \geq N$, the query is sent to all neighbors) to forward the request to. Each of these nodes evaluates the query against its local repository and if a hit occurs, the walker terminates successfully. On a miss, the query is forwarded to one of the node's neighbors. This procedure continues until all *k* walkers have terminated, either with a success or a failure. At each forwarding step, the current node appends its identifier in the search message (the corresponding entry in the `v` table) and keeps a soft state about the search it has just processed (the `requesterID`, `searchID` pair). If two walkers from the same request cross paths (i.e., a node receives a *duplicate* message due to a cycle), the second walker is assumed to have terminated with a failure and the duplicate message is discarded.

Index values stored at peers are updated in the following manner: When a node forwards the request to one or *k* of its neighbors, it pro-actively either increases the relative probability of the peer(s) it picked, assuming the walker(s) will be successful (*optimistic* approach), or it decreases the relative probability of the chosen peer(s), assuming the walker(s) will fail (*pessimistic* approach).

Upon walker termination, if the walker is successful, there is *nothing* to be done in the *optimistic* approach. If the walker fails, index values relative to the requested object along the walker's path must be corrected. Using information available inside the search message, the last node in the path sends an *"update"* message to the preceding node. This node, after receiving the update message, *decreases* its index value for the last node to reflect the failure. The update procedure continues along the reverse path towards the requester, with intermediate nodes decreasing their local index values relative to the next hops for that walker. Finally, the requester decreases its index value that relates to its neighbor for that walker. If we employ the *pessimistic* approach, this update procedure takes place after a walker succeeds, having nodes increase the index values along the



Indices	Initially	After walkers finish	After updates
A→B	30	20	20
B→C	30	20	20
C→D	30	20	20
A→E	30	20	40
E→F	30	20	40
A→G	30	30	30

Figure 2.1: Search for an object stored at node F using the pessimistic approach of APS with two walkers. The table shows how various index values change, where X→Y denotes the index value stored at node X for neighbor Y relative to the requested object.

walker's path. There is nothing to be done when a walker fails.

Figure 2.1 shows an example of how the search process works. Node A initiates a request for an object owned by node F using two walkers. Assume that all index values relative to this object are initially equal to 30 and the *pessimistic* approach is used. The paths of the two walkers are shown with thicker arrows. During the search, the index value for a chosen neighbor is reduced by 10. One walker with path (A,B,C,D) fails, while the second with path (A,E,F) finds the object. The update process is initiated for the successful walker on the reverse path (along the dotted arrows). First node E, then node A increase the value of their indices for their next hops (nodes F, E respectively) by 20 to indicate object discovery through that path. In a subsequent search for the same object, peer A will choose peer B with probability $2/9$ ($=\frac{20}{20+40+30}$), peer E with probability $4/9$ and peer G with probability $3/9$.

Our method utilizes “probabilistic” walkers with a *learning* feature that incorporates knowledge from past and present searches to enhance future performance. The learning process adaptively directs the walkers to promising parts of the network, while keeping bandwidth consumption low.

APS requires no message exchange on any dynamic operation such as node arrivals or departures and object insertions or deletions. Because the indices do not depend on content or its location but rather on the success or failure of search paths, the handling of these operations is simple: If a node detects the arrival of a new neighbor, it will associate an initial index value with that neighbor when a search will take place. If a neighbor disconnects from the network, the node removes the relative entries and stops considering it in future queries. No action is required after object updates, since indices are not related to file content. So, although *APS* actively uses information, its maintenance cost on any of these events is zero, a major advantage over most current approaches.

2.2.2 Discussion

Each node stores a relative probability (e.g., an unsigned integer value) for each of its neighbors for each (directly or indirectly) requested object. So, for $\mathcal{R}$ such objects and N neighbors, $O(\mathcal{R}N)$ space is needed. For a typical network node, this amount of space is not a burden. In nodes with limited storage capacities, index values for objects not requested for some time can be erased. This can be achieved by assigning a time-to-expire value on each newly-created or updated index, or by expunging the least recently (or frequently) used indices.

Let us calculate how many messages it will take for the *APS* method to terminate (in success or failure). In the worst case — all walkers travel TTL hops and then invoke the update procedure — the number of messages exchanged will be $2k \cdot \text{TTL}$, so the method has the same complexity with the *Random Walks* algorithm – $O(k \cdot \text{TTL})$. The only extra messages that occur in *APS* are due to the update process along the reverse path. This is where our two index update policies are used: If we expect or experience after a while that for a specific number of walkers k, only few of them terminate successfully, then the *pessimistic* mode should be employed. Conversely, if many of our walkers hit their targets on average, the *optimistic* approach should be considered.

Along the paths of all k walkers, indices are updated so that better next hop choices are made with bigger probability. Our learning feature includes both positive and negative feedback from the walkers in both update approaches. In the *pessimistic* approach, each node on the walker's path decreases the relative probability of its next hop for the requested object concurrently with the search. If the walker succeeds, the update procedure increases those index values by more than the subtracted amount (positive feedback). So, if the initial index value for a neighbor for a certain object was $\mathcal{J}$, it becomes bigger than $\mathcal{J}$ if the object is discovered through (or at) that node and smaller than $\mathcal{J}$ if the walker fails. This is the only invariant we require from our update process. In the next section, we compare several index update functions to empirically decide on their performance. The learning process in the *optimistic* approach operates in an opposite fashion, with negative feedback taking place after a walker fails. Our algorithm exhibits both *learning* and *unlearning* characteristics: *Learning* is important to achieve both high performance and discovery of newly inserted objects. *Unlearning* helps our search process adjust to object deletions and node departures, redirecting the walkers elsewhere.

Another characteristic of the algorithm is its ability to learn faster with more questions. The more feedback from the walkers, the more precise the indices become. This particularly suits the discovery of popular objects in the P2P network, which, according to studies [21], constitute over 60% of all searches. Another observation is that all nodes participating in a search will benefit from the process. This is a distinctive feature of our method, with indices being constantly updated during searches and not after object updates. In our case, *both* requesters and peers on the paths of all walkers actively adjust their knowledge about the specific object. A node that has never before requested an object but is "near" peers that have done so, inherits this knowledge by proximity. Besides standard resource-sharing in P2P systems, our algorithm achieves the distribution of *search knowledge* over a large number of peers.

2.2.3 Algorithm Improvements

APS produces update messages to adjust index values along the paths of some walkers. Our goal is to minimize these messages in order to further reduce the level of bandwidth consumption. Obviously, if fewer than $k/2$ walkers are successful, then the *pessimistic* approach should be employed instead of the *optimistic* and vice versa. Choosing one strategy over the other for queries over all objects is not optimal, as many unnecessary update messages would be produced for both popular and unpopular object requests. An improved version of *APS* is the *swapping-APS* (or *s-APS*), where the algorithm constantly monitors the ratio of successful walkers for each object and accordingly switches to the update policy that produces fewer messages. This makes our *s-APS* improvement even more bandwidth efficient. The number of objects for which nodes monitor the successful walker ratio depends on available node storage, although the overhead will be very small in most cases.

Another improvement relates to the index update procedure. The idea is to give preference to objects located near the requesters. In the original scheme, all index values are updated without any regard to the hop-distance from the requesters. In the *weighted* approach (*w-APS*), we incorporate a distance-based function for modifying the indices stored at each node. Index values for peers closer to the discovered object are increased more than those for distant nodes. Thus, the updated value of index $\mathcal{J}$ at node X would be $\mathcal{J} \leftarrow \mathcal{J} + \Gamma\gamma^{h}$, where h is X's hop-distance from the discovered object, γ is the *reinforcement* parameter, $\gamma \in (0,1)$ and Γ is a multiplicative factor (to assure a non-negligible index increase for the larger values of h). The smaller the value of γ, the more biased walkers become towards nearest-object paths. Distance information is directly accessible from the stored path inside the search messages. With this method, peers are biased to direct walkers towards closer objects in the overlay.

Both improved versions impose no extra burden to the search process, while they aim at reducing its average response time. This is achieved either by decreasing the produced messages or the distance to the objects.

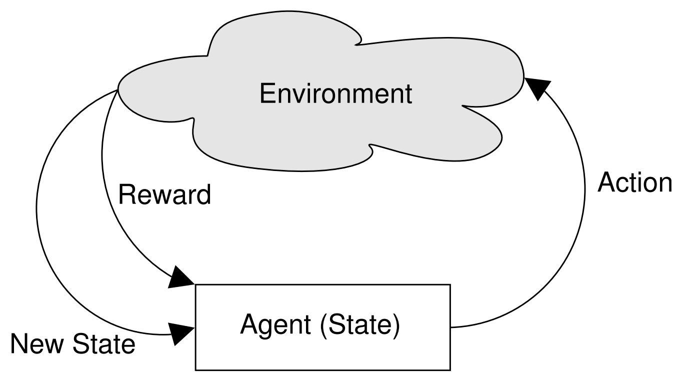


Figure 2.2: The model of interaction between agent and environment

2.2.4 APS and Reinforcement Learning

In this section we discuss the formulation of our problem as a reinforcement learning problem. First, we shortly describe the area of reinforcement learning and the general components of the problems it addresses. Later, we show that our search scheme can be similarly formulated and that a unique optimal policy can be reached.

2.2.4.1 Elements of Reinforcement Learning

Reinforcement Learning is defined as the task of learning how to behave in a certain environment [29]. Specifically, an *agent* or *learner* is expected to learn a mapping from states into actions which will eventually maximize its feedback from the environment (*reward*). The agent has to learn its behavior through trial-and-error, unlike supervised learning where the correct behavior is given through a series of training examples.

Figure 2.2 shows the model that reinforcement learning uses. At each time step t the agent is in state s_t. It can choose action $a_t \in A(s_t)$, that is one of the actions available given the state it is in. Upon acting, it finds itself in state s_{t+1}, while the environment signals a numerical value, r_{t+1} as a reward for entering that state. The goal of the agent is to select a *policy* π (choice of an action for each possible state) that will maximize the total reward it receives in the long run:

$$R_r = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$$

The parameter $\gamma \in (0,1]$ is the *discount factor* that allows the same formulation for both finite ($T < \infty$) and infinite ($T = \infty$) horizon problems. It represents a natural notion that rewards received in the future should not weigh as much as the immediate reward.

Besides the model and the reward function, the third important element in reinforcement learning methods is the notion of the *state value*. The *value* of a state s, $V^{\pi}(s)$, represents the expected total reward for the agent starting at state s and following policy π thereafter. Intuitively, state values represent a metric of goodness for the agent being in that state. Their computation is the ultimate goal in a reinforcement learning problem.

A very important property that greatly assists in the formulation and solution of reinforcement learning problems is the *Markov* property. It states that, given a finite number of states and actions, next or future states depend solely on the current state and action. Formally, the probability of transitioning from state s_i to state s_j taking action a_k is: $P\{s_{t+1} = s_j | s_t = s_i, a_k\}$, i.e., depends only on the current state and action. If a problem has this property, then it can be formulated as a *Markov Decision Process (MDP)*.

An MDP is formally defined by the tuple $(S, A, \{P_{sa}\}, \gamma, R)$, where S is the set of states, A is the set of actions, $\{P_{sa}\}$ is the set of transition probabilities, $\gamma \in [0,1)$ is the discount factor and R is a reward function: $R : S \mapsto \mathbb{R}$. As we described before, the MDP proceeds in the following manner: At each time step t, the agent finds itself in state s_t. Choosing action $a_t \in A$ drawn according to probabilities $\{P_{s_t a_t}\}$ brings the agent to the next state s_{t+1}, receiving a reward of $R(s_{t+1})$. Our total reward (or *return*) is given by the discounted sum of rewards. The goal then becomes the maximization of the return. The value function of a state, as me mentioned before, is defined as the expected return when the agent starts from that state. The value function satisfies the *Bellman equation*:

$$V^{\pi}(s) = R(s) + \gamma \sum_{\substack{s' \in S \\ a \in A(s)}} P_{sa} V^{\pi}(s')$$

Thus, the value of a state equals the immediate reward for being in that state plus the discounted values of the future states through the probability distribution.

The core of reinforcement learning methods is the estimation of these value functions for each state, since they represent a measure of how useful each state is in achieving a high return. The Bellman equations, which hold for a set policy π, define an $|S| \times |S|$ linear system that can be solved to give the respective $V(s)$ values. Among all policies, there exists at least one that maximizes the expected sum of rewards. Following that policy (π^*) produces the optimal value functions $V^*(s)$, which also satisfy Bellman's equation:

$$V^*(s) = R(S) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa} V^*(s')$$

This equation means that the optimal value function for state s is equal to the immediate reward plus the maximum (over all possible actions from that state) future sum of rewards.

For finite MDPs, there exists a unique set of $V^*(s)$ values that satisfy the Bellman equations, regardless of a policy. Given the optimal value functions, one merely has to perform greedy single-step choices that maximize the value functions: Given any start state s, the optimal policy(-ies) are those that choose actions that maximize $V^*(s)$. Thus, solving finite-state MDPs becomes equal to the task of producing efficient algorithms (e.g., dynamic programming, temporal-difference learning, Monte-Carlo methods, etc) to estimate the value functions.

2.2.4.2 Problem Formulation

In this section we present a simple formulation to our problem based on Markov Decision Processes. It shows that many parts of our scheme already fit into this framework that provably converges to optimal state values/policy.

First off, we show that the Markov property holds for our system. Indeed, each time a query arrives at a node, the path to be followed and the discovery of an object through this node does not depend on the previously visited nodes: What happens next depends on the peer that currently processes the request and the neighbor it will choose to forward it to.

In the following, *neighbors(x,y)* is a function that returns `TRUE` if nodes x,y are neighbors and `FALSE` otherwise. Similarly, $has(x,o)$ returns `TRUE` if node x has object o and `FALSE` otherwise. $\mathcal{V}$ is the set of nodes that have received query q. Finally, HL is a variable that at the initial state has a value of TTL, reduced by 1 each time q gets forwarded. Having asserted that our problem can be formulated as an MDP, we now define the tuple $(S, A, \{P_{sas'}\}, \gamma, R)$ that describes our problem:

S: The set of states. Let $S = \{S_1, S_2, ..., S_N\}$, where S_i represents a node in our overlay and $|S| = N =$ size of the network. We map the position of q at each time to a state of our MDP.

A: The set of actions. Let $A(s) = \{a_1, a_2, ..., a_N\}$, where a_i represents the action of node s choosing node i to forward the query to.

$P_{sas'}$: The transition probability matrix. The quantity p_{iaj} shows the probability of transitioning from state i to state j given action a was taken. For our setting:

$$p_{iaj} = \begin{cases} 1, & \text{if } \mathit{neighbors(i,j)} \text{ and } a \equiv a_j \text{ and } HL > 0 \\ & \text{and } j \notin \mathcal{V} \text{ and } \neg has(i,o) \\ 0, & \text{otherwise} \end{cases}$$

The transition probabilities describe that a state transition is only allowed if the object has yet to be found, there are still more hops to travel and we forward to a neighboring node, given that the query has not previously visited that node.

γ: The discount factor $0 \leq \gamma < 1$.

R: Numerical rewards associated with each state. Let r_i be the reward we receive when the query reaches state (node) i. For our system:

$$r_i = \begin{cases} 1, & \text{if } has(i,o) \\ 0, & \text{otherwise} \end{cases}$$

Given this formulation, we can use Bellman's equation on the optimal state value functions, which, given our definition of $P_{sas'}$ becomes:

$$V^*(s) = \max_{a \equiv a_{s'}} \{r(s) + \gamma V^*(s')\},$$

with neighbors$(s,s') =$`TRUE`.

To illustrate how these equations can actually help us determine, in an optimal way, the query forwarding policy at any peer, we consider the following example: Figure 2.3 shows a subgraph with eight nodes/states. Let us compute $V^*(s_1)$, assuming $HL = 6$:

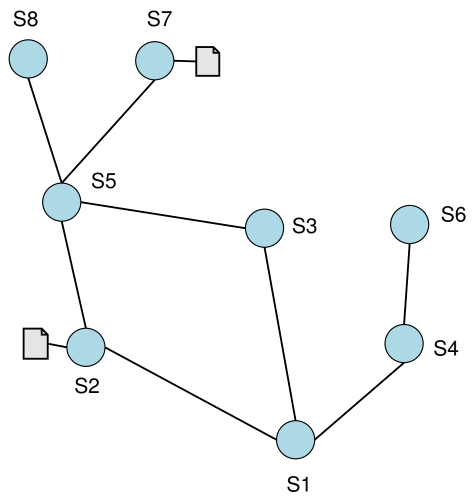


Figure 2.3: Part of an overlay for our example. Nodes 2 and 7 obtain the object in search

$$\begin{aligned}
V^*(s_1) &= r_1 + \gamma \max_{HL\leftarrow 4}\{V(s_2), V(s_3), V(s_4)\} \\
&= \gamma \max\{1, r_3 + \gamma_{HL\leftarrow 3} V(s_5), r_4 + \gamma_{HL\leftarrow 3} V(s_6)\} \\
&= \gamma \max\{1, \gamma^2 \max_{HL\leftarrow 2}\{V(s_2), V(s_7), V(s_8)\}, 0\} \\
&= \gamma \max\{1, \gamma^2 \max\{1,1,0\}, 0\} \\
&= \gamma \max\{1, \gamma^2, 0\} = \gamma
\end{aligned}$$

Similarly:

$$\begin{aligned}
V^*(s_3) &= r_3 + \gamma \max_{HL\leftarrow 4}\{V(s_1), V(s_5)\} \\
&= \gamma \max\{r_1 + \gamma \max_{HL\leftarrow 3}\{V(s_2), V(s_4)\}, r_5 + \gamma \max_{HL\leftarrow 3}\{V(s_2), V(s_7), V(s_8)\}\} \\
&= \gamma \max\{\gamma \max\{1, r_4 + \gamma \max_{HL\leftarrow 2}\{V(s_6)\}, \gamma \max\{1, 1, \gamma \max_{HL\leftarrow 2}\{V(s_8)\}\} \\
&= \gamma \max\{\gamma \max\{1, \gamma r_6\}, \gamma \max\{1, 1, \gamma r_8\}\} = \gamma^2
\end{aligned}$$

With this method, we arrive at the optimal solution:

$V^*(s_1) = \gamma, V^*(s_2) = 1, V^*(s_3) = \gamma^2, V^*(s_4) = \gamma^2, V^*(s_5) = \gamma, V^*(s_6) = \gamma^3,$

$V^*(s_7) = 1, V^*(s_8) = \gamma^2$

Given these values and assuming $\gamma < 1$, it is easy to implement the optimal forwarding policy: From any node s, if $V(s) < 1$, forward to a neighbor s' that maximizes $V(s')$. For example, node 1 should forward to node 2, node 5 can forward to either one of nodes

2 or 3, etc. The above hold for deployment of a single walker per query $(k = 1)$. For $k > 1$, we initially forward to the top$-k$ neighbors according to their values. The optimal policy behaves greedily in respect to the optimal value functions.

Moreover, given the discounted model we used, the query is directed towards the *nearest* replica in the graph. Nodes with no chance of locating an object have a value of zero. All other states maintain a value γ times the length of the shortest path to a replica. This is the index update model that our *w-APS* approach utilizes.

Value iteration is one of the methods that we can use to compute this solution. We update all values in steps: Initially, $V^{*(0)}(s_i) = r_i$. Then, compute

$$V^{*(t)}(s_i) = \max_{s'}\{r(s) + \gamma V^{*(t-1)}(s')\}$$

for all i and $t = 1, 2, \ldots$ until the values converge. Initially we have: $V^{*(0)}(s_1) = V^{*(0)}(s_3) = V^{*(0)}(s_4) = V^{*(0)}(s_5) = V^{*(0)}(s_6) = V^{*(0)}(s_8) = 0$ and $V^{*(0)}(s_2) = V^{*(0)}(s_7) = 1$.

Small changes in this formulation can alter the resulting policy. For example, since we assume each search has a limited scope, we can set $\gamma = 1$. This would make all state values equal to 1, meaning an object can *potentially* be discovered from any state[1]. Another formulation could drop the restriction that the search terminates upon object discovery. For this reformulation, the optimal value functions are: $V^*(s_1) = \gamma + \gamma^3, V^*(s_2) = V^*(s_7) = 1 + \gamma^2, V^*(s_3) = V^*(s_4) = \gamma^2 + \gamma^4, V^*(s_5) = \gamma, V^*(s_6) = \gamma^3 + \gamma^5, V^*(s_8) = \gamma^2$. We notice that the values of the states are increased. Indeed, each state's value now has one term for each object it can recover with one path of length at most HL.

[1]note that this choice may lead to non-optimal policies when deciding on state values. To avoid that, we should instead compute the optimal *action-value* functions

The previous analysis describes a theoretic formulation of our problem and the guarantee that, under certain assumptions, dynamic programming (among other) techniques can be used to show convergence to optimal state values (and thus query forwarding policy). Nevertheless, in most realistic scenarios, this computation is either very expensive or not desirable. In the area of reinforcement learning, simple greedy algorithms such as $\varepsilon-$greedy, *pursuit* and *softmax* methods are considered very effective means of solving similar problems [29]. This is also apparent in the *Ant-based* routing algorithms. They represent a family of reinforcement learning algorithms that base their operation on biological ants and their collective behavior. Such algorithms have proved extremely successful in providing shortest path routes in dynamic networks [30], yet they incorporate a variety of empirically-tunable parameters and variable convergence rates. In the chapter presenting related work we describe these approaches in more detail.

2.3 Simulation Results

To simulate the P2P overlay, we mainly used the *random* graph topology with the *pure* P2P model. We also experimented with the *hybrid* model for a comparison with *GUESS*. In the pure model, all peers equally pose and answer requests; in the hybrid model, nodes are organized in an ultrapeer-leaf hierarchy. Some experiments were run over *power-law* [31] graphs. We utilized two well-known topology generators: GT-ITM [32] for the pure and hybrid random graph models and Inet-3.0 [33] for the power-law graph model.

For the object placement and query strategies, we choose from two different distributions, namely uniform and zipf. Requesters are randomly chosen and always represent a noticeable fraction (10% or more) of the network. The default graph has 10,000 nodes with an average out-degree $d \cong 9$. The default value for k is 12 and for TTL is 5 hops. The minimum value of an index is 1, so that no nodes are precluded from the forwarding process.

To simulate a dynamic network behavior, we insert new nodes and remove online ones with varying frequencies. In the first setting (static), there are no dynamic operations. In the less dynamic setting, the topology changes more than 300 times during each run, while in the more dynamic one it changes more than 3000 times. Periodically, a portion of peers depart from the overlay and offline ones return. We always keep approximately 80% of the network nodes online. Departing nodes clear their local cache from all built knowledge.

We used 100 objects in most simulations for simplicity and speed. Objects are of varying popularity, which affects the respective number of replicas and received requests. An increase in the number of objects did not affect the quality of the results. We modeled the query and object placement strategies using a zipfian distribution to achieve results similar to the observations in [21]. The highest-ranked 10% of objects amount to over 40% of the total number of stored objects and receive about half of the requests. With our default parameters, the most popular object is stored in more than 10% of the peers, while the least popular only in 0.25% of them. Table 2.1 summarizes our simulation parameters and their default values.

In the figures that follow, the label "*APS*" is used when all variations of our method have very similar performance in a particular metric. If the results were taken under any

Table 2.1: Simulation parameters and their default values

Simulation Parameters	**Default Values**
Number of Nodes (N)	10000
Graph/P2P model	Random/Pure
Average node out-degree (d)	9
Walkers deployed (k)	12
TTL	5
Replication Distribution	Zipf ($a = 0.82$)
Query Distribution	Zipf ($a = 0.9$)
Number of Requester Nodes	1000
Number of Queries per Requester Node	3162
Reinforcement Parameter (γ)	0.3

of the two dynamic settings, this will be shown in parenthesis.

2.3.1 Comparing the Index Update Functions

Previously, we described that our index update strategy increases or decreases index values along walkers' paths in order to direct future searches. One can identify a variety of strategies in order to achieve that. Clearly, not every function can be as efficient in achieving fast learning of paths or redirect walkers after objects relocate. In our first set of simulations, we try to examine the behavior of several index update functions.

We take several 20-node connected parts from our main graphs and make queries originating from a single node each time. We consider two settings: In the first one, a single object exists 2 hops away from the initiator. After 10 requests, it gets deleted and relocated at another node (on a completely different path) 4 hops away. In the second setting, we instead place the item at a different peer 2 hops away. Note that this is more

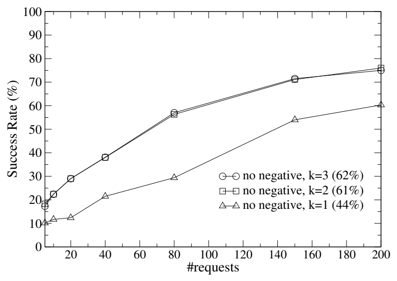


Figure 2.4: No unlearning, first setting

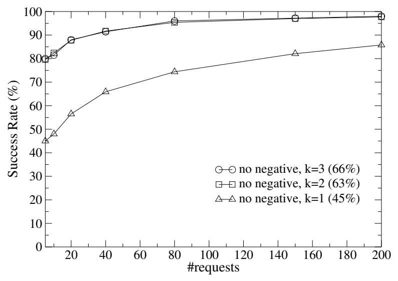


Figure 2.5: No unlearning, second setting

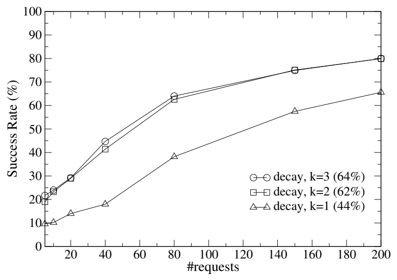


Figure 2.6: Learning with decay, first setting

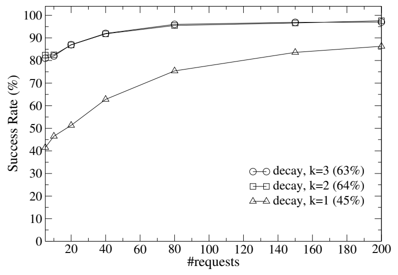


Figure 2.7: Learning with decay, second setting

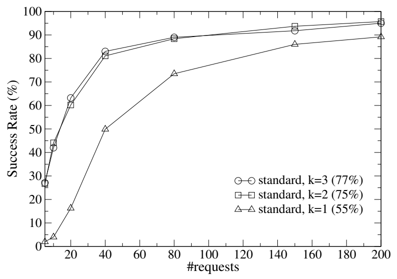


Figure 2.8: Standard update, first setting

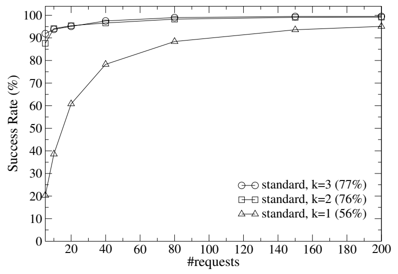


Figure 2.9: Standard update, second setting

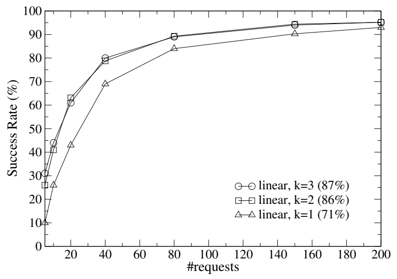


Figure 2.10: Linear update, first setting

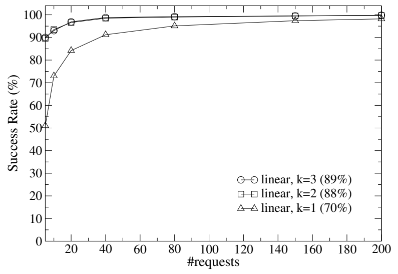


Figure 2.11: Linear update, second setting

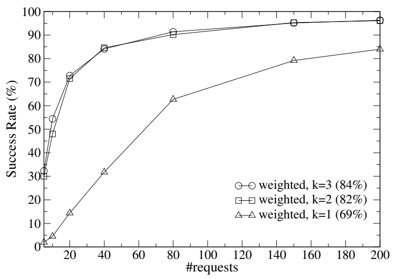


Figure 2.12: Distance-based update, first setting

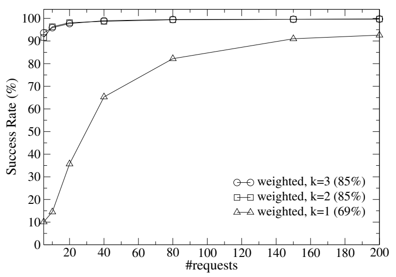


Figure 2.13: Distance-based update, second setting

challenging than simply removing the node with the initial replica. Keeping the node active forces *APS* to consider it for future requests with the already accumulated index knowledge.

We monitor the accuracy achieved by several functions after the deletion and present the results for five of them in Figures 2.4–2.13 (for different values of k). The achieved accuracy just before the deletion is shown in parenthesis. We evaluate a function with no negative reinforcement (no unlearning), one with “temporal” decay (small negative reinforcement at each time step), a flat update function (change indices by a set value each time, see Figure 2.1), a linear function (amount of change is a linear function of the current index value) and a weighted function (as described for the *w-APS* method).

We notice that all functions “learn” with more queries, although they do so with varying speed. Trying to learn the location of an object 4 hops away is harder than finding a new one 2 hops away, as we would expect. Utilizing more walkers mitigates this problem as more resources are now available for exploration of the network. The results for $k \geq 2$ do not differ significantly since in this experiment we deal with only two replicas. The linear method clearly performs better when $k = 1$ in both accuracy and fast unlearning. When $k > 1$, the standard, linear and weighted update schemes perform similarly.

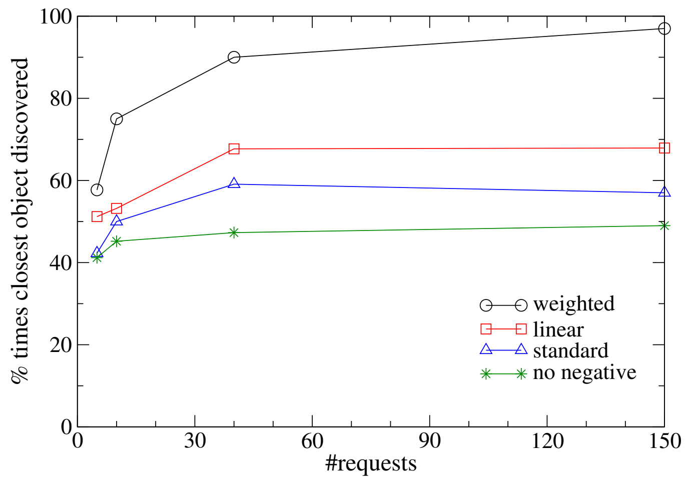


Figure 2.14: Percentage of finding the closest object for the various index update methods

Methods with negligible or no negative reinforcement show worse performance.

These results show that both learning and un-learning are necessary: The linear function increases its accuracy faster to match the initial success rate. Our observations show that unlearning is more effective if the amount of index *decrease* is proportional to its value. Similarly, the *w-APS* scheme proves more effective when the positive reinforcement is analogous to γ^h. Obviously, the rate at which nodes (and therefore paths to objects) depart affects the efficiency of the unlearning process. Also, these experiments do not take into account the interaction between different queries. This would enable the failure/success of previous queries to be considered by the current search.

In a similar experiment for the *w-APS* method, we monitor the percentage of hits for each replica, having only 2 of them at distances 2 and 4 hops away from the requester respectively ($k = 1$, Figure 2.14). The un-weighted functions find the nearest object about 45–60% of the time. A function with the amount of increase being proportional to γ^h and $\gamma = 0.3$, discovers the nearest replica with increasing frequency (i.e., over 95% of the time). This function will be used to evaluate the *w-APS* version. However, for larger k

this advantage diminishes, since the walkers quickly establish paths to objects and search for alternative ones. Indeed, for $k \geq 2$, our experiment shows that the method converges fast to an equal discovery ratio for the two objects.

2.3.2 Basic Performance Analysis

For the default graph, our simulations show that the standard flooding scheme with TTL=4 can be successful in over 99% of its searches, while producing over 9000 messages per query. These values are well-known, but mentioned here for direct comparison with the *Random Walks* and *APS* algorithms. In the following figures, if one or more of our algorithm's variations are compared, they will be specifically mentioned with their names (e.g., *w-APS*). The label "*APS*" is used to denote the *s-APS* version of the protocol with the linear index update scheme.

In our basic set of simulations, we try to validate the analysis of Section 2.2.2. We vary the number of walkers deployed (k) from 1 to 15 for the default parameters and test the two algorithms on all three settings. Figure 2.15 presents the detailed comparison on the three important metrics (accuracy, hits and messages per query).

Random Walks exhibits low success rate (below 50%) as a result of its nature. Moreover, it barely averages one hit per query in the dynamic settings, even with many walkers. Its message production is reduced during the dynamic runs (mainly when more walkers are utilized), since some of the unsuccessful paths inside the network are cut short with the departures of nodes. The performance decrease is relatively small though, as walkers are not directed according to object locations but randomly across the network.

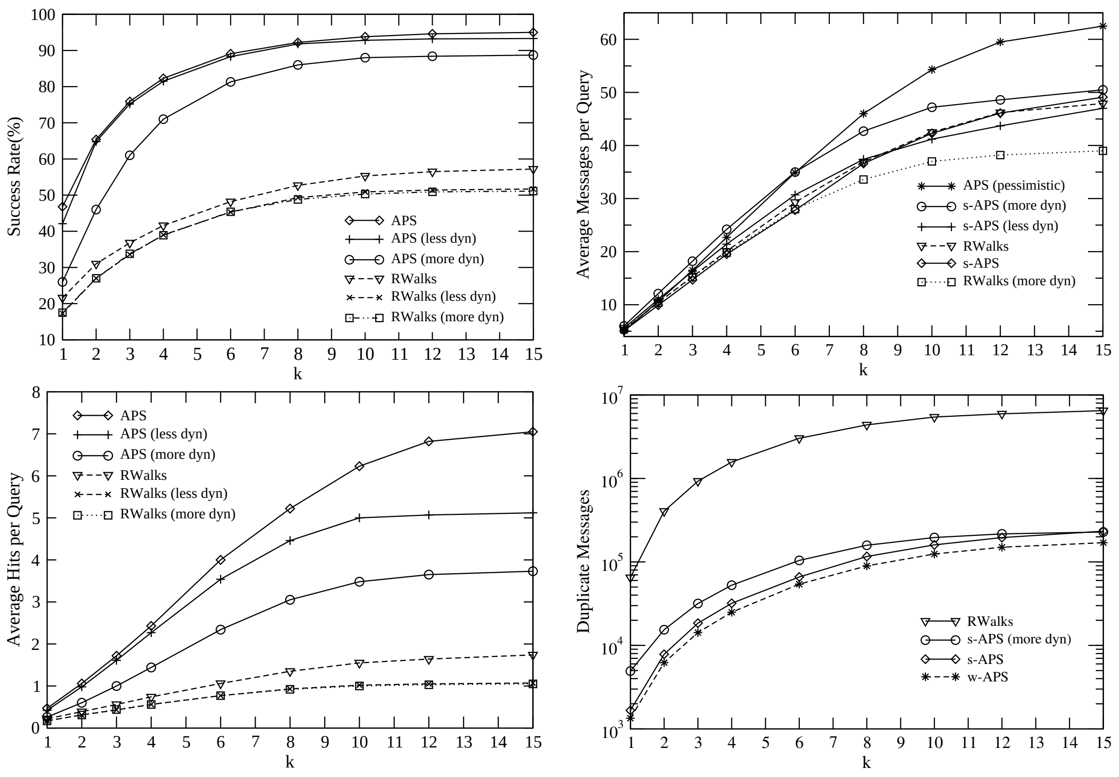


Figure 2.15: Success rate, message production, number of hits and number of duplicate messages of the two methods vs. number of deployed walkers in the three different settings

On the other hand, *APS* achieves high quality results in all these metrics. *APS* manages to maintain high levels of robustness for a variety of reasons: In the static environments, the learning process achieves fast direction of walkers towards objects. This is achieved with increasing accuracy as more queries are collected in the system. Nodes utilize indices built by all their neighbors or even other nearby peers.

In the dynamic environments, two things that affect a search may happen: First, objects may be removed and/or inserted at different locations. Second, peers may disconnect from the system, disrupting established paths. Because *APS* query forwarding is a probabilistic process, nodes with the largest values do not get necessarily chosen. Thus, no peers are excluded because of a low probability, enabling recovery from bad choices during query routing. Moreover, our algorithm performs unlearning (negative reinforcement), which enables walkers to be redirected if previously discovered objects are found missing. Finally, the probability of query failure is greatly reduced with the use of a large number of walkers. This achieves both *exploitation* of high index paths and *exploration* of less accurate neighbors in order to determine new object locations. The changes in topology or object locations must simultaneously affect all successful paths in order for a failure to occur. The metric we expect to be reasonably affected is the number of hits per search, as some paths to discovered objects frequently “disappear”.

We can see that *APS* achieves very high success rates (about 40% more accurate than *Random Walks*) even with few deployed walkers. As predicted above, the accuracy is not greatly influenced by node departures. For the less dynamic run, the amount of decrease is almost zero, while it remains within only 5% for relatively large ($k \geq 8$) values.

One would expect that our method produces a much larger number of messages compared to *Random Walks* due to the update process, but this is not the case, as the majority of walkers in *APS* are successful and only few of them reach TTL hops away. In *Random Walks*, about 70% of the walkers fail and travel TTL hops each. To a lesser extent, objects are equally discovered at all possible distances in the random method, while our scheme discovers more objects closer to the requesters. The results confirm our case: Using only the *pessimistic* approach, *APS* produces around 15 messages more per search compared to *Random Walks*. This proves that a single update policy is not suitable for all ranges of requests. The *s-APS* improvement has the same very low production as the random algorithm. This effect is enhanced if we recall that no message exchange is necessary for peer join/leave/update operations. Only in the highly dynamic setting do we

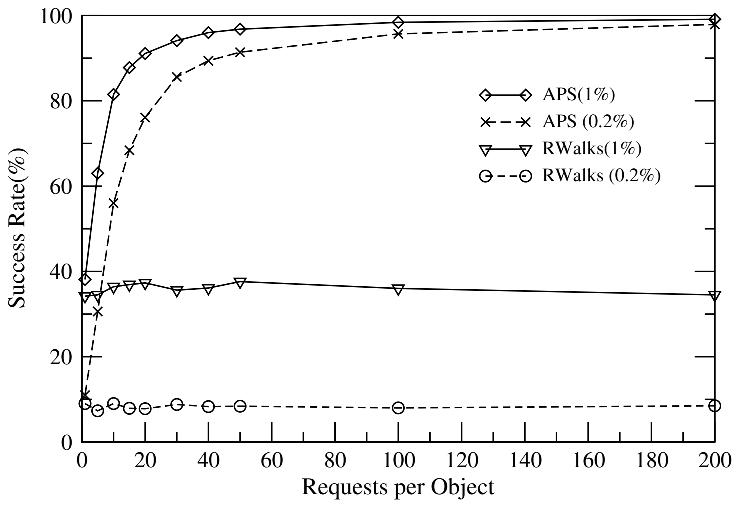


Figure 2.16: Success rate vs. number of requests per object

see an increase in the average production, which is at most 5–7 messages per search. This gap appears because of the frequent broken paths to objects, causing walkers to travel more inside the network.

Moreover, *APS* puts the walkers to a much better use, discovering around 4 times as many objects as the competing method. This is extremely important for current popular P2P applications, giving the user a much broader choice for download. This characteristic comes as a result of its high success rate and minimization of walker collisions (two walkers that cross paths forcing one of them to fail). In the dynamic settings, the maximum reduction in the number of hits is around 25% and 40% for the less dynamic and more dynamic runs respectively. These numbers occur for large values of *k*, where the probability of node departures affecting the walkers increases.

The last graph of Figure 2.15 displays the vast reduction that *APS* achieves in the number of duplicate messages. These occurrences are considered to be failure states for our walkers, therefore the learning process makes adjustments in order to minimize

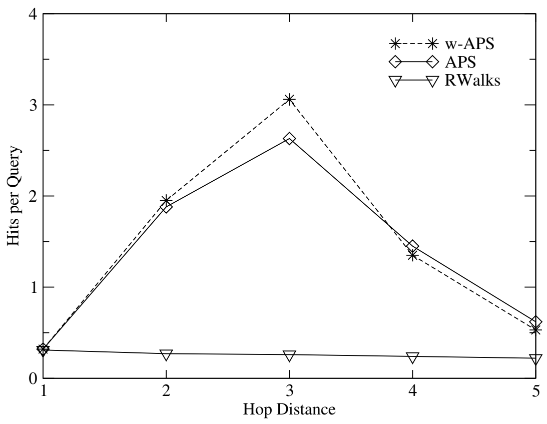


Figure 2.17: Hits per query vs. hop distance (static setting)

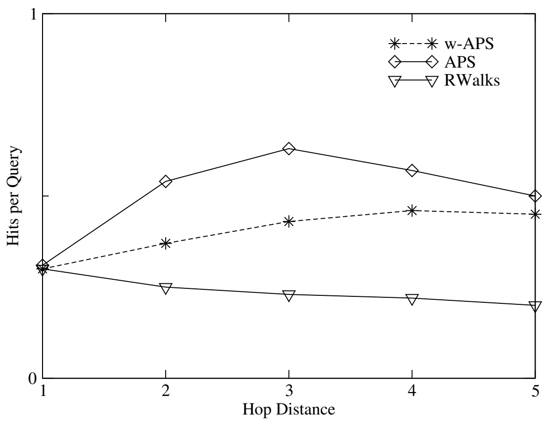


Figure 2.18: Hits per query vs. hop distance (more dynamic setting)

them. Our method constantly outperforms *Random Walks*, producing 1 to 2 orders of magnitude fewer duplicate messages. This is also important because it increases the useful processing time for each peer. The *weighted* approach exhibits almost 20% fewer duplicate messages than our default methods.

To demonstrate how *APS* increases its accuracy as more queries come into the system, we vary the number of requests per object on the default graph, using a uniform replication ratio of 0.2% and 1%. The results are presented in Figure 2.16. We can see that the accuracy of our method improves significantly with only a small increase in requests. For replication ratios greater or equal to 2%, our method exhibits almost perfect results. It is noteworthy that even for the rarest of objects, *APS* manages to build paths leading to them through learning and cooperation. At the same time, *Random Walks* is steadily below 40% and 10% respectively, regardless of the number of requests.

2.3.3 Discovered Objects vs. Distance from Requesters

Figure 2.17 shows how the hits are distributed over their distance from the requesters, for the default parameters in the static setting. While *Random Walks* discovers about the same amount of objects throughout the 1 to TTL range, *APS* makes an effort to discover closer ones. It displays a symmetric curve, finding the most objects 3 hops away from the requesters. The reason for this is its learning feature that promptly locates the closest ones (one and two hops away). The rest of the walkers are directed towards more distant content. Such objects exist in larger quantity (since nodes increase exponentially with distance) but are less easily accessible (more paths, walker collisions, etc). The results for the dynamic settings are similar, the only difference being the reduction in hits we mentioned. We have also noticed that our algorithm becomes more biased into discovering nearby objects as the number of replicas inside the network increases. This happens because the walkers have a broader selection of paths to objects and can, therefore, choose the shortest.

The *w-APS* technique marginally improves the *s-APS* performance by locating a small amount of extra content two and three hops away. While one would expect the weighted version to locate considerably more objects closer to the requesters, this is not the case: As *k* increases, paths to the nearest replicas are exploited by both methods. Furthermore, as Figure 2.18 shows, the un-weighted update method shows superior performance under dynamic environments, exhibiting both higher accuracy and hit count compared to the weighted index update method. Nevertheless, given a different setting, one could notice the difference in performance: By applying a uniform 10% replication

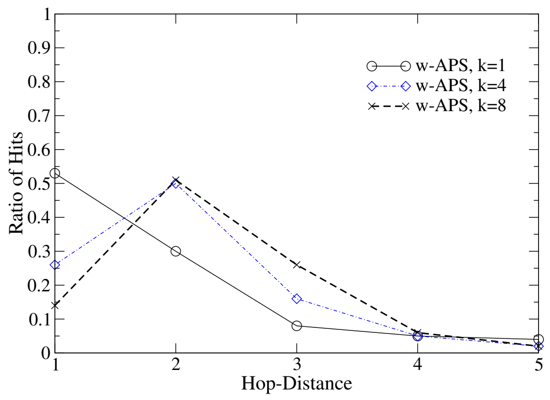


Figure 2.19: Ratio of hits per query vs. hop distance for *w-APS*

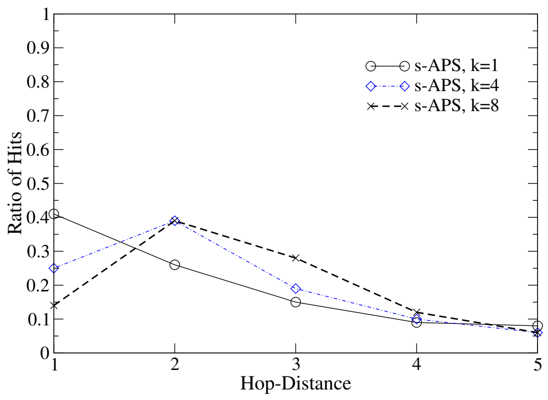


Figure 2.20: Ratio of hits per query vs. hop distance for *s-APS*

rate (to allow for more choice) and 100 requests per object, we measure the ratio of objects discovered at different distances by the weighted and un-weighted version of *APS* in Figures 2.19 and 2.20. As *k* increases, the difference between the more steep *w-APS* curve and the flatter one by *s-APS* is diminished. For larger values of *k*, the two schemes almost coincide (see Figure 2.17).

2.3.4 Effect of Object Popularity

Next, we analyze the behavior of our scheme's index values. *APS* is an inherently adaptive search algorithm, whose power lies in the use of the local indices. For the next experiment, we choose only one node from our default graph with degree 12 and examine how its local indices change. We make requests for 10 objects, with object 1 being the most popular and 10 the least. Replication and request distributions take their default values. Figure 2.21 displays the number of high-valued indices for that node for all 10 objects. Object popularity decreases from left to right on the x-axis. We monitor indices with large values (more than 20 hits) and indices that have a fairly large value (more than

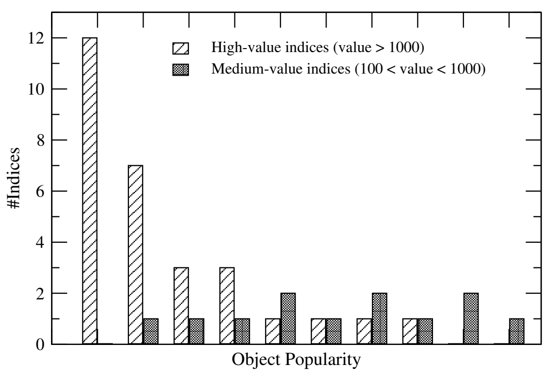


Figure 2.21: Distribution of index values according to object popularity for one peer and 10 objects

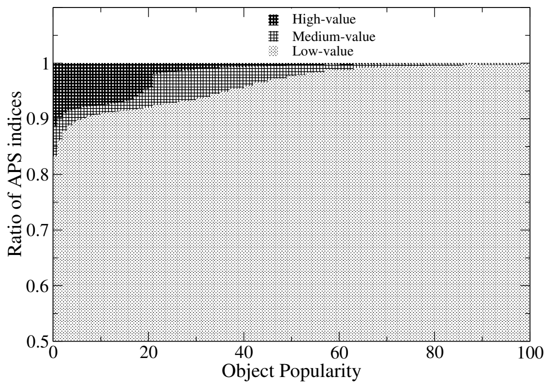


Figure 2.22: Distribution of index values versus object popularity in our default setting

5 but less than 20 hits). We notice that many indices with large values exist for the very popular objects, while this number decreases as popularity drops. Still, some indices with a relatively large value always exist for less popular objects. *APS* exhibits high precision for very popular objects, building up its “confidence” through large index values. On the other hand, the few fairly large indices for unpopular objects point out the algorithm’s ability to locate them with good probability.

Figure 2.22 presents how the entirety of index values changes for our default setting. The majority of the $O(Nd)$ indices are not used, since only 10% of the nodes are requesters. We notice that high-valued indices exist mostly for the 20% most popular items and medium-valued ones are prominent roughly between the top 20%–40% of objects. Nevertheless, some exist even for the least replicated content, giving few of *APS*’s walkers viable paths to discovery.

We conclude our analysis on object popularity and *APS* with the results of Figure 2.23. We show the success rates for individual objects grouped according to their popularity, using all default parameters (in the non-dynamic setting). Popularity decreases

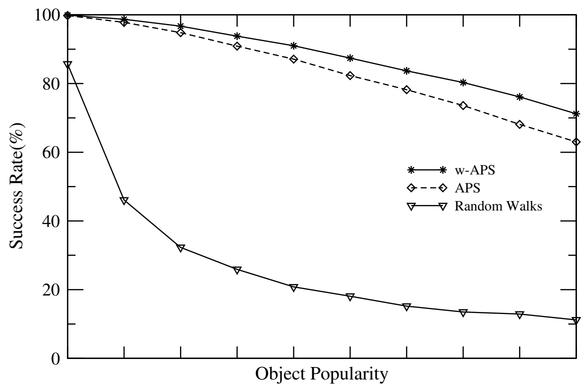


Figure 2.23: Individual success rate vs. object popularity

from left to right on the x-axis. *APS* shows almost perfect results for popular objects and displays a "graceful" decline for unpopular requests, while *w-APS* slightly improves on this for unpopular requests. On the other hand, *Random Walks*' accuracy drops significantly after requests for the highest-ranked 10% of objects, reaching a mere 11% for the least popular objects.

2.3.5 Results for Different Topologies

In this section, we compare *s-APS* with *Random Walks* over four different graphs: The default one, a 10,000-node random graph with $d = 4$ (similar to Gnutella-type graphs), a 50,000-node random graph with $d = 10$ and a 10,000-node power-law (PLAW) graph with $d = 4.4$. Table 2.2 presents the two algorithms' performance in the highly dynamic setting with the respective results from the static runs in parentheses.

First, we test the methods using a uniform distribution for both requests and storage in the default graph. The replication ratio for each object is set to 1% and each of them receives 30 queries by each requester node. We clearly notice that *s-APS* greatly benefits from such a setup, delivering over 94% in success rate (a mere 2% decrease from the

Table 2.2: Results for more environments

Graph-Distr.	*s-APS*			*Random Walks*		
	Succ%	Mesg	Hits	*Succ%*	Mesg	Hits
10K-Rand (d=10,Unif)	94.1 (96.1)	58.5 (53.5)	4.3 (7.2)	32.3 (38.2)	41.8 (49.6)	0.4 (0.5)
10K-Rand (d=4,Zipf)	70 (82.2)	17.3 (18.2)	1.4 (2.25)	26.0 (34.5)	12.0 (15.0)	0.3 (0.5)
50K-Rand (d=10,Zipf)	79.3 (87.6)	48.4 (47.0)	2.4 (5.7)	55.6 (57.6)	39.5 (45.7)	1.3 (1.4)
10K-PLAW (d=4.4,Zipf)	67.6 (76.1)	13.0 (14.9)	1.11 (1.76)	21.0 (31.6)	9.0 (12.0)	0.3 (0.5)

static run) and discovering more than 10 times more objects than *Random Walks*.

On a similar graph with smaller out-degree and $k = 5$, *s-APS* is still 40–50% more accurate, 5 times more effective in locating objects and almost as bandwidth-efficient as the random method. The results are worse compared to the default graph because of the smaller out-degree and fewer walkers used.

Our simulations on the 50,000-node random graph justify our prediction that the graph size cannot influence the performance of *APS*. The results were a little worse from the ones in the original graph, because the quality of the new graph was worse (many more disconnected components were present). We notice the success rate is about 8% lower from the static case, while the number of discovered objects is almost halved.

Our results on the 10,000-node power-law graph show an even greater gap in the performance of the two algorithms. Our method delivers about 4 times more results and exhibits a success rate three times bigger than *Random Walks*'. The success rate for *s-APS* drops by around 9% and discovered objects decrease by 37%, while message production slightly decreases.

Table 2.3: Comparison with GUESS

Metric	*s-APS*			*GUESS*		
	Succ%	Mesg	Hits	*Succ%*	Mesg	Hits
Messages	97.7	**16.3**	5.22	63.9	**16.1**	1.28
	98.6	**22.0**	7.01	65.6	**22.2**	1.87
	99.7	**33.2**	11.39	84.0	**33.1**	2.55
Hits	81.0	3.2	**1.33**	63.9	16.1	**1.28**
	94.6	8.7	**3.42**	86.4	45.0	**3.70**
	97.9	16.5	**5.42**	94.5	65.1	**5.60**

In these simulations, our method kept its message production at the same levels with the static runs, wasting at most 5 extra messages per search, a direct proof that it does not impose more burden on network traffic. As expected, the success rate shows only a small decrease, ranging from 2% to 12%. These results also show that our method maintains its relative performance gains over the different environments.

2.3.6 Comparison with *GUESS*

Lastly, we present results comparing *s-APS* with an implementation of *GUESS* [27] on a random *hybrid* graph with 6500 peers, 500 of them being super-peers (or ultrapeers in *GUESS*). Each ultrapeer is connected to 12 leaf-nodes on average. Links exist only between ultrapeers and between an ultrapeer and its leaf-nodes. In our *GUESS* implementation, initiating ultrapeers forward queries to *k* randomly chosen neighbor ultrapeers. Query and object placement distributions are set to their default values. Since it is impossible to directly compare the two methods for the same *k* and TTL values, we select simulations where the two algorithms had similar performance in one of two important metrics: Messages and hits per query. The results are presented in Table 2.3 and the comparison metric is typed in boldface. For similar message consumption, our scheme exhibits higher success rates and delivers 4 to 5 times more results. For similar hits per search, our scheme produces 4 to 5 times fewer messages and always outperforms *GUESS* in accuracy. *APS* achieves these results taking no advantage of the hybrid topology that *GUESS* utilizes.

2.4 Summary

APS deploys probabilistically directed walkers by utilizing information from past searches regarding their success or failure. This allows for fast learning with a low message consumption. Peers are required to keep indices only relative to their neighbors, while no message exchange is necessary for any dynamic network event, local or global. Our results show that *APS* exhibits effectiveness being almost as bandwidth-efficient as *Random Walks*. It discovers 4 times as many objects and delivers very high success rates compared to the *Random Walks* and *GUESS* methods, maintaining these features in dynamic environments. Appendix A contains a direct performance comparison between *APS, Random Walks, GUESS* and six more representative schemes described in the related work section.

Chapter 3

Content Dissemination to Groups of Peers: AGNO

3.1 Overview

Mass communication is defined as the process of data distribution to a greater number of people at the same time. The importance and applications of group communication schemes in computer networks and in distributed systems in particular have been well-defined in past and recent research work (e.g., [34–36]). A multicast transmission is defined as the dissemination of information to several hosts within a network. These hosts are interested in receiving the same content from an authority node (such as a web server) and naturally form a group. The lack of deployment of multicast communication in the IP layer has led to the development of various application-level multicast protocols, in which the end hosts are responsible for implementing this functionality. One-to-many communication is a very useful mechanism for a variety of network applications (e.g., [37, 38]).

As the applications that embrace the P2P paradigm grow, a number of methods have also been proposed to implement multicast communication utilizing some popular P2P overlays (e.g., [34, 35, 39, 40]). Nevertheless, these approaches take advantage of the structure that DHTs provide. As we mentioned before in our work, there exist many realistic scenarios where the topology cannot be controlled and thus DHTs cannot be used (e.g., ad-hoc networks or existing large-scale unstructured overlays). Explicit group formation schemes require frequent communication overhead between group members.

Nodes must go through a subscription process by contacting a special node and announce their intent to receive/transmit/forward group messages. These techniques often prove unsuitable because of the generated traffic for large and dynamically changing group populations.

In the area of unstructured P2P overlays, contacting large numbers of nodes is implemented by either broadcast-based schemes (e.g., Gnutella [6], Modified-BFS [22]), or *gossip*-based approaches, e.g., [36, 41, 42]. Both produce large numbers of messages by contacting many peers inside the network. Our work aims at providing peers in dynamic, unstructured environments with an effective yet inexpensive mechanism to disseminate content-related information to groups of nodes interested in their content. Specifically, we intend to provide a scheme that is:

- *Efficient:* It should be able to contact a high percentage of interested peers with low message overhead.
- *Scalable:* The scheme should be able to scale to very large group sizes (thousands of peers).
- *Robust:* We would like to avoid the necessity of a single point of contact or group leader as well as the burden of costly message exchanges in case of member arrivals and departures.
- *Adaptive:* It should adapt to changes in the group size and to dynamic workloads.

We assume a fully distributed and unstructured system, where peers share and request resources replicated inside the network. Users are interested in objects with changing content such as results of a sports meeting in real time, temperature readings, weather maps, stock quotes, security updates, etc. There exist some nodes (similar to the web servers or mirror sites in the Internet) that provide fresh content, but their connectivity or availability varies, as happens with all other network nodes. Peers that are interested in retrieving the newest version of the content conduct searches for it in order to locate a fresh or closer replica. In this environment, interest in a specific object is tied to the lookups generated for it. We argue for a push-based approach, where a server node forwards notifications (or other object-specific information) towards the interested hosts. Our assumption is that peers which have recently searched or retrieved an object would also be interested in receiving important updates about it. For example, it is safe to assume that a host frequently querying for the price of a quote or the temperature of an area would like to be informed about an update or another object-related notification.

It is important to note here that peers still search and retrieve objects in a distributed manner. The notification itself may or may not be directly related to a specific object: A severe weather alert to be effective in the next 3 hours is not related to the current area temperature. A change in the scores or quote prices, on the other hand, is directly linked to the content of the object. Group communication (especially for large groups) requires a considerable amount of bandwidth. Content providers can assess the importance of various updates/notifications and choose to push those that would be the most beneficial.

On a more technical note, the forwarding path between any two given peers in a DHT remains the same with high probability. This is a feature that many approaches utilize in order to construct efficient multicast paths. This is not the case for unstructured P2P networks: Peers have multiple (and dynamically changing) communication paths with each other. Therefore, a notification scheme for such networks can also be used to simulate that functionality and identify reverse paths from the destination (location of an object) back to the requesters. This information can in turn be used in a variety of problems (e.g., assist in dynamic replication, see Chapter 4).

In this Chapter, we present the *Adaptive Group Notification* (*AGNO*) method. Our approach combines the utilization of state accumulated during the search process together with probabilistically stored shortcuts. The first indicates the amount of demand for a specific object and can be used to infer membership and guide the dissemination of updates on a hop-by-hop basis. By also allowing peers to locally store a constant amount of requester addresses (called *backpointers*), we show that *AGNO* achieves a robust, scalable behavior in a variety of environments and group sizes. Our method utilizes a simple *binning* scheme as well as adaptive index *aging* to adjust its performance to different workloads and member joins/leaves. *AGNO* does not require any global knowledge, existence of a special contact node or any membership message exchange. It builds its knowledge by exclusively monitoring the independently conducted lookups. Finally, its performance can be easily tuned to fit specific application requirements.

3.2 AGNO Protocol Description

A multicast transmission (also referred to as the *notification* or *push phase* hereafter) in our setting is initiated by a content-holding peer (or *server*) and its target is to contact as many “group” members (i.e., requester nodes) as possible with the least amount of overlay messages. The focus of this work is to describe an efficient mechanism for such transmissions and not to define their content. The message relayed during the push phase will be referred to as a *notification* or *push message* and always relates to a specific object that is shared in the network.

The rationale behind *AGNO* relates to the observation that efficient group communication comes at a cost. In current approaches, this cost is paid by either a membership management protocol or an overlay infrastructure. Our goal is to provide with the missing state that can allow for content dissemination to a group of peers, but in a way consistent with the nature of an unstructured P2P system. In *AGNO*, the equivalent of group membership is the demand for an object (or a collection of them), realized through searches and object sharing that are *independently* conducted by peers. The granularity can be as coarse or fine-grained as the application requires. For the remainder of our discussion, we assume a per-object level of granularity.

After each search using the *APS* algorithm, peers accumulate knowledge about the relative success of a search through each of their neighbors. Intuitively, overlay paths that comprise of high index values are the ones most frequently used to connect requesters and object holders. In *AGNO*, nodes utilize this information in order to forward group messages towards possible group members during the push phase. Note here that, although we utilize the *APS* method as a means to provide this state, our approach can be used with a variety of search mechanisms, as long as they support a similar demand incentive.

We now describe the nature of the index values that are stored at each peer. *APS* keeps a local view (an index value) for each neighbor. For *AGNO*, each peer P needs to maintain the index values that P's neighbors hold relative to P. Let $X \xrightarrow{i} Y$ denote the APS index value stored at node X for neighbor Y and object i. Then, peer P must know

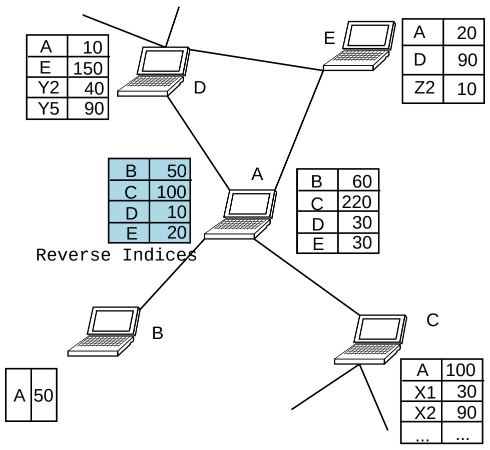


Figure 3.1: Graphic explanation of AGNO reverse indices. The filled table represents the reverse index values stored at node A, which coincide with the APS index values that nodes B,C,D,E store regarding A

$X \xrightarrow{i} P$, for each neighbor X (see Figure 3.1). These values can be made known to P either implicitly or explicitly: In the first case, peer P can infer the index $X \xrightarrow{i} P$ if it knows about the update process used (optimistic or pessimistic) and its initial value. In the explicit approach, whenever a search for object i is conducted and X forwards to P, it piggybacks $X \xrightarrow{i} P$. We call these new stored values the *reverse indices*, to distinguish them from the indices used by *APS* in searches. For the rest of our discussion, we assume that the explicit approach is used.

Reverse indices are not the only state that our method utilizes. During the search, intermediate nodes decide with probability p_r whether or not to cache the requester's address. Thus, for a search path h hops long, it will be stored on hp_r peers on average. With this scheme, we create a number of soft-state shortcuts called *backpointers* along the search paths which point to group members. Each peer can individually decide on the maximum number of backpointers stored. For simplicity, we assume that all nodes can store a maximum of c backpointer values. Backpointers are soft-state that gets invalidated

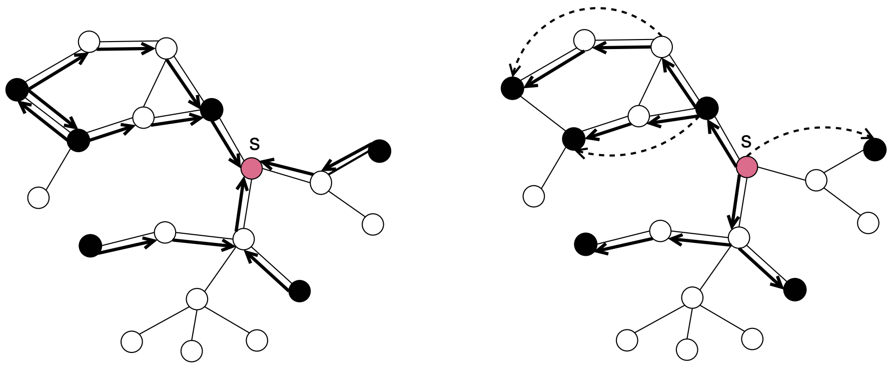


Figure 3.2: The black nodes search for an object stored at node *s* (left). On the right, *s* initiates a push phase in order to contact the requesters

after some amount of time.

Notifications are issued by peers that (authoritatively) serve objects. They are of the form `(nodeID, nodeIP, notificationID, objectID, TTL, content)`, where `(nodeID, nodeIP)` is the server's identifier and IP address, `notificationID` is a unique identifier for each push message generated by `nodeID` (to eliminate duplicate receptions), `TTL` is the maximum distance allowed for the message to travel and `content` holds the actual content of the notification that refers to object `objectID`.

During the push phase, peers issuing or receiving a notification forward it to their neighbors using the reverse index values. We consider the following forwarding schemes:

- Probabilistically forward to $k \geq 1$ neighbors using the reverse indices or forward to those with the top-k values
- Forward to all neighbors with reverse index value larger than a defined threshold

Moreover, a peer sends the push message directly to each of its valid backpointers with probability p_n. These messages have a TTL=1 and do not travel further.

Whenever an overlay link is crossed, the `TTL` field is decremented. A push message is discarded either when its TTL value reaches zero or when it is received more than once due to a cycle. Therefore, our scheme combines a selective, BFS-like forwarding augmented with shortcuts in order to contact the group members. This is shown pictorially in Figure 3.2.

We now discuss how the aforementioned state is maintained at each peer. The backpointer values expire after a certain amount of time. Since our incentive to push a message is the demand on a per-object basis, new backpointers replace the oldest valid ones (if a node has already c valid backpointers). As searches take place inside the system, the backpointer repositories get updated, while the probabilistic fashion in which they are stored guarantees a diverse collection of (ID, address) pairs. Reverse indices get updated during searches, but this is not enough: There may be peers that have searched for an object and built large index values, but are no longer interested in receiving notifications (i.e., stopped querying for that object). If searches are no longer routed through those peers, the reverse index values (which reflect *APS* indices) will not be updated and will remain high.

To correct this situation, we add an *aging* factor ξ to the reverse indices, which forces their values to decrease with time. Peers need to keep track of the time that a reverse index was last updated in order to acquire its correct value before using it. When a peer receives a search message, it sets the corresponding reverse index to the piggybacked value and its last modified field to the time of receipt. Figure 3.3 shows how this process works. The value of the index decreases exponentially, while two searches at times t_1, t_2 reset its value. A push message received at time t_3 will use the value as shown in the figure. The last modified value is also reset when a reverse index is used, since a peer computes

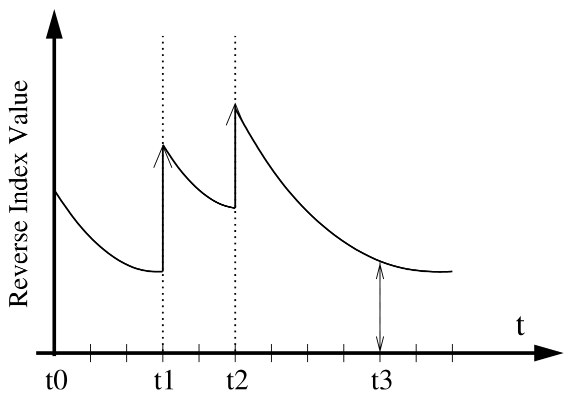


Figure 3.3: Example of computation of a reverse index value

its current value before using it. Obviously, a fixed value for ξ will perform suboptimal aging, by either reducing the reverse indices too much or by failing to reduce them enough for the push phase to prune out disinterested peers. The next section describes in more detail how our protocol proceeds in the computation of the parameters described above.

3.2.1 Protocol Specifics

1) Space Requirements: The amount of space required by the peers is $O(2d+2c)$ per object, where d is the average node degree in the overlay and c is the maximum number of backpointers stored. Each peer stores one reverse index value and its modification time and a backpointer with its creation time per object. Even if nodes want to keep track of large numbers of objects, the space requirements are in the order of a few tens of megabytes, definitely affordable by the vast majority of modern hosts (typical 1GB of main memory configurations). For about 1 million objects, assuming $c = d = 4$, each peer would need approximately 64MB of memory for *AGNO*.

2) Forwarding: Nodes use a threshold parameter *Thresh* in order to choose the neighbors to which a notification will be forwarded. Neither the probabilistic nor the top-

k value schemes are suitable, as they fail in certain cases. Consider for example a peer with very low values for all its neighbors. Thresholding enables peers to forward to the most "promising" (active in searches) parts of the overlay. A good first approximation is for each peer to use the average of all its neighbors' indices as *Thresh*. Nevertheless, both the average and the median values fail as well in various circumstances (e.g., when all indices have a very close low or high value). Thus, we have to identify a value for *Thresh* that will enable more high quality indices to be selected and less (or none) of the low-quality ones.

3) Local Threshold Computation: After each peer computes the average of its neighbors' reverse index values $\langle \mathrm{RIV} \rangle_t$ at time t, it uses a system-wide *binning* scheme to come up with the actual value for *Thresh*. The binning method divides the space of reverse index values into a set number of bins, $\{Bin_i = ([a_i, b_i), Thresh_i)\}$. Bin_i is characterized by its lower and upper limit values a_i, b_i ($a_0 < b_0 = a_1 < b_1 = a_2 ...$) and a $Thresh_i$ value. The final threshold value is *Thresh*= $Thresh_i$, if $\langle \mathrm{RIV} \rangle_t \in [a_i, b_i)$. For example, assume we use a 2-bin scheme, $\{Bin_0 = ([0, 50), 40), Bin_1 = ([50, \infty), 100)\}$. If $\langle \mathrm{RIV} \rangle_t = 75$, that node will forward to all neighbors with reverse index value over 100. Bins represent an approximation that maps reverse indices to a value representing their quality. Higher numbered bins represent higher quality indices. Values $Thresh_i$ are chosen such that:

$Thresh_{i-1} - b_{i-2} > Thresh_i - b_{i-1}$ and $Thresh_{i-1} < Thresh_i$, where we assume that $b_{-1} = a_0$. For small i values we should pick few neighbors (therefore a high threshold relative to the bin's interval), while for large i (i.e., high quality bins), most of the neighbors need to be chosen. Note that we do not require $Thresh_i$ to belong to $[a_i, b_i)$, nor do we require that $b_i - a_i = b_j - a_j, i \neq j$. As a simple heuristic that produces good results for selecting

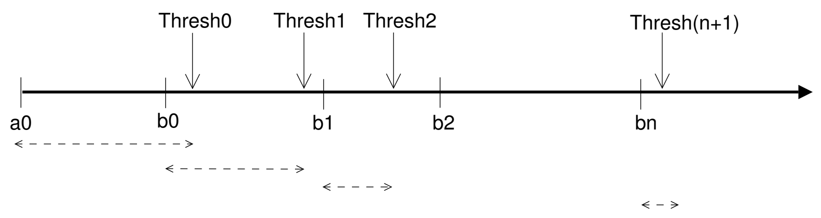


Figure 3.4: Sample binning scheme with the respective *Thresh* and $Thresh_i - b_{i-1}$ values

the $Thresh_i$ values, given $Thresh_0$ for bin $[a_0, b_0)$, we set $Thresh_i = Thresh_{i-1} + \frac{b_{i-1} - b_{i-2}}{2}$. Figure 3.4 gives a graphic description of our binning scheme. Its granularity, controlled by the number of defined bins, can be as fine-grained or coarse as our application requires.

4) Reverse Index Aging: APS updates its index values after either a success or a failure, achieving learning in both situations. This is very important for *AGNO* as well: Peers that lose interest in an object should be left out of the push phase as quickly as possible. Our scheme uses the aging factor ξ together with the last modified time of each reverse index to reduce the influence of inactive ones. Assuming index $P \to Q$ was last modified at time t_{last}, its value at time $t \geq t_{last}$ is: $P \to Q(t) = (1-\xi)^{t-t_{last}} P \to Q(t_{last})$, where $\xi \in [0,1]$. For $\xi = 0.2$, a reverse index value will be 80% of its last modified after one time unit.

The value of ξ dictates how aggressive our aging will be. It depends on the rate at which requests occur (and therefore index updates): The larger the rate of searches λ_r, the more aggressive the aging can be. Nevertheless, it is still application-dependent, since the rate λ_n at which notifications occur (or even their content) largely affects the aging factor. For example, in sharing stock market data, for the duration of a peer's online time it can be assumed that a user is always interested in her portfolio.

For the remainder of this paper, we assume that peers use the same value for ξ which satisfies the inequality: $(1-\xi)^T max\_reduced\_Thresh < \min_i(Thresh_i)$ (1). In effect, we pick ξ such that any reverse index with value less or equal to *max_reduced_Thresh* will be reduced below the lowest threshold (and thus will not be selected) if not used for T time steps (T is defined as our "tolerance" parameter). The maximum $Thresh_i$ represents the minimum high-quality index value, as this is defined by our binning scheme. Therefore, by setting $max\_reduced\_Thresh = \max_i(Thresh_i)$, we choose ξ such that all reverse indices up to that level of quality are discarded after a period of time T without getting updated. Choosing larger values results in a more aggressive aging. The same is true for choosing smaller T values. Assuming that, in the vast majority of cases, notifications are considerably less frequent than requests, we set $T = O(1/\lambda_r)$, which defines the tolerance interval to be in the order of the average request interarrival period. This is done in order to quickly identify and decrease idle indices in the overlay.

5) Estimation of λ_r: In order for our scheme to work without requiring a priori knowledge of the request rate but also to be able to adapt to changes in the workload, we need an effective yet inexpensive mechanism to estimate its value and compute the new ξ before each push. This value is then piggybacked downstream and used by all receiving nodes. In order to estimate λ_r, we need the zeroth and first frequency moment of the request sequence arriving at a server. F_0 is the number of distinct IDs that appear in the sequence, while F_1 is the length of the sequence (number of requests). Servers can easily monitor the number of incoming requests inside a time interval. Many efficient schemes to estimate F_0 within a factor of $1 \pm \varepsilon$ have been proposed (e.g., [43, 44]). We use one of the schemes in [43], which requires an extra of only $O(1/\varepsilon^2 + \log(m))$ memory bits (at server-nodes), where m is the number of distinct node IDs. In reality, m is in the order of the distinct peers within TTL hops from a server, since only these nodes can reach it: $m \simeq d^{\mathrm{TTL}} \Rightarrow \log(m) \simeq \mathrm{TTL} \cdot \log(d)$, which is usually very small. After each push phase, both estimates are reset and a new estimation cycle begins.

6) Backpointer Selection: Finally, we specify which backpointers are used by a node that receives a group notification message. Clearly, following the same number of backpointers at different peers and times is not efficient. Our method utilizes the local thresholding computation to assist in the process of selecting valid backpointers. As we mentioned before, the threshold value is representative of the average quality of a peer's reverse indices (higher bins choose on average more neighbors to forward to). Given that a peer's threshold bin is i at time t, the probability with which each stored backpointer will be followed is p_{n_i}, given from the set $\{p_{n_1}, p_{n_2}, \ldots p_{n_i}, \ldots\}$ (i.e., one p_n value for each bin). We choose those values such that $p_{n_i} > p_{n_j}\ \forall i < j$, since better quality bins forward to more neighbors and need not waste more bandwidth. With this scheme, *AGNO* adaptively balances the amount of forwarded messages per peer between the shortcuts and the neighbors according to the current quality of its reverse indices.

7) Summary: *AGNO* is a probabilistic group notification scheme that integrates search indices with a constant amount of shortcuts to effectively route messages in an unstructured overlay. It utilizes a binning scheme to choose between the exact amount of useful information from each source and an aging mechanism to gracefully adapt to member departures, requiring no explicit cooperation on their part.

3.3 Simulation Results

We use a message-level simulator written in C (about 2,100 lines of code) which runs on a linux-based platform using an Athlon 2.1GHz processor and 1GB of main memory. Requesters make searches for objects using *APS* at rate λ_r (exponentially distributed interarrival times), while servers initiate push transmissions at rate λ_n. At each run, we randomly choose a single node that plays the role of a server and a number of requesters, also uniformly at random. Results are averaged over several tens of runs.

We present results for both *random* and *power-law* graphs. There has been strong evidence [45] that connects large-scale unstructured P2P networks to a power-law topology. We utilize the *BRITE* [46] and *Inet-3.0* [33] topology generators to create the random and power-law graphs respectively. We consider 10K node graphs with average node degrees around 4 (similar to Gnutella snapshots [45]). Results for graphs up to 50K nodes and larger average degrees are qualitatively similar.

Finally, the following basic metrics are used to evaluate the performance of a scheme: The *success notification rate* (or *success rate* in brief), which is the ratio of contacted group members versus the total number of group nodes and the bandwidth *stress*, which we define as the ratio of the produced messages over the minimum number of messages in order to contact all members.

AGNO Parameters: We choose to set $c \cong d$, which reserves an amount of space for backpointers roughly equal to the average node degree. Potentially, for each of its neighbors, any peer can keep one backpointer address during a search. Ref. [45] shows that over 90% of the node pairs in Gnutella are around 5 hops away. Given this value as an estimate for the TTL parameter, we set $p_r \cong 1/\mathrm{TTL}$, so that on average one peer on the search path will store the requester's address. While we experimented with different distributions (e.g., favoring storage of backpointers for closer nodes), the results did not considerably vary from the uniform policy.

Given the index update policy used by *APS*, we employ a simple 3-bin scheme. The first bin represents indices below the initial value (very few to no successes, $p_{n_1} = 0.4$), the second those with some hits ($p_{n_2} = 0.15$) and the last one those with more successes ($p_{n_3} = 0.05$). The values of p_n for the second and third bin are chosen deliberately low since the values of the reverse indices are high enough and backpointers are less frequently useful. Finally, from equation (1) and setting $T = 2T_r$ (where $T_r = 1/\lambda_r$ is the average request interarrival period), we have: $\xi = 1 - 0.44^{0.5\lambda_r}$. The value of λ_r (and therefore ξ) is estimated right before each server push using $\varepsilon = 0.1$.

Compared Methods: We compare our method against the SCAMP protocol [47] which defines explicit membership procedures and the two rumor-spreading schemes in [41]: *Rumor Mongering* (RM) and its deterministic version (det-RM), where peers have complete topology information. All three schemes are gossip-based approaches for update dissemination/group communication in unstructured overlays. Furthermore, they do not require a single point of contact or frequent refresh messages, similar to *AGNO*.

In SCAMP, joining members subscribe by contacting a random existing member. Upon receiving a subscription request, a member forwards it to all the members in its local repository. Nodes decide probabilistically whether to store or forward the subscription. For the unsubscription process, a node notifies the locally known members to replace its ID with the IDs of the members it has received messages from. Group communication is performed in the standard gossip-based manner. SCAMP is shown to converge to a local state of slightly over $log(n)$ member IDs, which guarantees with high probability that all members will receive a notification.

In [41], peers that have received a message less than F times, forward it to B randomly selected neighbors, but only those that the node knows have not yet received it. The deterministic version of that algorithm requires global knowledge of the overlay. Nodes forward messages to all neighbors with degree equal to 1, plus to B remaining neighbors that have the smallest degrees. For SCAMP, we first run the membership phase, in which we favor the method by assuming joining peers know all already joined members. The parameters for those three methods are the `branching factor` B, which represents how many other peers shall be contacted per forwarding step and the `seen` value F that represents how many times a peer can receive the same message before dropping it.

Finally, for demonstration purposes, we design and implement a pure shortcut selection scheme (*Shortcuts*) inspired by the DHT-based multicast tree creation. Search packets carry the (ID, address) values of the last node along the path interested in the object so far. Initially, this pair contains the requester node's information. During the search, an interested peer that receives a search message, decides with probability p_r whether to store the last member's ID or not. Moreover, it replaces this ID with its own before forwarding the request. With this scheme, we create a small sub-overlay of soft-state backpointers with direction from the object holders towards the group members. For simplicity, we assume the same maximum number of shortcuts as in *AGNO*. In the push phase, a peer forwards to all valid shortcuts, using the standard TTL scheme (unlike *AGNO*, where backpointers are contacted with a TTL $= 1$).

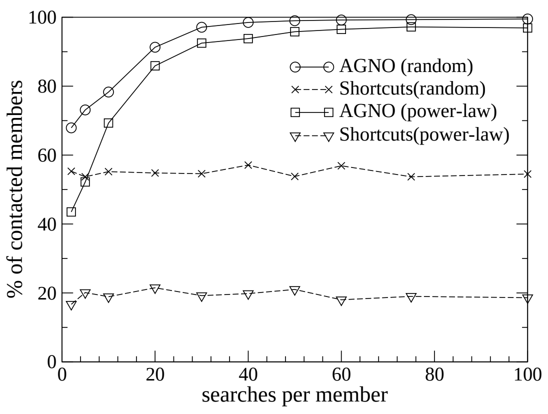


Figure 3.5: Success rate over variable number of searches

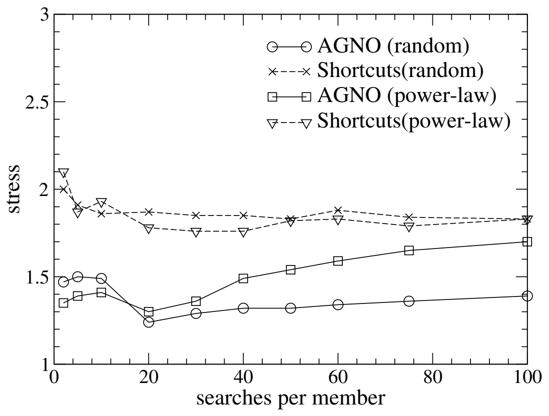


Figure 3.6: Stress over variable number of searches

Table 3.1: (Success rate, Stress) results for the remaining methods (500 requesters)

	SCAMP	**RM**	**det-RM**
10K Random	$(89\%, 2.7)$	$(89\%, 34.5)$	$(98\%, 31.1)$
10K Power-law	$(68\%, 2.1)$	$(27\%, 13.6)$	$(65\%, 10.8)$

3.3.1 Basic Performance Analysis

In this first set of experiments, using a group of 500 requesters, we vary the number of lookups each of them makes before a single push phase occurs. We report the results averaged over sets of 10,000-Node random and power-law topologies ($d = 4$ and $d = 4.1$ respectively). Figures 3.5 and 3.6 present the results for *AGNO* and *Shortcuts* which are affected by the number of searches.

We notice that the pure shortcut scheme cannot provide an efficient notification method by itself. *AGNO* quickly contacts the majority of requesters after only a few searches take place, while maintaining a low stress factor. As our scheme creates better quality indices, there exists a slight variation in the stress. This is due to the fact that after a certain number of queries, peers switch to a different (higher) bin on average.

In the power-law topologies, where about 34% of the peers have degree one, fewer

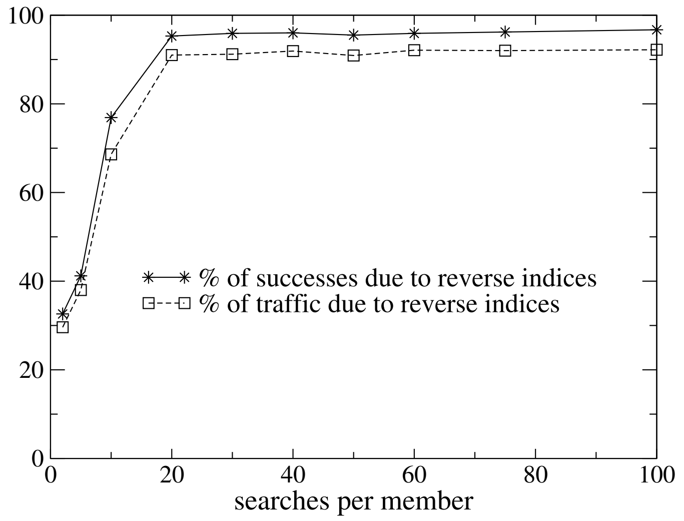


Figure 3.7: Utilization of pure forwarding vs. backpointers

paths are used compared to the random graphs. This, combined to the fact that $\xi = 0$ in these experiments, explains why the stress for *AGNO* slightly increases with more requests. The respective results for the remaining methods (not affected by searches) are shown in Table 3.1. *AGNO* proves very accurate (in the big majority of runs) and also the most bandwidth-efficient of the compared methods. All three rumor-spreading schemes show considerably worse numbers in the power-law topologies. *det-RM* is much more effective than *RM* in such graphs, which is in accordance to the findings of [41].

Figure 3.7 shows the percentage of contacted members and messages of *AGNO* purely attributed to forwarding (not backpointers). As we move from less to more precise reverse indices (from fewer to more queries), our method uses a decreasing number of backpointers. These results also depict the usefulness of the backpointer scheme as for less accurate indices they can provide with over 50% of the contacted members.

Table 3.2 summarizes the effect that a change in the number of maximum stored backpointers (c) has on the performance of *AGNO*. We select two runs from the previous

Table 3.2: Effect of parameter c

	10 queries/member		20 queries/member	
	success rate	**stress**	**success rate**	**stress**
c=1	68.7%	1.17	90.3%	1.16
c=2	73.5%	1.27	91.5%	1.20
c=4	77.9%	1.42	91.6%	1.23
c=8	79.6%	1.80	92.5%	1.37
c=16	81.2%	2.80	92.9%	1.49

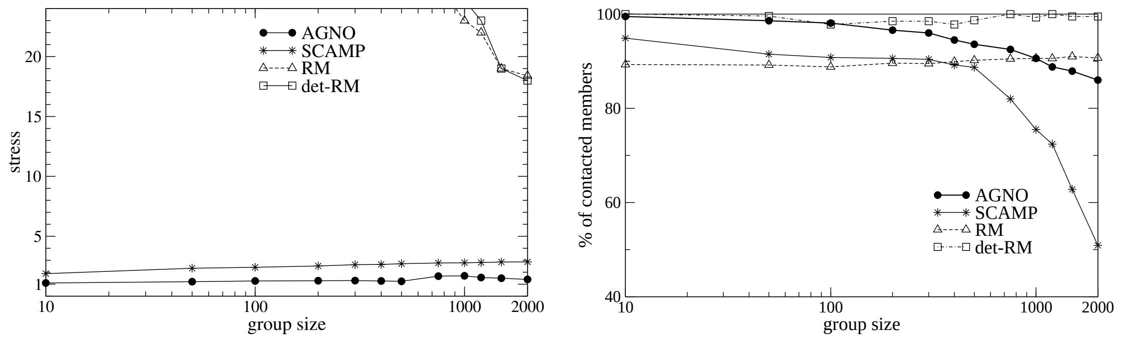


Figure 3.8: Stress and success rate over variable group size

experiment, where each of the 500 members make 10 or 20 queries in the random topologies. For 10 queries per requester, many peers fall into bins 1 and 2 on average, while the majority of nodes operate on bin 3 with twice as many queries. With less queries (and larger backpointer usage), the increase in the success rate over our selected $c = 4$ is very small compared to the increase in stress. As the indices get more accurate, the method becomes almost insensitive to the value of c.

Next, we measure the scalability of our method with group sizes ranging from 10 to 2,000 peers using the random topologies. Requesters make only 10 searches on average, immediately followed by a single push phase from the server. For SCAMP, the membership protocol is run before each different group size. For RM, det-RM and SCAMP, we set $B = 3, F = 1$, which proves the best combination taking into consideration both the

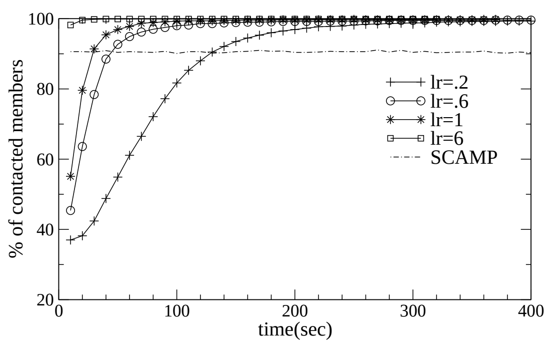


Figure 3.9: Success rate over variable λ_r values ($T_n = 10sec$)

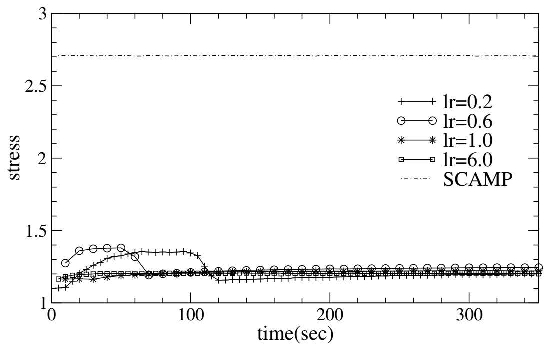


Figure 3.10: Stress values over variable λ_r values ($T_n = 10sec$)

success rate and stress metric. Figure 3.8 presents the results.

Our method is very successful in all group sizes, deteriorating only slightly as the members increase. This happens because with more requesters, their average distance from the server increases (the number of peers reachable from a node increases exponentially with the hop distance). This makes *APS* searches (and its indices) less accurate for some requesters. The RM schemes produce a similar number of messages regardless of the group size, while the closest competitor (SCAMP) has roughly twice the stress value of *AGNO*, without including the overhead of the membership phase. Our method manages to contact a very high percentage of the members (86-99.5%) using an almost constant message ratio over the group size.

3.3.2 Sensitivity to λ_r

In this section, we try to evaluate the effectiveness of our λ_r estimator and the computed ξ values over the random topologies. Results for the power-law graphs are qualitatively similar.

Assuming a group size of 1,000 peers, we try to evaluate the performance of *AGNO*

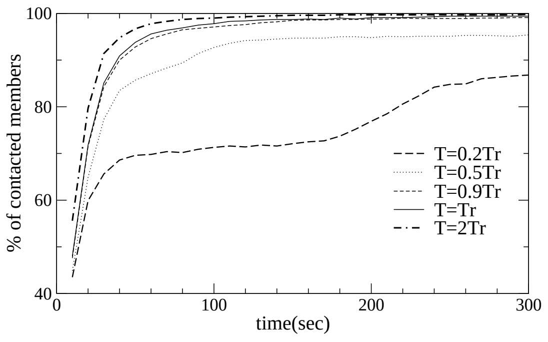


Figure 3.11: Success rate for different values of T ($T_n = 10sec$)

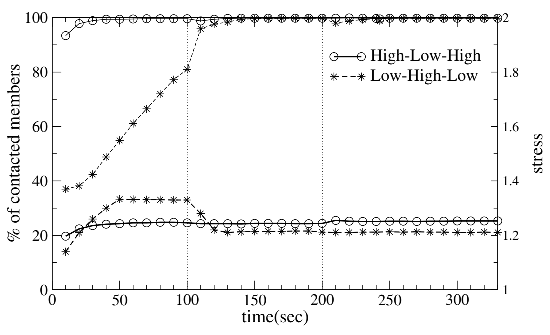


Figure 3.12: Adaptation to a change in λ_r by a factor of 20

for different λ_r values. Figures 3.9 and 3.10 show the results. Not surprisingly, the larger the value of λ_r, the faster the increase in the success rate, since indices get accurate faster. Another observation is that, regardless of the average request rate, our method asymptotically manages to contact all interested peers and reach a very low stress level (below 1.3). For most realistic scenarios ($T_n \gg T_r$), the choice of T_n does not affect *AGNO*'s performance. In the very rare cases that $T_n < T_r$, we just set $T = O(T_n)$ to achieve comparable adaptation. In all cases, our adaptive aging mechanism selects a suitable value for ξ such that the stress remains almost constant and below 1.4, half the value of the best of the remaining schemes (SCAMP). For small request rates, peers adapt using initially low and then higher quality bins (thus the slight variation in stress). The smaller the value of λ_r, the longer this adaptation takes.

The value of T defines how aggressive the aging is. The smaller it gets, the bigger ξ becomes and thus the bigger the reduction in the reverse index values. Figure 3.11 shows how the success rate of *AGNO*, given 1,000 peers making requests at $\lambda_r = 1/sec$ (and $T_n = 10sec$), varies by changing the value of T relative to the average request period $T_r = 1/\lambda_r$. Our default choice for $T = 2T_r$ yields very good results, while choosing values close to the request period also produces fast learning. As T decreases more, the success rates increase at much smaller rate. Surprisingly, even if we employ twice as aggressive an aging as the average request rate, over 80% of the members will be contacted after three *AGNO* pushes (30 seconds). Nevertheless, it is not safe to assume that the larger the value of T the better. This would be the case if, for example, we had a static group size (no aging necessary); a significant number of member departures combined with a large value for T would delay the adaptation to the new group size and cause more messages to be created than necessary.

Finally, Figure 3.12 shows how effective our adaptive λ_r estimation scheme is. We simulate the extreme case where the 1,000 requesters suddenly change their query rates by a factor of 20 (from $\lambda_r = 4/sec$ to $\lambda_r = 0.2/sec$ and vice versa). Our goal for the transition from high to low rate is to quickly decrease ξ so that our success rate is not affected. For the transition from low to high rate, we wish to quickly adjust the new ξ value according to the increased requests, such that no more than the necessary indices increase their value. We name our two runs high-low-high and low-high-low respectively: Starting with a rate of $\lambda_r = 4/sec$ (0.2/sec), requesters drop (increase) their average number of requests to 0.2/sec (4/sec) at time $t = 100sec$. At time $t = 200sec$, they increase (decrease) their rates back to 4 queries/sec (0.2/sec). The top two lines correspond to success rates while the bottom two to the respective stress values. The maximum observed decrease in the success rates at 100 or 200 seconds is only 2%, while the stress values remain almost unaffected (increase equal to 0.01).

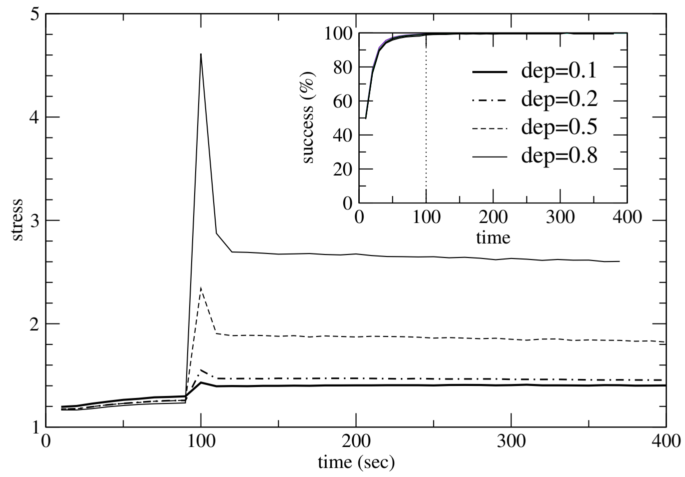


Figure 3.13: Stress and success rates when a different ratio of peers depart at time t=100sec ($\lambda_r = 1/sec, T_n = 10sec$)

3.3.3 Changes in Group Size

We now evaluate the performance of *AGNO* under dynamic changes in the group size. Our goal is to allow for members to join or leave the group with the minimum amount of performance degradation. Employing this approach that ties group membership to the interest (or lack thereof) of peers for objects, we require no coordination between members nor any single authority node.

Figure 3.13 shows how our two metrics are affected by having 10%–80% of the 1,000 requesters leave the group (stop making queries) at time $t = 100sec$. We assume that all these nodes jointly and instantly decide to leave the group (as a worst-case scenario). In all runs, the stress value peaks at the time of the departures, since the same number of peers are notified but fewer are now considered as members. The size of the departing subgroup directly affects the stress increase. The stress value instantly drops due to our aging mechanism, but it does not reach its previous value (though it decreases very slowly). This is due to the fact that a peer's indices get updated not only when it makes a request but also

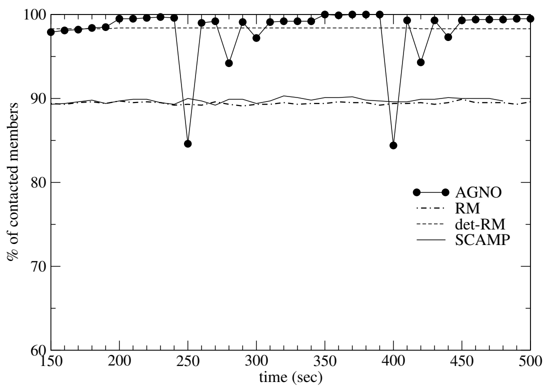


Figure 3.14: Success rate after a series of member departures and arrivals ($\lambda_r = 0.5, T_n = 10$)

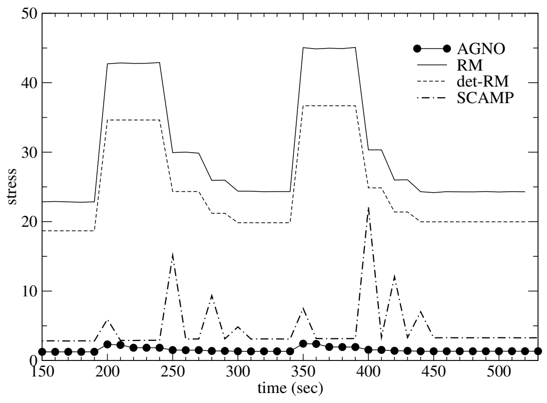


Figure 3.15: Stress after a series of member departures and arrivals ($\lambda_r = 0.5, T_n = 10$)

when any request passes through it. Therefore, while shortcuts for departing peers expire, indices leading to them may still have large values, depending on the relative positions of other requesters in the overlay. The amount of increase for {10%, 20%, 50% and 80%} of the members departing is {7%, 12%, 38% and 100%} respectively. The amount of increase gets reduced as the original group size gets smaller, which proves our previous point: Assuming 200 initial members instead, the respective stress increase percentiles are {7%, 9%, 16% and 25%}. On the other hand, as the included graph shows, our success rate is not affected at all. We show next that the decrease in stress after new members join compensates for the increase after peer departures.

Figures 3.14 and 3.15 display the performance of the compared methods under a combination of member joins and leaves. At times $t = \{200, 350\} sec$, 50% of the current group members decide to leave. At $t = \{250, 280, 300, 400, 420, 440\} sec$, 50% of the non-active requesters re-join the group. Members make requests at $\lambda_r = 0.5/sec$, while the group notification phase is performed every 10 secs.

The success rate shows an instant decrease at the exact time of arrival which is proportional to the number of joining peers. Nevertheless, always more than 85% of the current members are contacted, and *AGNO* has learned of their presence by the exact next transmission. In the next push phases, the method quickly reaches its previous levels. On the other hand, the value of stress decreases after member joins and balances the small increase that occurs after member departures. SCAMP and the two rumor spreading schemes show big variations in the stress metric. For RM and det-RM, this happens because of the change in the group size (same number of messages regardless of peer membership), while for SCAMP this is due to the subscription and unsubscription processes. *AGNO* contacts the vast majority of members at a cost 1 to 10 times lower than the closest compared method (SCAMP).

3.3.4 Sensitivity to the Binning Scheme

In all our experiments, we used the same binning scheme. The question is how sensitive *AGNO* is to different binning configurations. An adaptive process that will adjust an initial binning configuration according to the method's performance is a difficult task: Even if the server knows about the number of interested peers that received a notification (by members acknowledging through piggybacking), finding how many messages were sent in a distributed manner requires extra overhead. Furthermore, success rate and stress are often conflicting goals.

Assuming the simple solution of a single binning scheme, we evaluate AGNO using different bin configurations. We measure the success rate and stress of a single push phase to a group of 1,000 peers each having made 20 requests on our set of random graphs. We

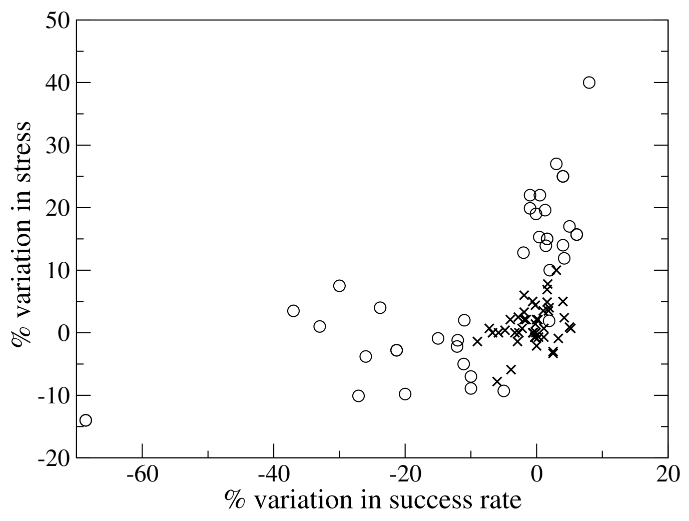


Figure 3.16: Comparison of 100 different binning configurations to the original one

produce 100 different binning configurations with 2, 3 and 4 bins for direct comparison to the original scheme. Our study confirms that if we follow the empirical conditions of Section 3.2.1, even with coarse granularity (2 bins), our method's performance exhibits small variation. On the other hand, random choices for the bin limits and/or threshold values result in performance significantly degraded. In Figure 3.16, each point represents the percentile variation in our 2 basic metrics of a bin configuration compared to the original one. Configurations marked with '$\times$' represent choices that follow our rules, while '$\circ$'s represent bin settings that do not adhere to those rules. Even random choices of the binning scheme which reasonably respect our simple conditions exhibit less than 10% variation.

3.3.5 Real Traces

We now present results from simulations using real traces. In our first experiment, we monitor the change in content for two very popular web sites, CNN and BBC

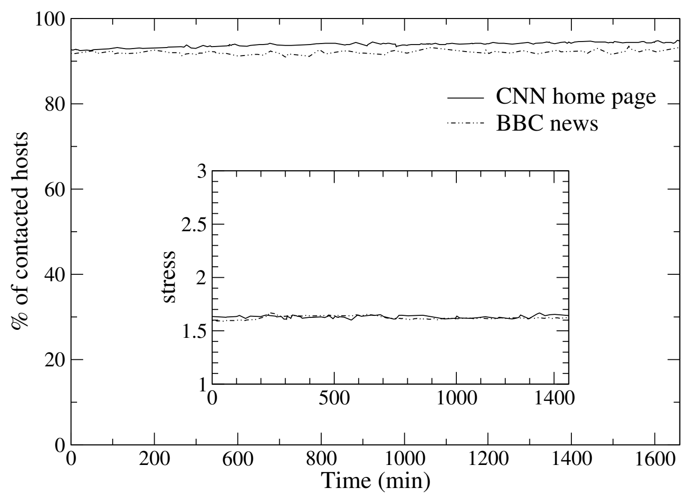


Figure 3.17: Average results for one-day periods for the CNN and BBC news front pages

news. We retrieve their home pages (*http://www.cnn.com* and *http://news.bbc.co.uk* respectively) at a minute granularity and record the time that their content has been modified. To determine that, we extract the official *Last Updated* string from the page and also directly compare the files [1]. Each page is preprocessed with *HTML Tidy* [48]. Taking advantage of the fact that the overall structure of the same page rarely changes, we discard code, advertisements and pictures that change after each browser refresh, focusing on content. We monitor the changes over a period of 2 weeks, from Feb. 16th to Mar. 1st, 2004.

The CNN home page changes every 18.1 minutes on average, while BBC's news page changes every 8.6 minutes. In our experiments, we use the same 10,000-Node power-law graphs of the previous sections and a group size of 1,000 requesters, making requests with exponentially distributed interarrival times ($\lambda_r = 0.1/min$) for those two pages. The notification phases occur each time a page is updated, as given by our col-

[1]This method was developed as part of a project for the CS724 Database graduate course in University of Maryland

lected data. At exponentially distributed intervals (an average of 1/15minutes), we choose with equal probability among the following events: 10% of the members stop requesting the pages or 80% of inactive members resume their requests or nothing happens. On average, we vary our setup over 60 times per run. Figure 3.17 shows the results over the 14 1-day periods (averaged over all graphs with multiple runs for each). *AGNO* manages to exhibit very high accuracy and adapts its notification mechanism such that the stress value always remains stable between 1.6 and 1.7.

Finally, we test the behavior of our scheme in a much more dynamic environment. We use real traces taken from NYSE stock trades, which describe the accesses, volumes and values of all quotes in a 10-day period (Apr. 3-14, 2000). Aggregating to minute granularity, we monitor quote activity (accesses-updates) during a busy time interval (11:00-11:59am) each day. For our simulation, using the same power-law topologies as in the previous experiment, we assume a standard client population (group members) equal to the maximum number of accesses recorded at any minute per individual quote. We model our system such that, given there were Q accesses at a given minute, only the first Q clients are assumed to query for that object. This is equivalent to having a variable request rate for each member. Pushes were conducted whenever a quote's value was updated, with a maximum of one notification per minute.

Figure 3.18 shows the results for three of the most active quotes, SUNW (Sun Microsystems Inc.), MSFT (Microsoft Corp.) and ORCL (Oracle Corp.) The statistics for each of these quotes are presented in Table 3.3. The interesting statistic here is the high standard deviation value for all three quotes, which translates to a wide range of different λ_r rates for each requester in our experiments. Updates (=push transmissions) were per-

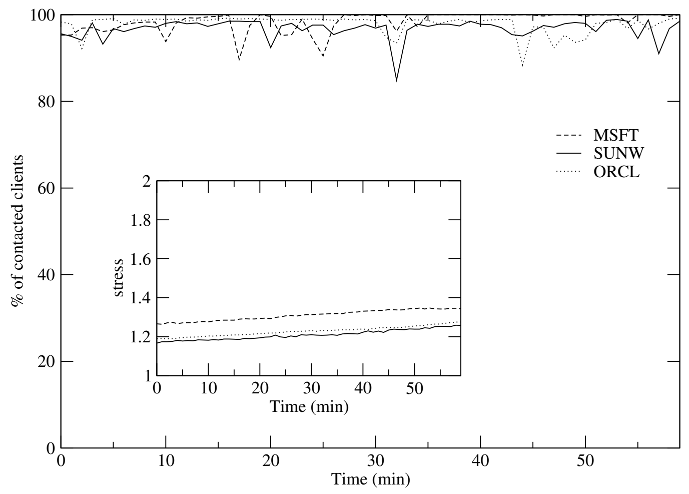


Figure 3.18: Results for a 7-day period for the Microsoft, SUN and Oracle quotes between 11:00am and 11:59am

	Mean	**Max**	**STD**
SUNW	148	1037	118
MSFT	240	1171	184
ORCL	165	1137	101

Table 3.3: Access statistics for the three quotes

formed almost every minute. For all three datasets, *AGNO* achieves a high success rate with few small spike-shaped decreases occurring. A more detailed analysis of the data shows that these coincide with sudden increases (often more than 400%) in the group size (or accesses per minute), as were observed in the data. Given traces for more days, those spikes would have less weight on the averages. We also depict the average stress values for the quotes, which are kept at a very low level throughout the whole interval. These results also show that our adaptive forwarding and aging mechanisms work effectively even in the most dynamic environments. Results for less popular quotes or for time intervals outside high-access periods are qualitatively similar and were not selected since the average group size was less than 100.

3.4 Summary

AGNO is an adaptive and scalable message dissemination scheme for unstructured Peer-to-Peer networks. Our method integrates knowledge accumulated during searches to enable content-providers contact interested peers with very small overhead. We described in detail our adaptive mechanisms to regulate message forwarding according to the quality of existing knowledge as well as to ensure efficient performance in all group operations. A variety of simulations using both synthetic and real traces show that *AGNO* adapts quickly to variable request rates and group sizes, being at least twice as bandwidth-efficient as the compared methods.

Chapter 4

Adaptive Replication for Unstructured Overlays

4.1 Introduction and Overview of our Approach

While Peer-to-Peer's success can still be largely attributed to file-sharing applications (e.g., [49–51]), an increasing number of different utilizations of this technology have emerged. P2P has been proposed to assist in web caching [52], instant messaging [53], e-mails [54], update propagation [36], conferencing [37], etc.

A basic requirement for every P2P system is fault-tolerance. Since the primary objective is resource location and sharing, we require that this basic operation takes place in a reliable manner. Nevertheless, in a variety of situations, the distributed and dynamic nature of the environment stresses the system's ability to operate smoothly. For example, the demand for certain content can become overwhelming for the peers serving these objects, forcing them to reject connections. *Flash crowds*, regularly documented surges in the popularity of certain content, are also known to cause severe congestion and degradation of service [55]. Failing or departing nodes further reduce the availability of various content. Consequently, resources become scarce, servers get overloaded and throughput can diminish due to high workloads.

Data replication techniques are commonly utilized in order to remedy these situations. Replicating critical or frequently accessed system resources is a well-known technique utilized in many areas of computer science (distributed systems, databases, file-systems, etc) in order to achieve reliability, fault-tolerance and increased performance. Resources such as content, location of replicas, routing indices, topology information etc, are cached/replicated by multiple nodes, alleviating single points of contact in routing and sharing of data. This has the additional benefit of reducing the average distance to the objects. Replication can be performed in a variety of manners: Mirroring, Content Distribution Networks (CDNs [56, 57]), web caching [58], etc.

However, these approaches often require full control and provide static replication. Static replication schemes require a priori knowledge of the popularity/workload distribution in order to compute the amount of replicas needed. In large scale unstructured P2P networks, peers usually operate on local knowledge, having variable network connectivity patterns and no control over the induced topology or workload. Data availability and efficient sharing dictate replication in this challenging environment. Structured P2P systems (DHTs) provide with the state necessary to accurately identify the paths that requests take. This information can be used to point out, with high probability, all possible replication locations. However, such information is not available in unstructured overlays. File-sharing applications implicitly handle replication through object downloads, while some force their users to maintain the new replicas for the benefit of others. Yet, this does not tackle the issue of real-time replication responsive to workload for unstructured environments.

In this part of our work we present *APRE* (*Adaptive Probabilistic REplication*), a replication method for unstructured overlays based on soft-state routing indices. Our approach focuses on providing an adaptive solution to the problem of availability together with minimizing the instances of server overloads and serious service degradation. Our

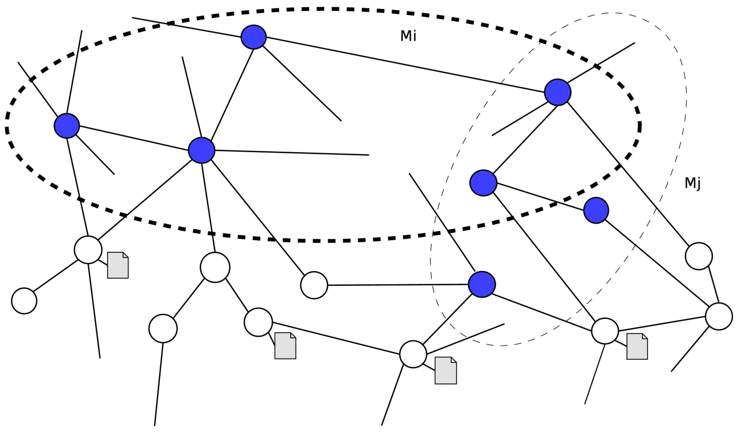


Figure 4.1: Part of the overlay network of our model. Dark nodes inside the bold dotted ellipse represent $\mathcal{M}_i$, while those inside the thin dotted ellipse represent $\mathcal{M}_j$. Peers with a file attached also serve objects *i* or *j*

system dynamically "expands" and "contracts" its resources according to the workload as perceived locally. New replicas are created in areas of high demand in the overlay, thus disposing of the need to advertise them. Moreover, this will be done in a completely decentralized manner, with minimal communication overhead and using absolutely affordable memory space per node.

The framework we use to describe our system is a model as general and realistic as possible, avoiding many unnecessary assumptions, thus following the general description of Section 1.3. As a motivating example, assume an unstructured P2P system, where peers share and request replicated resources. Objects are assumed to be requested regularly, since their content changes over time: results of a live sports meeting, weather maps, security updates, real time aggregated statistics, tactical data, etc. Some of the nodes provide fresh content, while others share versions they have recently downloaded. Peers that are interested in an object conduct searches for it in order to locate a fresh or closer replica.

Figure 4.1 gives a graphic representation of the *APRE* framework. For each object i, there exists a set of peers called the *server set* $\mathcal{S}_i = \{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\}$ that serve the specific object. These are the nodes that, at a given time, are online, store object i and are willing to share it. A subset of $\mathcal{S}_i$, the *mirror set* $\mathcal{M}_i \subseteq \mathcal{S}_i$ (the shaded peers) represents the set of peers that, if online, *always* serve i. This does not imply that all peers in $\mathcal{M}_i$ will always be online, their connectivity in the overlay will remain the same, or that they will never refuse connections. But we can assume, without loss of generality, that these nodes will be mostly available. Our assumption is not unrealistic: Imagine that these servers can represent mirror sites/authority nodes that provide up-to-date content. Apart from the mirror set, other peers that already host or have recently retrieved an object can serve requests for it (nodes with files attached to them in Figure 4.1). A server set comprises of these nodes plus the corresponding mirror set.

Naturally, peers may belong to server or mirror sets for multiple objects. While this is a symmetric environment, it is clear that nodes exhibit different sharing abilities. A variety of parameters, including storage and CPU capability, popularity of stored objects, system workload, connectivity, etc, contribute to this fact. Some of these factors remain more or less static over time (e.g., processing power or the maximum available bandwidth of a host), while others change dynamically.

Whichever the case, it is safe to assume that each peer in this system imposes a limit on the services it provides to other peers. This is something that is already utilized by several file-sharing applications (e.g., Kazaa [49], FTP servers, etc). There exist a variety of metrics that can be used to realize those limits. Peers may set restrictions on the number of concurrent connections, their upload bandwidth, the number of shared files, the rate of received requests, etc. In this work, we focus on two of these parameters, namely workload and object popularity as they are manifested through a single observable quantity, the request rate λ. It is obvious that servers of popular (or temporally popular) items receive a larger number of requests, which can possibly affect their sharing ability as well as the system's behavior.

Given this general framework, our goal is to design and implement a replication protocol that will provide efficient sharing of objects (in terms of providing low load operation), scalability and bandwidth-efficiency. *APRE* is a distributed protocol that automatically adjusts the replication ratio of every shared item according to the current demand for it. By utilizing inexpensive routing indices during searches, loaded servers are able to identify "hot" areas inside the unstructured overlay with a customizable push phase. Chosen nodes receive copies thus sharing part of the load. Under-utilized replicas are released, allowing their hosts to store more popular content. The rationale behind *APRE* is the tight coupling between replication and the lookup protocol which controls how searches get disseminated in the overlay. By utilizing search state, in a manner similar to *AGNO*, we are able to identify in real-time "hot" or "cold" paths and avoid the need of advertising constantly created replicas. Our experimental evaluation shows that this method proves very efficient in a variety of metrics and environments.

4.2 Adaptive Replica Expansion/Contraction: APRE

Our main goal is to provide a completely decentralized mechanism through which the system will adaptively expand its replica size when demand is increased and will

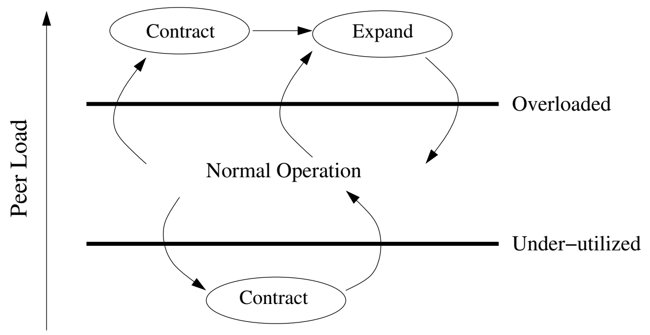


Figure 4.2: State transitions in our system

shrink when demand will fall. *APRE* is based on two basic operations: *Expand* and *Contract*.

The high-level behavior of our system can be described using a simple model (Figure 4.2): In normal mode, nodes can adequately serve requests and also retrieve objects. As load increases due to incoming requests, some reach their self-imposed limits. By invoking the *Expand* process, we aim at bringing the node status back to normal and lower the average load for a specific object through the creation of more replicas. Normal operation through the distribution of load will not be necessarily achieved in a single step. Consider, for example, that a peer initiating *Expand* may receive requests for multiple objects. Expanding with respect to one of them will probably lower its load, but will not necessarily bring its level back to normal. As load decreases, nodes can free up space (and the respective resources) and thus share a bigger portion of the workload.

Let us now discuss why the system would benefit from these two operations. When parts of the server set $\mathcal{S}_i$ receive too many requests for object i, the following may occur: Clients' connections get refused, while servers receive an increasing amount of requests and their performance deteriorates. Both groups would benefit from an increase in the number of replicas available, especially if those replicas were placed inside the areas of

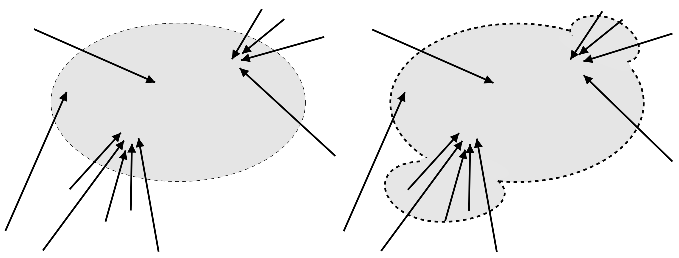


Figure 4.3: The shaded oval represents a server set for a specific object. Our system expands by creating replicas inside two areas where demand (depicted by arrows) is high.

Algorithm 1 *Expand*

```
1: if Replica $i$ at node $s$ reaches its limit then
2:     $P \leftarrow FindPossibleServers(i)$; $\{P \cap \mathcal{S}_i = \emptyset\}$
3:     $Activate(i)$ at $Y \subseteq P$ {Replicate at a subset of the nodes in the high-demand area}
4: end if
```

high demand in the overlay.

Conversely, consider that one or more subsets of $\mathcal{S}_i$ have recently received very few requests for object *i*. This practically means that an amount of their storage space is under-utilized. They could remove *i* to free up space or replace it with another object of higher demand. We have to stress here the point that the system will not force any peer to store or serve an object until this becomes necessary. Peers with available storage can play that role. *Contract* will also be invoked when a peer is called to join $\mathcal{S}_i$ but cannot do so without exceeding its limits (e.g., available storage). Note that peers can still choose to reject a certain action, e.g., refuse to remove an object in order to serve a new one.

Algorithm 1 describes the high-level operation of the *Expand* process. It is invoked by peers receiving more requests than those that they are willing to accept. Overloaded peers have to identify the set *P*, i.e., candidate nodes for replication inside query intensive areas. A subset *Y* of these nodes is selected and, upon their agreement, the new replicas are transfered (*Activate*). Figure 4.3 shows an example of our system expanding in re-

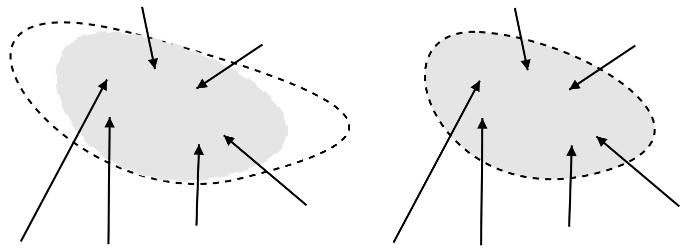


Figure 4.4: Due to low demand in certain regions of the server set (depicted as white areas inside the dotted line), our system contracts its replica set

Algorithm 2 *Contract*

```
1: if (Replica $i$ at node $s$ is under-utilized) or ($s$ receives $Activate(j)$) then
2:     $i \leftarrow ChooseObject()$; $\{i$ is among the candidates for eviction$\}$
3:     $Deactivate(i)$;
4:     if ($s$ received an $Activate(j)$) then
5:         $Activate(j)$;
6:     end if
7: end if
```

sponse to increased demand for a specific object. On the left, we see some initial server set (gray oval) and the demand for *i* (arrows from various parts of the network). Servers in two areas are overloaded with requests, thus forcing extra replicas in those two areas to be activated. $\mathcal{S}_i$ expands, as we see on the right part of the picture, in response to the current demand for object *i*.

Algorithm 2 describes our *Contract* process. It is invoked by a peer that either receives a low amount of requests for the object(s) it serves or is requested to serve a more popular one but cannot do so without freeing up some space. In any case, peers stop serving the object(s) that fall into these categories (*Deactivate*). Function *ChooseObject* decides at each point which object should be deactivated at nodes that have decided to serve a new object (i.e., received an *Activate*) but have reached their storage capacities. Natural choices are to have the new replica replace the least recently requested or the least popular one. Figure 4.4 shows that two areas of the server set (the areas inside the dotted line) do not receive any requests for object i. This leads to the contraction of $\mathcal{S}_i$ which is now the gray oval on the right part of the figure. Our goal is to achieve a system behavior that resembles the buffer management techniques in databases: Viewing the P2P network as a large buffer, we want to decide (in a distributed and dynamic manner) the ratios of objects in the buffer according to user-specified queries (i.e., workload).

4.2.1 Protocol Implementation

In this section we describe the actual implementation of the *APRE* protocol as described by the *Expand* and *Contract* algorithms. We assume that servers measure load and perform replication on a per-object basis, at the same level of granularity with lookup and reverse indices of *APS* and *AGNO* respectively. Vital to the success of our scheme are the following:

1. A mechanism to identify object popularity
2. A mechanism to create replicas inside high-demand areas
3. Minimization of communication within each server set

The conditions of line 1 in Algorithms 1 and 2 describe when *Expand* or *Contract* are initiated. We believe that each peer can independently choose when to initiate an expansion or when to deactivate a replica. Therefore, there is no need for any message exchange between servers.

We assume that each server s defines the maximum number of requests that replica i can accept per time unit $Limit_{s,i}^{up}$. If it receives less than $Limit_{s,i}^{down}$ requests for object i, this

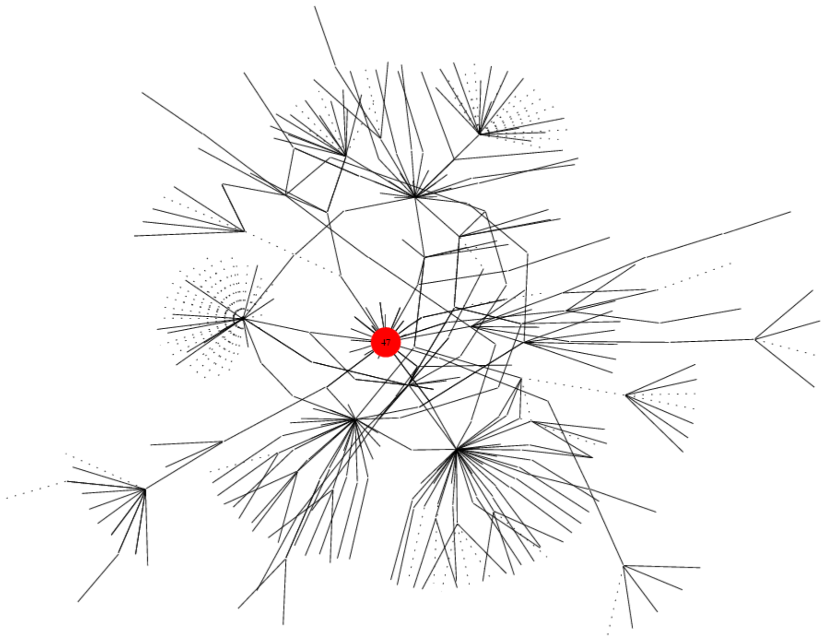


Figure 4.5: Visual representation of a sample power-law graph, after several searches for a single object using the APS method. Solid line arcs show high index value links between nodes

replica is deactivated/deleted from the node's cache without any further communication. Alternative measures such as the maximum number of allowed connections can be used. These limits can be optionally advertised inside the network upon connection or replica activation. If a peer cannot sustain its advertised rates, then it may choose to advertise new maximum capacities. This can potentially assist requesters by hinting them to avoid very high request rates. Nevertheless, it is not required by our approach. Obviously, the total maximum capacity for server s is equal to $\sum_i Limit_{s,i}^{up}$, where i refers to every object that s serves.

In order to identify "hot" areas inside the overlay (i.e., locate set P), we utilize a push phase similar to the one featured in *AGNO*. Paths with large APS-index values connect the requesters to the content providers (Figure 4.5). Peers store *reverse index* values for each of their neighbors. Reverse indices are used to identify (paths to) active

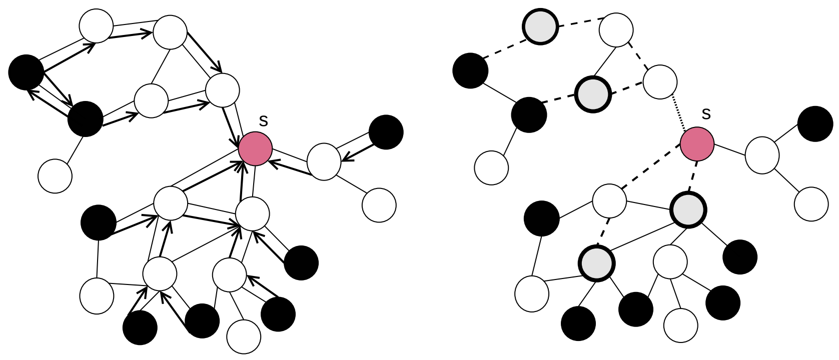


Figure 4.6: After searches for an object at *s* take place, reverse index values are updated and a push phase creates new replicas inside areas of high demand (dotted links)

requesters in the overlay on a per-object basis.

In order to discover candidate new servers to host replicas of *i*, whenever the local load for object *i* (measured in requests per time unit) $\lambda_i^s(t)$ exceeds the limit $Limit_{s,i}^{up}$, the respective server *s* issues a special message which is forwarded to *k* neighbors with the *k* highest reverse index values. The push message contains the tuple $(s, i, size(i), \text{TTL}, D_i(t))$: The overloaded server's ID, the ID and size of the object that caused the overload, the hop distance left and the amount of overload $D_i(t) = \lambda_i^s(t) - Limit_{s,i}^{up}$. Each node that receives this message, independently decides whether to join S_i according to our implemented replication policy. This phase continues with each intermediate node forwarding this message to *k* neighbors in a similar fashion until either its TTL value reaches zero or a duplicate reception is detected. Figure 4.6 shows an example of our scheme at work: Black nodes represent requesters of the item held at node *s*. *APS* searches are depicted by arrows. In the push phase, paths with high index values are visited (links with dotted lines). The new shaded nodes with bold outline represent possible replicas created.

Reverse indices get updated in the same manner as in *AGNO*, while an *aging* factor forces their values to decrease with time. In *AGNO*, push phases are assumed to be less frequent than request rates ($\lambda_r \gg \lambda_{push}$), thus the need for an estimate of λ_r, through which the tolerance parameter T is defined (see Section 3.2.1). However, since our priority is for the system to be as reactive as possible, we assume that servers may initiate *Expand* frequently (checking for overloads on *every* time unit). In this case, the aging scheme can be made simpler by avoiding the estimation of λ_r. Instead, we can set $T \leq 1/\lambda_i^s(t)$ in equation 3.2.1.(1), substituting the estimator with the observed rate at each time step.

Each node on the path independently decides whether it will join S_i according to our replication policy. Currently, we have implemented three: *FurthestFirst*, *ClosestFirst* and *Uniform*. In *FurthestFirst*, the probability of a node joining S_i increases with the message distance, while the opposite occurs in *ClosestFirst*. All nodes are given the same chance in *Uniform*. After subset Y has been identified, replicas are transmitted and activated.

In order for *APRE* to adapt to various workloads and avoid system oscillation [59] due to replicas with perceived load a little above or below the limits frequently entering and leaving $\mathcal{S}_i$, we introduce a *scaled* replication policy: We regulate the number of replicas activated per push phase according to the amount of overload for object i, $D_i(t)$, as observed by the server initiating the push at time t. To achieve that, we define a set of intervals $\{d_1, d_2, \ldots, d_m\}$ that group the different values of D_i. Each interval $d_k : \{(l_k, u_k), \{p_{k_1}, p_{k_2}, \ldots, p_{k_{TTL}}\}\}$ is defined by an upper and lower value and TTL probability values, one for each hop distance. For the interval limits, we require that $l_1 < u_1 = l_2 < u_2 \ldots < u_m$. When a server receives a push message, it joins $\mathcal{S}_i$ with probability p_{k_δ}, if $l_k < D_i \leq u_k$ and the TTL value in the message is δ. Probability values increase as D falls into higher number intervals (i.e., $p_{k_\delta} < p_{(k+1)_\delta}$). Thus, a heavily overloaded server will create more replicas than a less overloaded one and marginally overloaded peers will not alter $\mathcal{S}_i$ significantly. We note here that each server locally estimates $\lambda_i(t)$, the number of requests for object *i* per time unit.

Our experimental evaluation confirms that expanding relative to the size of excess load *D* is necessary to achieve smooth changes in the server-set. Our results also show that peers can further avoid oscillations by monitoring the number of times they joined or left $\mathcal{S}_i$ inside a small window of time. Peers that repeatedly leave and re-join a server set can choose a single state (either host the object or not) for the following τ time steps. If the size of the object and the peer's free space allow it, it is preferable that the node serves object *i* for τ time steps, regardless if $\lambda_i(t) < Limit_i^{down}$.

4.3 Performance Evaluation

We test the effectiveness of *APRE* using a message-level simulator written in C. Requests for object *i* occur at rate λ_i with exponentially distributed inter-arrival times. At each run, we randomly choose a *single* node that plays the role of the initial $\mathcal{M}_i \equiv \mathcal{S}_i$ set and a number of requesters, also uniformly at random. Our experiments involve a single object each time (thus $\lambda_i \equiv \lambda_r$). This is done for two reasons: First, the only dependency that exists between replication of different objects relates to a possible deactivation of an object before the activation of a new replica. In the previous section we described how *APRE* tackles this issue. The more practical reason relates to the amount of memory required to simulate multiple objects for our graphs.

Periodically, 10% of the current requesters stop querying for the object and are replaced by an equal number of other (previously non-requester) nodes. Results are averaged over several tens of runs over sets of 10,000-node *random* and *power-law* graphs with average node degrees around 4 (similar to gnutella snapshots [45]). These are created with the BRITE [46] and Inet-3.0 [33] topology generators.

To evaluate the replication scheme, we utilize the following metrics: The average load Λ which is the number of received requests per time unit averaged over the number of servers $|\mathcal{S}_i|$. Obviously, regarding our load-balancing requirement, we also need to measure the disparity of the load distribution. To that direction, we compute the standard deviation σ_Λ and the *Gini* coefficient [60]. High values for both these metrics indicate that load is unevenly balanced across $\mathcal{S}_i$. Besides the size of the server set, we also keep track of the number of replica activations/de-activations. Frequent changes in $\mathcal{S}_i$ incur huge overheads in terms of messages and bytes transferred.

APRE Parameters: We assume that $(Limit_s^{up}, Limit_s^{down}) = (18,3)$ requests/sec, for each server s. To calculate the decay of the reverse indices, we choose an aggressive value of $T = \frac{0.5}{\lambda_i(t)}$. Different values of $T \leq 1$ sec produce similar results. During the *Expand* push phase, peers forward to the two neighbors with the largest reverse index values. Servers check whether to initiate *Expand* and *Contract* every time unit for fast response. We assume no item can be replicated at more than 40% of the network nodes (maximum replication ratio). This external condition simulates the natural limitations in storage that exist in most systems. We present experiments that show our method's performance by altering this ratio.

We utilize a scheme with 3 distinct intervals for values of D: $[0,5],(5,20]$ and $(20,\infty)$. The results did not vary considerably for schemes with more intervals. As the intervals get fewer, *APRE* becomes less responsive to D, with the number of created replicas having smaller variation. This affects the performance of the replication during the warm-up state as well as during sudden surges in requests. The chosen configuration works well and is used in the entirety of our simulations. For the first interval and *Closest-First*, we use probabilities $p_1 = \{.12, .06, .03, .02, .01\}, p_2 = \{.22, .10, .06, .04, .02\}, p_3 = \{.35, .18, .10, .07, .04\}$. These are reversed when we use *FurthestFirst*. *Uniform* uses a $(.05, .08, .15)$ probability for each of the 3 intervals. Thus, we roughly double the p_{k_δ} value from one interval to the next and halve it from one hop to the next in the same interval. For *Uniform*, we roughly select the average of the p_{k_δ}'s of the other methods as the p_k value. Increasing these probabilities causes more objects to be created, often reaching the maximum replica count, while much smaller values delay the responsiveness of *AGNO* (in high-demand settings). Increasing the difference between p_{k_δ} and $p_{k_{\delta+1}}$ changes the ratio of replicas created at different distances, while decreasing it produces an effect similar to *Uniform*. While we experimented with different configurations, our results are based on the described one.

We compare *APRE* against the following methods: In the *random replication* scheme (hence *Random*), we randomly create the same number of replicas as our method in its steady state at the start of the simulation. In path replication (hence *path-cache*), each time a server is overloaded we replicate the object along the reverse path to the requester. This is similar to the replication applied by Freenet [61]. In all cases, the *APS* method is used for lookups, while in *path-cache* replica deactivation occurs using our *Contract* scheme. Obviously, by varying the push method and the replication probabilities, *APRE* can behave either as *path-cache, Random* or in between, with a variable rate of replica creation per workload. This allows for full customization according to the system's primary

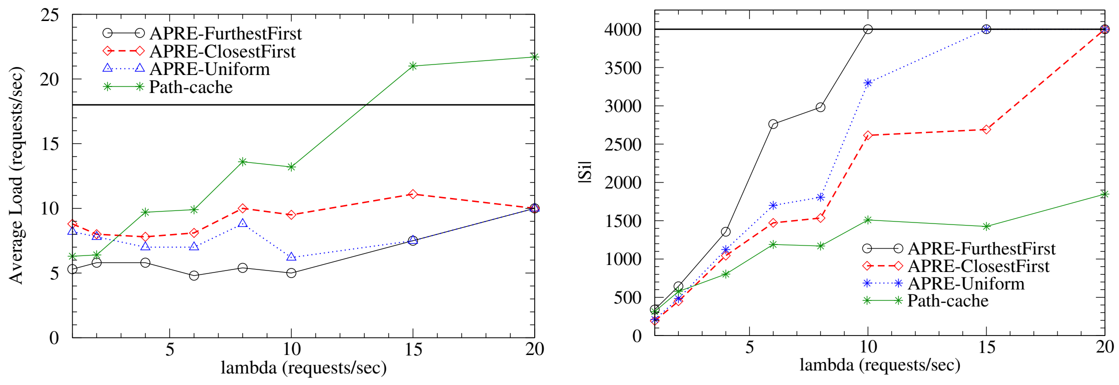


Figure 4.7: Variation in Λ and $|\mathcal{S}_i|$ over increasing λ_r values

objects, namely low load (more replicas) or space (replicas only where necessary).

4.3.1 Basic Performance Comparison

For our default setting, we assume 2000 requesters and vary their λ_r. The results are presented in figure 4.7.

APRE effectively expands $\mathcal{S}_i$ in order to accommodate the increased demand and manages to keep the average load into the “Normal Operation” zone, well below $Limit_s^{up}$ (identified by the bold horizontal line). Our first observation is that *FurthestFirst* achieves lower Λ values by creating more replicas than *ClosestFirst*. The paths traversed during the push phase contact an increasing number of nodes as their distance from the initiator increases, thus giving *FurthestFirst* an increased probability of replication. *Uniform* behaves in-between, creating replicas equally at all distances. *Path-cache* exhibits a steeper increase in Λ and fails to keep its value within the acceptable region for large λ_r. Choosing only single successful paths to replicate along prevents the algorithm from doing further replication. Increased demand merely forces the algorithm to utilize a few more paths, which is the reason why this method fails to increase the replica set to meet the limits.

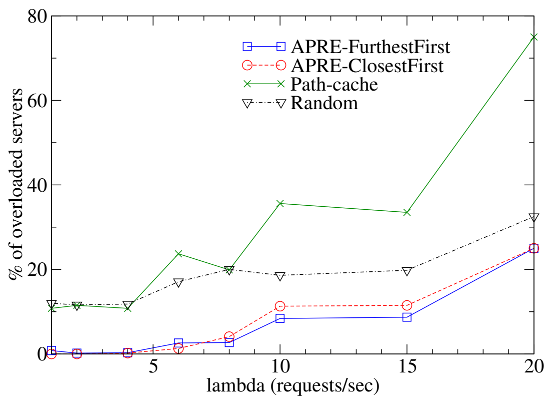


Figure 4.8: Ratio of overloaded servers vs. variable λ_r

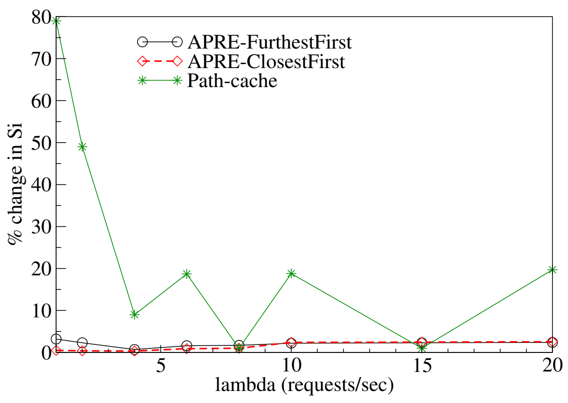


Figure 4.9: Percentage of change in $|\mathcal{S}_i|$ vs. variable λ_r

Figure 4.8 displays the average percentage of overloaded servers at any time for all three methods. Our technique clearly outperforms the two competing methods: For $\lambda_r < 10/sec$, less than 4% of servers are overloaded, while about 10% and 25% are documented as overloaded for the largest demand. *Random*, having the same number of servers, exhibits twice as many overloaded nodes. Even though the learning feature of *APS* helps in redirecting queries, yet the load cannot be evenly distributed. *Path-cache* shows the worst performance (at least 3 times larger ratio of overloaded peers than *APRE*), reaching 75% at the highest λ_r value. Replicating closer to requesters creates, as we saw, more service points, thus marginally reducing the number of overloaded instances for *FurthestFirst* (*Uniform* exhibits the same curve).

Moreover, we show that *APRE* achieves a much more robust replication. The stability of the server population constitutes an important metric to the evaluation of a replication scheme. This is measured by the average ratio of new replicas entering the server set per replication phase over the size of the server set. This quantity approximates the amount of marginally under-utilized replicas in the overlay: Receiving few requests, they

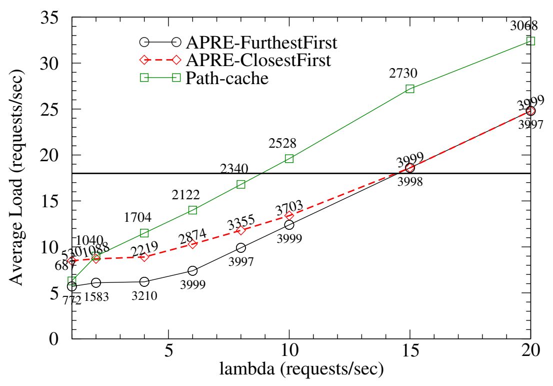


Figure 4.10: Variation in the average load vs. variable λ_r (5000 requesters)

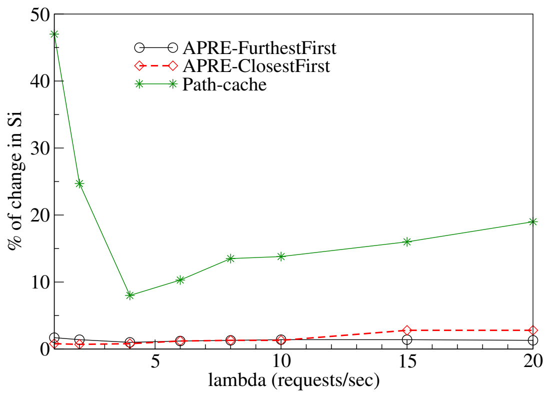


Figure 4.11: Percentage of change in $|\mathcal{S}_i|$ vs. variable λ_r (5000 requesters)

get deactivated. Server overloads force them to get re-activated, producing an oscillating effect. Obviously, this is a highly undesirable situation: network and local resources are burdened by a multiplicative factor, since replicas need both control messages and data transfer for reactivation. Figure 4.9 shows that *APRE* is particularly robust, altering at most 3% of $\mathcal{S}_i$ per push phase, while *Path-cache* oscillates and performs poorly in most runs. altering a large percentage of the server set. The variability in the amount of oscillation is due to the effect we described before: An increase in the demand is not always followed by an increase in the number of replicas. In these situations, the existing ones receive the extra amount of requests (assisted by the *APS* scheme), thus reducing the marginally idle servers.

The same experiment is repeated with 5,000 requesters, which constitute 50% of the overlay (see Figure 4.10 where we annotate the respective $|\mathcal{S}_i|$ value over each point). *APRE* again manages to keep the system within its limits, except for the two cases where even the largest replica set cannot achieve that (75k and 100k total queries per second). Our method documents its largest ratio of overloaded servers in those two settings (30%

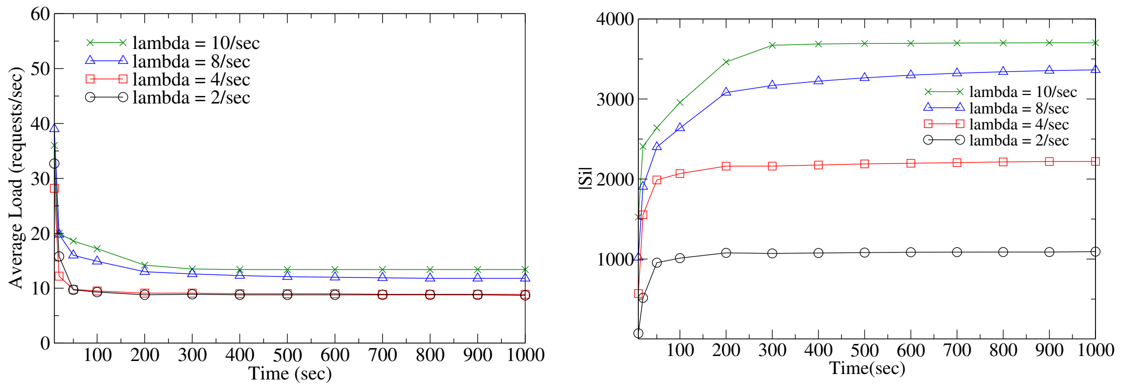


Figure 4.12: Λ and $|\mathcal{S}_i|$ over time for 5000 requesters and multiple λ_r values

and 60% respectively). Figure 4.11 shows the amount of change in the server sets of the two methods. *APRE* is much more stable in the $\mathcal{S}_i$ population for both strategies, while *Path-cache* deteriorates compared to the results for 2000 requesters.

Finally, Figure 4.12 shows how Λ and $|\mathcal{S}_i|$ vary with time, using *ClosestFirst*. For all values of λ_r, *APRE* manages to bring Λ to a steady state within few time steps, a state which is hence maintained with almost no deviation. The same is true for the size of $\mathcal{S}_i$, with the exception that for high total demand, it takes longer to reach the steady state. This is due to the fact that there is a limit to the maximum amount of replication per push phase for our method (as there is for *path-cache*) that causes the delay in reaching the constant values.

Table 4.1 summarizes our observations for this setup by documenting the performance of the three schemes for a variety of metrics. *APRE* manages to keep bandwidth consumption steadily low in all runs: The number of push messages during *Expand* remains constant, while replicating inside query-intensive areas allows for an active reduction to the average distance between requesters and servers. The *Random* method shows an increased average distance to the objects compared to *APRE* and *path-cache* that repli-

Table 4.1: Performance Comparison under a variety of metrics (5000 Requesters)

Method		$\lambda_r = 1/sec$	$\lambda_r = 4/sec$	$\lambda_r = 8/sec$	$\lambda_r = 20/sec$
APRE (ClosestFirst)	Search Mesg	9.6	7.8	7.8	7.8
	Push Mesg	41.7	39.7	39.2	39.4
	Hit Distance	3.4	2.2	2.0	1.9
	% Overloaded servers	1.6	6.8	20.2	60.8
Path-cache	Search Mesg	10.2	8.4	7.9	7.7
	Hit Distance	3.6	2.5	2.2	2.0
	% Overloaded servers	11.1	23.5	37.8	94.3
Random	Search Mesg	10.4	10.3	9.4	9.0
	Hit Distance	3.7	3.0	2.5	2.2
	% Overloaded servers	8.9	15.0	23.0	54.4

cate along search paths. Our method exhibits a far smaller percentage of overloaded servers compared to *path-cache* but it is comparable to *Random* for the largest value of λ_r. This happens because *APRE* has reached the maximum replication ratio by exhausting all possible paths where requests are coming from in the overlay.

4.3.2 Load Distribution Between Replicas

While a low number of overloaded servers and a low Λ value are important, we have not yet investigated how load is distributed among the replicas. Obviously, balanced distributions are preferred to those showing a high degree of variation. Various metrics that quantify the degree of disparity of a number or measurements have been proposed. We investigate two of them, namely the *standard deviation* and the *Gini coefficient*.

Returning to our 2000 requesters experiment, Figure 4.13 compares the standard deviation of Λ for the three methods. We note that *APRE* exhibits small σ_Λ values, rang-

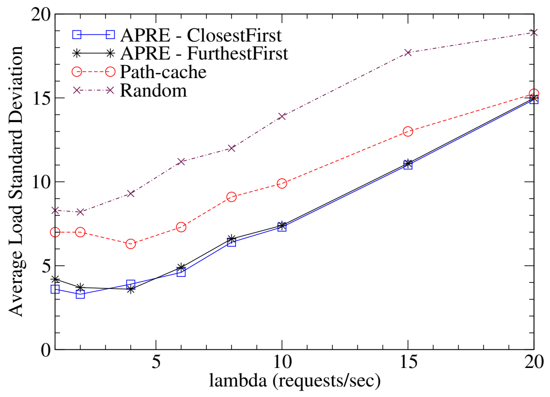


Figure 4.13: Variation of σ_Λ vs. variable λ_r (2000 requesters)

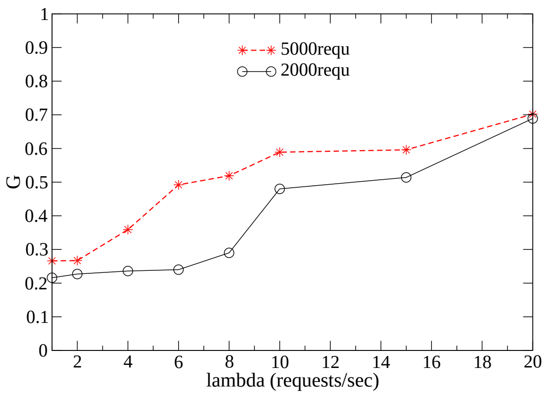


Figure 4.14: Average values of $\mathcal{G}$ as a function of λ_r for the *ClosestFirst* strategy

ing from 3.3 to 11. It increases to 14.9 only when $\lambda_r = 20/sec$. These values are either smaller or at most comparable to Λ, a good indication of load balancing. On the other hand, randomly placing the same number of replicas yields significantly worse load distributions, with σ_Λ values roughly twice as large. This is a clear indication of the need for correct placement inside structureless multi-path overlays. Finally, *path-cache* behaves in-between, with larger deviation values than *APRE* that converge as load increases. This happens since both methods base their replication on paths connecting requesters and servers. Our method utilizes *multiple* paths that consistently carry requests, thus it outperforms *path-cache* in almost every setting.

The *Gini coefficient* (or Gini ratio) $\mathcal{G}$ is a summary statistic that serves as a measure of inequality in a population [60]. The Gini coefficient is calculated as the sum of the differences between every possible pair of individuals, divided by the mean size:

$$\mathcal{G} = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} |x_i - x_j|}{2n^2\mu},$$

where n is the number of observations whose values are given by x_i, and $\mu = \frac{\sum_{i=1}^{n} x_i}{n}$ is their mean. The Gini coefficient has been used as a measure of inequality in size and fecundity in plant populations in numerous studies (e.g., Weiner 1985, Geber 1989, Knox et al. 1989, Preston 1998). Its value ranges between 0 and 1, where 0 corresponds to perfect equality and 1 corresponds to the theoretic case of an infinite population with only one individual having a non-zero value. Recent work [62] proposed its use as a load-balancing metric. Assuming our population comprises of the number of received requests by each replica, we calculate the value of $\mathcal{G}$ as an index of load distribution among servers. Note here that a low value of $\mathcal{G}$ is a strong indication that load is equally distributed among them, but does not necessarily imply that this load is low.

Figure 4.14 shows the average values of $\mathcal{G}$ for all different values of request rates in both settings (2k and 5k requesters). In low-load runs, servers show very similar loads. As the total load increases (either through an increase in λ_r or the requester population), so does the inequality between the received requests. The authors in [62] identify that $\mathcal{G} < 0.5$ presents very well-balanced configurations, while when $\mathcal{G}$ is approximately between 0.5 and 0.65, relatively fair distributions are achieved.

Our scheme, while not explicitly providing any mechanism to balance load inside $\mathcal{S}_i$, manages to provide very well-balanced configurations for medium to low loads and fair ones for medium to high loads. The reason for that is because the push phase (thus, by extension, the creation of new replicas) operates symmetrically on multiple (k) high-quality paths. So, neither the originating server gets starved of requests, nor the newly established ones differ substantially in their positioning. Only when $|\mathcal{S}_i|$ approaches our artificial limit of 4000 peers we notice that load allotment gets uneven.

To visualize and confirm these findings, Figures 4.15, 4.16 plot the load distribution

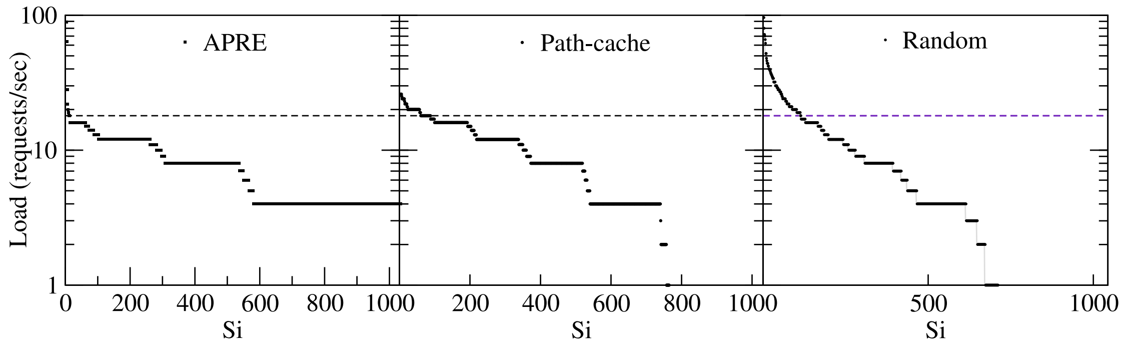


Figure 4.15: $\mathcal{S}_i$ load distribution for $\lambda_r = 4/sec$ (2000 requesters)

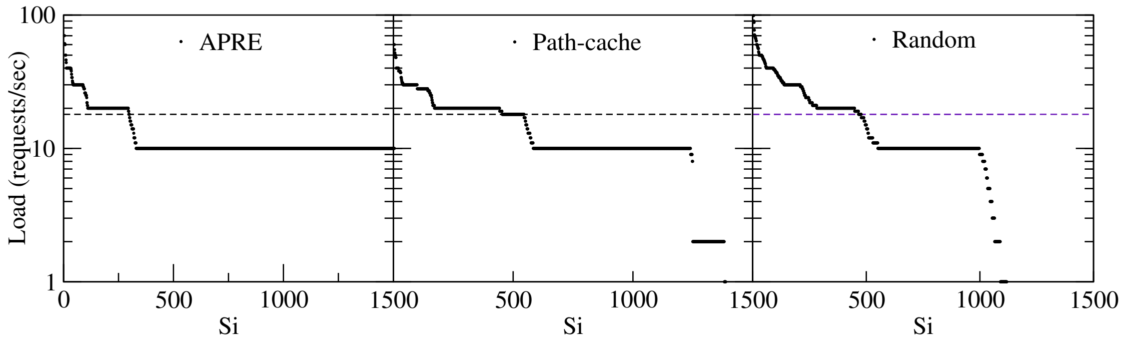


Figure 4.16: $\mathcal{S}_i$ load distribution for $\lambda_r = 10/sec$ (2000 requesters)

of the server set at a random point in time for $\lambda_r = 4/sec$ and $\lambda_r = 10/sec$. Servers are sorted in decreasing order of load. First, we notice that in both cases *APRE* has less servers above $Limit^{up}$ (the dotted line). Our method exhibits a less steep curve, with fewer groups of replicas with similar loads. *Path-cache* shows more unbalanced load and a larger number of servers below $Limit^{down}$. Random replication causes even more unbalanced load in all runs.

Thus far we established our basic premise, that replication along high demand paths in the overlay proves an effective and highly robust solution in a variety of metrics and workloads. Although our method does not explicitly offer load-balancing, it achieves a well-proportionate load distribution. We also showed that our method is advantageous to randomly replicating inside the network or merely choosing a single path and fully replicating along it. In the first case, few replicas receive the majority of requests, while in the second case the composition of the replica sets changes very frequently. Our method outperforms both alternatives by keeping fewer peers over the sharing limit and showing less disparity in the distribution of load among servers.

4.3.3 Flash Crowds

In the next set of simulations, we examine the behavior of our method when we experience a sudden surge in the workload. This is often referred to as a *flash crowd*, an unexpected rise in requests towards specific content, typically due to some newsworthy event that just took place. Flash crowds have been regularly documented in web traffic history (e.g., September 11th) and are known to cause severe congestion at the network layer. Requests may never reach the servers, while others do so with significant delays caused by packet loss and retransmission attempts. Content holders are unable to handle the volume of requests, while end-users experience long delays and failures in their queries.

To simulate this situation, we initiate our system with 500 requesters querying at rate $\lambda_r = 2/sec$. At time t=401sec, 10 times as many requesters start querying for this item at rate $\lambda_r = 12/sec$. The parameters return to their initial values at time t=601sec. On average, the total demand during the flash-crowd period increases by a factor of over 70. Note that this is the worst case scenario, when simultaneously both requesters and rates increase. We present the variations in Λ and $|\mathcal{S}_i|$ in the first 2 graphs of Figure 4.17.

APRE promptly manages to meet the surge in requests by increasing the replication

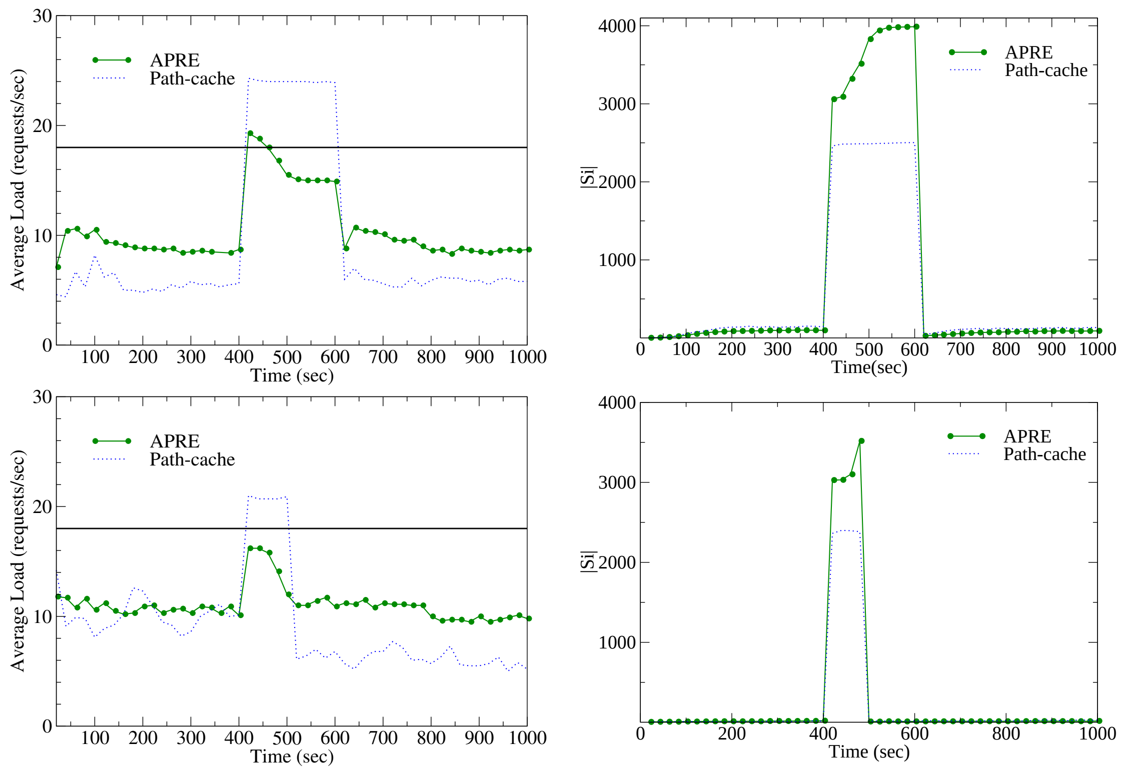


Figure 4.17: Effect of flash crowds in Λ and $|\mathcal{S}_i|$ in two different settings

ratio by a factor of 30. Excluding a very short window due to our mechanism's response, our method succeeds in keeping the load factor below the limit (with $\sigma_\Lambda < 10$) and steady through time. At both times that load changes, replicas are activated and de-activated rapidly to meet the extra requests and reduced traffic. While *path-cache* shows similar response speed, it creates more servers in the low-workload period and less than the minimum number required to keep content providers from overloading during the surge.

The bottom two figures show how the same two metrics vary in a more challenging flash-crowd setting. Here, we initially have 500 requesters with $\lambda_r = 1/sec$, while for time $t \in (400, 480]$ we set $\lambda_r = 10/sec$ for 5000 requesters. On average, the workload inside the overlay increases by a factor of 120. Our results show that, even for shorter and steeper changes, *APRE* very successfully adapts to the surge in requests. On average, $\mathcal{S}_i$

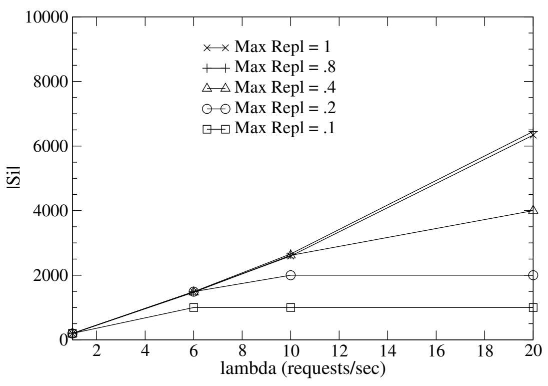


Figure 4.18: $|\mathcal{S}_i|$ variation for different maximum allowed replication ratio

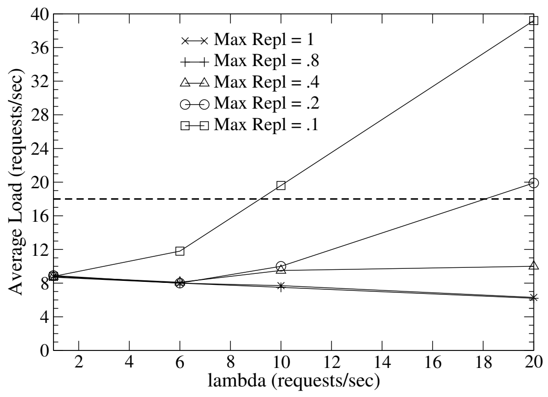


Figure 4.19: Λ variation for different maximum allowed replication ratio

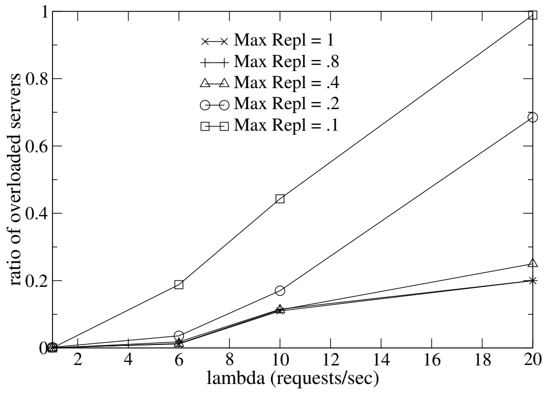


Figure 4.20: Percentage of overloaded nodes for different maximum allowed replication ratio

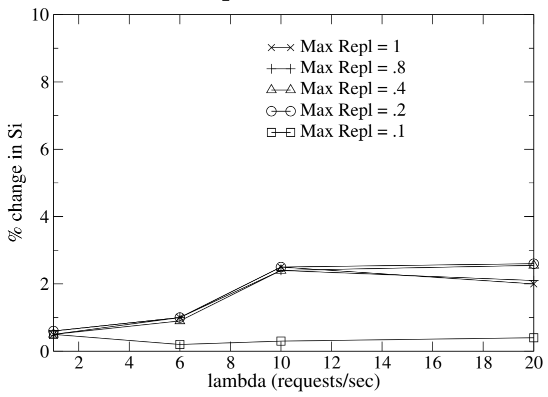


Figure 4.21: Percentage of change in $\mathcal{S}_i$ for different maximum allowed replication ratio

is expanded by a factor of 175 in order to reduce and balance load (our results document an average $\sigma_\Lambda \simeq 8.6$).

4.3.4 Effect of the Maximum Replication Ratio and *Limit*^{up}

Our default scenarios assumed a set value of $0.4N$ for the maximum allowed $|\mathcal{S}_i|$ (where $N = 10k$ = size of our overlay). In the next figures, we plot the performance of *APRE* (using *ClosestFirst* and 2000 requesters) while we vary the maximum replication ratio from 0.1 to 1.0. Too small values should force the system to quickly saturate, while

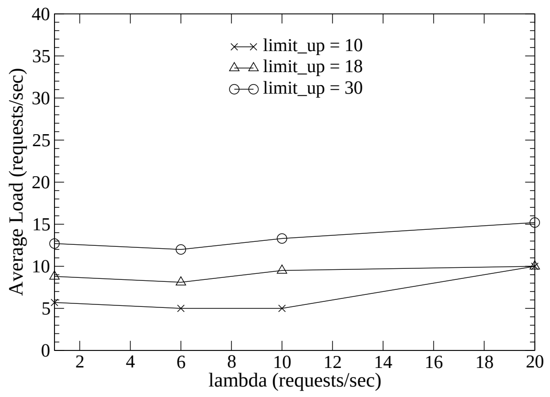


Figure 4.22: Λ variation for different values of $Limit_{s,i}^{up}$

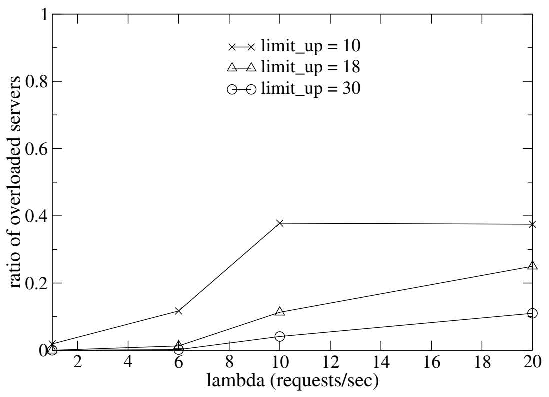


Figure 4.23: Percentage of overloaded nodes for different values of $Limit_{s,i}^{up}$

complete freedom to replicate should exhibit the best behavior. The results are presented in Figures 4.18, 4.19, 4.20 and 4.21.

When the ratio is too small (at most 1000 nodes are allowed to host the object), $\mathcal{S}_i$ quickly reaches this limit. This affects the values of Λ as well as the number of servers over $Limit^{up}$, which rapidly increase. As more peers are allowed to become servers, $|\mathcal{S}_i|$ increases and so does the percentage of servers below the limit. The interesting observation here is that even for the most optimistic case (no replication restriction), *APRE* manages to keep the ratio of change in the server-set below 3%.

In the next experiment, we vary the maximum advertised capacity $Limit^{up}$ uniformly. Sample results are presented in Figures 4.22 and 4.23. As we would expect, the smaller the upper limit gets, the faster our algorithm reacts to load, thus creating replicas sooner. Obviously, given some storage restriction (such as a maximum allowed number of peers able to enter $\mathcal{S}_i$), small $Limit^{up}$ values cause system saturation and more overloaded instances. On the other hand, for larger upper limits, the server-set increases more gracefully and significantly fewer server overloads are observed.

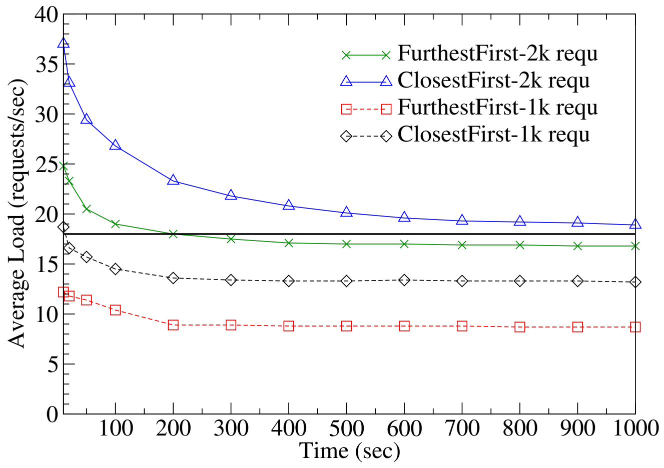


Figure 4.24: Average load for 1k and 2k requesters in power-law topologies ($\lambda_r = 6/sec$)

4.3.5 Simulations with Different Topologies

We tested our method on a set of 4,000-node power-law graphs created with the Inet-3.0 generator [33]. These graphs have an average degree of $d = 4.3$ (maximum degree equals to 855), while over 30% of the nodes have only one neighbor. Figure 4.24 shows how Λ varies with time for *ClosestFirst* and *FurthestFirst* using 1000 or 2000 nodes as requesters.

These topologies noticeably affect performance compared to our previous simulations. Even for average-range λ_r values, Λ moves close to the overload line, while Expand shows diminished ability to extend $\mathcal{S}_i$. This is consistent with results documented in previous work [15]. The tested topologies offer fewer paths between servers and clients, while a large percentage of the nodes only have one neighbor. This also explains why *FurthestFirst* outperforms *ClosestFirst*. Favoring replication close to the requesters quickly saturates available nodes due to lack of alternate paths. Nevertheless, its is worth noticing that our method still manages to keep Λ at lower levels. Even at the 2k-*ClosestFirst* run, where $\Lambda > Limit^{up}$, 14% of the servers are overloaded compared to 20% by *path-cache*.

We must note here that the replication protocol is not always responsible for overloaded servers. In many occasions, the amount of demand or the overlay connectivity cannot allow for more extensive or balanced replication. As we experiment with more densely connected graphs, *APRE* performs inside the load limits where it failed to do so over more sparse overlays. Moreover, with biased forwarding, as happens with most informed approaches, certain nodes will unavoidably receive a bulk of requests. This situation can only be corrected through maintaining additional state at each peer (such as the location of other servers) and changing the forwarding scheme. In an environment with rapid changes in workload and server sets, this locally maintained metadata can become frequently stale, thus incurring larger communication (besides the local storage) overhead. Our approach, on the other hand, does not require a change in *APS*, but relies on its ability to independently create and store that state through new object discoveries and reverse index built-up.

4.4 Conclusions

In this part of our work we presented our adaptive replication scheme for unstructured Peer-to-Peer systems based on probabilistic soft state. *APRE* aims at providing a direct response to workload changes, by creating server points in needy areas or releasing redundant servers in areas of low demand. Our approach couples lookup indices together with an aging mechanism in order to identify, in real-time, query intensive areas inside the overlay. Peers then individually decide on the time and extent of replication, based on local workload computation.

Our work shows that it is important to couple replication with the search protocol in unstructured systems. Random replication performs poorly with informed lookup schemes, unless extra state is added to enhance searches. Applying *APRE* over a scheme such as *APS* solves this problem. *APS*-indices store local, per-object state to direct queries to objects. While peers only keep metadata about their neighbors, this information can be used to identify, hop-by-hop, where the queries are coming from. Moreover, our scheme is highly customizable, allowing control of both the size and the location (as defined through reverse-indices) of replication.

Using thorough simulations, we show that *APRE* is extremely robust in eliminating server overloads, while minimizing the communication overhead and balancing the load. Specifically, we show that replicating along the reverse path is an extreme case of our protocol. By effectively discovering all reverse paths, *APRE* manages to distribute content proportional to demand in a variety of overlays and workloads. Finally, we show that our method succeeds in creating a very stable server set with minimal amount of oscillation.

Chapter 5

Related Work

Peer-to-Peer networks have been studied a lot in the last few years. A large amount of information for P2P computing with taxonomies, definitions, current trends, applications and related companies can be obtained at [63, 64], as well as individual sources (e.g., [7,8]). P2P computing is also described in [65], with basic terminology, taxonomies and description of some systems. A brief summarization of *Gnutella* [6] and *Napster* [5], together with approaches for structured networks are also included. Gnutella and Napster are the focus of two measurement studies: Reference [66] attempts a detailed characterization of the participating end-hosts, while the work in [21] measures the locality of stored and transferred documents. In [67], a traffic measurement for three popular P2P networks is being conducted at the border routers of a large ISP. Extensive results for traffic attributed to HTTP, Akamai and P2P systems are also presented in [68].

In this part of our dissertation, we present work related to each of our contributions: Search algorithms, group communication schemes and replication methods for unstructured P2P networks.

5.1 Search Algorithms for P2P Systems

As part of this thesis, we present a thorough description of many representative search algorithms for unstructured P2P networks. We first describe blind search algorithms and proceed with several informed techniques. Appendix A presents a direct experimental comparison between many of these schemes.

5.1.1 Blind Search Methods

GNUTELLA [6]: The original Gnutella algorithm (or *flooding* scheme) contacts all accessible nodes within TTL hops. Its basic characteristics are its simplicity and the huge overhead it produces by contacting many nodes (and possibly multiple times each).

Modified-BFS [22]: In this variation of the flooding scheme, peers randomly choose only a ratio of their neighbors to forward the query to. This reduces the average message production, but still contacts a large number of peers.

Iterative Deepening: Two similar approaches that use consecutive BFS searches at increasing depths are described in [14, 69]. These algorithms achieve best results when the search termination condition relates to a user-defined number of hits and it is possible that searching at small depths will satisfy the query. In a different case, they produce even bigger loads than the standard flooding mechanism.

5.1.2 Informed Search Methods

Super-Peer approaches: In *Gnutella2 (G2)* [70], when a super-peer (or *hub*) receives a query from a leaf, it forwards it to its relevant leaves and also to its neighboring hubs. These hubs process the query locally and forward it to their relevant leaves. No other nodes are visited with this algorithm. Neighboring hubs regularly exchange local repository tables to filter out unnecessary traffic.

Both *G2* and *GUESS* [27] rely on a dynamic hierarchical structure of the network. They present similar solutions for reducing the effects of flooding by utilizing the structure of hybrid networks. The number of leaf-nodes per super-peer must be kept high, even after node arrivals/departures. This is the most important condition in order to reduce message forwarding and increase the number of discovered objects.

Intelligent-BFS [22]: This is an informed version of *modified-BFS*. Nodes store (query, neighborID) tuples for recently answered requests from (or through) their neighbors in order to rank them. First, a peer identifies all queries similar to the current one, according to a query similarity metric; it then chooses to forward to a set number of its neighbors that have returned the most results for these queries. If a hit occurs, the query takes the reverse path to the requester and updates local indices. This approach focuses more on object discovery than message reduction. At the cost of an increased message production compared to *modified-BFS* (because of the update process), the algorithm increases the number of hits. It achieves high accuracy, enables knowledge sharing and induces no overhead during node arrivals/departures. On the other hand, its message production is very large and only increases with time as knowledge is spread over the nodes. It shows no easy adaptation to object deletions or peer departures, because the algorithm does not utilize negative feedback and forwarding is based on ranking. Finally, its accuracy depends highly on the assumption that nodes specialize in certain documents.

Local Indices (LI) [69]: Each node indexes the objects stored at every peer within a certain radius *r* and can answer queries on behalf of all of them. A search is performed in a BFS-like manner, but only nodes accessible from the requester at certain depths process the query. To minimize the overhead, the hop-distance between two consecutive depths must be $2r+1$. This approach resembles the two search schemes for hybrid networks. The method's accuracy and hits are very high, due to the indexing scheme. On the other hand, message production is comparable to flooding, even if the processing time is smaller because many nodes just forward the query. The scheme requires a flood with TTL = r whenever a node joins/leaves the network or updates its local repository, so the overhead becomes even larger for dynamic environments.

GIA [28]: In *GIA*, requesting nodes deploy biased walkers in order to discover various objects. Each peer chooses to forward the query to the neighbor with the highest announced *capacity*. This is a user-defined metric that reflects the processing power of a node inside the system. Moreover, the protocol requires that each peer indexes the documents of its neighbors. This scheme also utilizes a topology-adaptation algorithm which re-configures the overlay connectivity such that each node is connected to a number of peers proportional to its capacity. The biased walkers are then directed towards highly connected neighbors and, probabilistically, to those with the highest number of indexed objects. Finally, the scheme provides a flow-control mechanism which allows peers to control the rate at which they can accept and process requests from their neighbors. Once the topology has been set, we expect *GIA* to perform very bandwidth-efficient searches with several hits. On the other hand, the adaptation algorithm plus the indexing of the neighbors' repositories increase the responsibilities of each peer as well as the communication overhead. Another issue is how fast can the algorithm work for joining peers and at what cost for their neighborhood.

Routing Indices (RI) [26]: Documents are assumed to fall into a number of thematic categories. Each node stores an approximate number of documents from every category that can be retrieved through each outgoing link (i.e., not only from that neighbor but from all nodes accessible from it). The forwarding process is similar to DFS: A node that cannot satisfy the query stop condition with its local repository will forward it to the neighbor with the highest "goodness" value. Three different functions which rank the out-links according to the expected number of documents discovered through them are also defined. The algorithm backtracks if more results are needed. This approach trades index maintenance overhead for increased accuracy. While a search is very bandwidth-efficient, RIs require flooding in order to be created and updated, so the method is not suitable for highly dynamic networks. Moreover, stored indices can be inaccurate due to thematic correlations, errors in the categorization of documents and network cycles.

In [23], each node holds a number of *bloom* filters for each neighbor. The i^{th} filter summarizes documents that can be found i hops away through that specific link. Nodes forward queries to the neighbor whose smaller depth bloom filter matches a hashed representation of the object ID. After a certain number of steps, if the search is unsuccessful, it is handled by a deterministic algorithm instead of backtracking. The scheme's expectation is to find only one replica of the object with high probability. Index maintenance requires flooding messages initiated from nodes that arrive or update their collections.

Distributed Resource Location Protocol (DRLP) [24]: Nodes with no information about the location of a document forward the query to each of their neighbors with a certain probability. If an object is found, the query takes the reverse path to the requester, storing the document location at those nodes. In subsequent requests, nodes with indexed location information directly contact the specific node. If that node does not currently obtain the document, it just initiates a new search as described before. This algorithm initially utilizes flooding to find the locations of an object. In subsequent requests, it might use a single message to discover it. A low message production is achieved only with a large workload that enables the initial cost to be amortized over many searches. In rapidly changing networks, this approach fails and more nodes have to perform blind search. This also affects the number of hits: If many blind searches are made, then many results are found; if many direct queries take place, then only one replica is discovered.

Gnutella with Shortcuts (GS) [25]: In this work, the authors propose the addition of *shortcuts* (i.e., direct links to peers that have recently proved useful in answering queries) to a Gnutella-like overlay. The original flooding mechanism is initially used to locate documents. Peers that provide answers are indexed by the requesters, following the assumption that they could provide answers to more requests. When a new query is made, nodes first forward it to their shortcuts (ranked in a descending order of usefulness — usually the success rates). If all shortcuts fail, the standard flooding scheme is again used to locate the object. This approach resembles the *DRLP* scheme but stores more than one pointer and keeps statistics on them. For semantically related queries, we expect it to quickly identify relevant peers and mostly use the shortcuts for object location. Moreover, we anticipate a very high success rate since the fall-back mechanism is flooding. On the other hand, if peers make many unrelated queries or they do not store relevant content, it is possible that the shortcuts will fail, which in turn means that the system pays the price with a full-scale flooding. The same is true when objects are removed or peers depart frequently.

New Approaches: Recently, there has been an effort to combine the advantages of structured systems (DHTs) and unstructured ones. In [71], an *immediate neighborhood* area is defined for each peer. Object placement inside these overlapping areas is performed in a DHT-like fashion. Searches use the standard flooding mechanism except that only certain areas are probed. In [72], peers are grouped into *possession rules,* according to whether they contain a specific item or not. Nodes search inside one possession rule in a blind fashion. The possession rule is chosen by a greedy mechanism according to past query results. Finally, the work in [73] combines random walks in unstructured overlays with DHT-like replica placement: The owner of each object places replicas of the object on several nodes. The replicas are assigned to nodes which have IDs numerically close to the object. During a search, random walks are used to locate several *minima* for a given object (i.e., nodes inside a neighborhood that have the closest ID to the object).

Finally, we describe a family of algorithms which are based on traditional reinforcement learning and inspired by dynamics observed in biological colonies. Several algorithms have been proposed to mimic the collective foraging behavior of ants that self-organize in order to locate and transfer food back to the nest in an almost-optimal manner. They are known as *ant-based algorithms* [30]. The problem of routing data packets in dynamic communication networks has characteristics well-suited for ant-based solutions. Indeed, a variety of schemes based on mobile agents (or *ants*) have been proposed in order to discover shortest routes between any pair of nodes in data networks (e.g., [74, 75]). These schemes utilize some ideas similar to our probabilistic *APS* walkers.

The main characteristics of ant-based routing (as seen in *AntNET*) [75] can be summarized as follows:

- Each node holds probability values per neighbor per destination. These values are used to guide the ants. Moreover, it holds statistics for network traffic as seen locally.

- At regular intervals, *forward* ants are launched from nodes to randomly selected destinations. Forward ants keep memory of the visited nodes and traffic characteristics during their route.

- If the destination is reached, the agent creates a *backward* ant that travels along the reverse path and updates probability values and local traffic measurements. Specifically, the probability of taking the successful path is increased using the standard reinforcement-learning rule: $P \leftarrow P + r(1 - P)$.

These algorithms feature a plethora of tunable parameters that actively affect performance: The rate at which forward ants are created, the reinforcement learning parameter r (should depend on the roundtrip time and the local traffic measurements), the probability of exploration versus exploitation, etc. *APS* differs from such schemes as it updates probabilities on both success and failure with respect to message minimization. Second, it deploys multiple walks thus actively exploring and exploiting at the same time, while it requires neither a regular query dissemination nor the calibration of many parameters.

5.2 Data Dissemination

The problem of distributing content to multiple hosts is well-studied. We categorize existing methods into general application-layer multicast protocols, multicast for structured P2P overlays and, finally, approaches for unstructured networks.

5.2.1 Application-layer Multicast

Proposed approaches roughly fall into three categories: The mesh-first category (e.g., *Narada* [76]), where nodes form a random mesh between them and then compute unicast paths for each pair of members. This approach requires control overhead quadratic to the group size with refresh messages. In the tree-first approach (e.g., *Yoid* [77]), peers directly form a data delivery tree and also maintain a few extra links to exchange control messages. Finally, in the implicit approach (e.g., *NICE* [38]), both control and delivery structures are implicitly defined by the underlying routing protocol. For example, *NICE* arranges members into a hierarchy of layers and clusters and defines processes for member arrival/departure and cluster merge/split. All these approaches require the existence of a designated host to initiate the membership process, periodic exchange of control messages and also significant overhead for member joins/leaves.

5.2.2 Multicast over P2P Overlays

The algorithm described in [40] describes a broadcast mechanism that operates over *CAN* [13]. Nodes forward to their neighbors in the d-dimensional space, as this is defined in CAN. There are also provisions made to eliminate duplicate messages and prevent looping of the packets around the coordinate space.

Scribe [39] is implemented on *Pastry* [10]. Interested hosts route their requests towards the node responsible for the group's key (the root). Each node on the path checks if it is a current member of the group. If this is the case, it registers the source node as its child in the multicast tree and stops the forwarding process. Otherwise, it stores the

ID of the source and makes a join request towards the root. Scribe is a decentralized and scalable protocol that takes advantage of the overlay structure to produce a balanced delivery tree.

Bayeux [34] is implemented on *Tapestry* [11]. The difference with Scribe is that join/leave operations go through the root of the tree, making it less scalable. *Overcast* [35] also requires coordination with the root node, while it builds its multicast tree in a manner similar to Yoid. The work in [78] contains thorough descriptions and performance comparisons for representative schemes from this category.

5.2.3 Group Communication in Unstructured Overlays

Many search schemes for unstructured P2P networks have been proposed that implement flooding or its modifications in order to contact large numbers of nodes. Examples include the gnutella flooding algorithm [6], the modified-BFS scheme [22], the iterative deepening method [69], etc. All these techniques produce a large number of messages, cannot adapt to variable group sizes and use blind forwarding, which results in many non-members receiving the message.

An alternative solution to the problem is presented by a variety of gossip algorithms, where each member is responsible for forwarding a notification to a randomly selected subset of the group. These approaches have been used in a variety of different scenarios (e.g., distributed databases [79], publish-subscribe systems [80]) and have proved to be a robust solution in the face of member/network failures at the cost of inducing extra traffic to the network.

In *Lpbcast* [80], membership is achieved by a periodic gossiping of subscriptions: peers transmit a set of subscriptions that they recently heard to a random subset of their locally known group members. Upon receiving such a message, nodes replace a random subscription from their local lists with the new one. To achieve the probabilistic guarantees offered by similar schemes, the size of the group and the local list size must be fixed, which is not the case in highly dynamic networks.

SCAMP [47] is a decentralized membership protocol that utilizes gossiping. Joining members subscribe by contacting a random existing member. Upon receiving a subscription request, a member forwards it to all the members in its local repository. Nodes decide probabilistically whether to store or forward the subscription. For the unsubscription process, a node notifies the locally known members to replace its ID with the IDs of the members it has received messages from. Group communication is performed in the standard gossip-based manner. SCAMP is shown to converge to a local state of slightly over $log(n)$ member IDs, which guarantees with high probability that all members will receive a notification.

In [36], the push phase of an update algorithm for unstructured P2P networks is a rumor-spreading scheme: each peer receives an update message along with a partial list of other peers to which the update has been sent. If the update has not been received before, it is forwarded to a different subset of members with a certain probability. In [41], peers that have received a message less than F times, forward it to B randomly selected neighbors, but only those that the node knows have not yet received it. The deterministic version of that algorithm requires global knowledge of the overlay. Nodes forward messages to all neighbors with degree equal to 1, plus to B remaining neighbors that have the smallest degrees.

In contrast, our approach requires no group subscription/unsubscription process nor any centralized or distributed storage of the current group members. Its forwarding scheme is an adaptive selection between neighbors and shortcuts, relative to the quality of the local search knowledge.

5.3 Replication

Replication is a well-known technique utilized to achieve high availability and fault-tolerance in large-scale systems. While applied to a variety of contexts, we focus in the area of distributed (P2P) systems.

Structured overlays (DHTs) balance routing between network nodes, due to the nature of the hashing functions used. Moreover, in systems like *CFS* [81] and *PAST* [82], each item (or chunk of it) is replicated on a set number of network nodes. DHTs take advantage of the routing structure, which in effect allows for almost-deterministic paths between two nodes, thus identifying “hot” areas easily. Nevertheless, DHTs are not optimized for skewed access patterns and direct such traffic to few nodes responsible for popular content.

DHash [83] is a replication method applied on *Chord* [12]. The protocol allows for r copies to be stored at the r immediate successors of the initial copy’s home. In [84], the authors propose the storage of at most R replicas for an object. Their location is determined by a hash function, allowing requesters to pro-actively redirect their queries. The work in [85] proposes replicating one hop closer to requester nodes as soon as peers are overloaded.

Lar [86] is a DHT-based approach similar to *APRE*, in that it adapts in response to current workload. Overloaded peers replicate at the query initiator and create routing hints on the reverse path. Hints contain some other locations that the content has been previously replicated, so queries are randomly redirected during routing. The method takes advantage of the DHT substrate in order to place the hints. Our scheme does not attempt to re-route queries or shed load to the initiator, but rather places replicas inside forwarding-intensive areas using multiple paths. Moreover, the state kept is accessible at any time, not only at the time of the query arrival. Finally, it appears that *lar* would suffer from a slow propagation of hints in lower-demand scenarios as well as from stale caches in dynamic settings.

HotRoD [62] presents a load-balancing approach for DHTs handling range queries for relational database systems. It is based on a locality-preserving DHT and replication of overloaded arcs (consecutive modes on the DHT ring). The work in [87] employs a minimization function that combines high availability with low load to replicate video content inside a DHT. The approach requires knowledge of peer availabilities, workload and data popularity. In [59], the authors show that load-balancing based on periodic load statistics suffers from oscillation. By directing queries towards the maximum capacity replica location, both heterogeneity and oscillation issues are tackled. However, this approach assumes knowledge of all existing replicas and that replicas regularly advertise their capacities to the network.

The work in [14] discusses static replication in unstructured networks that use *Random Walks* as a lookup method. Various replication strategies are compared and it is concluded that replicating proportionally to the square root of the access frequencies of objects (which must be known a priori) minimizes the size of a search. In [88], replicas install pointers to their locations on $O(\gamma\sqrt{n})$ random peers using a random walk (n being the number of peers and γ a parameter). Searches are conducted in the same manner, contacting $O(\gamma\sqrt{n})$ random nodes with a single walk. This approach utilizes replication of object locations in order to provide guarantees for the success of a search and not for load-balancing or adaptive replication purposes.

The replication method utilized by *PlanetP* [89] attempts to tackle the problem of resource availability in unstructured environments. Peers regularly gossip metadata about their online status, free space and stored objects to other nodes. Each peer periodically chooses an object it hosts and decides, based on information collected from all peers, on its availability in the network. Given a low estimate, it fragments the file and pushes all fragments to nodes using hints about their free space. This approach relies heavily on the collection of data from all network nodes to achieve high-availability.

In most P2P file-sharing applications, replication is naturally handled through content sharing among users. In general, the following two approaches exist: Files comprise of equal size chunks and are individually indexed, or peers dynamically decide the portion that is retrieved from each source peer. The first approach is utilized by *Overnet* [90], *BitTorrent* [91] and *Slurpie* [92]. Each file is divided into a number of standard-size fragments (9500KB, 256KB, 256KB for those systems respectively). A peer may then download different fragments from various sources. Upon completion, each fragment becomes available for sharing with other nodes.

The second approach [93] (or modifications of it [94]) is currently used by other P2P applications (e.g., *Morpheus* [95]). A requester contacts many source peers and retrieves small portions of the file from each of them. When each small chunk is retrieved, more is asked from that specific source. There also exist several schemes (e.g. [96, 97]) which allow for increased robustness in reconstructing a file by receiving a few extra parts of it.

There has also been considerable amount of work on flash crowd avoidance. In [98], overloaded servers redirect future requests to mirror nodes to which content has been pushed. This approach does not tackle the issue of which node to replicate to. *PROOFS* [99] explicitly constructs a randomized overlay to locate content under heavy load conditions or unwilling participants. In effect, the method relies on the combination of a custom overlay and a gossip-based lookup scheme to locate objects, without involving any replication.

Chapter 6

Conclusions

In the last few years, the research community has provided a plethora of powerful tools in the area of distributed communications. The interest in P2P computing produced a variety of systems and schemes that facilitate the two important primitives in large decentralized environments: Content sharing and open communication.

While we are still unsure about the future applications of P2P, no one can deny their popularity and attractive features that favor them as a choice to become the basis of future platforms. Our thesis focuses exclusively on providing adaptive, bandwidth-efficient protocols for data search, retrieval and one-to-many communication in unstructured overlays. Our schemes offer deployable, low-cost solutions for current applications (in the case of *APS-AGNO-APRE*), with a look towards the future and scientific collaborations (in the case of *GrouPeer*, described in Appendix B).

There exists a set of common characteristics in all these methods. First and foremost, we aim for algorithms that are adaptive to the environment they operate on. To achieve that, we enable a *learning* feature in each of our protocols: Peers learn from experience and interactions with other peers in order to both increase their performance and adapt to changes. Second, we aim for a collaborative operation among the users. In order to achieve that, we design our schemes such that individual experience (in the form of state stored at nodes) can be shared and refined collectively. Third, we identify the need for bandwidth-efficient operation in such systems. Consequently, we utilize *directed walkers* instead of flooding in order to locate content. We regulate our push phases such that only the interested peers participate. Finally, *clustering/grouping* of peers according to content or demand is used in order to improve data sharing or our communication flexibility.

In the future, we expect an increase in the number and size of P2P collaborative applications. While our focus will still be on efficient distributed algorithms, more attention will be given to security, reputation and trust issues. The combination of lack of central authority with the reality that not all users are equal or play fair, is the biggest, in our opinion, bet that P2P has to win and decisively so.

Appendix A

Analysis and Comparison of P2P Search Methods

A.1 Overview

With the increasing interest in P2P systems, a plethora of search schemes for unstructured P2P networks has been proposed. In this supporting work, we try to analyze the performance of many representative lookup protocols alongside *APS*. We focus on the behavior of these algorithms for each of the following metrics: Efficiency in object discovery (*accuracy* and number of *hits*), bandwidth consumption and adaptation to changes in topology and object locations. While discovering many objects is very important, as it enables efficient object retrieval, minimizing search messages always represents a high-priority goal for distributed systems. Finally, it is important that any search algorithm adapts to changing conditions, since in most P2P networks users frequently enter and leave the system, as well as update their collections.

To evaluate our analysis, we simulate nine methods and present a direct quantitative comparison of their performance. We identify the relative advantages and disadvantages of each method as well as the conditions under which they can be most or least effective. To our knowledge, this is the first work that attempts a direct comparison of such a diverse set of search techniques proposed for unstructured P2P systems. We believe this is an important contribution that can provide a better understanding of the various mechanisms and assist in choosing an algorithm that best fits a particular application.

A.2 Performance Evaluation

A.2.1 Algorithm Implementations

In this section we present results for nine of the methods described in the Related Work Chapter: *G2, Modified-BFS, Intelligent-BFS, Local Indices, DRLP, GS* and *GIA*, together with the already described *APS* and *Random Walks*. The simulated methods are representative blind and informed schemes, both flood and non flood-based, with or without user-initiated index updates (that is, updates triggered strictly by the search process). In our experiments, we utilize the GT-ITM [32] and Inet-3.0 [33] topology generators to produce sets of random and power-law graphs respectively. For each setup, the results are averaged over a set of 10 similar graphs for each described topology. We also present results on a real gnutella graph [100], with 61,685 nodes and average degree $d = 4.6$.

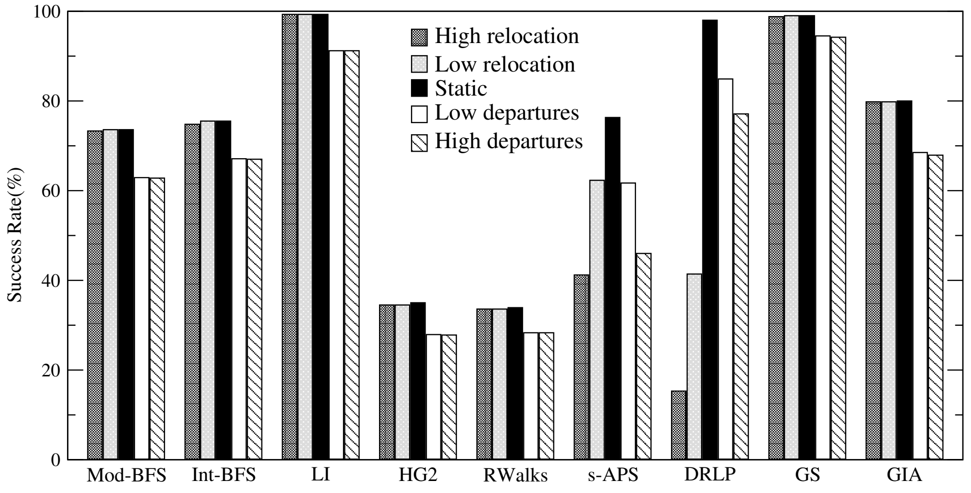
For the default parameters, we mainly follow the model described for the *APS* evaluation (see Table 2.1). Requester nodes are randomly chosen and represent about 20% of the total number of nodes. Each requester makes about 1,500 queries over a time period. We do not allow extra replicas to be stored (i.e., we only consider the search phase, not object retrieval). Finally, besides keeping a dynamic node population, we also redistribute objects to model file insertions and deletions. Object re-location always follows the initial distribution parameters.

The *Intelligent-BFS* method was modified to allow for object-ID requests. Index values at peers now represent the number of replies for an object through each neighbor and nodes choose the neighbors with the highest index values when forwarding a query.

For *Modified-BFS*'s, *DRLP*'s and *Intelligent-BFS*'s flood-based search, nodes choose an equal number of neighbors to forward a query in order to make direct comparisons. For *G2*, peers randomly choose k neighbors to forward the query to. The chosen nodes forward the query to all their neighbors. By modifying the value for k we can simulate the operation of both *G2* (with k always larger than the number of neighbors) and *GUESS*. In our simulations, *G2/GUESS* operate on a pure (instead of a hybrid) model in order to achieve uniformity in our results. Moreover, they both function in a blind manner, so no cache or repository table exchange takes place. We name this approach *HG2* (Hybrid G2/Guess). For our *LI* implementation, nodes index the objects of their neighbors ($r = 1$). To ensure that the search is equivalent to a flood with TTL=5, only peers at depths 1 and 4 process the query. We also ensure that no object from the same peer is being discovered multiple times. Finally, our *GIA* implementation deploys k walkers, with each peer forwarding to the neighbor with the highest out-degree, while the overlay adaptation process is not simulated. Peers index the documents of their immediate neighbors. For our *GS* implementation, we use 5 shortcuts and rank them by their success rates.

A.2.2 Basic Comparison

In our first set of experiments, we use a set of 10,000-Node random graphs (average degree $d = 4$) to compare the nine methods over 5 different environments: A static one, one with low/high object relocation frequency and one with low/high peer departure frequency. In the two low-frequency scenarios, relocation and departures/arrivals occur about 300 times per run, while in the high-frequency ones they occur 10 times more often.



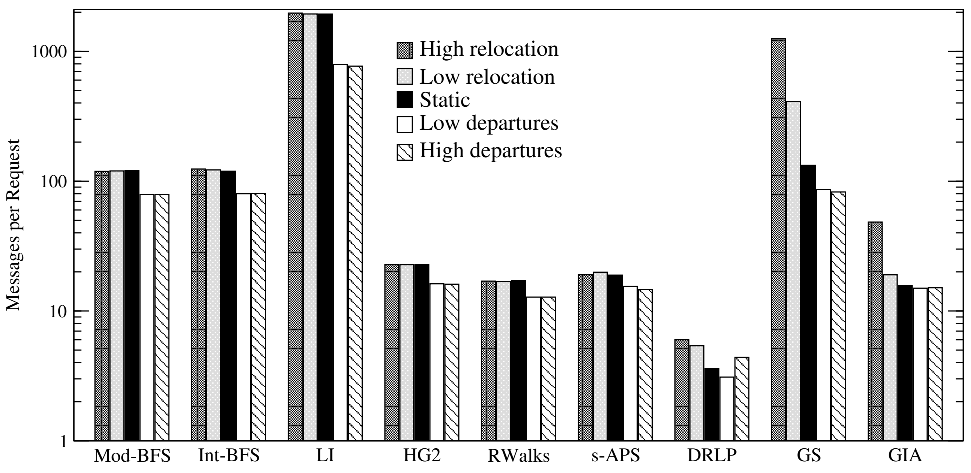


Figure A.1: Success rate and message production of the methods using a set of 10,000-node random graphs with average degree $d = 4$

DRLP and *Int/Mod-BFS* forward to 3 neighbors, while $k = 7$ for *s-APS, GIA, HG2* and *Random Walks*. Figures A.1 and A.2 present the results.

Blind methods show a fairly stable performance between the static and dynamic settings, since the dynamic operations do not interfere with the forwarding scheme. Flood-based schemes discover many objects at a higher cost. Nevertheless, only *LI* and *GS* with the pure-flooding scheme achieve very high accuracy. This happens because of the small out-degree of our network. We also notice that blind and flood-based techniques do not get affected by object relocation, but only by peer joins/leaves. While our relocation process does not substantially alter anything in those algorithms' operation, peer arrivals/departures alter the topology and the amount of available resources.

Mod/Int-BFS show relatively high accuracy and return many hits. Their performance is very similar, with the informed method showing marginally better results. For environments resembling this setup, *Mod-BFS* will be preferred, since its performance is equally high and it is much simpler. We expect the informed method to perform better in richer or more specialized environments (like the one described in [22]), mainly in the number of hits.

Random Walks displays low accuracy ($<$34%) and finds less than 0.5 objects on average. Its bandwidth consumption is quite low (about 15 messages), while its performance is hardly affected by the dynamic operations. *HG2* behaves similarly, with the exception of producing about 5 more messages per search. In general, these algorithms exhibit poor performance and appear very robust to increased network variability. This is reasonable, as walkers are randomly directed with no regard to topology or previous results.

The *s-APS* method achieves a success rate of over 75% in the static run, a number that drops by around 30% in the highly dynamic settings, but only around 12% in the two less dynamic ones. The metric that is reasonably affected is the number of discovered objects, which are almost cut to a third. This happens because it takes some time for the learning feature to adapt to the new topology and paths to discovered objects frequently "disappear". On the other hand, it manages to keep its messages almost as low as *Random Walks*. The scheme is equally affected by relocations and departures/arrivals, since walkers are directed towards specific locations which are altered by both types of events.

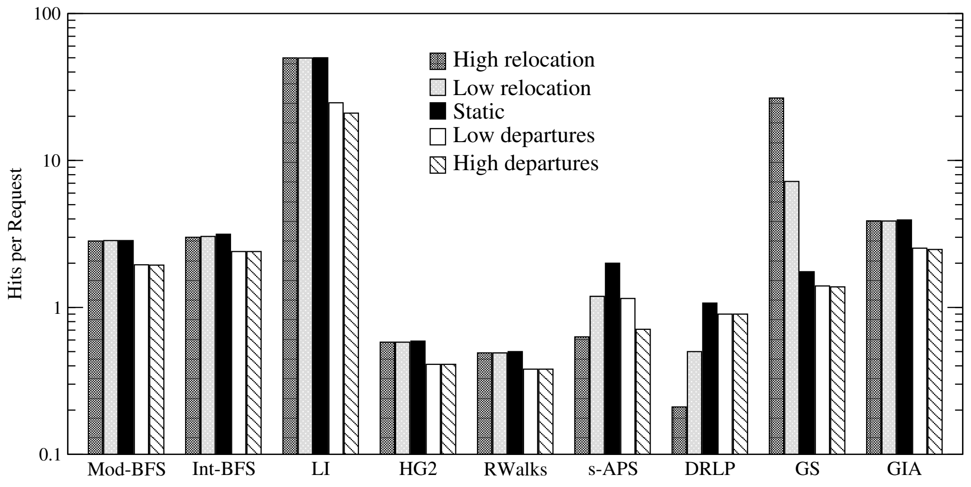


Figure A.2: Hits per query of the methods using the set of 10,000-node random graphs with average degree $d = 4$

Nevertheless, it exhibits a good overall performance compared to the non-BFS related schemes, without indexing other peers' repositories.

The *DRLP* algorithm exhibits some interesting characteristics. First, its message production is very low (less than 6 messages per request). Our simulations count the direct contact of a node (both for *DRLP* and *GS*) as one message, although a link between them might not exist in the overlay. Dynamic behavior causes the stored addresses to become more frequently "stale", thus the initial flooding is performed more often. This is the reason for the decrease in its accuracy from 99% in the static case to 77% and 15% in the highly dynamic ones. *DRLP* produces the same amount of messages for its initial search with *Modified-BFS*, so it needs many successful requests to amortize this initial cost. The number of objects it discovers is very small, ranging from 1.4 to 0.2. If *DRLP* is forced to use flooding many times, then the number of hits increases. If it is successful and produces few messages, then it only finds one replica per request. Despite this, we notice that it proves very bandwidth-efficient and flooding is scarcely used. This is due to the fact that, with many nodes making requests, most of them obtain a pointer for every object after a while. So, even if some node initiates a flood, most of its neighbors will only forward to one other node. The large number of requests per run helps *DRLP* achieve a very low average message consumption. This scheme seems ideal for relatively static environments and large workloads, with the exception that the number of hits will be very close to one. Another important observation is that *DRLP* is affected far more by object relocation than by node departures. This is reasonable if we consider that with departures there still exist nodes with a valid pointer to an object, whereas object relocation may make many pointers become stale at once.

The *LI* scheme proves the most productive in terms of discovered locations and the most costly in message production. It produces one order of magnitude more messages than the other BFS-related methods but also discovers about 10-20 times more objects, taking advantage of its index scheme. Its performance is only affected by the dynamic joins and leaves, with a decrease of more than 50% in located objects. The cost of the index updates, even under the more dynamic settings, is negligible compared to the cost of a search (at most 2% over the total number of messages). On the other hand, this cost is considerable for nodes that stay idle (and possibly alter their local repositories), since it induces traffic without any search involved.

GS shows very high accuracy, since it can always fall back to the flooding scheme. Nevertheless, when peers do not have shortcuts or when these fail (this happens mostly when objects get relocated), message consumption increases dramatically. On the other hand, similarly to *DRLP*, the more flood searches are performed, the more objects are discovered. Shortcuts are mostly used in the static and dynamic arrival/departure modes,

Table A.1: Comparison on 10,000-node random graphs with degree $d = 10$

Metric	*ModBFS*	*IntBFS*	*LI*	*HG2*	*RWALKS*	*s-APS*	*DRLP*	*GS*	*GIA*
Success(%)	98.8	99.8	100	70.2	53.4	91.7	100	100	97.0
Messages	875	1233	39710	108.7	43.6	43.0	8.0	2344	35.0
Duplicates(%)	10.3	0.4	18.7	8.3	0.2	0.1	1.8	17.8	0.9
Hits	20.2	32.6	300.0	2.9	1.2	6.1	1.4	18.9	9.5
Hit Distance	4.58	4.61	3.99	1.88	2.78	3.16	1.90	4.60	3.1

since 5 shortcuts proved sufficient for at least one of them to provide an answer most of the times.

Finally, *GIA* manages to perform as well as *Mod/Int-BFS* but being more bandwidth-efficient. The combination of one hop indexing and biased walkers achieves a good, robust performance at relatively low cost. Only in the high relocation setting we notice a considerable increase (200%) in the average message consumption since peers have to refresh their indices frequently.

A.2.3 Results on Denser Graphs

In the next set of simulations we use a random graph set with an average degree $d = 10$ to compare the 9 methods over two different environments: A static one, and one where both object relocation and peer departures occur about 600 times per run. *DRLP* and *Int/Mod-BFS* forward to 4 neighbors at each step, while $k = 12$ for *s-APS, Random Walks, HG2, GIA*. All other parameters remain the same. The results for the static case are shown in Table A.1. We also report the percentage of messages per search that are duplicates and the average distance of the hits in overlay hops.

Blind forwarding causes a large amount of messages to be dropped. Informed meth-

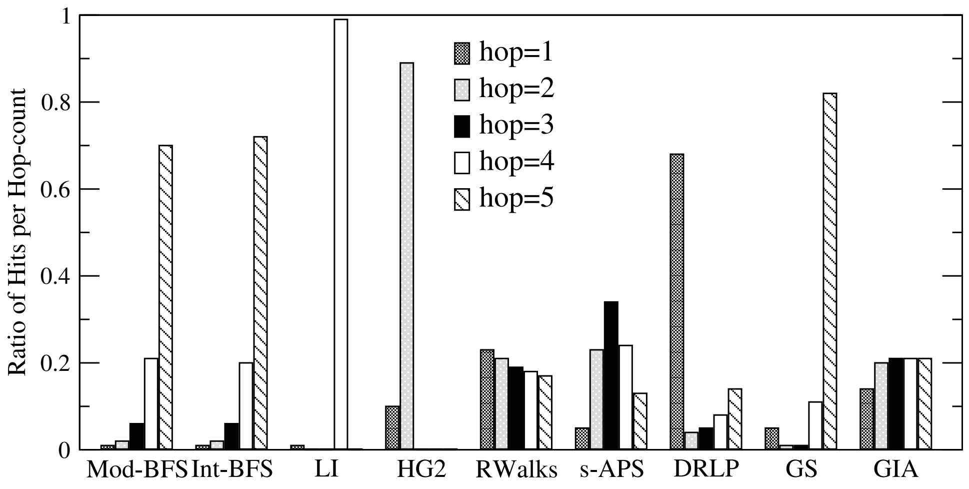
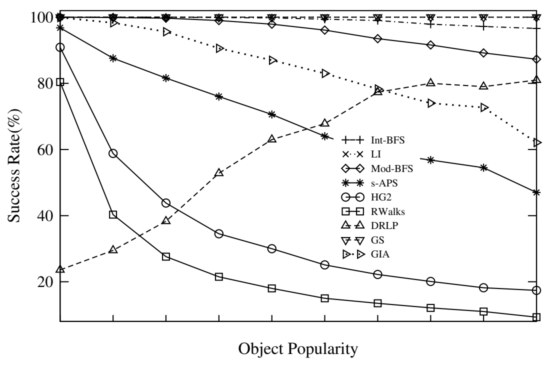


Figure A.3: Hits per hop distance from the requesters

ods with no direct indices perform much better (*s-APS, Int-BFS* wasting only 0.1% and 0.4% of their messages respectively). Flood-based schemes also exhibit large hop distances for their hits.

All algorithms produce a larger number of messages per request in the new graph, taking advantage of the larger number of connections. *DRLP* still averages less than 10 messages per request. *Random Walks* and *s-APS* roughly double their hits and increase their accuracy. On the other hand, *Int/Mod-BFS* produce 10 times more messages. *HG2* performs in between, producing about 5 times more messages. *LI* increases its bandwidth production by more than an order of magnitude. The overhead due to update messages is even less apparent now, since its search messages overshadow their effect. *GS*'s performance increases similarly to *LI*'s since they use the same underlying mechanism. Finally, *GIA* exhibits a very good performance again, having low message consumption and increased accuracy/hits.

Another interesting metric is the percentage of hits discovered at various distances by the methods (Figure A.3). It shows how many objects each method locates with few or



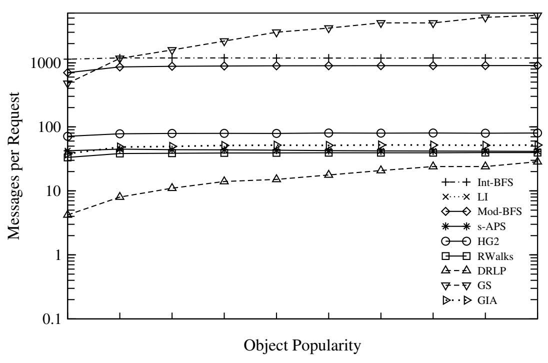


Figure A.4: Accuracy and message production vs. object popularity in the dynamic setting

more messages. Our discussion is based on the static setting. Flood-based schemes discover the vast majority of the objects TTL hops away, since the available nodes increase exponentially with distance. *LI* always locates about 99% of its objects 4 hops away, and the rest only 1 hop away from the requesters (since only nodes at these two depths process the queries), while *HG2* discovers about 90% of the objects with its flooding phase (2 hops away). *Random Walks* discovers almost the same number of objects per distance, since the query forwarding is done randomly. *GIA* also uses walkers and exhibits a similar behavior as requesters are randomly chosen in our simulations. *DRLP* finds almost 70% of its hits using its indices (which also explains why its hit average is close to one). *s-APS* displays a symmetric curve. After a certain distance, possible paths become too many and the accuracy of the indices drops. Finally, we notice that *GS* only discovers about 5% of its hits using the shortcuts, whereas in the smaller graph the respective number was 50%. This can be explained by the fact that the flooding scheme now finds 2 orders of magnitude more objects than in the previous graph, while shortcuts still find one object.

Figure A.4 shows how object popularity affects the methods' accuracy and message production in the dynamic environment. Popularity decreases as we move to the right along the x-axis. The first data point represents the accuracy/messages of the methods for the top-10%, the second for objects ranked between 11–20%, etc. This is an important comparison, because different applications or users target objects of varying popularity.

The three BFS-related methods together with *GS* exhibit very high accuracy, with *Mod-BFS* showing a noticeable decrease only for the least popular items. *Random Walks, HG2, s-APS* and *GIA* show decreasing accuracy as popularity drops, with *GIA* and *s-APS* clearly performing better. *DRLP* performs very poorly for the very popular documents (about 20%), but its accuracy increases as popularity drops. This can be explained by the fact that less popular objects receive considerably fewer queries. Therefore, object relocations and node departures which affect the algorithm happen less frequently during requests for such objects. All algorithms — except *DRLP* and *GS* — waste roughly the same amount of messages per request for each popularity group. *DRLP* and *GS* increase their consumption with a popularity decrease for the sole reason that the cost of the initial floods is now amortized over a smaller number of requests. Finally, we noticed that all algorithms except *DRLP* and *GS* discover a decreasing number of objects as popularity drops, exactly because this means there exist fewer objects to be located.

In the dynamic environment, we also measure the percentage of messages per request sent due to index updates (for relevant methods only). We found that *Int-BFS* requires 11%(=131 mesg) of its messages for index updates. The respective numbers for *LI, GIA* and *s-APS* are 14.4%(= 2968 mesg), 31.7%(= 14 mesg) and 18.5%(= 8 mesg). Although *GIA* and *s-APS* appear to require a larger portion of updates, they are much more bandwidth-efficient than the other methods in absolute numbers.

Our previous simulations depicted the relative performance characteristics of the

Table A.2: Comparison on 10,000-node random graphs with degree $d = 20$

Metric		*Mod-BFS*	*Int-BFS*	*HG2*	*RWALKS*	*s-APS*	*DRLP*	*GS*	*GIA*
Messages	Success(%)	63.6	67.6	63.5	62.2	93.4	100	90.8	99.9
	Messages	**73.4**	**83.0**	**77.0**	**72.5**	**70.6**	**79.2**	**77.0**	**70.0**
	Hits	1.9	2.3	2.1	2.0	10.7	5.3	1.12	14.9
Hits	Success(%)	75.8	77.0	71.9	75.0	80.2	100.0	100.0	92.2
	Messages	134.4	117.1	115.1	125.2	31.4	43.0	356.5	32.1
	Hits	**3.5**	**3.2**	**3.1**	**3.2**	**3.8**	**3.4**	**3.6**	**3.8**

nine algorithms. To some extent, that sort of comparison was not direct either because of the different nature of the methods or because of the single choice of the various parameters. Since it is impossible to directly compare the methods for the same parameter values (e.g., k, TTL), we select simulations on a third set of 10,000-node random graphs ($d = 20$), where the algorithms had similar performance in one of two important metrics: Messages and hits per query. These results were obtained by experimenting on various values for k, TTL, number of neighbors to forward and number of requester nodes. The results are presented in Table A.2 and the comparison metric is typed in boldface. *LI* is omitted from this table because its large number of messages and hits could not be matched by the other methods.

For similar message consumption, first *GIA*, then *s-APS* discover the most objects (followed by *DRLP* with about 10 extra messages per search). These three methods also prove extremely accurate, while the rest of the schemes (either flood-based or random) do not perform well. For similar hits per search, again *GIA* and *s-APS* stand out above *DRLP*, which wastes a few more messages but is perfectly accurate. From the rest of the methods, only *GS* is 100% successful, but exhibits the highest message consumption.

Table A.3: Comparison of the nine methods with a 20,000-object pool

Graph		*Mod-BFS*	*Int-BFS*	*LI*	*HG2*	*RWALKS*	*s-APS*	*DRLP*	*GS*	*GIA*
RAND	Succ(%)	68.4	69.7	89.9	30.7	29.8	75.2	99.0	89.2	74.4
	Mesg	118.8	115.4	1511.6	24.9	18.6	24.1	7.1	563.5	18.3
	Hits	2.3	2.4	37.7	0.5	0.4	2.2	1.2	5.0	3.2
PLAW	Succ(%)	56.8	62.3	93.3	76.7	22.9	75.7	98.3	88.4	85.7
	Mesg	73.3	82.0	1473.0	750.3	13.1	15.1	5.0	355.9	19.1
	Hits	1.5	1.8	86.1	17.7	0.3	1.9	1.2	3.0	13.9
GNUT	Succ(%)	67.8	76.2	94.7	63.3	33.7	70.1	99.1	83.6	78.8
	Mesg	145.6	217.4	1325.1	282.1	24.3	33.1	17.1	886.5	20.9
	Hits	2.6	4.4	59.8	5.7	0.5	3.0	2.0	15.3	6.0

A.2.4 Increased Number of Objects

Our previous model was mainly tailored for a system where peers continuously search for specific objects. The wide range of replication ratios together with the network dynamics best enables us to observe the effect of popularity, dynamic behavior and forwarding scheme. We now consider a more general situation, with a large number of objects (20,000) and 5,000 requester nodes, each making 2,000 queries. This could be an example of a P2P search engine application, with users having their own preferences (changing with time). Table A.3 presents our comparison using three sets of graphs, our original 10,000-node set ($d = 4$, RAND), a 10,000-node power-law graph set ($d = 4.4$, PLAW) and a Gnutella topology snapshot ($d = 4.6$, GNUT). For larger graphs (simulations up to 50,000 nodes), results are qualitatively similar.

Compared to the previous results, we clearly notice a small performance degradation, which is natural if we consider that now more queries are made for sparsely located objects, while flooding is used more by some of the methods. Nevertheless, first *DRLP*, followed by *s-APS* and *GIA* achieve numbers closest to the original ones. With the power-law topology, although the average out-degree is the same as with the random graphs, various neighborhoods differ substantially, since there are few nodes with very high connectivity. *GIA* clearly takes advantage of this to increase its discovered objects. Another observation is that pure flood-based schemes also discover substantially more objects (compared to the respective runs over the random topologies with 20,000 objects). *HG2* achieves more than 10 times more hits with a 150% increase in accuracy, using 30 times more messages. *LI* doubles its hits without any message increase. The rest of the schemes perform very similarly to the previous simulation. The results for the real topology resemble those for the power-law graphs if we also take into account the size increase as well as an increase in the average out-degree and the number of poorly connected neighborhoods (possibly due to crawling imperfections). In general, most methods show increased messages and hits compared to the random topologies. While they effectively locate popular objects, they either fail to be as accurate or greatly increase their message production for the bulk of the non-popular items.

A.3 Conclusions

In this work we presented many of the search techniques available for unstructured P2P networks, along with a quantitative comparison through simulation. Our analyses focus on the performance metrics of search accuracy, bandwidth consumption, discovered objects and behavior under dynamic operations.

The specifics of the problem play a big role in choosing the appropriate method. Each scheme has its own goals and it is important that these goals match the application's. Important parameters that could influence our decision include the primary purpose of the application (e.g., fast discovery, many hits, bandwidth-efficient and accurate, easy deployment, etc), the underlying topology, expected workload, etc. We offer some general-purpose observations based on our analysis and simulations, hoping they will prove useful in evaluating the plethora of different schemes.

a) Blind forwarding is not adequate for both high numbers of hits and low message production.

b) Index semantics play an important role: Direct location information is effective but sensitive to changes and more demanding (becomes obsolete if a failure/relocation occurs, requires update messages). Indirect information (e.g., success rates in *s-APS, Int-BFS* or connectivity/capacity in *GIA*) is much more robust but less accurate.

c) Indexing other peers' repositories is very useful but must be carefully applied, since it requires updates to keep the indices up-to-date.

d) Adaptation is a key characteristic through which peers that have a prolonged stay in the network enhance their knowledge with time. *GS, s-APS* and *Int-BFS* learn from system searches and improve their performance.

e) In many cases, the simple protocols are the preferred ones. The simplicity of the mechanisms behind flooding or Random Walks make them powerful and easy to implement. They can be used either by themselves or in combination with other schemes to improve their performance.

Appendix B

Sharing Relational Data in Unstructured Overlays

B.1 Introduction

In this Appendix we describe the problem of sharing relational data in unstructured overlays. This is joint work with Verena Kantere and Professor Timos Sellis of the Department of Electrical and Computer Engineering, National Technical University of Athens, Greece.

In the past few years, there has been a growing interest in the Peer-to-Peer (P2P) paradigm, primarily boosted by popular applications that enable massive data sharing among millions of users. Our research, thus far, has been focusing on applications with exact-match queries: Users are requesting for an object by either providing a unique identifier (e.g., filename, system-wide file-ID, etc) or a single `attribute-value` pair, evaluated always as `TRUE` or `FALSE`. While this formulation covers a significant portion of real-life scenarios, it is certainly not restrictive. Scientific collaborations, enterprise data integration and sharing in the World Wide Web are only examples of applications that require more powerful data and query formulations.

In contrast to data integration architectures, P2P data sharing systems do not assume a mediated schema to which all sources of the system should conform in order to share data. In such a system, each peer is an autonomous source that has a local schema and individually stores and manages its data, revealing only part of its schema to the rest of the peers. Due to the lack of global schema, users express and answer queries based on their local schema. In a P2P data management system, peers also perform local coordination with their *acquaintees*, i.e., their one-hop neighbors in the overlay. During the acquaintance procedure, the two peers exchange information about part of their local schema and create a mediating mapping semi-automatically [101]. The establishment of an acquaintance implies an agreement for the performance of data coordination between the acquaintees based on the respective schema mapping. However, peers do not have to conform to any kind of data or schema transformation to establish acquaintances with other peers and participate in the system.

As we mentioned before, many popular P2P applications operate on unstructured networks, with peers joining and leaving the system in an ad-hoc fashion, while maintaining only local knowledge. In such systems, joining peers usually become acquainted to the first randomly available nodes and not to the ones that best meet their need for information. Therefore, they have to direct queries not only to their neighbors, but to a greater part of the system. One can roughly identify two common approaches in order to query and retrieve answers in such a system:

The first approach is to propagate queries on paths of bounded length in the overlay. At each routing step, the query is rewritten to the schema of its new host based on the respective acquaintance mappings (see Figure B.1). A query may have to be rewritten several times from peer to peer till it reaches nodes that are able to answer it sufficiently in terms of quality but also quantity. It is obvious that the successive rewritings decrease or restrict the information that can be returned by a query and, thus, also reduce the possibility of accurate query answering. Moreover, it is the case that peers may not be able

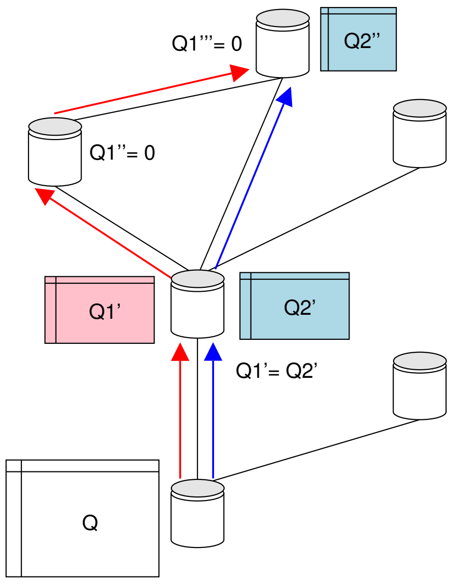


Figure B.1: Propagation of queries among acquaintees. The size of the rectangles reflects the amount of degradation after a rewriting. $Q_1''' \neq Q_2''$ because the queries followed different paths

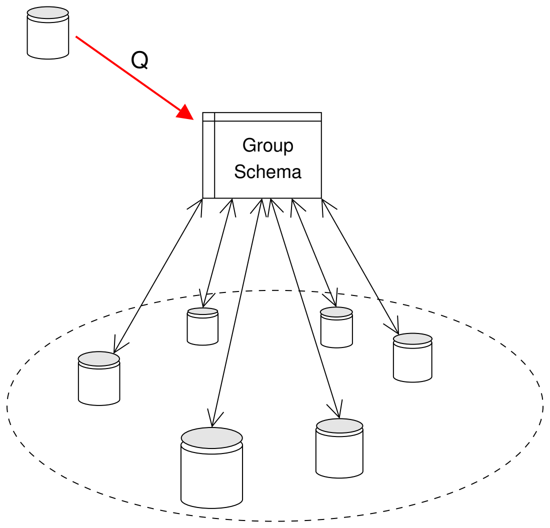


Figure B.2: Query directed towards a group schema which holds mappings with all group members

to sufficiently answer received queries, not because their local schema does not match the initial query adequately, but because the incoming rewritten version has been gradually reduced or corrupted. Therefore, the performance of the query processing procedure is degraded during the rewritings on intermediate peers.

In the second approach, nodes are organized (usually by one or more administrators and application experts) into groups of peers that store semantically related data. The administrators, using schema matching tools as well as domain knowledge, create a mediated schema that is representative of the group. Group schemas hold mappings with each of the local databases. This configuration corresponds to multiple data integration system realizations, one per semantic group. Queries are then globally expressed on this mediated schema (see Figure B.2). Obviously, this approach requires human involvement,

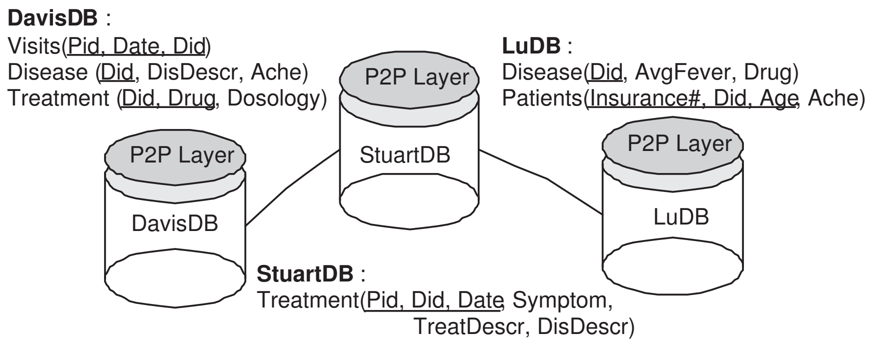

DavisDB :
Visits(Pid, Date, Did)
Disease (Did, DisDescr, Ache)
Treatment (Did, Drug, Dosology)
LuDB :
Disease(Did, AvgFever, Drug)
Patients(Insurance#, Did, Age, Ache)
P2P Layer
StuartDB
P2P Layer
DavisDB
P2P Layer
LuDB
StuartDB :
Treatment(Pid, Did, Date, Symptom, TreatDescr, DisDescr)

Figure B.3: Part of a P2P system from a health-related environment

extensive peer coordination and repetition of the process each time the group changes.

B.1.1 Motivating Example

As a motivating example, envision a P2P system where the participating peers are databases of private doctors of various specialties, diagnostic laboratories and databases of hospitals. Figure B.3 depicts a small part of this system, where nodes are: DavisDB - the database of the private doctor Dr. Davis, LuDB - the database of pediatrician Dr Lu and StuartDB - the database of the pharmacist, Mr Stuart. A P2P layer on top of each database is responsible for all data exchange between a peer and its acquaintees. The P2P layer is also responsible for the creation and maintenance of mappings between local schemas during the establishment of acquaintances towards the line of [101]. Moreover, each peer owns a query rewriting and a query-schema matching mechanism. The local schemas exported by these peers are shown in Figure B.3.

Suppose that Dr Davis would like to collect from the system general information about patients that have had diseases. He expresses the following query on his database:

Q_{orig}:

```
SELECT  V.Pid, D.DisDescr, D.Ache, T.Drug, T.Dosology
```

```
FROM    Disease D, Treatment T, Visits V
WHERE   V.Did = D.Did AND D.Did = T.Did
```

Having only one acquaintance, the pharmacist's database, Dr. Davis's database propagates Q_{orig} to it. We assume GAV, LAV, or GLAV (i.e. Global, Local, Global and Local As View) mappings between acquaintees [102]. We assume the following LAV mapping between DavisDB and StuartDB databases:

$M_{StuartDB\_DavisDB}$: Treatment(Pid, _, _, Symptom, TreatDescr, DisDescr) :-

Visits(Pid, _, Did), Disease(Did, DisDescr, Ache), Treatment(Did, Drug, _),

where correspondences Symptom = Ache, TreatDescr = Drug are implied[1]. Thus, the rewritten query on StuartDB is the following:

$Q_{StuartDB\_sr}$:

```
SELECT  T.Pid, T.DisDescr, T.Symptom, T.TreatDescr
FROM    Treatment T
```

Obviously, the new query has lost the attribute referring to information about drug dosology, since it cannot be mapped in StuartDB. The node of Mr Stuart passes the rewritten version $Q_{StuartDB\_sr}$ to Dr Lu with whom he has the following GAV mapping:

$M_{StuartDB\_LuDB}$: Treatment(Pid, _, _, Symptom, _, _) :-

Disease(Did, AvgFever, _), Patients(Insurance♯, Did, _, _), Age < 13[2],

[1]The mapping is actually a view defined on StuartDB.Treatment, which is matched with a join on DavisDB relations such as: View1(Pid, Symptom, TreatDescr, DisDescr):-Treatment(Pid,Did, Date, Symptom, TreatDescr, DisDescr)

View1(Pid, Ache, Drug, DisDescr):- Visits(Pid, Date, Did),Disease(Did, DisDescr, Ache), Treatment(Did, Drug, Dosology). We summarize mappings by omitting view definitions and introducing '_' for attributes that are not matched.

[2]Because he treats children

where correspondences Pid = Insurance♯, Symptom = AvgFever are implied. Thus, the rewritten query on LuDB is the following:

Q_{LuDB_sr}:

```
SELECT  P.Insurance#, D.AvgFever
FROM    Disease D, Patients P
WHERE   D.Did = P.Did, P.Age < 13
```

Clearly, the new query has lost more attributes, which refer to the description of the disease and the respective drug. Moreover, the new query is more restrictive than the original, since it has an additional condition on 'Age'. Finally, it is clear that the 'Ache' attribute of the original query has been poorly rewritten to 'AvgFever', even though the schema of LuDB contains an attribute that represents the exact same concept. Yet, if Dr Davis was acquainted with Dr Lu, among the supported mappings could be:

$M'_{LuDB\_DavisDB}$: Visits(Pid, _, Did), Disease (Did, _, Ache), Treatment (Did, Drug, _) :- Disease(Did, _, Drug), Patients(Insurance♯, Did, _, Ache),

where the correspondence Pid = Insurance♯ is implied. Using the above mapping, Dr Davis would ideally like his query to be translated as follows:

Q_{LuDB_ideal}:

```
SELECT  P.Insurance#, D.Ache, D.Drug
FROM    Disease D, Patients P
WHERE   D.Did = P.Did
```

The above version overcomes the degradation of successive rewriting in terms of query information loss and further query restriction, as well as the poor matching of the 'Ache' attribute.

Our approach enables DavisDB to evaluate Dr Lu's query translations (e.g., suggest that Ache = AvgFever is not a good correspondence and Pid = Insurance is a good one) and gradually help him improve the quality of its query rewriting. Through iterative evaluations, Dr Davis notices the average answer quality from Dr Lu is high enough to add him as an acquaintee.

Following the clustering of peers into semantic neighborhoods, our system can initiate the creation of a mediating schema $\mathcal{G}$ representative of all three databases. $\mathcal{G}$ holds mappings with each of the creating nodes and functions as a point of contact for all incoming queries, whether from inside or outside the cluster. Thus, requesters need only evaluate answers and mappings against one schema, instead of multiple ones. Furthermore, they can effectively speed-up the learning process by directing queries to semantically relevant clusters known system-wide.

B.1.2 Our Contribution: *GrouPeer*

GrouPeer is a system designed to enable accurate query evaluation through semantic overlay clustering and automatic creation and maintenance of semantic groups in relational P2P databases without prior schema or meta-schema information. *GrouPeer*'s contributions are twofold:

- *Clustering of semantically related peers:* Nodes individually decide whether to answer the successively rewritten query or automatically rewrite its original version. Requesters evaluate the replies along with the returned rewritings and gradually build mappings with remote peers. Eventually, peers with similar local schemas become acquainted and clusters are created around active peers.

- *Group schema creation and maintenance:* Nodes in well-formed semantic clusters are candidates for initiating the group inference process. The process contacts nodes similar to the initiator inside the already formed cluster, creating a schema representative of the participants. Group schemas are then propagated inside the network, enabling all nodes to direct relevant queries towards a single mediated schema.

Our focus will be on the second part of *GrouPeer*, i.e., the creation of group schemas from semantically similar nodes in a completely distributed manner. In Section B.2 we present a brief overview of the clustering process. A detailed description can be found in [103]. Section B.3 discusses group schema creation, while Section B.4 presents our experimental evaluation.

B.2 Clustering Peers for Accurate Query Answers

GrouPeer proposes a procedure that supports the evasion of successive rewritings along every query's propagation path. This methodology enables peers to discover others with similar interests and schemas, given that no form of global knowledge (e.g., each peer's schema [104]) is assumed system-wide. Learning is performed through making queries and evaluating their answers, and is formed through mappings between the respective schemas. As pairs of peers build more mappings, query rewriting becomes more accurate. Eventually, peers with similar schemas become acquainted, gradually restructuring the overlay into semantic neighborhoods. This process is referred to as the *clustering process*. We should note here that *GrouPeer* focuses on queries and mappings that can be expressed as SPJ queries (or else conjunctive queries with arithmetic comparisons)

B.2.1 Query Reformulation and Similarity for P2P Database Systems

The goal of the reformulation mechanism is to transform a query so that it can be answered, fully or in part, by an acquaintee. We assume that each peer exports a relational schema to its acquaintees. For the remainder of this work, we shall refer to it as the peer's schema, regardless if it represents the whole schema or a part of it (e.g., for security/anonymity purposes). Each pair of acquaintees holds peer mappings between their schemas, which are considered to be of the well-known GAV/LAV/GLAV form.

The available query rewriting algorithms restrict their usage to queries that can be *completely* rewritten under a set of mappings. Yet, this is not suitable for a P2P environment, where peers are satisfied with information that shares characteristics similar to those of their query, not necessarily (and precisely) all of them.

In *GrouPeer*, peers utilize a modified reformulation mechanism based on existing rewriting algorithms. Our mechanism allows a rewritten version of a query to maintain only the attributes and conditions that "survive" the query translation that is performed among acquaintees.

After the translation of a query to their local schemas, peers can proceed with its computation. Our goal is to measure the similarity between different versions of a query and its original formulation in order to decide which constitutes a more accurate rewriting. The similarity function M_{sim} proposed in *GrouPeer* measures the semantic deviation of the target query Q_{rewr} from the original Q_{orig}. Assuming that every query is defined as a set of elements, one for each '`select`' attribute and one for each '`where`' condition, the rewritten version can deviate from the original one in the following ways: Elements of Q_{orig} cannot be mapped in Q_{rewr}, or extra conditions are introduced in Q_{rewr}.

Formally, for two query versions Q_{orig}, Q_{rewr} and a set of user-specified weights $w_{Q_{orig}}$ that denote the importance of each element in the semantics of Q_{orig}, the similarity of the rewritten version to the original is:

$$M_{sim}(Q_{orig}, Q_{rewr}) = 1 - \frac{\left(\sum_{\substack{\text{elements i} \\ \text{not mapped}}} w_i \right) + \#\text{ of conditions in } Q_{rewr} \text{ but not in } Q_{orig}}{\sum_i w_i}$$

M_{sim} is structured such that dissimilar elements diminish its value. Perfect similarity is represented by $M_{sim} = 1$.

B.2.2 Description of the Clustering Process

In order to achieve the discovery of remote relevant peers, the key idea of our method is to propagate along the query path not only the successively rewritten version, but also the original one. In this way, peers can individually decide which one to answer. Peers are assumed to be equipped with a query rewriting mechanism and an automatic schema-matching tool. The rewriting mechanism is used to reformulate queries received from acquaintees based on the respective mappings. The automatic schema-matching tool is used in order to translate queries (or parts of them) expressed on schemas for which mappings are not available.

Successive query reformulation produces query versions that deviate from the original query. Obviously, if the chain of peer mappings used for the rewriting is poor in information relevant to the query (i.e., query elements cannot be reformulated accurately), this can result in fast degradation within a few hops. Query elements that cannot be translated through existing mappings are eliminated in the rewritten version. Although the following nodes on the query path may encapsulate the eliminated concepts in their schemas, they still cannot contribute them to the original query, because the version they receive does not include them. Our approach keeps the eliminated concepts and tries to match them in subsequent reformulations.

Overall, an initiated query Q_{orig} is propagated along the query path. On each node, the query is rewritten through mappings with the previous node to Q_{sr}, which is augmented with automatically rewritten query elements to Q_{sra}. Also, Q_{orig} is automatically rewritten from scratch to Q_{ar}. The answering node compares the two rewritten versions with the original one, using our similarity function and answers the version it seems most similar to it. The query initiator evaluates the satisfiability of the received answer and sends its feedback to the answering peer about the query version it chose to reply to. According to the evaluation, the query replier keeps record of bad and good rewritings on the initiator's schema elements. Gradually, content providers build mappings with the initiators through the queries they receive and answer on their behalf. Moreover, the initiators log the evaluation of query answers from each replier. Based on this, the initiators can decide that they have common interests with a remote peer and ask to become acquainted. New acquaintees can base their communication on mappings already created.

B.2.3 GrouPeer Protocol Internals

In the following we describe basic algorithm internals, specifically the query routing scheme and the addition/deletion of acquaintances.

1) Routing: Our method utilizes informed walks with a TTL parameter in order to propagate queries to nodes in the overlay. The requester deploys k walkers, each following independent paths. A node forwards a query to the neighbor(s) whose schemas have the highest similarity value with respect to this query. Note that these values can be computed, since neighbors share this information by default in our protocol.

2) Adding/dropping acquaintees: We augment our clustering algorithm by allowing the dropping of existing neighbors in order to gradually improve on the random initial setup: New acquaintees are added whenever the local evaluation average is over θ_{P_I} and existing ones are dropped when its value is below $\theta_{P_I Low}$, provided we have received at least *THR* replies from that node. This confidence parameter is important to ensure that the local evaluation is based on an adequate number of queries. We also define a maximum number of connections per peer, MAXDEGREE, which forces a neighbor addition to be preceded by the dropping of the neighbor with the smallest schema similarity if this limit is reached. A link is dropped whenever the local evaluation average is below $\theta_{P_I Low}$, provided the degrees of both nodes are at least MINDEGREE. This ensures that peers do not get disconnected from the network.

B.3 Interest Groups in *GrouPeer*

We now describe *GrouPeer*'s group schema creation process. Our goal is to materialize the creation of semantic clusters by combining the overlay clustering presented in the previous section with a distributed process that iteratively merges local schemas into the final group schema.

After the performed clustering, peers with similar information are close in the overlay, achieving an increase in the number and quality of answers. Yet, this overlay clustering is implicit, in that there is no information about the identity and characteristics of a cluster or the peers that participate in it. We define explicit knowledge of a cluster to consist of knowledge about its participants, their schemas, the cluster's schema and the relations between each participant's schema and the group schema. Such information has multiple advantages:

First, it enables peers to direct relevant queries towards a single, authoritative, schema. Instead of traversing multiple paths and performing learning with multiple sources, query initiators interact with a single "virtual" schema (the group schema). Participants already hold complete mappings with this target schema.

Joining nodes can also benefit by selecting appropriate acquaintees or speeding up the learning process instead of choosing random entry points. Finally, since our system operates in a dynamic environment, with node arrivals/departures and possible schema or workload changes, dynamically created group metadata can be automatically refreshed.

We call these explicit clusters *interest* or *semantic* groups and the process of creating them the *group inference* process.

B.3.1 Group Inference

The process comprises the following steps: (a) Initialization - who and when initiates the group inference, (b) Propagation - how does the process advance among peers of the same group, and (c) Termination and Refinement - when is the process over/reiterated.

Initialization: There are two main considerations in the initialization process: First, the nature of our application requires that the group inference is performed in a distributed manner, without global coordination. Hence, peers should independently decide to start the process that creates the respective schema. Second, we must ensure that the initiator(s) are qualified representatives of a semantic group. Given that, the only proof of group existence in our system is implied by the changed overlay topology and local state stored at each node. In *GrouPeer*, a peer may consider itself part of a semantic cluster and initiate the group inference process if the following requirements are met:

- The similarity of answered queries to the original ones (measured at this node) is above a certain threshold T_o.
- The average rate of queries sent from this node is over a certain threshold.

For example, we may require that any prospective *initiator* has received replies of average similarity greater than 0.7 and that it has made at least 50 queries in the last 2 hours. These requirements satisfy both conditions described before, by ensuring that the respective peer is an active participant (rate of queries) and is a member of a well-formed cluster (similarity threshold).

Since any prospective initiator is a qualified representative of the group, its local schema will also become a point-of-reference regarding the inferred one. Thus, the peer schemas considered for the formation of the group schema should not substantially differ (in semantic distance) from the schema of the initiator. The following function calculates the *directed* semantic similarity, *SS*, of two relational schemas:

$$SS(S,T) = \frac{\sum_i \sum_j w_{ij} Mapped_T(SR_{ij})}{\sum_i \sum_j w_{ij} SR_{ij}}$$

In the above function, S is the source schema and T is the target schema. *SS* calculates the portion of *S*'s attributes (*SR*) that are mapped on T. Obviously, $SS(S,T) \neq SS(T,S)$ in general. In order for *SS* to be computable, we have to know the mapping between i, j. This requires a composition of mappings between acquaintees until a mapping from i to j is produced. Related work [105] describes efficient composition schemes that can be utilized. Nevertheless, *GrouPeer* assumes mappings that are simple 1-1 correspondences and can be easily composed. *SS* achieves to measure semantic similarity because it takes into consideration the mapping of concepts beyond their structural interpretations on the schema level. Moreover, since *SS* ignores the schema structure, it is very easily calculated.

In *GrouPeer*, we require that all considered local schemas be at least t-similar to the initiator's schema: $SS(S_I,T) \geq t, \forall T$. The initiator peer I is called the *originator* of the group, its schema S_I is the *origin* of the group schema and the maximum similarity distance between the origin and the peer schemas that participate in this process is the *semantic radius t* of the group.

Propagation: Initiator I (with schema S_I) initializes the group schema to its own and creates a stack $ST(I)$ with its acquaintees that are part of the cluster. Specifically, $ST(I) = \{A_1, A_2, ..., A_m\}$ is an ordered set of elements $A_j = \{P_j, SS(S_I, S_{P_j})\}$, where P_j is a peer with schema S_{P_j}. Elements A_j refer to the I's most similar acquaintees: $SS(S_I, S_{P_j}) \geq$

t, $j = 1,..,m$ and $SS(S_I, S_{P_j}) \geq SS(S_I, S_{P_{j+1}})$, $j = 1,..,m-1$. The initiator propagates the inference procedure to the first peer on the stack. Each intermediate node P merges its own schema with the group schema it receives. P then determines its acquaintees P_j for which $SS(S_I, S_{P_j}) \geq t$, adds the respective pair $\{P_j, SS(S_I, S_{P_j})\}$, to $ST(I)$ and orders it. $SS(S_I, S_{P_j})$ is calculated indirectly, as the product: $SS(S_I, S_P) \cdot SS(S'_P, S_{P_j})$, where S'_P is the part of S_P mapped on S_I. Essentially, $SS(S_I, S_P)$ aims to measure how much of the semantics of S_I can be found on schema S_P, independently of other semantics that the latter captures. The only way to measure this (without automatic matching) is through the chain of mappings of S_I all the way to S_P. As such, the value of $SS(S_I, S_P)$ depends on the path that the inference process follows and fails to consider concepts that exist both in S_I and S_P but not in the schemas of intermediate nodes. However, this formula produces a satisfactory result, since nodes are visited in decreasing order of similarity with I and clustering precedes this process, so a peer P will have higher similarity with the originator than successor nodes in the stack. Moreover, if a peer P already in $ST(I)$ is considered for addition, the entry with the highest $SS(S_I, S_P)$ value is kept.

Even though the participation or not of peers in the inference process is judged by a part of their schemas, their whole schema contributes to the inferred group schema. The goal of the inference process is to produce a schema that represents semantics encapsulated in the cluster. In order to determine the cluster's semantic borders, we use the semantics of the initiator as a reference. In this way, the process is safe from producing a schema much broader or distorted from the initiator's interests.

Termination: The group inference procedure ends when the stack of participating peers becomes empty. However, if many peers have schemas very similar to the originator's schema or the similarity threshold t is small, (i.e., the semantic radius is big), then it may be the case that the stack grows at each step. The inference procedure is prolonged, taking into account a large number of peers. After a certain number of iterations, there is usually no point in considering more schemas, because they do not contribute significantly. In order to reduce the time of the inference and save valuable network resources, we add a limit to the maximum number of encountered peer schemas, *MaxP*, as a termination condition. *MaxP* is not a TTL condition, since successive hops are not always on the same path; *MaxP* refers to the total number of participating nodes.

Finally, there may be situations where the inference procedure terminates due to *MaxP* while important semantic information is still added, or continues until *MaxP* is reached while little information is assimilated. To rectify this, *GrouPeer* also considers the *degree of change* that occurs to the inferred schema during each merging step. In case of a poorly chosen *MaxP* value, this criterion can be used to calibrate this parameter.

B.3.2 Discussion on the Group Inference Process

In this section we briefly cover issues related to the inferred groups, such as schema creation and merging, group broadcast, maintenance and interaction.

Group Schema Creation: As mentioned earlier, the inference of the interest group schema is achieved gradually by merging the schemas of peers in consecutive steps. The goal of this procedure is to produce a schema that represents the majority of the peers that belong to the respective cluster. Therefore, the merged schema is neither the intersection nor the union of the members of the cluster. Assuming that such a cluster comprises of numerous peers, it is straightforward that the intersection of their schemas would probably be empty and their union would be too large. Thus, our merging procedure has to incorporate only the most "popular" elements of the respective peer schemas in the merged schema.

Yet, inferred schemas should also be representative of almost all their source peer schemas, therefore our merging procedure should also perform high compression before discarding schema elements. Finally, we note that the whole procedure is based only on available information in the peers, i.e., schemas and mappings between them. Specifically, we assume that peer mappings are GAV/LAV/GLAV and peer schemas are relational, (i.e. the only internal mappings are foreign key constraints). One mapping is considered to be a set of 1-1 correspondences between attributes that hold with an optional set of value constraints on some attributes. Moreover, peers do not carry semantic information about their schemas and mappings. Following is the description of the merging algorithm:

Input: the merged schema S_{IG}; the peer schema S_P and a set of mappings M between them; a set of intra-schema mappings M_i

Output: the new merged schema S'_{IG}, a set of mappings M' with the following node on the path, a set of intra-schema mappings M'_i and a dictionary D

Initialization: $S_{IG} = \emptyset$, $M = \emptyset$, $M_i = \emptyset$, $D = \emptyset$

On each peer of the network path perform the following steps:

Step1: Add to S_{IG} all the relations of S_P

Step2: If $M = \emptyset$ set $S'_{IG} = S_{IG}$ and go to step 7

Step3: Merge relations that share the same key

Step4: While the number of relations is over the limit do:

a. Select pairs of relations that have the most correspondences between their attributes and that do not depend on value constraints

b. From pairs of (a) select the pairs of relations that have the fewest not mapped attributes and merge them

c. Remove from M the mappings used for the merge of (b) and add the involved correspondences in the dictionary D

Step5: Set $S'_{IG} = S_{IG}$, $M'_i = M_i \cup M$

Step6: Select the next node, P', of the network path from the acquaintees of peer P. Set M' equal to the set of mappings between P, P'. Change attribute and relation names of P in M' to the respective names in the merged schema S'_{IG}

Step7: Send S'_{IG}, M'_i and M' to P' □

Steps 3 and 4 refer to the merging of a pair of relations. The following procedure performs the merging of two relations:

Input: A pair of relations R_1, R_2 a set of mappings M

Output: The merged relation R

Initialization: $R = \emptyset$

Step1: Add to R all attributes of the relations R_1, R_2

Step2: Until the number of attributes is above the limit, if it is possible do:

a. if there are any, merge attributes that are involved only in one correspondence; else go to b

b. merge the attributes that are involved in at least one correspondence, starting from those participating in the fewest correspondences □

At the end of the schema merging procedure, i.e., when all relevant peer schemas have been merged, relations and relation attributes that have been met very rarely during the procedure can be dropped.

Example: Assume that Dr Davis is a doctor with a peer database the schema of which is:

$S_{DavisDB}$:

Visits($\underline{\text{Pid}}$, $\underline{\text{Date}}$, $\underline{\text{Did}}$)

Disease ($\underline{\text{Did}}$, DisDescr, Symptom)

Treatment ($\underline{\text{Did}}$, $\underline{\text{Drug}}$, Dosology)

And Dr Lu is another doctor with a peer database, the schema of which is:

S_{LuDB} :

Sickness($\underline{\text{Did}}$, AvgFever, Drug)

Patients($\underline{\text{Insurance}\sharp}$, $\underline{\text{Did}}$, $\underline{\text{Age}}$, Ache)

The schemas of DavisDB and LuDB are presented in Figure B.4. The databases have the following mapping:

$M1_{LuDB\_DavisDB}$:

Disease (Did, _, Symptom), Treatment (Did, Drug, _):-Sickness(Did, AvgFever, Drug), where the correspondences Symptom = AvgFever and Disease = Sickness are implied.

In this case, as shown in Figure B.5, there are three correspondences that are encapsulated in mapping $M1$. We assume that the peer of Dr Davis initializes the schema merge. Thus, S_{IG} is initialized to $S_{DavisDB}$. After the 1st step of the schema merging algorithm, S_{IG} contains all the relations of $S_{DavisDB}$ and S_{LuDB}. Since there is a mapping among the relations,the algorithm goes on to Step3: relations *Disease* and *Sickness* are merged

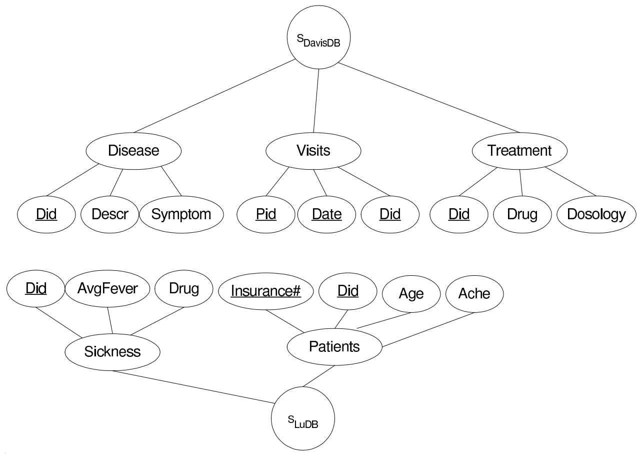


Figure B.4: Two schemas to be semantically merged

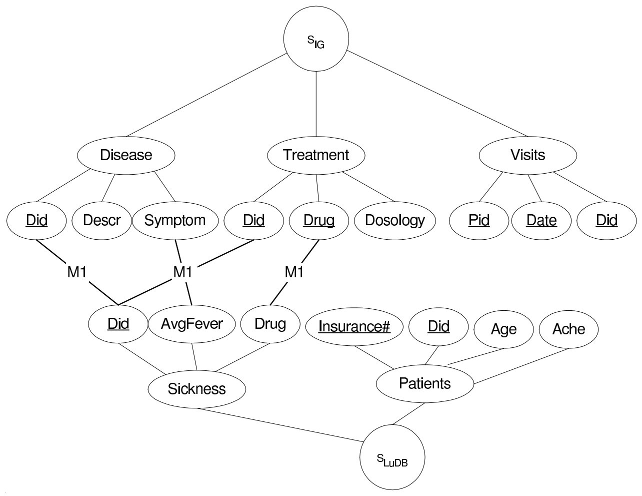


Figure B.5: S_{IG} is initialized to $S_{DavisDB}$ and there is mapping $M1$ between S_{IG} and S_{LuDB}

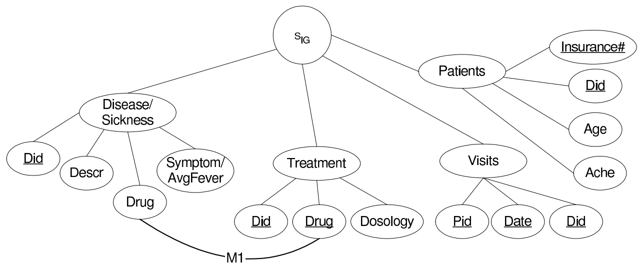


Figure B.6: Relations Disease and Sickness of Figure B.5 are merged

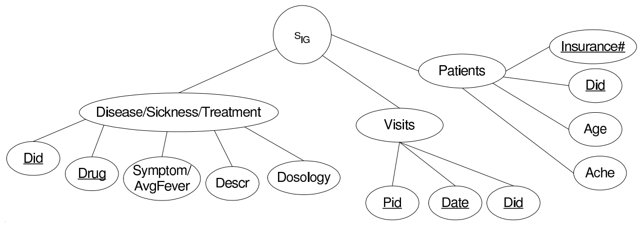


Figure B.7: Relations Disease/Sickness and Treatment of Figure B.6 are merged

in one, since they share the same key (*see* Fgure B.6). Thus, attributes *Symptom* and *AvgFever* are merged. The correspondence $Disease/Sickness.Drug = Treatment.Drug$ is kept as an internal one. Also, the dictionary D is enriched with correspondences $Disease = Sickness$ and $Symptom = AvgFever$; actually the schema keeps one name for each relation or attribute from the alternative ones. At the end of the schema merging procedure we propose that the schema keeps for relation and attribute names the most common ones encountered during the procedure.

Assuming that the algorithm goes on to Step4, relations $Disease/Sickness$ and *Treatment* are merged (Figure B.7), since they are the only ones related with a mapping. Now there is one attribute named 'Drug' and it is part of the relation key, even though just one of the attributes that where merged was a key. Additional iterations can merge relations based on foreign key constraints, since no other internal mappings exist.

The schema merging procedure produces the interest group schema but also a set of internal mappings and a dictionary. The internal mappings are the peer mappings that were not consumed in the successive schema merges. These hold additional syntactic and implicitly semantic information for the interest group schema elements; thus, they can be very helpful to peers that would like to join the group and create mappings with their local schema. Moreover, this set of mappings has the collection of all mappings with value constraints met during the merging procedure. These kind of mappings cannot be consumed: the involved relations/attributes cannot be merged, since they are mapped under certain conditions (the value constraints).

Group Broadcast: After a group schema is created, metadata about this group is periodically propagated across the overlay. This metadata includes the group schema, some or all of the IDs of participating nodes (*contact list*), the time of creation and the originator. Any peer in the system can rewrite its queries to the group schemas available. Queries can then be directly forwarded to the group members. In this way, we manage to bypass the information loss of multiple rewritings, since a query is translated only once, through the group schema. Making the participating nodes known to all peers enables any remote node to enter the cluster. Peers can now become acquainted with group nodes that have very similar schemas with them, without having to wait to be gradually clustered.

Group Interaction and Merging: While our completely decentralized approach in group creation is necessary, it also raises some consistency issues, since more than one groups can be created, even simultaneously. This can affect correct behavior only if nodes similar to the initiator choose to create a group *and* the two processes overlap in the overlay. Topologically close peers initiating the process over different semantic groups pose no problem. The same is true if the initiators' hop distance is such that would not allow either procedure to incorporate both groups in its progress.

In order to avoid extended negotiation rounds between competing potent originators, we require that initiators announce their intention to create a group to their neighborhood. In effect, this forces competing initiators with schemas similar to the first initiator to postpone or abort their process, if they are inside the announcement neighborhood. We note that the announcement neighborhood must have a radius proportional to the semantic radius of the group to be inferred. If such peers do not eventually participate in the group inference, they can add themselves to the overlay neighborhood or participate in the consequent *maintenance* of the group.

Nevertheless, peers are eligible to initiate a new group if they have not received a relevant announcement or if they incorrectly calculate their similarity with a known initiator. It is possible that such originators will create groups that have a significant semantic overlap with existing ones. Thus, these groups are subject to be merged into a unified schema. After both groups are advertised, the respective originators can detect the similarity between the inferred schemas and initiate the merging process. This involves choosing a new originator among the two existing ones, merging the two schemas and advertising the new group using the new initiator and the union of the contact lists.

We must note that an important property must hold: Groups created by similar initiators will also be similar and groups by dissimilar initiators will be dissimilar. This is essential because it justifies that authoritative peers can independently initiate the process (and thus block other similar ones from doing it). *GrouPeer*'s clustering process assures that this property holds, something also evident in our evaluation.

Group Maintenance: The maintenance process refers to updates in the contact list as well as the group schema itself. Maintenance is necessary, since peers join the group while others that belong to the group leave or change their local databases in time. There are two ways to decide how to maintain a group schema: The first is to allow the originator to initiate the inference process periodically. The second is to allow *any* eligible peer re-start the process. In order for both approaches to work, we define an *epoch* factor to represent the maximum life-span of a group, after which it will become invalid. Then, the originator can invoke the inference process every *epoch* minutes and re-transmit the new group inside the overlay. This way, group metadata are kept in a form of soft state inside our network and get promptly updated. By allowing any eligible peer to undertake the role of the originator, we eliminate inconsistencies created by changes in the original initiator and also ensure that the inferred schema does not specialize. Obviously, there is a trade-off between the cost of repeating the process over the anticipated query performance using stale groups.

B.4 Experimental Evaluation

To evaluate the performance of *GrouPeer*, we use a message-level simulator written in C. By default, we randomly choose 100 nodes that play the role of the requesters, each making 100 queries to the system. We present results for 1,000-node random graphs (an adequate number of participants regarding our motivating application) with average node degrees around 4, created by the *BRITE* [46] topology generator. Results are averaged over 20 graphs of the same type and size, with 100 runs in each.

For the schemas stored at each node, we use two initial relational schemas, whose tables and attributes are uniformly distributed at nodes. The initial schema comprises of 5 tables and 33 attributes. Seven attributes are keys with a total of 11 mappings (correspondences) between them. Each peer stores 10 table columns (attributes) on average. Queries are formed on a single or multiple tables if applicable (join queries). We experimented with larger schemas (90 attributes over 12 tables) and a flat 100-attribute single table (no mappings between attributes). Because the creation of the individual schemas is computer-generated, an increase in the schema reduces the amount of the default similarity between nodes (unless more attributes are distributed per node). Nevertheless, the important observation is that, in all cases, *GrouPeer* maintains its relative advantages and behaves in a similar fashion.

Our basic performance metrics are the average similarity or *accuracy* of answers to the original queries (i.e., the similarity of the answered query over the original one evaluated at the requester), as well as the number of nodes that provide an answer.

B.4.1 Clustering Results

For the automatic rewriting of the original query, we simulate the possible erroneous outcome by altering the “perfect” rewriting by 50%. This is then gradually ameliorated through our learning process. We set the maximum number of allowed hops per query TTL=6, the number of deployed walkers $k = 3$, as well as $\theta_{P_I} = 0.7$ and $\theta_{P_I Low} = 0.3$ using a threshold parameter of *THR*=5 replies. Finally, we assume that the returned tuples do not play any role to the answer evaluation.

Figure B.8 shows the performance of our algorithm by varying the number of queries posed by each of the 100 randomly selected requesters. Our method manages to return far more accurate results, achieving a similarity of around 85% in the steady state. The accuracy increases fast as more queries are created, since new acquaintees are added and neighbors with no contribution are dropped. We also present the respective values for answering the original and the rewritten versions of the query. Both the original and

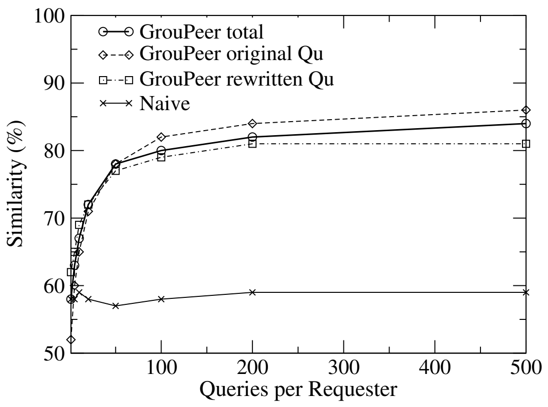


Figure B.8: Similarity of answers to the original and rewritten query versions over variable queries per requester

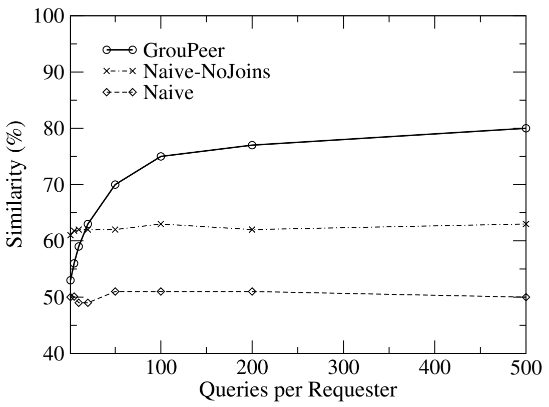


Figure B.9: Similarity of answers to join queries over variable queries per requester

the consecutive rewritten queries are answered with more precision. Our method's learning feature allows the automatic rewriting of the original query to improve over time as mappings are built between requester-replier pairs. Our clustering mechanism helps into bringing more information-rich nodes closer to requesters which also increases the accuracy of the consecutive rewritings. Our scheme is compared against *Naive*, which uses the same forwarding scheme as our method but answers only the successively rewritten query version. Our method can never fall below *Naive*'s performance but steadily performs better with more queries. Finally, it is almost as bandwidth-efficient as *Naive*, since the few additional messages reported are due to the communication between sources and requesters during the learning mechanism, as well as the message exchange when a new acquaintance is made.

Next, we monitor *GrouPeer*'s performance by specifically tracking join queries in the same setting as the previous experiment. Figure B.9 shows the results for our method and two different versions of *Naive*: The regular one we described before (which allows the rewriting of a join query even if the join is not mapped – like *GrouPeer*) and one

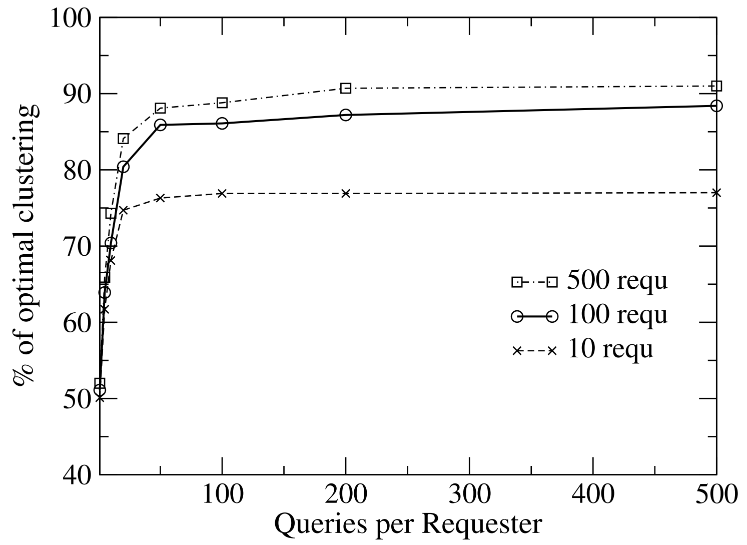


Figure B.10: Ratio of *GrouPeer*'s clustering versus the optimal, given an equal number of acquaintees

that returns an empty query if the join(s) are not preserved. As before, we notice that *GrouPeer* performs at least as good as the original naive method and quickly increases in accurate answers as more queries are generated. The more strict naive method returns more similar results for few queries compared to our scheme. This happens as this method favors a complete (and thus more accurate) rewriting. Nevertheless, this comes at a cost of retrieving an answer from about 1/3 of the peers that *GrouPeer* gets answers from.

We also examine the quality of the clustering process as a means of locating nodes with similar schemas. For each requester, we measure the average similarity with its acquaintees at the end of the querying process and compare it with the best possible scenario: Having all top-*m* nodes in the overlay with schemas most similar to the initiator being its acquaintees, where *m* is equal to the total number of acquaintees this node has at the end of the querying process. We report the ratio of the actual average similarity to this optimal value in Figure B.10.

Table B.1: Performance varying the number of query attributes

	Similarity	**Clustering**
$attr = 2, queries = 100$	0.87	80.2%
$attr = 2, queries = 500$	0.89	82.1%
$attr = 4, queries = 100$	0.80	86.1%
$attr = 4, queries = 500$	0.84	88.4%
$attr = 6, queries = 100$	0.71	83.0%
$attr = 6, queries = 500$	0.76	84.5%
$attr = 8, queries = 100$	0.67	80.0%
$attr = 8, queries = 500$	0.71	81.0%

Our methodology achieves clustering that is very close to the best achievable value in the steady state, while its quality quickly reaches that level. As more nodes become active, the process improves, since in *GrouPeer* nodes can take advantage of their neighbors' knowledge/connectivity. The ideal restructuring is hard to be achieved because of the random initial connectivity: The most similar nodes may not all receive queries and thus are not considered by the clustering process. Specifically, nodes may either be outside the query range or be left out of walkers' paths. By having more active nodes, our method effectively reduces the influence of the latter, since query initiators get replies by better nodes, taking advantage of other requesters' clustering. Figure B.10 shows that in the steady state and with 10, 100 and 500 requesters, *GrouPeer* achieves 77%, 88% and 91% of the optimal clustering respectively. We can identify 88% of the optimal nodes in the entire network by having only 10% active nodes and each of them contacting at most $k \times TTL = 18$ nodes per query (this amounts to less than 2% of the peers).

Table B.1 summarizes the performance of *GrouPeer* with a different number of query attributes (each requester making 100 or 500 queries). As the number or attributes per query increases, the accuracy of the answers slightly drops, since a smaller percentage of attributes has the chance to be satisfied. Note that the quality of the clustering increases up to a point, after which it starts to slightly decrease. This is due to the fact that there are two competing factors that affect the clustering process: The more attributes in a query, the more precise the clustering process becomes, since the initiator learns more information for its schema as a whole; the query similarity (which affects clustering through the Ev function), on the other hand, decreases with the number of attributes.

We tested our method in graphs of different sizes (from 100 to 4K nodes) and different connectivities (power-law). Results of these runs are qualitatively similar to the presented ones.

B.4.2 Group Inference Results

In this section we present results on the group schema creation of *GrouPeer*. Our basic setup remains the same, with the exception that queries on created groups are reformulated using the inferred schema(s). Our metrics are the percentile increase/decrease in accuracy and number of replies compared to clustering as these are measured on the *first* created group. The maximum size of the inferred schema is always in the order of the size of the initial schema used to produce the local ones during start-up. When the first group is created, we direct relevant queries to the inferred schema and measure their similarity compared to the clustering produced at the time of group creation. Initiators that belong to the group hold the complete mappings with the group schema, avoiding reformulation errors. Non-members utilize the same learning feature as with normal nodes, assuming a

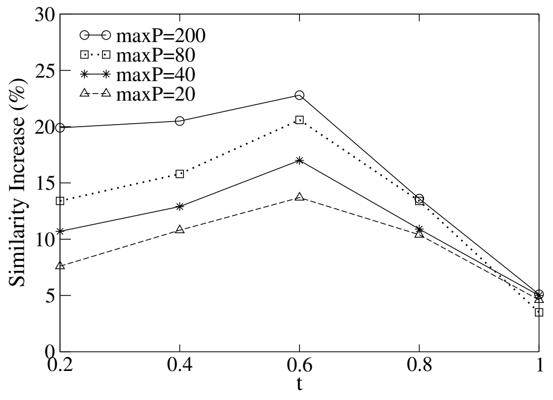


Figure B.11: % Increase in answer similarity over variable MaxP and t

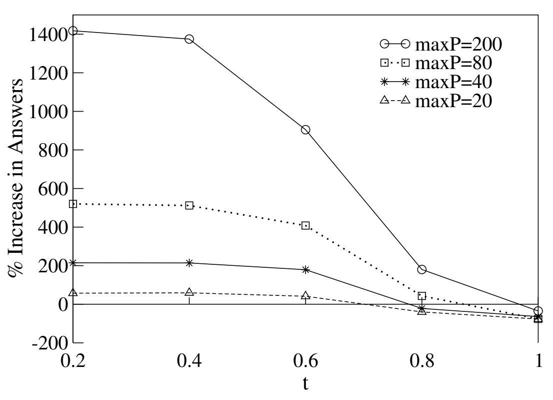


Figure B.12: % Increase in number of answers over variable MaxP and t

"virtual" host holding the group schema as their contact.

First, we vary the maximum group size limit, *MaxP*, as well as the minimum similarity of participating peers to the initiator node, t. Figures B.11 and B.12 show the obtained results for 100 requesters and maximum 100 queries each. As t increases, the group becomes more specialized and less general. In contrast, small similarity values produce groups too general that incorporate many concepts foreign to the initiator. Initiators choose to send queries to a schema if they deem it advantageous. This has the effect that *specialized* groups (i.e., high value of t) receive fewer queries, while more "general" ones receive more but cannot answer them all satisfactorily. Thus, there exists a point where grouping ceases to increase its relative gains to clustering, as our graphs show.

Both metrics increase as *MaxP* increases. This is reasonable since more nodes can participate and produce results. Very specialized grouping causes significantly less populated groups, which in turn affects the number of returned answers. As groups get more general (around $t = 0.6$), an improvement of 13-23% in accuracy is achieved, while the gains in replies are 40-900%. As t decreases, the gains in accuracy decrease but more

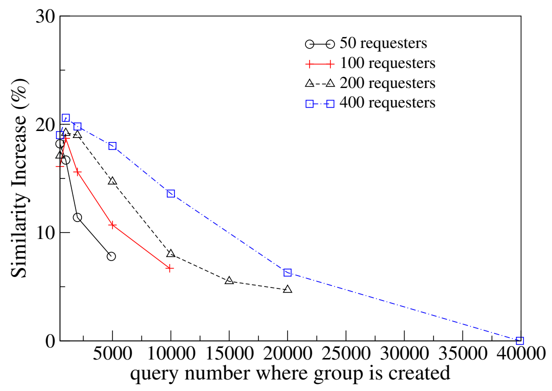


Figure B.13: % Increase in answer similarity over variable group creation time

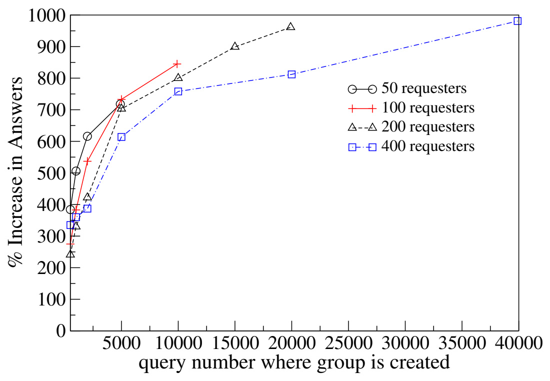


Figure B.14: % Increase in number of answers over variable group creation time

results are generated. These curves show that a t value of around 0.65 with the group initiator and $MaxP = 80$ achieve good results without too much generalization. These will be our default values for the rest of this discussion. Also, in all experiments, we set $T_o = 0.7$.

Next, we try to determine the quality of the created group based on its creation time, i.e., the number of queries at which it was created. Figures B.13 and B.14 show the percentile improvement in our basic metrics when the first group is created at various points in the clustering process. Our observations show a decrease in the relative gains in accuracy and an increase in the corresponding number of answers. This happens because clustering improves with time while the number of results slightly decreases due to the forwarding process: now more walkers cross paths on relevant nodes. What is important is that groups that are allowed to be created as soon as possible (which would be the frequent case) show about 20% more accurate answers and return about three times more results compared to clustering, even though the inference procedure is performed on a less optimally clustered overlay. Groups that are created later exhibit noticeable gains, espe-

Table B.2: Performance comparison with clustering

	100 requ		400 requ	
qu/requ	**Sim**	**#Answ**	**Sim**	**#Answ**
10	0.68 (+17.8%)	55.0 (+411%)	0.70(+19.9%)	53.7 (+387%)
50	0.71(+17.7%)	51.8 (+370%)	0.71(+19.0%)	61.6 (+461%)
100	0.72(18.2%)	55.8 (+413%)	0.72(+19.2%)	60.0 (+444%)

cially in terms of the number of replies. When more requesters are active, the clustering process is expedited, which suits the purposes of grouping.

Table B.2 shows the exact performance figures using our default parameters for various requesters/queries-per-requester combinations. The figures in parentheses show the percentile increase compared to simple clustering for the same number of queries. We notice that querying the inferred groups results in an average 18% increase in accuracy and around 400% increase in number of replies. This is true regardless of the requesters or their querying rates. It is interesting to note that, in all these results, the queries from nodes inside the created groups are less than 10% of the total. This proves that group creation and propagation effectively helps all nodes in the overlay.

One of the basic assumptions of our scheme is that each peer can individually choose to initiate the group inference process. This allows for completely distributed behavior only if semantically close initiators produce similar groups and the opposite. We measure the similarity between the first and randomly selected thereafter initiators as well as of the group schemas created respectively. Figure B.15 displays results over different runs, where either the two initiators were over 70% or less than 40% similar. Clearly, for very similar initiators the process yields very similar groups. On the other hand, for fairly dissimilar initial schemas, the created groups are 40-50% similar. This value is a

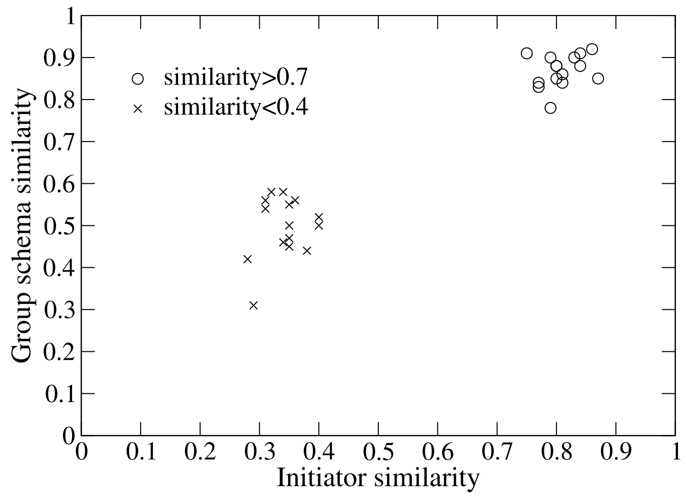


Figure B.15: Relationship between initiator and inferred schema similarity

little higher than expected due to the high overlap and semantic relations between stored attributes at various peers. When data is placed in a non-overlapping manner, such groups have less than 20% similarity. So, there clearly exists a correlation between initiator and inferred schema similarity value.

As we just showed, peers with similar schemas generate similar groups. To do so simultaneously is undesirable for two reasons: First, the system will perform a redundant operation and second, it will force our merging process to be invoked regularly. As we mentioned in Section B.3.2, initiators broadcast their intention to create a semantic group. Nevertheless, broadcasts that reach many nodes are very costly. Furthermore, our clustering process assures that a non-negligible number of semantically close nodes will also be close to the initiator in the hop-distance metric. To demonstrate this, we measure the hop-distance distribution of peers not included in the group creation process with similarity greater or equal to D to the initiator, given our default parameters. Table B.3 presents our results.

We notice that the minimum distance increases as we search for more similar peers,

Table B.3: Estimating group broadcast range

	D = 0.5	D = 0.7	D = 0.8	D = 0.9	D = 1.0
Min/Max Distance	1.1/5.9	1.9/5.6	2.1/5.3	2.9/4.8	3.8/4.2
#nodes	597	235	113	27	17
%nodes $\leq$ 4 hops	78	80	80	78	70

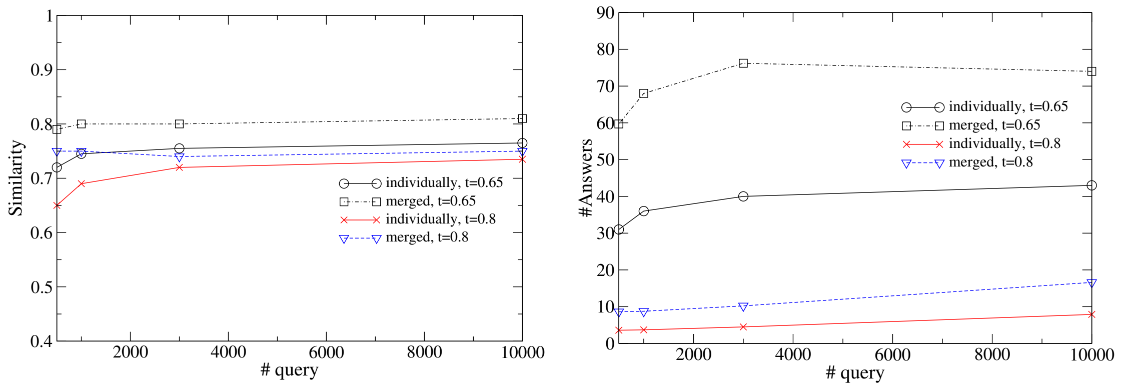


Figure B.16: Similarity and number of answers of the initial and merged groups vs creation time

while the maximum decreases. This is due to the clustering process: Similar peers get closer in the overlay. Grouping includes most of these peers, so the minimum distance to a non-grouped similar node increases. Moreover, the ones that have been left out of the group inference are now closer than before. The results show that a broadcast range of 4 contacts around 80% of our target nodes. Nevertheless, as D increases, these nodes become scarce. Thus, assuming that $D \simeq 0.65$ for practical reasons, a TTL=4 would suffice. In our experiments, a broadcast of that scope blocks an increasing number of nodes with time. For larger values of D, broadcasting with large range causes the majority of messages to be delivered to dissimilar peers.

Finally, we present some results concerning *GrouPeer*'s merging process. When two similar groups are identified (through broadcasting of the group metadata), the merge process is initiated. We measure the similarity and number of replies by the two groups as well as the merged one and present the results in Figure B.16. We notice that, while the two groups and the merged one do not substantially differ in the accuracy of the results (although the merged group always outperforms them), the new schema delivers almost twice as many. A very important observation is that the time of creation of the individual groups plays almost no role in their performance, which shows that *GrouPeer* will keep operating without performance degradation.

B.5 Related Work

The Chatty Web [106] considers P2P systems that share (semi)-structured information. The authors are concerned about the gradual degradation, in terms of syntax and semantics, of a query propagated along a network path. This approach considers peers that own very simple relational schemas and GAV mappings with their acquaintees. Instead, we are interested in more complex schemas and we consider GAV, LAV or GLAV mappings.

In [104], the authors propose optimization techniques for query reformulation in P2P database systems. They focus on minimizing the rewriting of a query and pruning the propagation path in order to avoid redundant reformulations. It is indicated that pre-computation of the query reformulation path-tree proves to accelerate the procedure despite the disadvantage of the necessary maintenance of pre-computed mappings. Our approach is designed for large-scale unstructured overlays. First, it evades reformulation at peers poor in query-relevant information by adaptively choosing the version of the query to be answered. Also, while in [104] central knowledge of the system structure is required, our scheme enables nodes to operate in a completely decentralized fashion, utilizing the standard lookup operations to refine their local knowledge.

PeerDB [107] features relational data sharing without schema knowledge. Query matching and rewriting is based on keywords. First all nodes within a TTL radius are contacted, returning prospective answer meta-data. Then the user selects those that are relevant to the query and the selected sources are contacted directly for the results to the various rewritten versions of the query. Instead, our approach employs an automated technique based on a combination of successive query rewriting and query-schema matching, while it utilizes bandwidth-efficient walks compared to the costly flooding scheme.

Some of the well-known projects that deal with the data heterogeneity problem in P2P systems are [108–111]. Piazza [108] presents a solution to the heterogeneity issue in P2P data management systems and proposes a language for schema mediation between peers. It also presents algorithms for query reformulation based on GAV/LAV query answering.

Edutella [109] is a schema-based network that holds RDF data. Peers have services (e.g. querying, mapping, mediating etc) that they share with other peers. Peers can formulate complex queries that are translated in wrappers to queries on the Edutella Common Data Model. Peers register the query-types they can answer to mediators, which route queries to appropriate peers. Edutella is an effort towards the solution of the heterogeneity problem of data and services. However, it does not focus on semantic clustering, neither does it propose sophisticated methods for distributing queries to semantically relevant peers.

GridVine [110], and pSearch [111], are based on a structured P2P overlay. Grid-

Vine hashes and indexes RDF data and schemas, and pSearch represents documents as well as queries as semantic vectors, which are the keys of a DHT structure. Both these projects base search efficiency on the underlying DHT, and, thus, do not solve the semantic diversity problem in an unstructured P2P system. Another disadvantage of p-Search is that documents of newly-joined peers, with terms that are not encapsulated in the existing vector, cannot be indexed by them.

Beyond semantic clustering, the work in [112] looks into the problem of discovering connectivity clusters of nodes in P2P networks, detecting the transmission of the same query multiple times at the same node. In [72], peers are grouped into *possession rules*, according to whether they contain a specific item or not. Nodes search inside one possession rule in a blind fashion. The possession rule is chosen by a greedy mechanism according to past query results.

B.6 Summary

GrouPeer is a system that effectively implements both popular approaches of answering queries in P2P data management systems: Propagation along paths of bounded depth and querying a mediated schema. First, it performs a gradual formulation of semantically similar clusters. Our system creates and maintains, in an automated way, a schema representative of the cluster. Requesters can direct relevant queries to the advertised groups and join relevant (or interesting) ones in a more effective and timely manner. Our results show that grouping results in significant gains in both the answer quality and quantity compared to the original clustering method.

BIBLIOGRAPHY

[1] Clay Shirky. What Is P2P ... And What Isn't. *OpenP2P.com*, 2000.

[2] A. Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.

[3] RFC 1036. Standard for Interchange of USENET Messages.

[4] RFC 1034. Domain Names - Concepts and Facilities.

[5] http://www.napster.com. Napster website.

[6] http://www.gnutella.com. Gnutella website.

[7] http://www.jxta.org. Project JXTA.

[8] http://www.microsoft.com/net. Microsoft .NET.

[9] The impact of file sharing on service provider networks. An Industry White Paper, Sandvine Inc.

[10] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science*, 2001.

[11] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, 2001.

[12] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *SIGCOMM*, 2001.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. Technical Report TR-00-010, University of Berkeley, CA, 2000.

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

[15] D. Tsoumakos and N. Roussopoulos. Adaptive Probabilistic Search for Peer-to-Peer Networks. In *3rd IEEE Intl Conference on P2P Computing*, 2003.

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

[17] D.Tsoumakos and N. Roussopoulos. Analysis and Comparison of P2P Search Methods. In *INFOSCALE*, 2006.

[18] D.Tsoumakos and N. Roussopoulos. AGNO: An Adaptive Group Communication Scheme for Unstructured P2P Networks. In *Euro-Par*, 2005.

[19] D. Tsoumakos and N. Roussopoulos. APRE: An Adaptive Probabilistic Replication Method for Unstructured P2P Networks. In *CoopIS*, 2006.

[20] D. Fallows, L. Rainie, and G. Mudd. The popularity and importance of search engines, 2004. ComScore data Memo.

[21] J. Chu, K. Labonte, and B. Levine. Availability and Locality Measurements of Peer-to-Peer File Systems. In *SPIE*, 2002.

[22] V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti. A Local Search Mechanism for Peer-to-Peer Networks. In *CIKM*, 2002.

[23] S. Rhea and J. Kubiatowicz. Probabilistic Location and Routing. In *INFOCOM*, 2002.

[24] D. Menascé and L. Kanchanapalli. Probabilistic Scalable P2P Resource Location Services. *SIGMETRICS Perf. Eval. Review*, 2002.

[25] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems. In *INFOCOM*, 2003.

[26] A. Crespo and H. Garcia-Molina. Routing Indices for Peer-to-Peer Systems. In *ICDCS*, July 2002.

[27] S. Daswani and A. Fisk. Gnutella UDP Extension for Scalable Searches (GUESS) v0.1.

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

[29] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[30] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.

[31] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-law Relationships of the Internet Topology. In *SIGCOMM*, 1999.

[32] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Infocom*, 1996.

[33] C. Jin, Q. Chen, and S. Jamin. Inet: Internet Topology Generator. Technical Report CSE-TR443-00, Department of EECS, University of Michigan, 2000.

[34] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of NOSSDAV*, 2001.

[35] J. Jannotti, D. Gifford, K. Johnson, F. Kaashoek, and J. O'Toole. Overcast: Reliable multicasting with an overlay network. In *OSDI*, 2000.

[36] A. Datta, M. Hauswirth, and K. Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *ICDCS*, 2003.

[37] Y. Chu, S. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay muilticast architecture. In *SIGCOMM*, 2001.

[38] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *SIGCOMM*, 2002.

[39] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *NGC*, 2001.

[40] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. *Lecture Notes in Computer Science*, 2001.

[41] Marius Portmann and Aruna Seneviratne. Cost-effective broadcast for fully decentralized peer-to-peer networks. *Computer Communications*, 26, 2003.

[42] A. Ganesh, A. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comp.*, 2003.

[43] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM*, 2002.

[44] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC*, 1996.

[45] M. Ripeanu and Ian Foster. Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems. In *IPTPS*, 2002.

[46] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An Approach to Universal Topology Generation. In *MASCOTS*, 2001.

[47] A. Ganesh, A. Kermarrec, and L. Massoulie. SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Networked Group Communication*, 2001.

[48] http://tidy.sourceforge.net/. HTML Tidy Project Page.

[49] http://www.kazaa.com. Kazaa website.

[50] http://www.emule project.net/. eMule project.

[51] http://www.bittorrent.com/index.html. BitTorrent home page.

[52] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *PODC*, 2002.

[53] http://web.icq.com/. ICQ web site.

[54] J. Kangasharju, K. Ross, and D. Turner. Secure and Resilient Peer-to-Peer E-Mail: Design and Implementation. In *IEEE Intl Conf. on P2P Computing*, 2003.

[55] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In *WWW*, 2002.

[56] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally Distributed Content Delivery. *IEEE Internet Computing*, September 2002.

[57] M. Freedman, E. Freudenthal, and D. Mazires. Democratizing Content Publication with Coral. In *NSDI*, 2004.

[58] http://www.squid-cache.org/. Squid Web Proxy Cache.

[59] M. Roussopoulos and M. Baker. Practical load balancing for content requests in peer-to-peer networks. Technical Report cs.NI/0209023, Stanford University, 2003.

[60] C. Damgaard and J. Weiner. Describing Inequality in Plant Size or Fecundity. *Ecology*, 81, 2000.

[61] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Lecture Notes in Computer Science*, 2001.

[62] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. Replication, Load Balancing and Efficient Range Query Processing in DHTs. In *EDBT*, 2006.

[63] http://www.openp2p.com. openP2P website.

[64] http://www.peer-to-peerwg.org/. Peer-to-Peer working group.

[65] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP, 2002.

[66] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. Technical Report UW-CSE-01-06-02, Un. of Washington, 2001.

[67] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. In *SIGCOMM Internet Measurement Workshop*, 2002.

[68] S. Saroiu, K. Gummadi, R. Dunn, S. Gribble, and H. Levy. An Analysis of Internet Content Delivery Systems. In *OSDI*, 2002.

[69] B. Yang and H. Garcia-Molina. Improving Search in Peer-to-Peer Networks. In *ICDCS*, 2002.

[70] M. Stokes. Gnutella2 Specifications Part One: http://www.gnutella2.com/gnutella2_search.htm.

[71] P. Ganesan, Q. Sun, and H. Garcia-Molina. YAPPERS: A peer-to-peer lookup service over arbitrary topology. In *INFOCOM*, 2003.

[72] E. Cohen, A. Fiat, and H. Kaplan. Associative search in peer to peer networks: Harnessing latent semantics. In *INFOCOM*, 2003.

[73] R. Morselli, B. Bhattacharjee, M. Marsh, and A. Srinivasan. Efficient Lookup on Unstructured Topologies. In *PODC*, 2005.

[74] D. Subramanian, P. Druschel, and J. Chen. Ants and reinforcement learning: A case study in routing dynamic networks. In *IJCAI*, 1997.

[75] G. Di Caro and M. Dorigo. AntNet: Distributed Stigmergetic Control for Communications Networks. *Journal of Artificial Intelligence Research*, 9:317–365, 1998.

[76] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *SIGMETRICS*, 2000.

[77] P. Francis. Yoid: Extending the internet multicast architecture, 2000. White Paper.

[78] M. Castro, M. Jones, A. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *INFOCOM*, 2003.

[79] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, 1987.

[80] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *DSN*, 2001.

[81] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[82] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *SOSP*, 2001.

[83] J. Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.

[84] M. Waldvogel, P. Hurley, and D. Bauer. Dynamic replica management in distributed hash tables. Technical Report RZ–3502, IBM, 2003.

[85] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *IPTPS*, 2002.

[86] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems. In *ICDCS*, 2004.

[87] W. Poon, J. Lee, and D. Chiu. Comparison of Data Replication Strategies for Peer-to-Peer Video Streaming. In *ICICS*, 2005.

[88] R. Ferreira, M. Ramanathan, A. Awan, A. Grama, and S. Jagannathan. Search with Probabilistic Guarantees in Unstructured Peer-to-Peer Networks. In *IEEE P2P*, 2005.

[89] F. Cuenca-Acuna, R. Martin, and T. Nguyen. Autonomous Replication for High Availability in Unstructured P2P Systems. In *SRDS-22*, 2003.

[90] http://www.overnet.com/. eDonkey2000-Overnet.

[91] Bram Cohen. Incentives build robustness in bittorrent, 2003.

[92] Rob Sherwood, Ryan Braud, and Bobby Bhattacharjee. Slurpie: A cooperative bulk data transfer protocol. In *IEEE Infocom*, 2004.

[93] Pablo Rodriguez and Ernst W. Biersack. Dynamic parallel access to replicated content in the Internet. *IEEE/ACM Transactions on Networking*, 10(4), August 2002.

[94] John W. Byers, Michael Luby, and Michael Mitzenmacher. Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In *INFOCOM*, 1999.

[95] http://www.morpheus.com. Morpheus website.

[96] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical loss-resilient codes. In *STOC*, 1997.

[97] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *JACM*, 36(2), 1989.

[98] P. Felber, T. Kaldewey, and S. Weiss. Proactive hot spot avoidance for web server dependability, 2004.

[99] A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust p2p system to handle flash crowds. In *ICNP*, 2002.

[100] M. Ripeanu and I. Foster. Mapping the Gnutella Network: Macroscopic Properties of Large-Scale Peer-to-Peer Systems. In *IPTPS*, 2002.

[101] V. Kantere, I. Kiringa, J. Mylopoulos, A. Kementsientidis, and M. Arenas. Coordinating P2P Databases Using ECA Rules. In *DBISP2P*, 2003.

[102] M. Lenzerini. Data Integration: A Theoretical Perspective. In *21th ACM PODS*, 2002.

[103] V. Kantere, D. Tsoumakos, T. Sellis, and N. Roussopoulos. GrouPeer: Dynamic Clustering of P2P Databases. Technical Report DBLAB–2006/4, National Technical University of Athens, Department of Electrical and Computer Engineering, 2005. http://www.dbnet.ece.ntua.gr/pubs/.

[104] I. Tatarinov and A.Halevy. Efficient Query Reformulation in Peer-Data Management Systems. In *SIGMOD*, 2004.

[105] J. Madhavan and A. Halevy. Composing mappings among data sources. In *VLDB*, 2003.

[106] K. Aberer, P. Cudre-Mauroux, and M. Hauswirth. The Chatty Web: Emergent Semantics Through Gossiping. In *WWW Conference*, 2003.

[107] B. Ooi, Y. Shu, K.L. Tan, and A.Y. Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *ICDE*, 2003.

[108] A. Halevy, Z. Ives, D. Suciu, and I. Tatarinov. Schema Mediation in Peer Data Management Systems. In *ICDE*, 2003.

[109] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. Edutella: A p2p networking infrastructure based on rdf. In *WWW*, 2002.

[110] K. Aberer, P. Cudre-Mauroux, M. Hauswirth, and T. Van Pelt. Gridvine:Building internet-scale semantic overlay networks. In *International Semantic Web Conference*, 2004.

[111] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM*, 2003.

[112] L. Ramaswamy, B. Gedik, and L. Liu. A Distributed Approach to Node Clustering in Decentralized Peer-to-Peer Networks. In *IEEE Transactions on Parallel and Distributed Systems*, 2005.

Index