THESIS

DECENTRALIZED AND DYNAMIC COMMUNITY FORMATION IN P2P NETWORKS AND PERFORMANCE OF COMMUNITY BASED CACHING

Submitted by
Chepchumba Soti Limo
Department of Electrical and Computer Engineering

In partial fulfilment of the requirements
For the Degree of Master of Science
Colorado State University
Fort Collins, Colorado
Summer 2015

Master's Committee:

Advisor: Anura P. Jayasumana

Liuqing Yang
Christos Papadopoulos

ABSTRACT

DECENTRALIZED AND DYNAMIC COMMUNITY FORMATION IN P2P NETWORKS AND PERFORMANCE OF COMMUNITY BASED CACHING

Distributed Hash Tables (DHT) are commonly used in large Peer-to-Peer networks to increase the efficiently of resolving queries. Minimizing the resource discovery time in P2P networks is highly desirable to improve system-wide performance. Distributed caching is an approach used to reduce the look-up time. File sharing P2P networks have shown that there exists nodes/users who share similar interests based on semantics, geography, etc., and a group of nodes that share similar interests are said to form a community. A Community Based Caching (CBC) algorithm where nodes make caching decisions based on personal interests is investigated. One of CBC's major contributions is that it alleviates the issue of nodes being limited to caching resources that are popular relative to the entire network. Instead, caching decisions are primarily based on a node's community affiliations and interests. Community discovery algorithms that currently exists either need a centralized source(s) to aid in community discovery or require additional messaging and complicated computations to determine whether to join a group or not. In many cases, nodes are also limited to being members of only one community at a time. A dynamic and decentralized community discovery algorithm, Dynamic Group Discovery (DGD), is proposed. DGD also allows nodes to be members of multiple communities at the same time. DGD's behavior and performance is then evaluated in conjunction with the Community Based Caching algorithm. To aid in group discovery during run time (i.e., dynamically), DGD uses special keys with embedded group identification information. Oversim, a flexible overly network simulation framework is used to evaluate the proposed DGD algorithm. Performance of DGD is compared to Chord and Static Group Allocation (SGA), in which group

identification is done only once. Performance is evaluated for different network sizes, community sizes, and asymmetry among communities. Performance results are presented and analyzed when queries are resolved using cache data versus when queries are resolved using non-cache data. The analysis shows that DGD generally improves lookup performance when cache data is used to resolved queries. However, when non-cache data is used, DGD occasionally performs slightly worse than Chord and SGA. For example, in a network with 10,000 nodes, asymmetrical communities and no churn group churn, DGD outperforms Chord by approximately half a hop and 0.1 seconds in latency. When churn was introduced to the same network, DGD performance drops by approximately one hop and 0.15 seconds in latency. The results also show that approximately 90% of the queries are resolved using non-cache data and therefore, even though DGD is guaranteed to reduce lookup time when asymmetrical communities are present and cache records are to used to resolve queries, it is often not enough to significantly improve overall system performance. The results however confirm that caching resources based on personal interests really does reduced lookup performance when resolving queries using cache records.

ACKNOWLEDGMENTS

I am elated to have completed my research work and this accomplishment would not have been possible without the many people who have helped me along the way. First and foremost, I would like to express my deepest and utmost gratitude to my advisor Dr. Jayasumana. He gave me the opportunity to work under him and always found a way to chuck the best in me. I am also very grateful for his patience, guidance and encouragement when my research was not looking very promising. His fresh perspectives, rigorous reviews and constructive criticism were invaluable and I would not have made it this far without him.

I would also like to sincerely thank my other committee members: Dr. Liuqing Yang and Dr. Christos Papadopoulos who were not only my professors, but also amazing mentors. Their advice and perspective is largely the reason I made it through graduate school. My deepest gratitude to my colleagues: Dr. Vidarshana Bandara, Yi Jiang, Negar Mshorraf and Aly Fathi Boud for their comments on my work and the many laughs we shared which helped create an amazing work environment. A very special thank you to two of the most special friends I have made in Fort Collins: Aisha Jama and Donnyale Ambrosine, who have played instrumental roles in molding me into the woman I have become.

Finally, I would like to sincerely thank my parents and siblings. I am forever indebted to them for the sacrifices they made for me and for their unending and unconditional love and support that brought me this far. Thank you and am truly blessed to call you my family.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Introduction

Peer-to-peer (P2P) network applications have evolved from the days of predominantly using it for file sharing purposes to now having a vast array of applications that include digital crypto currency applications like Bitcoin [1]; free movie streaming multimedia applications such as the recently discontinued Popcorn Time [3]; video conferencing applications such as Skype [35]; and even online gaming sites like World of Warcraft [36], only to name but a few.

One way of guaranteeing improved system-wide performance in P2P networks is by reducing the amount of time a node/user takes to find a resource in the network. In an attempt to minimize resource discovery time, various caching schemes such as Beehive [37] and PoPCache [38] have been proposed. However, [37] and [38] are designed to favor the most popular requests and/or resources with respect to the entire network. It is also worth noting that popularity of queries in P2P networks generally follow a Zipf's distribution [2] [40], which essentially means that there are few items which are very popular resulting in a large number of queries looking for them, while a large number of items that are significantly less popular thus accounting for a very small number of queries. Therefore, if Beehive or PoPCache are selected as the caching scheme of choice, then queries to the popular resources will be favored and majority of the nodes will not benefit from caching.

Within a P2P network, there exist nodes with similar interests and a cluster of such nodes are said to belong to a community [29]. For example, BitTorrent [41] has a large variety of user groups with varying interests such as music, movies and software etc., and these users form

associated clusters within the BitTorrent network. Similarly, Tomnod, a website that harnesses the power of crowdsourcing to identify satellite images with the aim of solving real world problems, has projects in various areas of interest and users who choose to be involved are essentially joining a community of people who are trying to achieve a common goal. An example of a well-known Tomnod project is that of attempting to locate the disappeared Malaysia Airlines MH370 [5] [6] [7].

The existence of communities in the network can be used to implement "smart" routing which yields improved quality of search results [10] and also lowers resource discovery (also referred to as lookup) times. A proactive caching algorithm that exploits the existence of communities is presented in "Community-Based Caching for Enhanced Lookup Performance in P2P Systems" [29]. In [29] sub-overlays, based on pre-assigned static community identification, are formed and the proposed Community Based Caching (CBC) scheme (a potential replacement for Beehive and PoPCache) is proven to enhance lookup performance in P2P networks.

1.2. Motivation

Community behavior is dynamic in that they appear, grow, diminish and eventually disappear over time and this phenomenon has been observed repeatedly in the Internet. For example, the Tomnod project where a community of users identified possible airplane remains of the missing Malaysia Airlines MH370 began with the unfortunate disappearance of the aircraft [5] [6] [7], grew as the news of the incident spread across the globe and then began to die out as media coverage also reduced, and finally the project was closed and with that the community disappeared. A second example on community formation, growth and disappearance in the Internet is that of the rumor about the discovery of a new particle with similar characteristics as

the Higgs boson; which is considered the most elusive particle in modern science and whose existence has been hypothesized since 1964 [8] [9]. Domenico et al., [9] analyzed the number of Twitter messages associated with the rumor between July 1st and July 7th 2012 for their study. In their findings, the number of messages grew steadily as the rumor spread with the peak observed around July 4th when the official announcement about the discovery was made. After July 4th, the number of Twitter messages steadily declined and over time and fewer messages on the rumor were observed.

Nodes joining a network tend to have varying interests; they may even have multiple interests and as such should be allowed to be members of multiple communities at the same time. For example in standard file sharing P2P network, a node that joins may be interested in music, software and movies. Ideally, such a node should be able to join the music, software and movie communities at the same time.

In simulation and testing of the Community Based Caching (CBC) algorithm [29], a community is viewed as a group of nodes that query the same website (e.g. BitTorrent) for resources. Even though there is commonality between such nodes, i.e., querying form the same website, BitTorrent has access to a wide variety of resource. Therefore, the main limitation of viewing communities from this perspective is that nodes are unable to take advantage of the fact that they may have similar interests at a lower level other than that of querying the same website. For example, let there be a *Node_A* that is interested in music and movies and another *Node_B* that is interested in software and computer games. If both *Node_A* and *Node_B* query site *X* for resources, then according to [29], *Node_A* and *Node_B* belong to the same community. However, based the definition of a community above, this is not very accurate since *Node_A* and *Node_B* have different interests at a lower level.

Also, there are two more assumptions made in [29] to simplify the simulation and testing of the Community Based Caching (CBC) algorithm that need to be modified in order to truly test the efficiency of CBC: 1) community assignments of nodes was done statically at the beginning for the simulation, and 2) nodes were limited to being members of one community at a time. These two assumptions, are not in line with the basic principles of community existence discussed above, i.e., they should appear and disappear dynamically and that a node could be a member of multiple communities at a time.

In conclusion, the Community Based Caching algorithm is a novel concept that has been proven to enhance lookup performance when communities in the network are static and restricted to belonging to one group. However, as demonstrated by the examples given above, appearance and disappearance of communities is essentially a dynamic process and it is imperative for nodes to have the ability to be a part of multiple communities at the same time. Therefore, it is necessary to ensure that the CBC algorithm truly enhances lookup performance when nodes are allowed to join communities dynamically and also have the ability to be a part of multiple communities at the same time.

1.3. Problem Statement and Contribution Summary

Community Based Caching has been proven to work for nodes in the network that belong to one community for the entire lifetime of the node [29]. This thesis proposes a novel approach for community formation that takes into consideration the dynamic nature of communities and also allows nodes to belong to multiple communities at any given time. In addition, the proposed community formation algorithm involves having special keys created using a novel approach

which allows for unsupervised appearance and disappearance of communities because no static community assignment is done.

1.4. Outline

The rest of this thesis is organized as follows. Chapter 2 reviews the background and related work. We expound on the problem statement in Chapter 3. In chapter 4, a detailed discussion of the Dynamic Group Discovery algorithm is presented. Chapter 5 covers the simulation implementation of the Dynamic Group Discovery algorithm and also includes case studies and results used to determine the algorithm's optimal region of operation. Chapter 5 also provides simulation results that compare Community Based Caching algorithm performance when communities in the network are statically assigned versus when communities are formed dynamically. Summary and future work is given in chapter 6.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1. Introduction

Existence of communities in P2P networks has been established in numerous publications. Since group presence in the network is a characteristic that can be exploited for gains in different areas, it has been studied extensively and as result multiple community discovery algorithms have been proposed. In this section, we shall first look into understanding community behavior in networks in order to take advantage of all known community characteristics to develop a decentralized community discovery algorithm. Next, we shall review some publications that motivated the proposed Dynamic Group Discover algorithm (DGD). In addition, since DGD is dependent on a novel key generation technique, will also look into existing key generation procedures that have been used in P2P networks. Lastly, because the proposed DGD algorithm is built on top of an existing P2P architecture, we shall also look into the common types of P2P structures that are currently in existence.

2.2. Understanding Community Behavior in Networks

This section looks into research done on communities in the Internet, in an effort to: 1) understand natural community existence and formation in the network, and 2) pin-point characteristics that can be used to design an efficient algorithm that exploits existence of communities and their properties.

Before diving into the details of community behavior in networks, we will give a formal definition of communities that will apply whenever “communities” or “groups” are referred in

this thesis. A community is simply defined as a subset of peers or nodes with similar interests [29]. Figure 2.1 below shows a structured overly network with 12 nodes labeled A to K. From the figure, there are two communities (also referred to as suboverlays) that have formed on top of the already existing overlay. This is shown by the solid blue line and the dashed black line. Notice that the nodes connected with the solid blue and dashed black line are a subset of the nodes that make up the network. Also note that not all nodes in the network belong to a community for example node B and node F. Another example is illustrated in Figure 2.2 below, which shows an unstructured overlay that consists of nodes which are grouped into three communities. Therefore, regardless of whether the network is structured or unstructured, a cluster of nodes that share similar interests are referred to as a community.

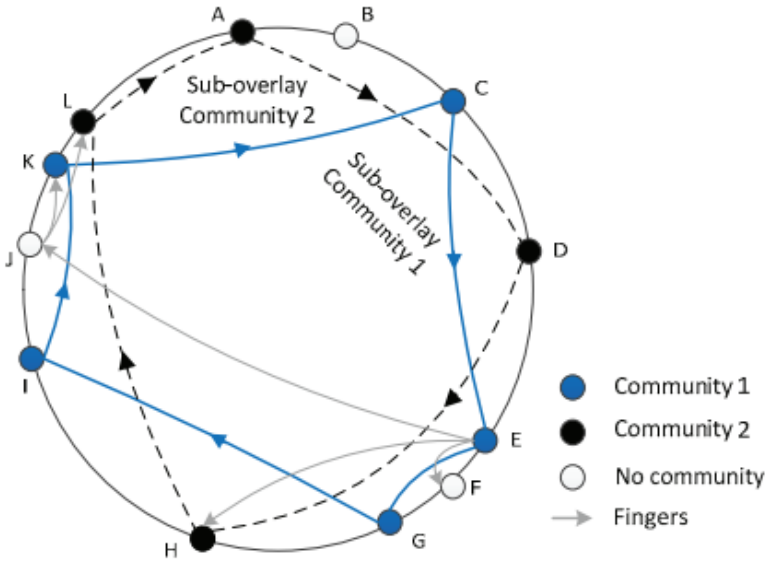

Figure 2.1: A structured overly network (chord) with two communities formed on top of the overlay [29].

Next we look at the three main causes of correlation in social networks as described in [11] which consequently leads to grouping of nodes with similar interests. These causes include: 1) homophily, which is the tendency to befriend others who perform similar actions [12]; 2)

influence/induction, which suggests doing something because a friend did it, for example buying a pair of shoes; and 3) environment, which suggests sharing the same information with people living in the same geographical area. The causes highlighted above can be viewed as a basis of community existence and identification in networks. Even though the findings are based on studies done on social networks, the proposed bases above could be generalized and applied to the entire Internet and particularly to P2P networks.

The existence of strong and weak ties in the network also influence natural community existence, discovery and expansion of the same. Strong ties, which are considered connections to neighbors that are used regularly, exist between neighbors who have a lot in common. Conversely, weak ties are connections between nodes that are not regularly used. Weak ties are valuable because they are believed to have more diverse information about resources available in the network and also tend to exist between nodes that occupy different sections of the graph [13]. This is attributed to the fact that the tie frequency of communication is directly proportional to mutual interest between nodes [13]. Weak ties are therefore crucial in discovering nodes in the network that may share similar interest so as to potentially form mutually beneficial relationships such as communities.

Another characteristics of naturally occurring communities in a network is the existence of densely linked clusters. This is partly due to society's natural way of dividing people into social groups for example families, villages, etc., [18][19]. In the World Wide Web's, topically related pages link more densely among themselves creating communities [21]. Within the clusters, there exists densely connected nodes and some nodes in a cluster form loose connections with other nodes in different clusters [14][15]. These loose inter-cluster connections

ensure connectivity throughout the network and that no cluster is isolated from the network as demonstrated in Figure 2.2,2 below.

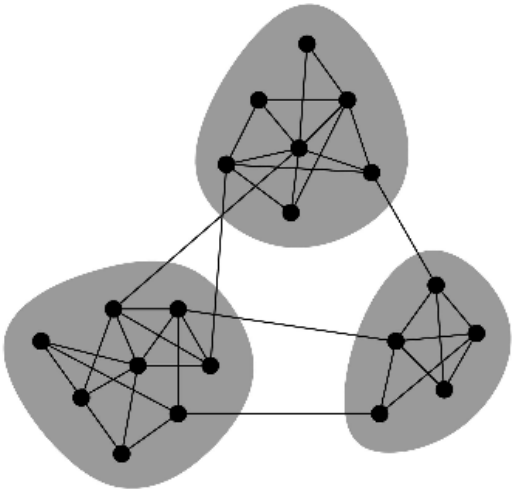

Figure 2.2: The three shaded regions represent three different naturally occurring communities that have dense connections within but loose connections with other nodes in different communities [15].

The definition of a cluster as described above does not account for situations where there is overlap between communities; which is also a naturally occurring phenomenal in networks as described in [16] [17]. Therefore, there is a possibility of having dense inter-cluster connections and loose internal links within a cluster [20] as illustrated in Figure 2.3 below.

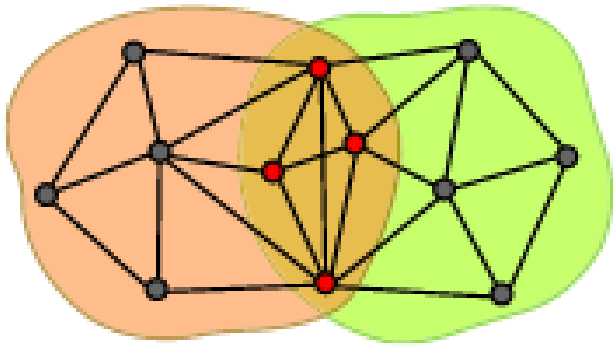

Figure 2.3: Illustration of two overlapping communities with dense external links.

In summary, the four general characteristics of naturally occurring communities in networks described in this section are:

i. Correlation of nodes due to homophily, influence/induction and environmental influences
ii. Existence of strong and weak ties in the network (and more so the existence of weak ties) that tend to disseminate information about more diverse information available in the network
iii. Natural occurring clusters due to society's natural way of dividing people into social groups and
iv. The natural occurrence of overlapping communities in the network

These characteristics are used as guidelines in the design of the proposed decentralized DGD algorithm that is discussed in Chapter 4.

2.3. Key Generation Techniques Used in P2P Networks

Most structured P2P networks utilize a Distributed Hast Table (DHT) with <*key, value*> entries that are used to assign ownership to resources [4]. Resources are represented using <*key, value*> can be assigned using centralized key generating infrastructure or consistent hashing algorithms such as SHA-1 in [4]. Based on our research, there are no key generating algorithms that include information about the type of content represented by the key; for example, does the generated key represent a music file or software etc.

In order to facilitate dynamic discovery of communities in the network, it becomes imperative for nodes to be able to learn about groups at run time, and preferably not have to query a central source for this information. The idea of decentralized key generation is proposed

in [27], which demonstrates that it is possible to have a key generating algorithm that is publicly known by all nodes in the network and still have secure communication in the network, even when up to 75% of the network edge is compromised; for example, when nodes are controlled by an adversary.

Since there is a constant stream of messages containing keys being forwarded in P2P networks, it becomes increasingly attractive to have these keys contain some form of meta-data appended to them using a key generation algorithm that is publicly known by all nodes in the network. This could be advantageous because when a node receives a query, it can extrapolate information about the key and consequently learn about other interests in the network at runtime. This information could also potentially aid in decentralized community discovery at runtime.

2.4. Previous Work Aimed at Community Discovery in P2P Networks

Comtella [45] is a small scale P2P file sharing system used to develop a trust-based community formation approach to help users get more relevant information from searches done in a network. In Comtella, a user can search for papers and/or comments, and based on the ratings of papers in the results lists the user can decide what paper is worth reading. However, ratings are biased in that users can rate papers at their discretion and not everybody has the same interests or the same level of background knowledge to rate papers "accurately." To overcome this usability problem, a community formation algorithm based on trust is proposed in [22]. Every node in the network has an agent that is used to track similarity in ratings given by users and an agent is limited to tracking ratings in only one category. The community is maintained by the creator and a node will prefer to build a community that is only useful it itself because community maintenance needs dedicate resources. Before building a community, the creator

starts by building trust in other agents and this is done by experience and once it finds agents it can trust, it invites them to join its community C_{NEW}. If an agent that is invited to join C_{NEW} is already part of another community C_0, it can suggest to the creators of both C_{NEW} and C_0 to join and form a bigger community. Members of each community decide if joining communities is good based on trust for each other, and if majority of the members vote not to combine, the individual agents decide whether to join C_{NEW} or C_0 based on trust. Community maintenance is done by the creators and it is done periodically. If an agents reputation is tainted and falls below the specified threshold to maintain trustworthy status, or if new agents not cleared by the creators have joined the community and are found to be untrustworthy, they are expelled from the community by the creator [23]. The main limitations of this algorithm is that community maintenance is centralized and nodes are limited to being members of one community per category.

Liu et al., [24] propose an algorithm where communities can be established using a hierarchical structure. Web domain visitations and frequency of visits are used as the basis to determine a node's interest profile. An order-based approach is presented to determine similarity between nodes in the network such that if the similarity measure between P_i and P_j is greater than a certain threshold Ω then P_i can determine, with a specified confidence level q, that P_j is considered to be among the top (1-p) quantile most similar nodes in the network. P_i chooses its threshold value Ω such that if similarity value calculated is greater than Ω then the node will be among its top (1-p) fraction of nodes in its community. The initiator node specifies N random walks to peers in the network where N is the number of samples needed by the node to calculate its Ω, p and q value. Whenever a peer is chosen, it sends a reply to the initiator and an end-to-end connection is established. After Ω, p and q are set, the initiator node sends out a discovery

message with a set time-to-live defining maximum number of hops. If a node P_k's similarity with the initiator is within a specified range, it sends a reply to the initiator and also forwards discovery message to all its neighbors except the source of the message. The initiator analyzes P_k's similarity and decides whether or not to send an invitation based on the quantiles estimated above. Once P_k gets the message, it either accepts or declines it. If it accepts, initiator is added to P_k's local cache. With an aim of expanding the network, if an invitation is accepted, then both nodes exchange each other's cache contents [24].

A robust self-adjusting algorithm that has the capacity to form communities and search for resources efficiently is proposed in [25]. Data inserted into the network is arranged in abstract categories called information profiles. When a node A wants to find other nodes with similar interests, it sends out a discovery message with the desired information profile. The message is routed using either flooding [33] or greedy routing, which in this case means preferring to forward messages to neighbors in your community over regular neighboring nodes. Each node that receives the message calculates the similarity between the profile in the message and its own, determines a similarity measure and compares this measure with a predetermined threshold. A search hit occurs when a node B determines that the similarity measure is greater than or equal to the threshold. If a search hit occurs, then a direct connection is made between the nodes A and B. B then replicates the discovery messages and forwards them to its neighbors. The idea behind proliferation of the discovery message at the node where the hit occurred is that, since nodes with similar interests tend to be present in the same neighborhood, then the likelihood of other nodes in the network discovering similar communities is increased. Each node is limited as to how many edges it can have connected to other community members. If both A and B reach their

maximum at the same time, then the edge AB is deleted. If either A or B reach its maximum, then another random edge is selected at random and deleted.

Khambatti et al. [26] propose a community discovery algorithm on which the node interests are represented using attributes which can be classified as either personal, claimed or group. All interests of a node are classified under personal, and the claimed attributes are a subset of personal attributes that represent interests the node chooses to make public because by default all attributes are private for security purposes. Nodes determine group attributes based on geographic location, e.g., if a node is at the Colorado State University campus, then it is part of the CSU group. Before a community of nodes is fully formed and considered stable, nodes send out group discovery messages that include a set of its claimed attributes. When a peer receives the group discovery message, it finds the intersection between the set of claimed attributes in the message and its personal attributes. If a personal attribute in the resulting set is not claimed, the interest is then added to the claimed. This is done because it is beneficial for a node to make public as many attributes as possible in order to take advantage of the benefits of being in a community but at the same time having the ability to only share information that it wants to make public. Community membership/joining is based on weighted calculations done by the node to determine if the node should join a community it has discovered.

In conclusion, the algorithms summarized above either need a centralized node to maintain community formation or need to send out additional messages in order to discover nodes with similar interests in the network. The proposed DGD algorithm discussed in Chapter 4 utilizes messages that are already being propagated in the network to disseminate information about existing communities. Piggy-backing group information in key also alleviates the need for additional information profile match calculations to determine potential community members.

2.5. P2P Architectures

There are two main types of P2P architectures, structured P2P and unstructured P2P. Nodes in structured P2P networks use a specified algorithm to determine where to forward queries whereas in unstructured P2P networks, flooding [33] is mostly utilized. The latter can be expensive and does not have any guarantees in terms of performance. Figure 2.4 below from [32] illustrates the unpredictable nature of flooding versus Chord [4], Pastry [30] and Content-Addressable Networks (CAN) [31]. The last three are algorithms are all types of structured overlay P2P networks.

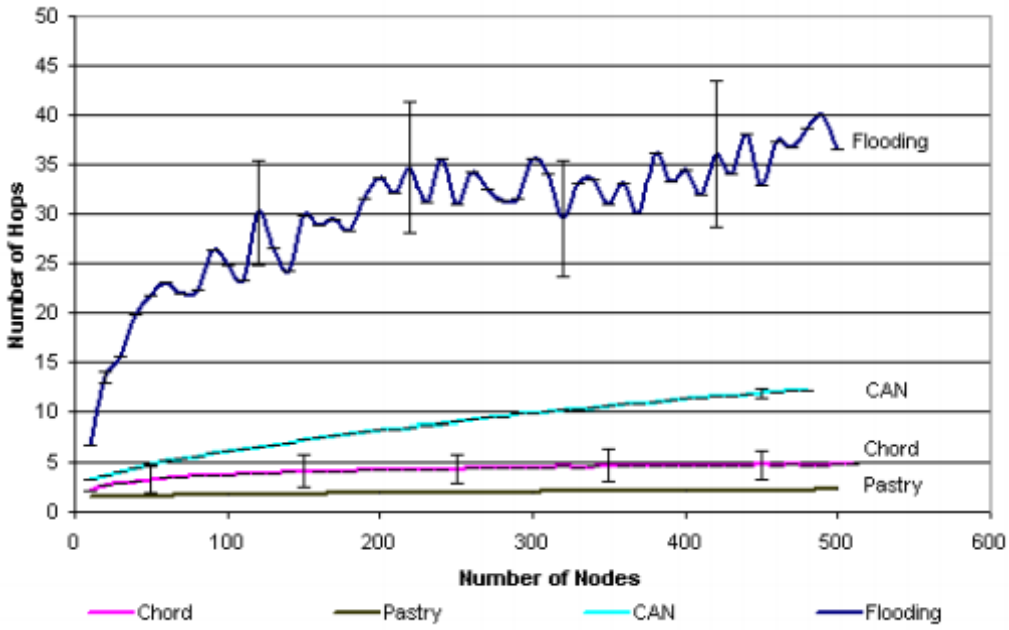

Figure 2.4: Cost of discovering resources in terms of average number of hops against varying network sizes using different forwarding algorithms.

In conclusion, a decentralized dynamic group discovery algorithm that utilizes already existing messages being forwarded in the network is desired and from the discussion above, is achievable. A public key generation algorithm can exist without compromising the security of the other nodes in the network. Designing the algorithm on top of a structured network would be

more beneficial versus that over an unstructured network because of the guarantee in performance given by the former.

CHAPTER 3

PROBLEM STATEMENT

3.1. Introduction

Studies discussed in Chapter 2 show that communities occur naturally in the network, and in an aim to minimize resource discovery times, exploiting the existence of communities is increasingly desirable. This section will cover three main methods establishing similarity between nodes in P2P networks. These include and may not be limited to: commonality between nodes based on querying from the same website; a node's personal interests; and acquired interest due to queries processed by a node. We conclude this section by reiterating the problem statement that drives the algorithm discussed and evaluated Chapter 4 and 5 respectively.

3.2. Community Identification Based on Websites Queried

In [29], communities are defined as a subset of nodes that share similar interests. For example, if a group of nodes are constantly accessing the local news website for updates, then these nodes can be said to belong to the news community. Similarly, in file sharing P2P networks if a node queries the same website, for example BitTorrent [41] or The Pirate Bay [43] for files, then these nodes can be grouped together and identified as belonging to either the BitTorrent or The Pirate Bay community.

3.3. Community Identification Based on Personal Interests

Nodes join P2P networks to either query for resources, make resources available or both. For example, a *Node_A* joins the network and not only would it like to query for music files, but

it also has a movie file that it is willing to share with other nodes in the network. A second *Node_B* would like to query for music files while a third *Node_C* would like to query for movie files. In this case, ideally *Node_A* and *Node_B* would join the music community based on their personal music interest and similarly, *Node_A* and *Node_C* would join the movie community. Note that the decision to join a community is independent on whether a node would like to share or query for a resource.

3.4. Community Identification Based on Acquired Interests

The last type of community identification scheme we shall look at is based on acquired interest. Depending on where a node is located in the network graph and the algorithm used to forward queries, a node could decide to join a community based on the number of queries it receives for keys that belong to a particular group. For example, if *Node_A* joined the network and it is interested in querying for music files, then naturally based on its personal interest *Node_A* would join the music community. In addition, because of *Node_A*'s location in the network it is constantly being requested to resolve queries for keys that belong to the movie community and therefore, *Node_A* could join the movie community to potentially improve lookup performance for movie files. If *Node_A* chooses to join the movie community, it would be because of an acquired interest largely due to the type of query requests processed.

3.5. Summary

Communities occur naturally in the network and it is desirable to take advantage of this phenomenon to potentially improve resource lookup times. To accomplish this, a decentralized

community discovery algorithm that does not require complex computation and/or multiple additional messages between nodes is desired. The proposed Community Based Caching (CBC) scheme in [29] was proven to work for: 1) communities that are static (i.e., a node could not change interests), 2) nodes that are restricted to belonging to one community at a time and 3) similarity basis for community formation was based on websites queried as described in Section 3.1 above. To further validate CBC's capabilities of enhancing lookup performance in P2P networks, it is tested in conjunction with the proposed Dynamic Group Discovery (DGD) algorithm whose design goals are to have a robust decentralized and dynamic community discovery algorithm that allows nodes/users to: 1) be members of multiple communities at the same time, 2) add, join or leave communities at will and 3) have similarity basis for communities based on personal interests and acquired interests as described in Section 3.2 and 3.3 above.

CHAPTER 4

DYNAMIC COMMUNITY FORMATION IN P2P NETWORKS

4.1. Introduction

This chapter covers the algorithm used to achieve dynamic community formation in P2P networks. First, the proposed key generation technique is introduced. Second, details on how establishing group interest is covered and this is followed by logic used to maintain group links in the dynamic network environment. Finally, we conclude the chapter with a summary of the proposed algorithm.

4.2. Key Generation and Group ID Finger Table Mapping

The approach used in the community discovery algorithm described below is dependent on resources being represented using a special key value. This approach is inspired by the user profiles defined in [14] where it is important to create accurate profiles for each resource in the network to facilitate community discovery between nodes.

In the proposed key generation design, resources have a corresponding key value that is used to identify it. The last 12 bits of every key are reserved for interest identification purposes where there are three levels of identification; four bits for each level. The three levels of identification used are as follows:

1. Level 1 (mandatory) – reserved for general data classification e.g., music, movies, and software. The idea is that every key in the network can be associated with some high level class and this information is made available using this level.
2. Level 2 (optional) – reserved to specify geographical location e.g., USA, Canada etc.

3. Level 3 (optional) – reserved to specify genre e.g., comedy, hip-hop, drama etc.

Having more levels of identification would be beneficial because this would allow for more accurate community matchups. However, for out work three levels is considered sufficient and a good trade-off between the amount of detail included about the key (i.e., metadata appended at the end of each key) and the number of bits left to represent the actual key. Note that the latter dictates the actual size of the key space and consequently determines the maximum number of nodes that can exists in the network at a given time. Therefore, it is important to ensure that a sufficient number of bits is allocated to represent the key.

The first level is made mandatory because every key in the network can to be classified under some category for example music, software, movies etc. as seen in common P2P resource discovery sites such as BitTorrent [41] and Pirate Bay [42]. If the level identification is not available or applicable for level 2 and 3 then a 0 is assigned to it. Note that level 1 cannot have a 0 assignment since it is mandatory. Table 4.1 below shows examples of keys generated with appended group ID information.

Table 4.1: An example of keys to be inserted in the network.

Key	Group ID Level 1	Group ID Level 2	Group ID Level 3	Final Key
0123456789abcdef	*music* => 1	*Canada* => 2	*hip-hop* => 3	0123456789abcdef 123
a123456789bcdef0	*music* => 1	*N/A*=> *0*	*drama* => 9	a123456789bcdef0 109
b123456789bcdef0	*movies* => *3*	*USA* => 3	*comedy* => 6	b123456789bcdef0 236

In order to guarantee a certain level of performance, we opt to use a structured P2P network versus a random network for our simulations. The structured network selected for this thesis is Chord [4], which is popular and widely used in P2P networks. As part of the Chord

algorithm, each node maintains two tables, i.e., a finger table mapping and a distributed hash table (DHT) for keys it is responsible for.

To aid in dynamic community discovery, in addition to the two tables mentioned above, each node maintains a map of group(s) it is interested in and fingers to nodes belonging to the said group(s). This information is stored in a table we shall refer to as the group ID finger table. Nodes learn about the presence of other with similar interests in the network through regular messages forwarded to it. In the case of Chord, these messages include PUT and GET. PUT messages are used by nodes to insert keys into the network for resources that it chooses to make available the network, while GET messages are used by nodes to query for resources. An example of the group ID finger table is shown in Table 4.2 below.

Table 4.2: An example of entries in group ID finger table.

Group ID	Finger To Group Member
123	node A
109	node B
206	node C
345	node D

An important property of the entries in the group ID finger table is that initially the mappings begin as weak ties (as described in [14] and [15]) and depending on the type of queries the node receives, the entries have the potential to turn into strong ties.

4.3. Community Discovery Algorithm

4.3.1. Establishing Group Interest

When a node joins the network, it is assumed that it already knows what communities it would be interested in joining based on its personal interests. A node's personal interests are the

first entries to be inserted into the group ID finger table and they are represented using the group ID number and a finger to itself. These entries are very important in the algorithm because they form the basis for future entries. As an example, Table 4.3 below shows the first two entries in a node's group ID finger table. When the node (identified as node A) joins the network, it knows that it is interested in music from America and comedy movies from Europe.

Table 4.3: Example of a node's initial group ID finger table.

Group ID	Finger To Group Member
120	node A
239	node A
xxx	TBD
xxx	TBD

In accordance with the Chord algorithm, when a node is ready to make a resource available to the entire network, it sends out a PUT message containing the key representing the resource. Take for example a *Node_A* which would like to insert a key is whose value is 1120. The Chord algorithm guarantees the existence of a *Node_X* in the network that is responsible for keys in a specified range [α, β) such that $\alpha \leq 1120 \leq \beta$. When *Node_A* sends out the PUT message, it is forwarded towards *Node_X* and when *Node_X* receives the message, it adds an entry to its DHT with the key 1120 and a pointer to the *Node_A*. Forwarding of the message towards *Node_X* is aided by Chord's finger table which also guarantees that the PUT message destined for *Node_X* will be delivered within $\log_2 N$ hops, where N is the number of nodes in the network.

Note that there is a little overlap in the information stored in all the three tables discussed above. The whole point of creating the additional group ID finger table is to potentially reduce the cost of resolving queries by using the premise, "ask those who probably know first," and in

our case this translates to ask members within the community first before deciding to use Chord's prescribed finger table. The details of how forwarding decisions are made are covered below in Sections 4.2.2 and 4.2.3.

4.3.2. Maintenance of Group ID Finger Tables

This section focuses on how nodes grow and maintain the group ID finger table using GET messages. A *Node_A* receives a GET message querying for a key 2912, however, it is unable to resolve the query and therefore needs to forward it to another node in the network that may be able to resolve it. To do this, *Node_A* first extracts to the group ID appended to the query (in this case it is 912) to see if it is already a member of the group. Based on *Node_A*'s initial group ID finger table defined in Table 4.3.1.1 above, it is only interested in joining group 1xx and/or 2xx. Also note also that at this point, *Node_A* has not discovered any other nodes in the network with similar interest. Therefore, since it is unable to resolve the query and it does not know of any other nodes in group 9xx, *Node_A* uses its Chord fingers to determine the query's next hop and also logs that it has attempted to resolve a query belonging to group 912.

Node_A receives another GET message but this time the key being queried is 2122. Let us assume that *Node_A* is unable to resolve the query and so far there have been no updates to its group ID finger table. In this case, *Node_A* uses its Chord finger table to determine its next hop and logs that it has attempted to resolve a query belonging to group 122. This is also the first time that *Node_A* gets a hint that there may exist other nodes in the network that have similar interests.

As *Node_A* continues to process GET messages, as it keeps track of the number of processed queries based on group. Note that from above, *Node_A* is keeping track of messages

that belong to all groups regardless of its personal interests. When *Node_A* processes λ number of queries for a group it is interested in, for example 122, it attempts to discover other nodes in the network with similar interests by sending out a FIND GROUP message to its neighbors. If *Node_A* processes μ number of queries for a group it is not interested in, for example 912, this is an indication that *Node_A* occupies a position in the network where it is constantly being requested to process queries belonging to group 912 and therefore, it is in *Node_A*'s best interest to join group 9xx to potentially take advantage of forwarding queries to community members and Community Based Caching. To do this, *Node_A* adds an entry to its group ID finger table and also sends out a FIND GROUP message. Note that there are two different thresholds used i.e., λ for already existing interests and μ for acquired interests due to queries processed. λ is used to ascertain that indeed there are other nodes in the network with interests similar to a node's initial personal interest and μ is used to justify joining a group that a node was not initially interested in, with an aim of attempting to improve general system performance. Ideally $\lambda << \mu$ because a node should favor joining communities based on personal initial interest rather than acquired interest.

Table 4.4: Updated group ID finger table after *Node_A* decides to join group purely based on GET messages processed.

Group ID	Finger To Group Member
120	node A
239	node A
912	node A
xxx	TBD

If a FIND GROUP reply is received, then it will contain finger(s) to other nodes in the network that share similar interests and these entries are added to the group ID finger table. For example, if a FIND GROUP reply for group 122 contained fingers to *Node_X*, *Node_Y* and

Node_Z, then *Node_A* updates its group ID finger table accordingly and this signifies *Node_A* joining community 1xx which already has *Node_X*, *Node_Y* and *Node_Z* as members. Table 4.5 below illustrates this update. Note that a FIND GROUP reply could have information about more than 1 node in the network. This is because all the nodes in the network are trying to find communities it can join and if a node has been in the network longer than another, then the probability it will have more knowledge of existing communities is high.

Table 4.5: *Node_A*'s updated group ID finger table after receiving a reply for FIND GROUP request done for group 122.

Group ID	Finger To Group Member
120	node A
239	node A
912	node A
122	node X
122	node Y
122	node Z
xxx	TBD
xxx	TBD
xxx	TBD

Due to limited resources, the group ID finger table can only hold so many entries. To ensure maximum use of the group ID finger table, each node is limited to having σ number of fingers to nodes in the same group. In addition, each node keeps track of how often it uses each finger and whenever σ limit is reached, the least used finger is deleted to make room for a potentially better finger. If more than 1 node have the same frequency, then a node is picked at random and deleted. For example, if σ = 3 and *Node_A* discovers a *Node_H* belonging to group 135 and it would like to update its group ID finger table whose current state is represented by Table 4.6 below, entry 122 with finger to *Node_X* will be deleted and a new entry for group 135 to *Node_H* will be added. Similarly, *Node_A* discovers a group 545 to *Node_C*, since both

entries for group 122 pointing to *Node_X* and group 420 pointing to *Node_X* have only been used once, *Node_A* will pick one of entries at random and replace it with an entry for 545 to *Node_C*.

Table 4.6: Sample snap shot of *Node_A*'s group ID when it is full.

Group ID	Finger To Group Member	Frequency of Use
120	node A	N/A
239	node A	N/A
912	node A	N/A
122	node X	1
122	node Y	7
122	node Z	9
912	node X	20
235	node E	4
420	node G	1

Max

$\sigma = 3$ reached

Finally, whenever a node is ready to leave the network, based on the chord algorithm, a LEAVE message is sent and nodes in the network adjust their chord finger table and Distributed Has Tables (DHTs) accordingly. Similarly, nodes would be able to use the same LEAVE message to determine what entries in its group ID finger table are no longer valid. For example, if *Node_G* leave the network, the entry in *Node_A*'s group ID finger table is no longer valid. Therefore, whenever a node learns of another node's departure, it can also delete any associated entries in it group ID finger table.

4.3.3. Summary of Algorithm

Below is pseudo code summarizing the algorithm described above. The main addition to the algorithm highlighted below is that messages are not forwarded within the community

indefinitely. The number of hops when trying to resolve a query within a community is limited to$(\log_2 N)/2$. This limitation is put in place to guarantee a certain level of performance with our algorithm. If a query cannot be resolved within the community in half the time it would have taken Chord, then nodes result to using chord fingers to avoid worsening system performance.

```
// Maintaining group ID finger table
void forward(key, msg, nextHop*)
{
	if msg type = GET
	{
		extract group ID from key;
		if group ID finger table // is there interest
		{
			if finger not pointing to me and hops < (log2 N)/2
			{
				set nextHop using entries in group ID finger table;
			}
			else
			{
				Use chord to set nextHop;
			}
		}
		else
		{
			Keep track of specific GET message;
			Use chord to set nextHop;

			if specific GET messages received λ OR μ times
				Send FIND GROUP request;
		}
	}
}

void handle_FINDGROUP_response(groupID, finger)
{
	if <group ID, finger> pair already exist // done to avoid duplicates
		return;
	if group ID finger table is at capacity
	{
		if one least used finger can be identified
			delete it;
		else // i.e., multiple fingers with same low frequency use number
			pick one at random and delete;
	{
	add new found finger;
}
```

CHAPTER 5

SIMULATION AND PERFORMANCE ANALYSIS

5.1. Introduction

In this section details of the simulation implementation used to test the Dynamic Group Discovery algorithm as described in Chapter 4 is discussed. Oversim [36], an event driven and flexible network simulation framework is used. This section will cover the following: how keys for specific simulation runs were created; query generation for a simulation run based on the keys created; group discovery and formation at runtime; how group churn is introduced in the network at runtime; steps taken to validate the simulator and finally case studies used to present results and analyze DGD's performance.

5.2. Key Creation

40 character long keys are generated using a programming script. As discussed in Chapter 4, the first 37 characters are reserved for the key hash value and the remaining 3 characters are reserved for group identification information. The following subsections detail steps taken to generate keys and queries that were used for simulations.

5.2.1. Determining Size and Symmetry of Community

Communities can either be symmetrical (i.e., all the same size) or asymmetrical. In an attempt to reach the desired symmetry at runtime, the number of keys belonging to a particular group is limited.

If symmetrical communities are desired, the maximum number of keys K that can belong to a particular group is simply given by $K = \frac{Number\ of\ Keyes\ to\ be\ Generate}{Desired\ Number\ of\ Groups}$. On the contrary, if K is

```
// K_1, K_2, … , K_n represent max number of keys in group 1, 2 , … , n

newMax = totalDesiredKeys;

K_1 = random_number(1, newMax);
newMax = totalDesiredKeys - K_1;

K_2 = random_number(1, newMax);
newMax = totalDesiredKeys - (K_1 + K_2);

…

// For last group
K_n = totalDesiredKeys - (sum of all groupMax set up to this point);
```

5.2.2. Key Generation

The key hash (i.e., the first 37 characters of each key) consists of randomly selected hexadecimal numbers. After the 37 character key is generated, group information is selected and appended to the key in order to create the desired 40 character key. To do this, a random number between 1 and the desired number of groups is selected for level 1 identification. If the selected number belongs to a group that has already reached its limit K, then another random number is selected at random. This process is repeated until a valid level 1 ID is picked. For level 2 and 3 identification, a random number between 0 and 9 is picked. Note that no additional checks are done for these last two levels because even though important, they are not involved in the big picture division of groups which will be discussed in the next section.

The keys created in this section are used to generate queries and this is discussed in Section 5.3 below.

5.3. Key Sorting and Query Creation

The queries used during a simulation are based on keys which already exist in the network. Querying for resources that are known to exist in the network guarantee that all queries will be resolved, allowing for accurate measurements of query resolution time. This section describes steps taken to create queries based on the keys that have been generated in Section 5.2 above.

The first step is to sort the keys from Section 5.2 into separate files based on their group ID. The granularity of sorting is limited to key type (i.e., level 1) information only. Sorting is done so that during runtime, nodes can know what query file to read from based on their interests. For example, if a node is interested in music it will be able to search for music files that already exists in the network by reading the query file with music keys. Also, for simulation purposes, nodes are limited to querying for resources that belong to groups of keys it has inserted into the network. For example, if a *Node_A* inserts keys into the network which belong to music and movie resources, *Node_A* will be limited to querying for music and movie files.

Sorting is also necessary since query popularity within different communities in the network are known to follow Zipf's distribution with different α parameters. Therefore, sorting of queries allows us to assign a different α to each community. It is also important to note that even though queries in individual communities follow Zipf's distribution, when all the queries in the network are aggregated, the result may not necessarily follow a Zipf's distribution. This is illustrated in Figure 5.2.1 below.

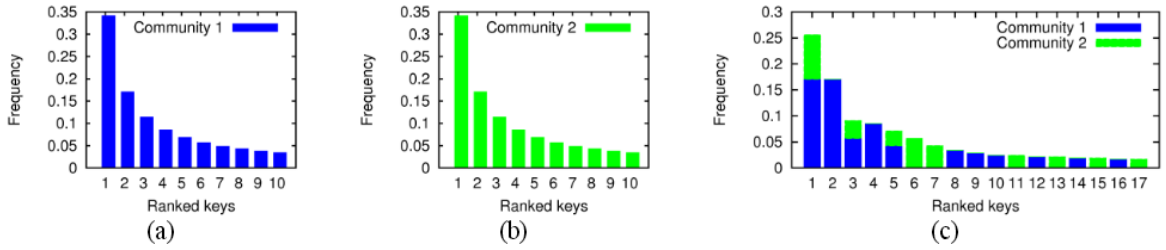

Figure 5.2.1: (a) Community 1; (b) Community 2; (c) Aggregate popularity distribution of both community 1 and 2.

After sorting, each key in the sorted file is assigned a frequency number f based on the Zipf's distribution with a specified parameter α. Ideally, different communities should be assigned a different α parameter [29], but to simplify the simulation, all groups are assigned the same $\alpha = 1.0$.

5.4. Runtime Simulation Logic Flow

In this section, the Overism implementation of the dynamic group discovery algorithm is discussed.

5.4.1. Network Initialization Inserting Keys

Network formation and initialization begins. This stage uses the network formation rules defined for the Chord algorithm [4]. To ensure the network is fully formed and stable, the simulation is allowed to run for 1000 seconds.

After network initialization, nodes begin to read from the key file generated in section 5.1 above. If *Node_A* reads a key *abcdef123* from the key file, it first extracts the group ID information from the key (i.e., 123) and adds an entry into its group ID finger table with the finger pointing to itself. Entries added to the group ID finger table at this stage will determine

what query file the node will read from when it is ready to start querying for resources. Table 5.1 below shows entries in *Node_A*'s group ID finger table after it has read from the key file twice. This implementation detail is based on the premise that a node joining the network would be interested in joining a community whose members have resources similar to its own. For example, if *Node_A* joins the network and it is willing to make available some music files, then there is a high probability that *Node_A* will be querying for music files and therefore *Node_A* would want to join the music community.

Table 5.1: Example of entries in the group ID finger table after *Node_A* reads from the provided key file twice.

Group ID	Finger To Group Member
123	node A
256	node A
xxx	TBD
xxx	TBD

Once the node has added the associated entry into its group ID finger table, it creates a PUT message and inserts it into the network. The PUT message is how 1) a node makes its resources available to the network and 2) how the node which will be responsible for resolving queries for the key being inserted learns about the key and its source. To further explain how PUT messages are handled in the network, an example of a simplified network is shown in figure 5.1 below. The network shown can have up to 8 nodes at a time but currently only consists of three nodes which occupy position 0, 1 and 3 respectively. As an example, if node 1 inserts a key with value 6 into the network using a PUT message, node 1 will use its Chord finger table to determine the PUT message's next hop. According to its finger table the next hop would be node 0. Since node 0 is the successor for all keys between 4 and 0 it knows that it is responsible for

resolving all queries for keys that are in this range. Therefore, once node receives the PUT message from 1, it adds an entry to its DHT with a finger pointing to node 1.

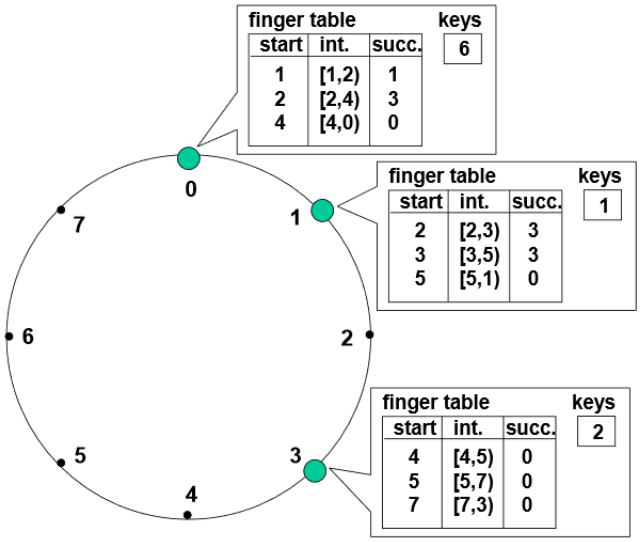

Figure 5.1: Example of a simplified Chord network [4].

It is also important to note that querying only starts when all the keys in the key file provided have been inserted into the network. The logic used to insert keys into the network using Oversim is summarized below.

```
void handle PUT event
{
      key = read from key file;

      if key is unspecified // i.e., all keys have been read from key
file
      {
            schedule GET event;
      }
      else
      {
            extract group ID information from key;
            add entry to group ID finger table;
            create PUT message and send it out to network;
            schedule next PUT event;
      }

`
```

5.4.2. Querying Resources and Community Formation

Queries are sent only when all the keys intended for the simulation run have been inserted into the network. Query messages are sent out using a GET message and these messages are sent out when a GET event is processed. Nodes read from the query files created in Section 5.2 above based on "personal interests" established in Section 5.4.1 above. For example, if *Node_A* is only interested in group 1, it will always read from the query file with group 1 keys. However, if *Node_A* is interested in multiple groups for example group 1 and 2, at runtime *Node_A* will choose one group and read from the associated query files. How often *Node_A* reads from either of the query files depends on how many keys it inserted for each group. For example, if 20% of the keys *Node_A* inserted belonged to group 1 and the other 80% belonged to group 2, then *Node_A* will read from the group 1 query file 20% of the time and the rest of the time it will read from the group 2 query file.

When a *Node_B* receives a query *Q* it first checks to see if it can resolve it. If *Node_B* cannot resolve *Q*, it either uses its group ID finger table or Chord finger table to decide where to forward the query as summarized in Chapter 4 Section 4.3.3. Whenever a node is processing a query, the first thing it does is extract the group ID information embedded in the query's key. If the key belongs to a group the node is already interested in, this is an indication that there exists other nodes in the network with similar interest. Ideally at this point, the node processing the query should send out a FIND GROUP message to its neighbors and wait to see if it gets a response. However, nodes do not have infinite resources and therefore limited as to how many entries it can have in its group ID finger table. Section 4.2.3 in Chapter 4 describes in detail the logic implemented to ensure entries added to the group ID finger table yield optimal results. After a GET event is processed, the logic below is used to schedule the next one.

```
void handle GET event
{
      select group ID file to read from; // Based on personal interest
      query = read from key from query file;

      if query is unspecified // i.e., all keys have been read from query
file
      {
            return; // do nothing
      }
      else
      {
            create GET message with query;
            send out GET message;
            schedule next GET event;
      }

}
```

5.4.3. Group Churn at Runtime

To simulate growing, diminishing and finally disappearance of groups, a third type of message called REMOVE is introduced. The purpose of this message is to tell nodes to remove a particular group entry from their group ID finger tables. Measurements are then taken to determine the effects of deleted entries on query resolution time.

After a specified period of time has elapsed, each node schedules an event to read from a file that contains group IDs that exist in the network. Equipped with the information retrieved from the file, the node creates a REMOVE message and sends it out to all its neighbours. Since the source node information in the REMOVE message may not be accurate (i.e., the node inserting the REMOVE message may not belong to the group contained in the message) a max hop count is included in the message to limit the number of times the message is forwarded before being dropped.

Nodes are limited to removing group ID entries with fingers to all nodes other than itself. This is done to ensure that the basis of a node's interest, which was established when inserting keys using PUT messages, is preserved. Ensuing a node's interest basis is not lost during group churn is critical because it is used to facilitate re-growing of communities since GET messages and REMOVE messages are circulating in the network at the same time. The logic below summarizes how REMOVE messages are processed by each node.

```
void hand_REMOVE_request(group id, finger, maxHops, curHops)
{
      if <group ID, finger> pair exist
            delete entry

      if curHops < maxHops
            forward message to all nodes in group ID finger table
}
```

In the next section, parameters needed to test the algorithm is discussed.

5.5. Performance Parameters

In order to test the Dynamic Group Discovery algorithm in conjunction with Community Based Caching algorithm proposed in [29], we are interested in measuring query resolution times in terms of hops and latency with respect to:

1. The entire network
2. Within the community

These two scenarios are considered mainly because our goal is to improve lookup performance by taking advantage of existence of groups without degrading overall system performance. In order to achieve this, every query reply is tagged with information on whether the reply is based

on cached data or application data (i.e., non-cache data.) To ensure that the cache response is truly from a community member, each node is limited to caching resources of communities it has joined.

5.6. Simulator Validation

Verification of implemented dynamic group discovery (DGD) algorithm was tested under various conditions to validate that the simulator works as expected. To achieve this, a series of simulations were run to:

1. Verify that the performance of DGD is comparable to Chord when the group implementation feature is switched off.
2. Validate that communities are able to grow and/or decline over time when PUT, GET and REMOVE messages are processed.
3. Verify that community formation works as designed by ensuring that the number of members in a community does not exceed the number of nodes in the network at runtime, and that the communities formed exhibit the features discussed in Chapter 2.

The two parameters that were varied during the validation process are network size and community symmetry. The algorithm was tested with a “big” (2000 nodes) and “small” (500 nodes) network and with both symmetrical and asymmetrical communities present. This was done to ensure that performance of the algorithm is consistent regardless of network size and symmetry of communities present.

5.6.1. Algorithm Performance without Using Group ID Table

Figure 5.2 to 5.6.1.2 below summarize results of Chord compared to the dynamic group discovery (DGD) algorithm when group ID finger table is not being utilized. As expected, DGD's performance over time is very similar to Chord and the difference between average number of hops and latency negligible in all cases presented above.

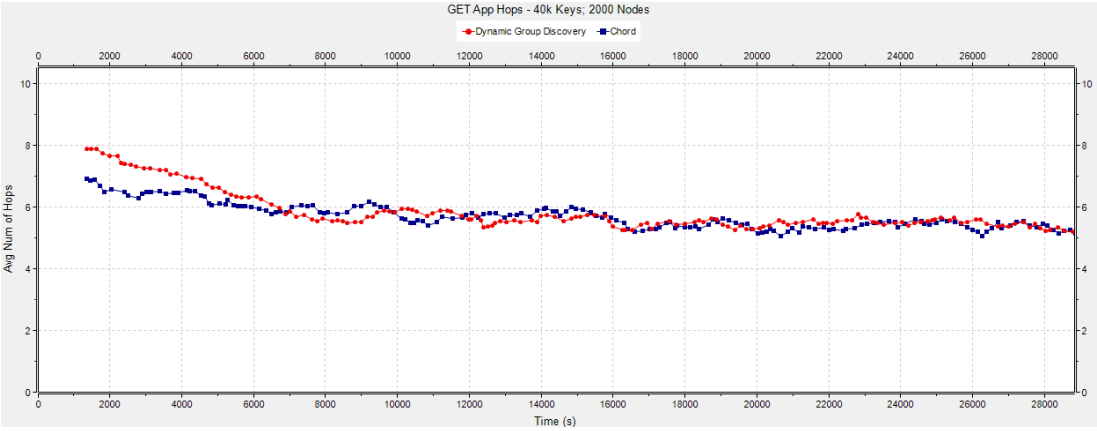

Figure 5.2: Average number of hops to resolve a query with application data using Chord and a network using Dynamic Group Discovery.

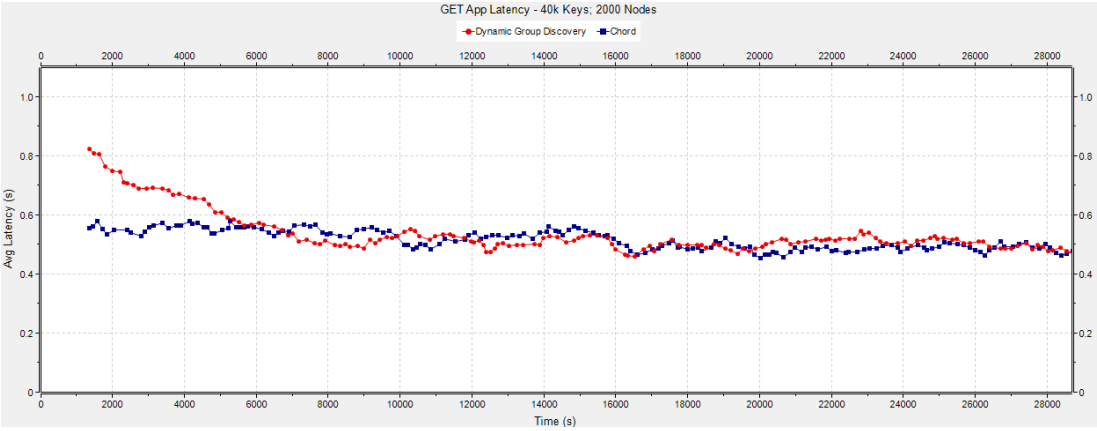

Figure 5.3: Average latency to resolve a query with application data using Chord and a network using Dynamic Group Discovery.

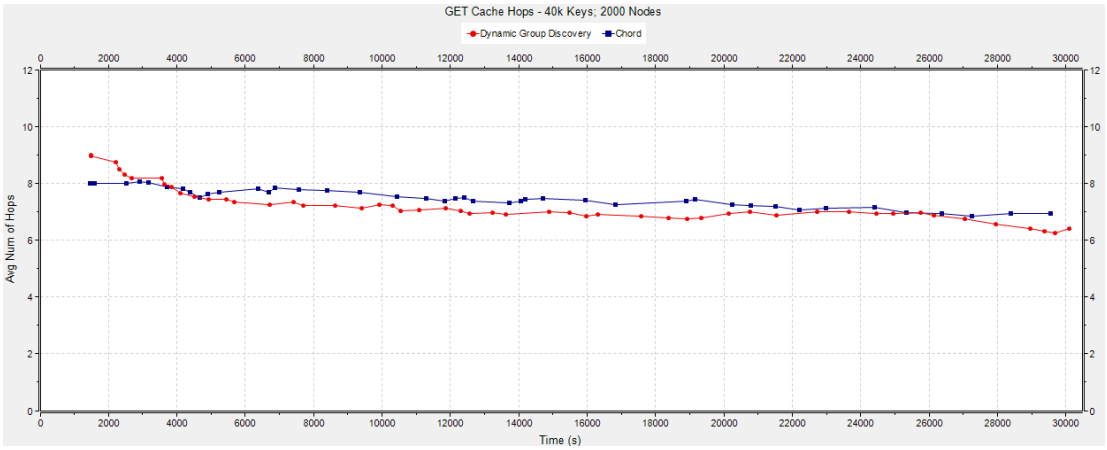

Figure 5.4: Average number of hops to resolve a query with cache data using Chord and a network using Dynamic Group Discovery.

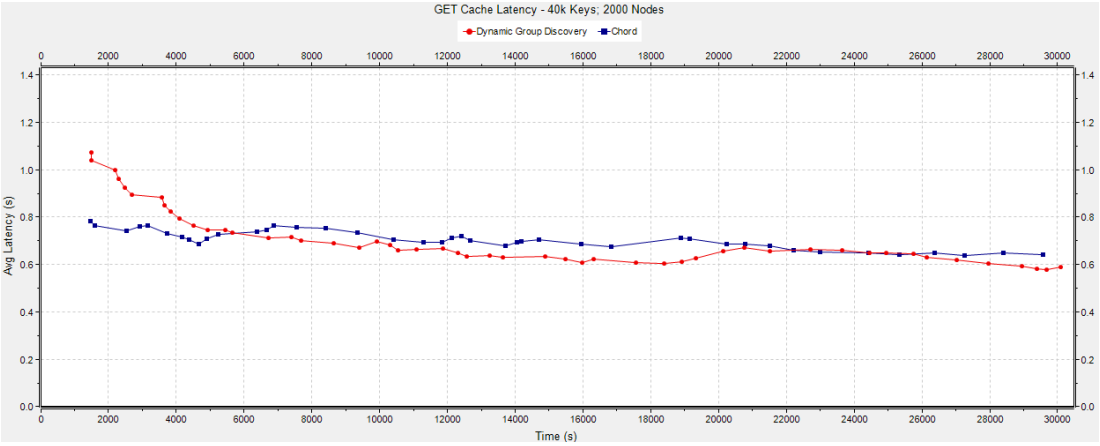

Figure 5.5: Average latency to resolve a query with cache data using Chord and a network using Dynamic Group Discovery.

5.6.2. Community Discovery and Performance

5.6.2.1. Community Growth and Boundary Conditions

Figure 5.6 below shows group growth in a network with 2,000 nodes and 5 communities.

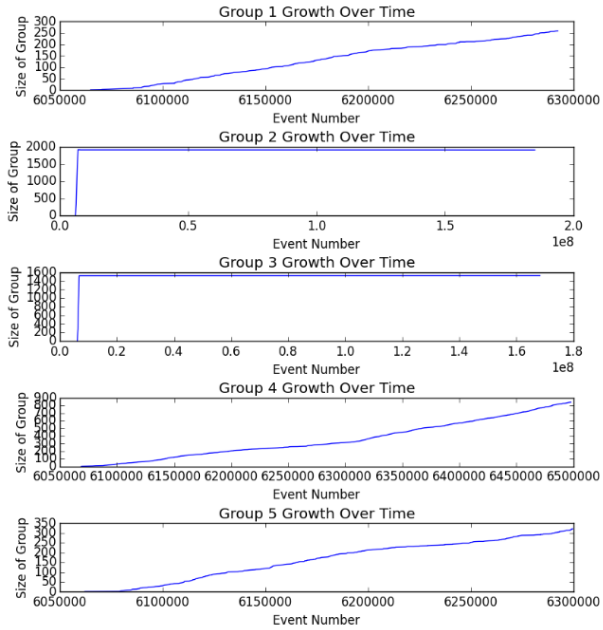

Figure 5.6: Group growth in a network with 2,000 nodes, and asymmetrical groups and no group churn is introduced.

First observation made is that no community grows beyond the number of nodes that are currently in the network and this is the first indication that DGD is working as expected. Second, notice that group 2 has approximately 2000 members compared to other groups such as 3 and 4 that have 900 and 1600 members respectively. These numbers imply that there are nodes in this network of 2000 nodes that belonged to more than one community. This observation is in line with the group membership requirement discussed in Chapter 2, which is that nodes should be allowed to be members of more than one community at the same time. This leads into the third observation that since nodes in the network can belong to more than one community at time, then

inevitably, there is overlap between the communities. The degree of overlap varies and from [17] [18], we know that the degree of inter-community connections is directly proportional to the community overlap. Therefore, since group 2 and 3 have the largest number of member nodes, we expect to see a lot of overlap between the two communities. On the contrary, group 1 and 5 that only have 300 and 350 member nodes and therefore the overlap between these two communities will be minimal. Lastly, the graphs show community growth varies over time and even though the growth patterns may be similar, the growth rate of a community is independent to others. The abrupt group growth seen in group 2 and 3 is as a result of many nodes inserting keys that belong to group 2 and 3 at the beginning of the simulation which is expected behavior.

To verify that REMOVE messages affect the size of communities in the network, a smaller network of 500 nodes is used. Note that all communities present in Figure 5.7 and 5.6.2.3 are symmetrical.

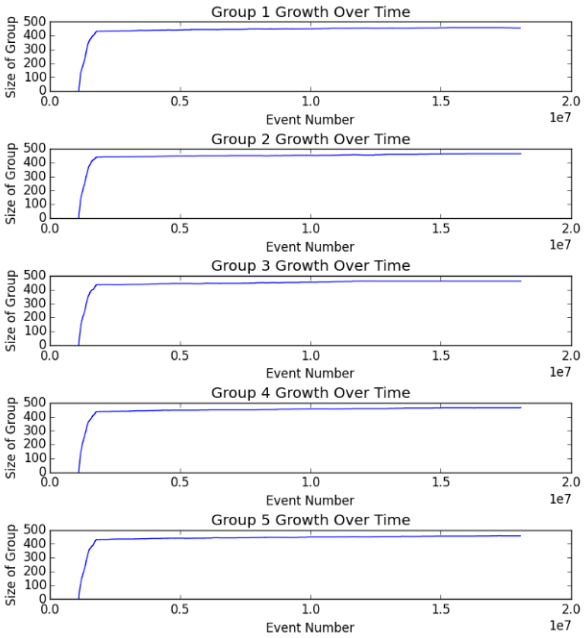

Figure 5.7: Group growth in a network with 500 nodes, and symmetrical groups and no group churn is introduced.

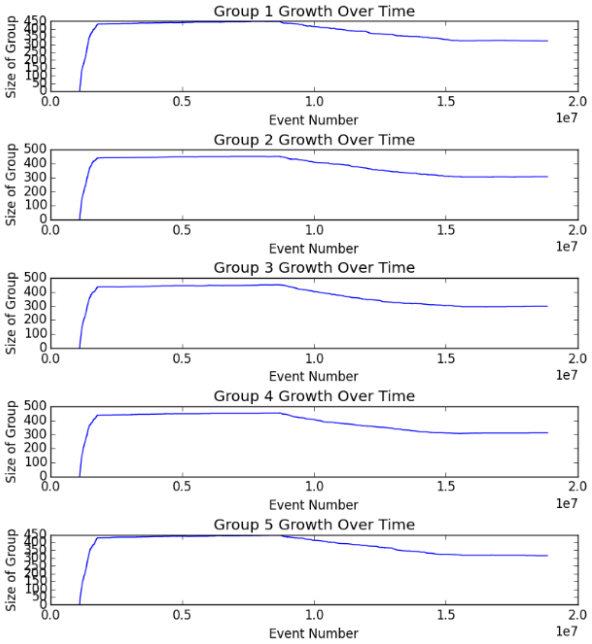

Figure 5.8: Group growth in a network with 500 nodes, and symmetrical groups and group churn is introduced.

In Figure 5.7, no group churn was introduced for the entire simulation run. However, REMOVE messages were introduced in the simulation run represented by Figure 5.8 and it is clear that all communities experience a steady decline in size. The steady and similar decline in all communities is due to the fact that communities are symmetrical.

The results presented in this section (5.6.2) confirm that:

1. Communities that exist in the network are discoverable by other nodes as seen by the steady and sometimes abrupt increase group size over time.
2. Nodes are able to join and leave communities at will.
3. Nodes are able to be members of multiple communities at time.
4. Because of observation 2 above, there exists a certain degree of overlap between independent communities.

5.6.3. Performance When Group Churn is Introduced

Figure 5.9 and 5.6.3.2 below show how resolving queries using cache records vary when there is no group churn versus when group churn is introduced in terms of hops and latency. REMOVE messages were sent beginning at 5000 seconds and the yellow plot represents performance in the simulation when group churn was introduced. The erratic behavior exhibited after 5000 seconds to could be attributed to invalid group ID finger table entries caused by the REMOVE message which resulted in longer and unexpected query resolution times.

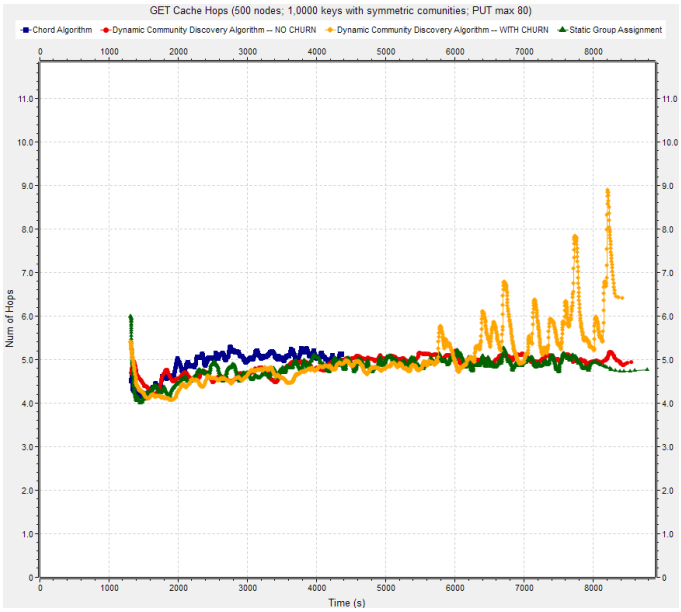

Figure 5.9: Average number of hops needed to resolve a query using cache data. Results include Chord, CBC + DGD without churn, CBC + DGD with churn, CBC with static group assignment.

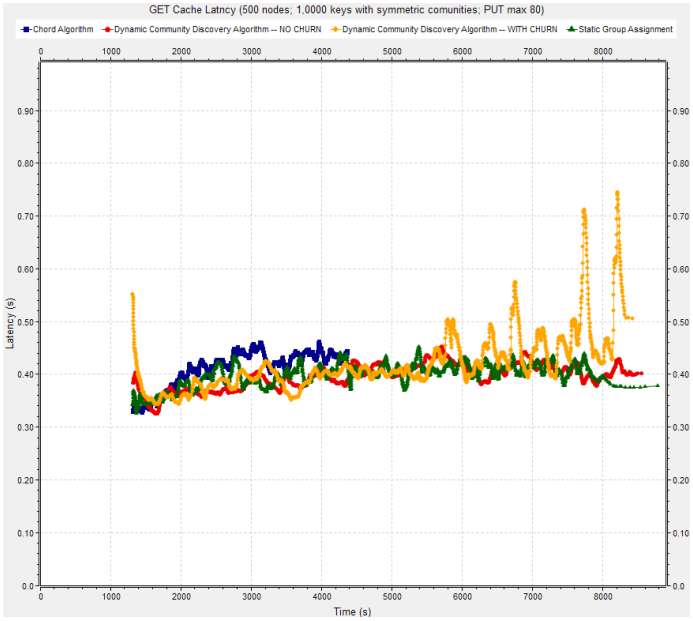

Figure 5.10: Average latency to resolve a query using cache data. Results include Chord, CBC + DGD without churn, CBC + DGD with churn, CBC with static group assignment.

Based on the results presented in this section we can conclude that all the features in the simulator work as expected and in the next section, various cases are presented to characterize DGD and determine its optimal performance region.

5.7. Results and Analysis

Highlighted in the subsections that follow are cases under which DGD was tested. The different scenarios are used to analyze DGD's performance and determine the optimal conditions for the algorithm to perform best. The goal of this section is to characterize the proposed dynamic group discovery algorithm and to compare Community Based Caching performance when group discovery is both static and dynamic.

5.7.1. Case 1: Varying Number of Nodes – Asymmetrical Communities

The number of nodes was varied between 500 and 10,000 while the number of keys for each run was kept constant. Each simulation used 40,000 keys and the group distribution was as follows: 40% of the keys belonged to group 1; another 40% of the keys belonged to group 2 and the remaining 20% of the keys was equally divided between groups 3 to 9. Each node in the network maintained a maximum group ID finger table of 160 and was limited to having a maximum of 3 fingers to other nodes in the network which belonged to the same group. Depending on the network size, an attempt to resolve queries within the group was limited to $(\log_2 N)/2$ where N is the number of nodes in the network. λ (threshold to search for group members based on personal interests) and μ (threshold to search for group members based on messages being forwarded) used by each node were set to λ=2 and μ=20. Per the simulation

discussion covered in section 5.4 above, a node queried for resources based on personal interests which is established when PUT-ing keys in the network.

Figure 5.11 and 5.12 below summarize the average cost of resolving queries using application and cache data respectively. A high level analysis of the results show that DGD's performance is generally better than Chord's. The trend in both DGD and Chord is that the higher the number of nodes in the network, the longer it takes to resolve queries. This is expected because both algorithms are designed to resolve queries in *O(logN)* hops or less. Even though the average number of hops to resolve queries follow an upward trend as nodes increase in the network, it is not always the case for latency. This is because latency is measured based on the query round trip time (RTT) which could be affected by many varying network factors such as link loss, varying buffer sizes in different nodes etc.

Figure 5.11 shows that DGD begins to perform significantly worse than Chord when the network size is greater than 2,000 nodes. This can be attributed to a couple of factors which include: growing network size, number of queries being propagated and how the DGD algorithm works. To expound on this observation, from Table 5.2 to 5.7.1.10, shows the average number of queries propagated during the entire simulation run is approximately 245,000 for all network sizes. Note that DGD depends on query messages being propagated to learn about nodes with similar interests. We therefore expect that the more queries in the network, the higher the chances of group discovery. This theory can be confirmed by the number of links recorded within each group (see Table 5.2 to 5.7.1.10). In the 500 node network, both group 1 and 2 show that 1,500 links were formed within each group which implies that each node had an average of 3 links. This observation is in line with the parameters set for the case study above, i.e., each node is limited to having a maximum of 3 fingers to other nodes within the same group. The same

conclusion can be drawn for networks with 1,000 and 2,000 nodes. However, for networks with 5,000 and 10,000 nodes, the average number of links per node is 1.12 and 0.45 respectively; and again this can be attributed to the relatively low number of query messages in larger networks. Due to low linkage within groups in larger networks and the fact that DGD is a hybrid of Chord and "smart flooding," there is a high probability that precious hops will be used to forward queries to group members who are unable to resolve the query. Essentially the "smart flooding" seizes to be smart and consequently affects performance of the algorithm.

An important observation made is that DGD's performance based on cache records (see Figure 5.12) is always better than Chord's. This is because caching records in DGD is primarily based on personal interests whereas this is not the case in Chord. Therefore as expected, query resolution using cache data is better when nodes base their caching decisions on personal interests and not based on system wide popularity.

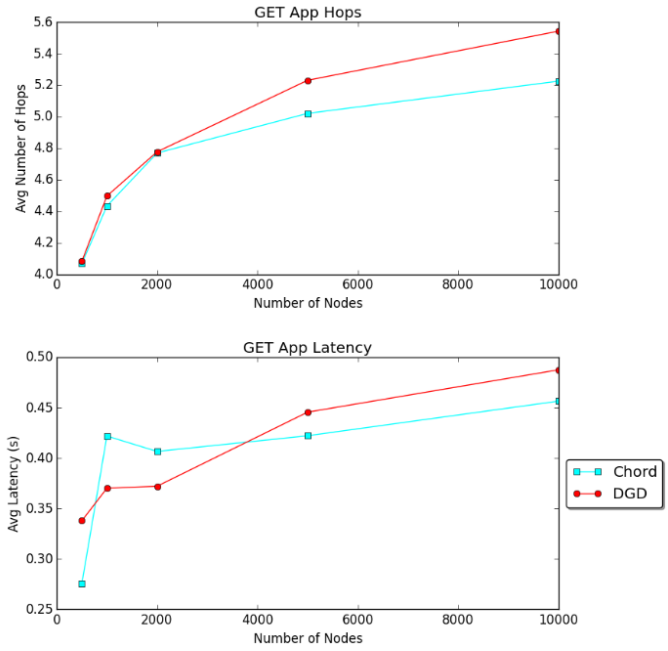

Figure 5.11: Average cost of query resolution based on non-cache data for varying network sizes in Chord and DGD.

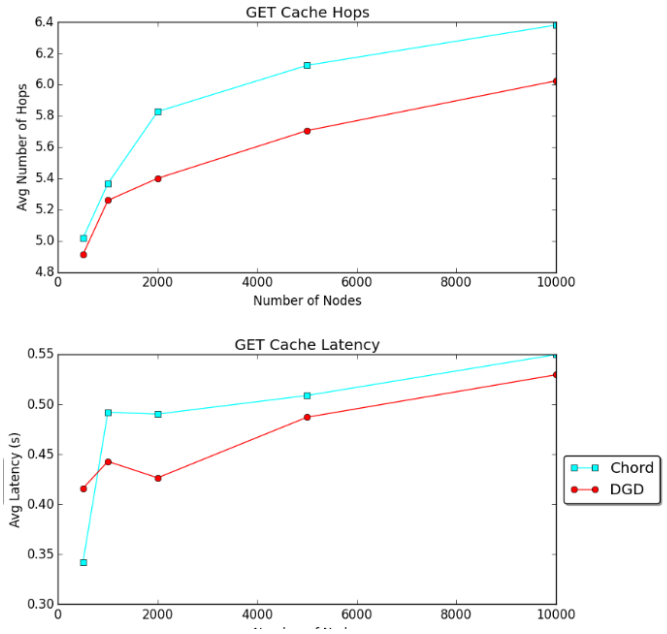

Figure 5.12: Average cost of query resolution based on cache data for varying network sizes in Chord and DGD.

To understand DGD's behaviour characterized above, Table 5.2 to table 5.11 below summarize the total number of queries resolved using application data, cache data, number of links formed within each group and the percentage of nodes in the network that are members of the group.

The first observation made is that the average DGD readings represented in Figure 5.11 and 5.7.1.3 follow the exact same pattern as average query resolution times recorded within group 1 and 2 and is illustrated in Figure 5.15 and 5.7.1.8. This behavior can be attributed to the fact that majority of the queries being circulated in the network are for keys which predominantly belong to group 1 and group 2. Also, as expected the counts recorded in Table 5.2 to 5.7.1.10 verify that majority of the queries in all five simulation runs belonged to group 1 and 2.

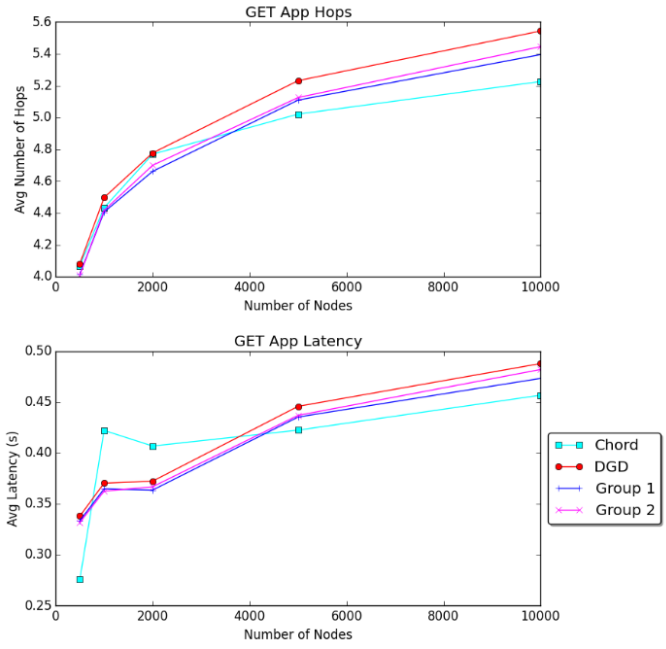

Figure 5.13: Average cost of query resolution based on non-cache data for varying network sizes in Chord, DGD, DGD group1 and DGD group 2.

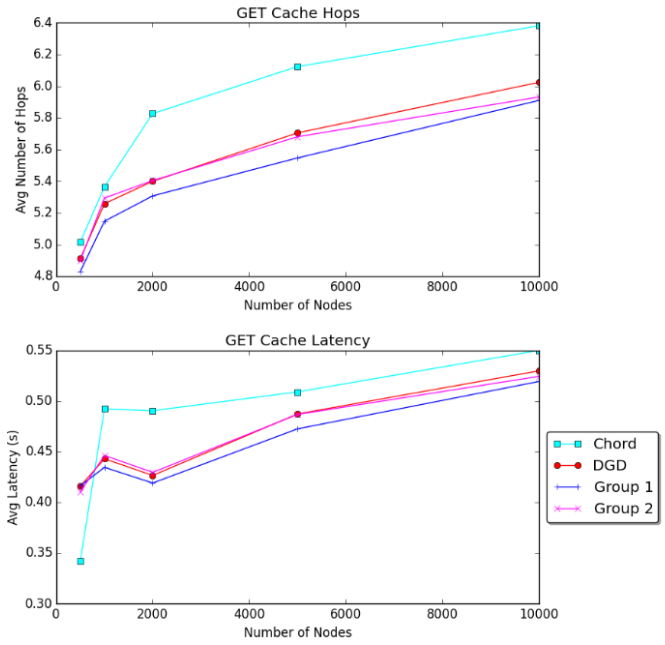

Figure 5.14: Average cost of query resolution based on cache data for varying network sizes in Chord, DGD, DGD group1 and DGD group 2.

Second, even though DGD's average performance is mainly driven by the values recorded in group 1 and 2, its average is slightly higher. This is because of degraded performance observed in group 3 to 9 which is due to the fact that there are few links within these groups and this results in not so "smart" routing as described above. Also, the effects of group 3 to 9 averages does not affect DGD's average greatly because the number of data points in these groups is significantly less than those in group 1 and 2 combined. Figure 5.15 and 5.7.1.6 which show the average performance between DGD's average and individual group averages between 3 and 9.

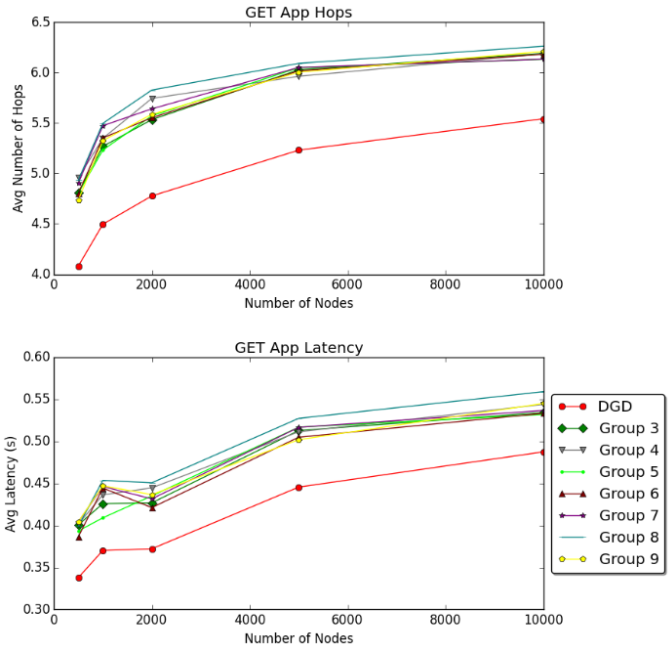

Figure 5.15: Average cost of query resolution based on non-cache data for varying network sizes in DGD and DGD group 3 and DGD group 9.

The unpredictable trends observed in group 3 to 9 in Figure 5.16 below is because of the low number of queries resolved using cache data and this theory is confirmed by Figure 5.23 where all there were "enough" queries resolved using cache data to establish a trend. Query counts for the different network sizes are summarized in Table 5.2 to 5.7.1.10.

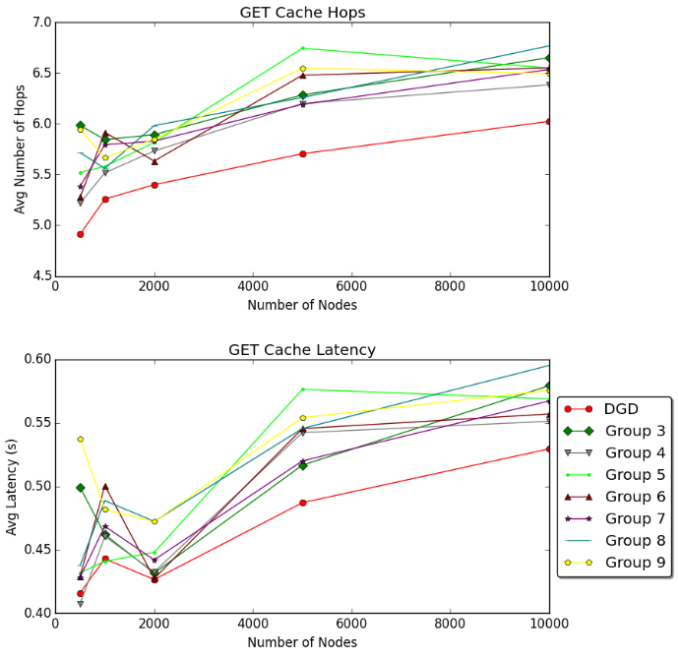

Figure 5.16: Average cost of query resolution based on non-cache data for varying network sizes in DGD and DGD group 3 and DGD group 9.

Table 5.2: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 500 node network.

500 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	98629	6482	1500	100%	46%
	Group 2	98423	5921	1500	100%	46%
	Group 3	2494	139	181	96%	1%
	Group 4	2566	151	180	96%	1%
	Group 5	2480	100	192	96%	1%
	Group 6	2499	123	223	96%	1%
	Group 7	2458	112	172	96%	1%
	Group 8	2541	155	168	96%	1%
	Group 9	2610	111	194	96%	1%

Table 5.3: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 500 node network.

500 Nodes									
		App Mean	App Min	App Max	App Std Dev	Cache Mean	Cache Min	Cache Max	Cache Std Dev
	Group 1	4.02	0	12	2.26	4.83	2	12	1.79
	Group 2	4.01	0	12	2.29	4.90	2	12	1.78
	Group 3	4.81	0	11	1.81	5.99	2	11	2.25
	Group 4	4.96	0	12	1.64	5.22	2	11	1.90
	Group 5	4.80	0	13	1.81	5.52	2	11	2.10
	Group 6	4.80	0	11	1.78	5.28	2	11	1.89
	Group 7	4.91	0	11	1.71	5.38	2	11	2.10
	Group 8	4.94	0	12	1.69	5.72	2	11	1.88
	Group 9	4.74	0	11	1.82	5.95	2	11	2.25

Table 5.4: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 1,000 node network.

1,000 Nodes						
		App Count	Cache Count	# of Links	% of Mem. Nodes	%age of Queries
	Group 1	98632	8353	2996	100%	45%
	Group 2	98590	8730	2998	100%	46%
	Group 3	2815	180	114	70%	1%
	Group 4	2859	180	126	70%	1%
	Group 5	2707	150	101	70%	1%
	Group 6	2725	227	123	70%	1%
	Group 7	2845	248	146	70%	1%
	Group 8	2811	203	125	70%	1%
	Group 9	2789	174	113	70%	1%

Table 5.5: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 1,000 node network.

1,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.41	0	13	2.46	5.15	2	13	1.92
	Group 2	4.42	0	13	2.46	5.29	2	13	1.93
	Group 3	5.27	0	13	1.86	5.84	2	13	1.82
	Group 4	5.35	0	10	1.74	5.52	2	9	1.70
	Group 5	5.23	0	11	1.83	5.58	2	11	1.73
	Group 6	5.35	0	13	1.89	5.92	2	13	1.86
	Group 7	5.47	0	13	1.78	5.79	2	13	1.69
	Group 8	5.50	0	12	1.80	5.56	2	12	1.76
	Group 9	5.33	0	10	1.88	5.67	2	9	1.60

Table 5.6: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 2,000 node network.

2,000 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	98631	9877	5294	100%	45%
	Group 2	98591	10101	5344	100%	45%
	Group 3	3066	226	58	50%	1%
	Group 4	3108	351	43	50%	1%
	Group 5	3215	213	71	50%	1%
	Group 6	3232	240	52	50%	1%
	Group 7	3211	317	63	50%	1%
	Group 8	3169	405	62	50%	1%
	Group 9	3171	265	41	50%	1%

Table 5.7: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 2,000 node network.

2,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.66	0	15	2.54	5.31	2	15	2.21
	Group 2	4.70	0	16	2.54	5.40	2	16	2.18
	Group 3	5.53	0	13	1.89	5.89	2	13	1.75
	Group 4	5.74	0	13	1.79	5.73	2	9	1.51
	Group 5	5.57	0	13	1.91	5.82	2	10	1.73
	Group 6	5.55	0	14	1.88	5.63	2	14	1.73
	Group 7	5.64	0	14	1.84	5.83	2	9	1.60
	Group 8	5.83	0	11	1.79	5.98	2	10	1.64
	Group 9	5.59	0	11	1.84	5.85	2	11	1.67

Table 5.8: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 5,000 node network.

5,000 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	98633	12025	5745	100%	44%
	Group 2	98591	12466	5803	100%	44%
	Group 3	3943	434	11	24%	2%
	Group 4	4151	500	10	24%	2%
	Group 5	3963	456	22	24%	2%
	Group 6	4081	490	23	24%	2%
	Group 7	3940	468	8	24%	2%
	Group 8	4064	493	13	24%	2%
	Group 9	4049	422	12	24%	2%

Table 5.9: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 5,000 node network.

5,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	5.11	0	18	2.70	5.55	2	18	2.58
	Group 2	5.13	0	18	2.71	5.68	2	18	2.65
	Group 3	6.03	0	12	2.05	6.29	2	10	1.75
	Group 4	5.96	0	12	2.07	6.20	2	11	1.79
	Group 5	6.05	0	16	2.13	6.75	2	16	2.22
	Group 6	6.02	0	12	2.06	6.48	2	12	1.91
	Group 7	6.05	0	12	2.04	6.19	2	11	1.88
	Group 8	6.09	0	12	2.06	6.26	2	11	1.73
	Group 9	6.00	0	13	2.09	6.55	2	12	2.00

Table 5.10: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 10,000 node network.

10,000 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	98633	14822	4574	90%	42%
	Group 2	98590	14863	4599	90%	42%
	Group 3	5373	729	9	13%	2%
	Group 4	5307	804	4	13%	2%
	Group 5	5423	745	6	13%	2%
	Group 6	5479	837	4	13%	2%
	Group 7	5516	829	8	13%	2%
	Group 8	5424	931	7	13%	2%
	Group 9	5327	796	8	13%	2%

Table 5.11: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 10,000 node network.

10,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	5.40	0	20	2.80	5.91	2	20	2.95
	Group 2	5.44	0	20	2.78	5.93	2	20	2.88
	Group 3	6.19	0	16	2.19	6.65	2	16	2.00
	Group 4	6.19	0	13	2.18	6.38	2	11	1.98
	Group 5	6.13	0	13	2.26	6.55	2	12	1.96
	Group 6	6.18	0	13	2.20	6.55	2	11	1.92
	Group 7	6.13	0	14	2.22	6.54	2	12	1.92
	Group 8	6.26	0	13	2.15	6.77	2	12	1.93
	Group 9	6.21	0	13	2.24	6.49	2	12	1.98

In conclusion, the number of links within group members is directly proportional to the number of queries propagated for the group. If there exists groups in the network that are predominantly large, and majority of the queries being forwarded in the network belong to these groups, then we see that DGD yields better results than Chord.

5.7.2. Case 2: Varying Size of Communities

For all simulation runs the total number of keys used was divided into two parts; subset 1 consisted of 80% of the keys and subset 2 had the remaining 20%. To vary the size of the communities, the number of communities in each subset was varied in such a way that there was always a total of 9 communities. To start, all the keys in subset 1 belonged to group 1 and group 2 to 9 equal shared the keys in subset 2. For the second run, group 1 and 2 equally shared keys in subset 1 and group 3 to 9 equally shared keys in subset 2. For the third run, group 1 to 3 equally shared keys in subset 1 and group 4 to 9 equally shared keys in subset 2. This pattern was

repeated until group 1 to 8 equally shared keys in subset 1 and all the keys in subset 2 belonged to group 9. Finally, the two subsets were merged and all the keys shared equally between group 1 to 9.

All runs used 40,000 keys and the network consisted of 2,000 nodes. Each node in the network maintained a maximum group ID finger table of 160 and was limited to having a maximum of 3 fingers to other nodes in the network which belonged to the same group. Depending on the network size, an attempt to resolve queries within the group was limited to $(\log_2 N)/2$ where N is the number of nodes in the network. λ (threshold to search for group members based on personal interests) and μ (threshold to search for group members based on messages being forwarded) used by each node were set to λ=2 and μ=20. Per the simulation discussion covered in Section 5.4 above, a node queried for resources based on personal interests which is established when PUT-ing keys in the network.

Figure 5.17 and 5.7.2.2 shows how DGD's performance generally compares to Chord's. Since the network size does not change, Chord's performance stays relatively the same for all simulation runs. However, for DGD we see an upward trend in the cost of resolving queries which can be attributed to the reduced number of links within groups. As noted in case 1 above, the number of links within a group is directly proportional to the cost of query resolution using DGD.

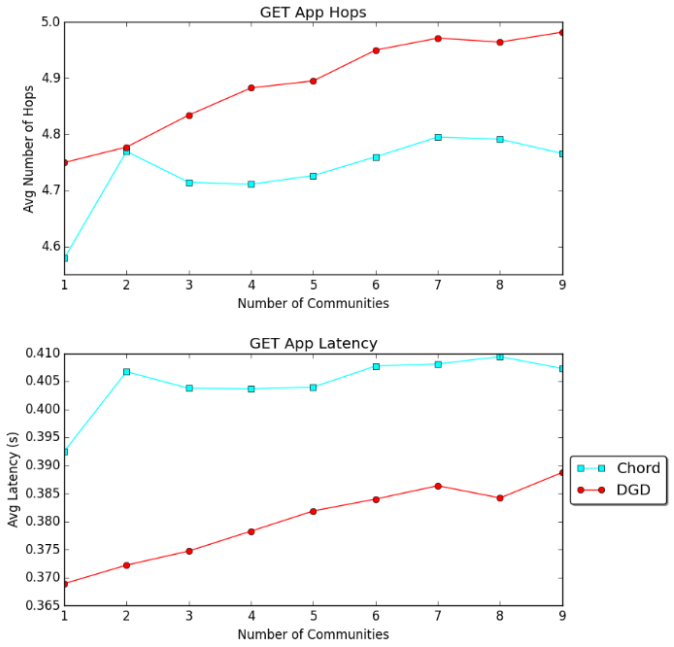

Figure 5.17: Average cost of query resolution based on non-cache data for varying community sizes in Chord and DGD.

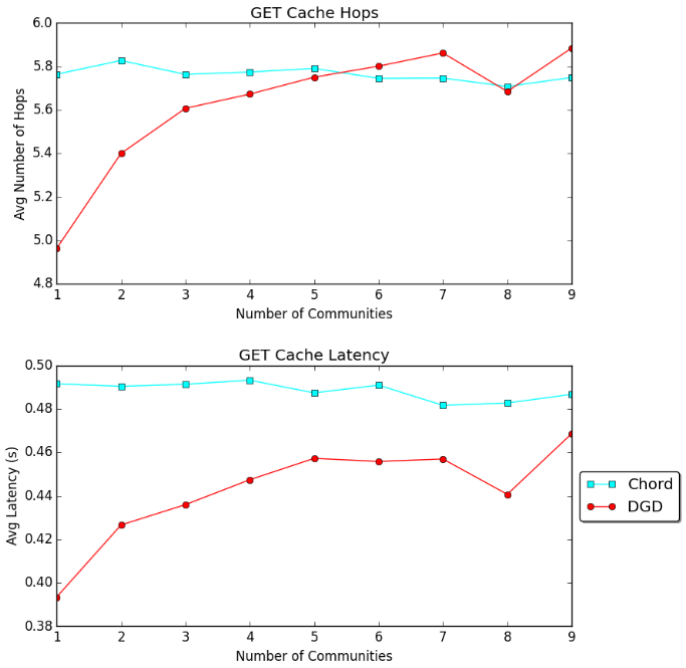

Figure 5.18: Average cost of query resolution based on cache data for varying community sizes in Chord and DGD.

As expected, DGD performance using cache records is always better than Chord's but we see performance drops when the size of community reduces. Again, the degraded performance can be attributed to reduced community sizes which translates to reduced linkage within the groups and this trend is clearly illustrated in Figure 5.18 above. A break in the upward trend is represented by a sudden dip when 8 communities are equally sharing keys in subset 1 and there is only 1 group in subset 2. This sudden change is an indication of improved performance in DGD and to understand why this is the case, we shall use Figure 5.19 and Table 5.12 to 5.7.2.18 below.

When subset 1 had only group 1 keys and group 2 to 9 shared the keys in subset 2, group 1 keys were the majority and consequently, queries propagated in the network were for keys that predominantly belonged to group 1. Table 5.12 confirms this by showing that 92% of the queries belonged to group 1. Since query propagation is what aids in group discovery, we see that approximately 5990 links were formed within group 1 which implies that on average, each node had 2.9 fingers to other nodes in group 1. It is also worth noting that this observation is in line with the parameters that were set at the beginning of the simulation, i.e., nodes are limited to having a maximum of three fingers to other nodes within the same group. When group 1 and 2 are the only groups sharing keys in subset 1, notice the drastic improvement in query resolution performance observed in group 2. Table 5.14 again confirms that the improved performance in group 2 is due to the fact that majority of the queries being propagated belong to the group 1 and 2 which aids in group discovery and again, improved linkage within the community. In this case, nodes that belonged to group 1 and 2 both had an average of 2.6 fingers to other nodes.

The trend of seeing significant improvement in performance whenever a group is removed from subset 2 and added to subset 1 is consistent only up until when there are 6 groups

in subset 1 and 3 groups in subset 2. However, when group 7 is added to subset 1 and group 8 and 9 are the only ones sharing keys in subset 2, the performance of all groups is relatively the same. This is because in this case, the communities are almost exactly symmetrical since 7 groups are equally sharing 80% of the keys and 2 groups are equally sharing 20% of the remaining keys. To confirm this observation, the percentage of queries propagated during run 7 (see Table 5.18) is very close to those recoded during run 9 (see Table 5.20) when all communities were intentionally made perfectly symmetrical.

During run 8, DGD's average performance suddenly improves as shown in Figure 5.17 and 5.7.2.2. From Figure 5.19 and table 5.19 below, it is clear to see that DGD's general improved performance is due to group 9. Again in this case, majority of the queries propagated belonged to group 9, which aided link growth within the group and consequently led to improved query resolution times.

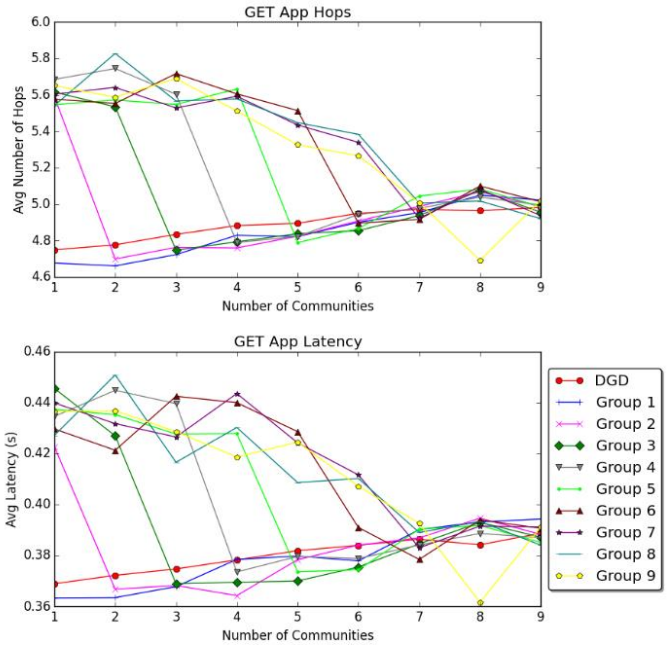

Figure 5.19: Average cost of query resolution per group based on non-cache data for varying community sizes in DGD.

Table 5.12: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for 1 community in subset 1 and 8 communities in subset 2.

1 Community						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	114807	10245	5990	100%	92%
	Group 2	1194	100	10	42%	1%
	Group 3	1222	90	19	42%	1%
	Group 4	1213	99	28	42%	1%
	Group 5	1227	110	8	42%	1%
	Group 6	1198	100	6	42%	1%
	Group 7	1243	109	22	42%	1%
	Group 8	1205	101	9	42%	1%
	Group 9	1268	93	3	42%	1%

Table 5.13: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for 1 community in subset 1 and 8 communities in subset 2.

1 Community									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.68	0	16	2.53	4.90	2	16	2.18
	Group 2	5.59	1	10	1.85	5.81	3	10	1.91
	Group 3	5.62	0	13	1.92	5.67	2	9	1.84
	Group 4	5.69	0	15	1.78	5.77	3	15	1.81
	Group 5	5.55	0	10	1.82	5.61	2	9	1.60
	Group 6	5.58	0	10	1.86	5.60	3	10	1.61
	Group 7	5.60	0	11	1.91	5.86	2	10	1.64
	Group 8	5.54	0	11	1.75	6.11	2	10	1.73
	Group 9	5.65	0	11	1.82	5.98	2	9	1.83

Table 5.14: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for 2 communities in subset 1 and 7 communities in subset 2.

2 Communities						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	98631	9877	5294	100%	45%
	Group 2	98591	10101	5344	100%	45%
	Group 3	3066	226	58	49%	1%
	Group 4	3108	351	43	48%	1%
	Group 5	3215	213	71	49%	1%
	Group 6	3232	240	52	48%	1%
	Group 7	3211	317	63	48%	1%
	Group 8	3169	405	62	48%	1%
	Group 9	3171	265	41	50%	1%

Table 5.15: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for 2 communities in subset 1 and 7 communities in subset 2.

2 Communities									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.66	0	15	2.54	5.31	2	15	2.21
	Group 2	4.70	0	16	2.54	5.40	2	16	2.18
	Group 3	5.53	0	13	1.89	5.89	2	13	1.75
	Group 4	5.74	0	13	1.79	5.73	2	9	1.51
	Group 5	5.57	0	13	1.91	5.82	2	10	1.73
	Group 6	5.55	0	14	1.88	5.63	2	14	1.73
	Group 7	5.64	0	14	1.84	5.83	2	9	1.60
	Group 8	5.83	0	11	1.79	5.98	2	10	1.64
	Group 9	5.59	0	11	1.84	5.85	2	11	1.67

Table 5.16: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for 3 communities in subset 1 and 6 communities in subset 2.

3 Communities						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	93508	9665	4623	100%	30%
	Group 2	93500	10444	4687	100%	30%
	Group 3	93091	10375	4651	100%	30%
	Group 4	5417	498	144	53%	2%
	Group 5	5378	457	108	53%	2%
	Group 6	5467	670	115	53%	2%
	Group 7	5353	496	126	52%	2%
	Group 8	5449	514	90	53%	2%
	Group 9	5306	753	112	53%	2%

Table 5.17: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for 3 communities in subset 1 and 6 communities in subset 2.

3 Communities									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.72	0	15	2.52	5.51	2	15	2.11
	Group 2	4.76	0	15	2.52	5.61	2	15	2.17
	Group 3	4.75	0	15	2.53	5.54	2	15	2.18
	Group 4	5.60	0	15	1.93	5.95	2	14	2.04
	Group 5	5.55	0	14	1.90	5.82	2	12	1.62
	Group 6	5.72	0	15	1.90	6.30	2	15	1.86
	Group 7	5.53	0	15	1.95	5.92	2	15	1.93
	Group 8	5.57	0	14	1.90	5.88	2	14	1.74
	Group 9	5.69	0	14	1.87	6.31	2	10	1.68

Table 5.18: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for 4 communities in subset 1 and 5 communities in subset 2.

4 Communities						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	89867	10023	4111	99%	22%
	Group 2	89869	9727	4064	99%	22%
	Group 3	89849	9985	3944	99%	22%
	Group 4	89685	9465	4205	99%	22%
	Group 5	9164	1156	293	61%	2%
	Group 6	9097	1089	277	61%	2%
	Group 7	9191	1111	202	60%	2%
	Group 8	8950	1034	259	60%	2%
	Group 9	9320	888	235	60%	2%

Table 5.19: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for 4 communities in subset 1 and 5 communities in subset 2.

4 Communities									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.83	0	15	2.50	5.59	2	15	2.16
	Group 2	4.76	0	15	2.51	5.64	2	15	2.22
	Group 3	4.79	0	15	2.51	5.68	2	15	2.10
	Group 4	4.79	0	15	2.52	5.66	2	15	2.14
	Group 5	5.63	0	15	1.93	5.95	2	15	1.82
	Group 6	5.61	0	14	1.98	5.88	2	14	1.95
	Group 7	5.59	0	15	1.94	5.78	2	11	1.67
	Group 8	5.58	0	15	1.91	5.97	2	15	1.88
	Group 9	5.51	0	15	2.00	5.94	2	15	2.00

Table 5.20: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for 5 communities in subset 1 and 4 communities in subset 2.

5 Communities						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	86457	9393	3569	98%	18%
	Group 2	86668	9625	3663	97%	18%
	Group 3	86225	9422	3633	97%	17%
	Group 4	86667	9191	3842	97%	18%
	Group 5	86669	9275	3700	98%	18%
	Group 6	15198	1829	560	69%	3%
	Group 7	15346	1761	509	69%	3%
	Group 8	15326	1565	533	68%	3%
	Group 9	15387	1358	631	69%	3%

Table 5.21: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for 5 communities in subset 1 and 4 communities in subset 2.

5 Communities									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.82	0	15	2.46	5.64	2	15	2.07
	Group 2	4.82	0	15	2.54	5.88	2	15	2.13
	Group 3	4.84	0	16	2.52	5.71	2	16	2.16
	Group 4	4.82	0	15	2.51	5.72	2	15	2.06
	Group 5	4.79	0	15	2.49	5.60	2	15	2.15
	Group 6	5.51	0	14	2.03	5.91	2	14	1.88
	Group 7	5.43	0	15	2.13	6.10	2	15	1.98
	Group 8	5.45	0	15	2.15	6.24	2	15	2.21
	Group 9	5.33	0	14	2.14	5.87	2	14	1.97

Table 5.22: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for 6 communities in subset 1 and 3 communities in subset 2.

6 Communities						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	84775	9838	3252	96%	15%
	Group 2	84775	9638	3272	95%	15%
	Group 3	83928	9344	3540	95%	14%
	Group 4	83532	9760	3303	96%	14%
	Group 5	84800	9191	3415	95%	15%
	Group 6	84717	9208	3359	96%	15%
	Group 7	24507	2707	763	75%	4%
	Group 8	24084	3020	927	76%	4%
	Group 9	24758	2407	968	75%	4%

Table 5.23: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for 6 communities in subset 1 and 3 communities in subset 2.

6 Communities									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.90	0	15	2.46	5.72	2	15	2.18
	Group 2	4.91	0	16	2.50	5.80	2	16	2.16
	Group 3	4.85	0	15	2.49	5.76	2	15	2.13
	Group 4	4.94	0	15	2.48	5.92	2	15	2.13
	Group 5	4.87	0	15	2.49	5.77	2	15	2.06
	Group 6	4.90	0	15	2.48	5.68	2	15	2.07
	Group 7	5.34	0	15	2.19	6.03	2	15	2.01
	Group 8	5.38	0	15	2.19	5.83	2	15	2.11
	Group 9	5.27	0	15	2.25	6.08	2	15	2.23

Table 5.24: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for 7 communities in subset 1 and 2 communities in subset 2.

7 Communities						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	82996	9318	3034	93%	12%
	Group 2	82773	10275	3074	93%	12%
	Group 3	83058	9566	2936	92%	12%
	Group 4	82996	9179	3096	93%	12%
	Group 5	80885	9737	3065	93%	11%
	Group 6	83001	8800	2980	94%	12%
	Group 7	81141	9098	2992	93%	11%
	Group 8	67683	7267	2681	90%	9%
	Group 9	66691	7551	2604	89%	9%

Table 5.25: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for 7 communities in subset 1 and 2 communities in subset 2.

7 Communities									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.95	0	15	2.45	5.84	2	15	2.04
	Group 2	4.99	0	15	2.44	5.87	2	15	2.00
	Group 3	4.94	0	15	2.43	5.77	2	15	2.08
	Group 4	4.98	0	15	2.46	5.87	2	15	2.13
	Group 5	5.04	0	15	2.41	5.81	2	15	2.07
	Group 6	4.92	0	15	2.47	5.90	2	15	2.15
	Group 7	4.92	0	15	2.44	5.82	2	15	2.03
	Group 8	5.01	0	15	2.46	5.95	2	15	2.13
	Group 9	5.01	0	15	2.44	5.97	2	15	2.16

Table 5.26: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for 8 communities in subset 1 and 1 community in subset 2.

8 Communities						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	30406	3288	1387	89%	9%
	Group 2	30430	3515	1501	89%	9%
	Group 3	30541	3466	1569	89%	9%
	Group 4	31834	3541	1458	89%	10%
	Group 5	30358	2974	1370	90%	9%
	Group 6	30863	3581	1574	88%	9%
	Group 7	30496	3267	1480	90%	9%
	Group 8	30037	3512	1452	90%	9%
	Group 9	89848	9907	4319	100%	27%

Table 5.27: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for 8 communities in subset 1 and 1 community in subset 2.

8 Communities									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	5.05	0	14	2.30	5.63	2	14	1.92
	Group 2	5.07	0	15	2.34	5.79	2	15	2.16
	Group 3	5.08	0	15	2.31	5.74	2	15	2.03
	Group 4	5.04	0	14	2.34	5.82	2	14	2.11
	Group 5	5.08	0	15	2.32	5.74	2	15	2.12
	Group 6	5.10	0	15	2.31	5.71	2	15	2.15
	Group 7	5.08	0	15	2.33	5.79	2	15	2.12
	Group 8	5.02	0	15	2.28	5.73	2	15	2.07
	Group 9	4.69	0	15	2.52	5.53	2	15	2.14

Table 5.28: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for 9 groups equally sharing all the keys in the network.

9 Communities						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	81274	10742	3012	92%	11%
	Group 2	81621	10232	2873	92%	11%
	Group 3	82723	9411	2978	92%	11%
	Group 4	82921	10002	3060	93%	11%
	Group 5	82295	9374	3026	92%	11%
	Group 6	82288	9819	3079	92%	11%
	Group 7	82878	9084	3028	93%	11%
	Group 8	82758	8562	3018	92%	11%
	Group 9	79907	9433	2861	92%	11%

Table 5.29: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for 9 groups equally sharing all the keys in the network.

9 Communities									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	5.02	0	15	2.42	5.83	2	15	2.08
	Group 2	4.99	0	15	2.43	5.81	2	15	2.14
	Group 3	4.96	0	15	2.48	5.94	2	15	2.12
	Group 4	5.00	0	16	2.44	5.70	2	16	2.06
	Group 5	4.99	0	15	2.46	5.97	2	15	2.15
	Group 6	5.01	0	15	2.46	5.92	2	15	1.98
	Group 7	4.94	0	16	2.47	5.87	2	16	2.02
	Group 8	4.92	0	15	2.48	5.95	2	15	2.15
	Group 9	5.02	0	15	2.47	6.01	2	15	1.99

This case study has reaffirmed the conclusion made in case 1; that the number of links within group members is directly proportional to the number of queries propagated for the group. Another observation made is that even though Figure 5.17 shows that DGD's average hop count using non-cache data is significantly worse than Chord's, the average hop count observed never goes above 5 which is still within acceptable range by Chord's standards; i.e., queries are resolved in $O(\log N)$ hops or less and in this case where the network consists of 2,000 nodes, this translates to 10 hops or less.

5.7.3. Case 3: Varying Number of Nodes – Symmetrical Communities

This case is very similar to case 1 above. The only difference is that the instead of having asymmetrical communities, all communities are the same size. Figure 5.20 and 5.7.3.2 show that in general Chord outperforms DGD which is the opposite of what was observed in case 1 above. Combining analysis of case 1, 2 and 3, we see that in order for DGD to truly improve lookup performance, two conditions need to be met:

1. First, there needs to be a sufficient number of links within group members. From Table 5.17, sufficient can be taken to mean that on average, each node needs to know of at least 1.5 other nodes in the group.
2. Second, at least 15% of the total number of queries (see Table 5.17) in the network need to belong to a community before any improved performance is observed.

From Figure 5.22 and 5.7.3.4 it is clear to see that the variance in performance of query resolution per community in DGD is minimal. This is mostly due to the fact that all queries are divided equally among all the available groups and therefore no one group has an advantage over

the other. Also, for all runs, the percentage of queries per group is 11% which is lower than the 15% determined above for improved performance. Therefore, if there are symmetrical communities in the network, then there is a low likelihood that DGD will outperform Chord.

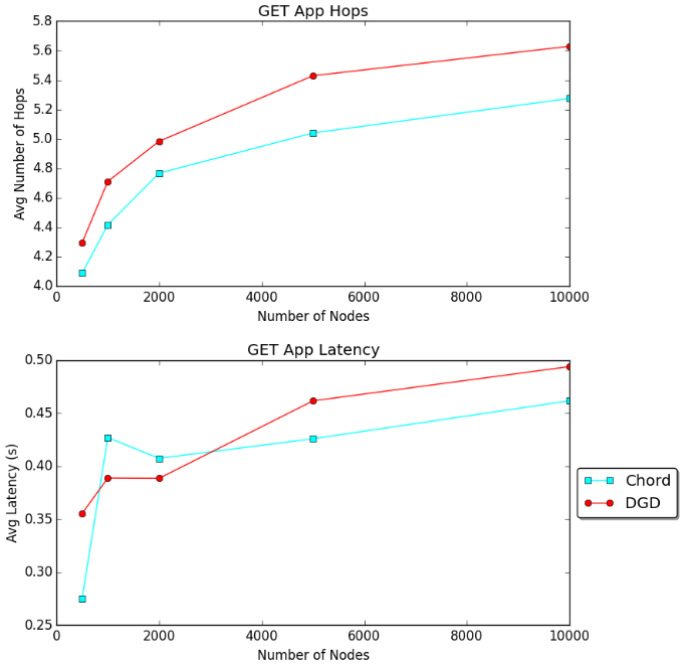

Figure 5.20: Average cost of query resolution based on non-cache data for varying number of nodes in Chord and DGD but with uniform communities.

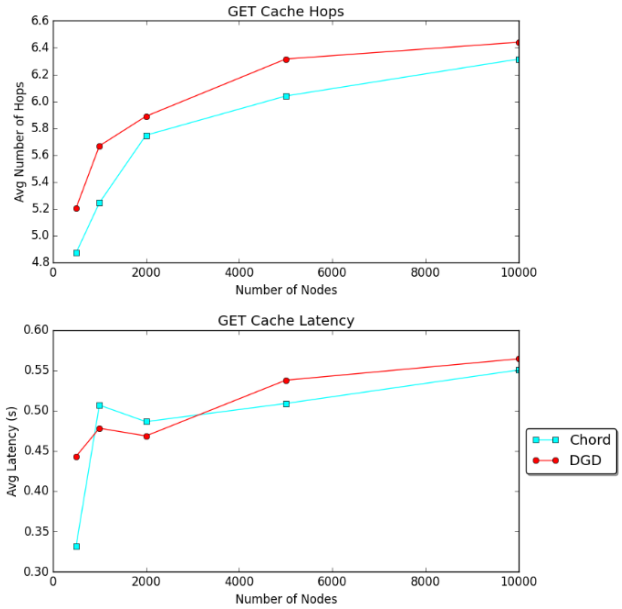

Figure 5.21: Average cost of query resolution based on cache data for varying number of nodes in Chord and DGD but with uniform communities.

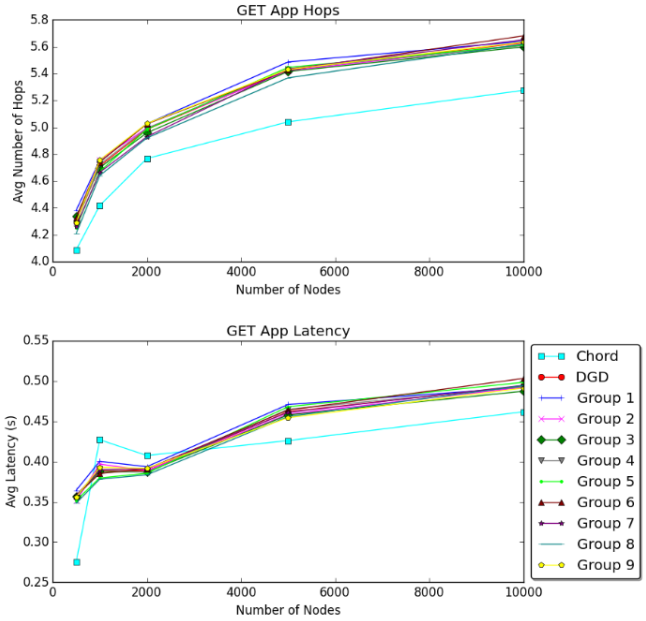

Figure 5.22: Average cost of query resolution per group based on non-cache data for varying number of nodes in Chord and DGD but with uniform communities.

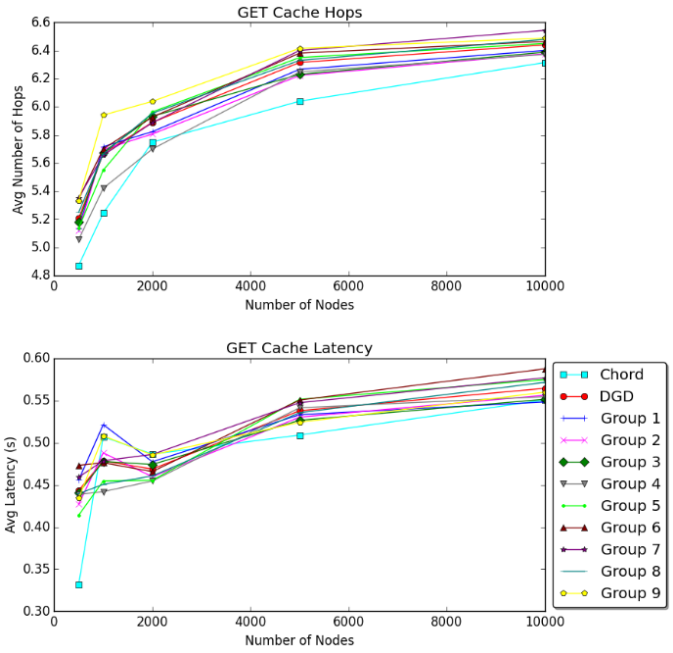

Figure 5.23: Average cost of query resolution per group based on non-cache data for varying number of nodes in Chord and DGD but with uniform communities.

Table 5.30: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 500 node network.

500 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	75322	6727	1493	100%	11%
	Group 2	78439	5491	1496	100%	11%
	Group 3	80964	5640	1490	100%	11%
	Group 4	82919	6145	1493	100%	12%
	Group 5	79194	4822	1493	100%	11%
	Group 6	82344	5854	1496	100%	12%
	Group 7	82871	4952	1497	100%	11%
	Group 8	80129	4528	1492	100%	11%
	Group 9	75972	4596	1491	100%	11%

Table 5.31: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 500 node network.

500 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.38	0	13	2.10	5.13	2	13	1.69
	Group 2	4.30	0	13	2.14	5.12	2	13	1.78
	Group 3	4.34	0	13	2.14	5.18	2	13	1.86
	Group 4	4.29	0	13	2.15	5.06	2	13	1.79
	Group 5	4.25	0	13	2.16	5.14	2	13	1.81
	Group 6	4.33	0	12	2.17	5.34	2	12	1.72
	Group 7	4.26	0	13	2.21	5.35	2	13	1.85
	Group 8	4.21	0	13	2.24	5.25	2	13	1.75
	Group 9	4.29	0	13	2.22	5.33	2	12	1.76

Table 5.32 Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 1,000 node network.

1,000 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	80116	8233	2467	99%	11%
	Group 2	82218	7969	2542	99%	11%
	Group 3	82820	7357	2533	99%	11%
	Group 4	82224	7881	2562	100%	11%
	Group 5	82789	6884	2554	99%	11%
	Group 6	80906	7769	2550	99%	11%
	Group 7	81504	6647	2554	99%	11%
	Group 8	82070	6327	2616	99%	11%
	Group 9	78546	7191	2557	100%	11%

Table 5.33: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 1,000 node network.

1,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.75	0	14	2.37	5.72	2	14	2.00
	Group 2	4.76	0	13	2.34	5.70	2	13	1.86
	Group 3	4.70	0	13	2.35	5.68	2	13	1.87
	Group 4	4.72	0	13	2.32	5.42	2	13	1.91
	Group 5	4.67	0	13	2.35	5.55	2	13	1.95
	Group 6	4.74	0	14	2.35	5.70	2	14	1.81
	Group 7	4.66	0	14	2.37	5.66	2	14	1.89
	Group 8	4.64	0	13	2.40	5.66	2	13	1.89
	Group 9	4.76	0	14	2.42	5.94	2	14	1.84

Table 5.34: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 2,000 node network.

2,000 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	81581	10765	3033	92%	11%
	Group 2	81786	10167	2871	92%	11%
	Group 3	82491	9394	2930	92%	11%
	Group 4	82922	9819	3111	93%	11%
	Group 5	82252	9310	3046	92%	11%
	Group 6	82584	10073	3140	92%	11%
	Group 7	82878	9011	3019	93%	11%
	Group 8	82761	8522	3072	92%	11%
	Group 9	80264	9314	2893	92%	11%

Table 5.35: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 2,000 node network.

2,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	5.03	0	15	2.42	5.82	2	15	2.07
	Group 2	4.99	0	15	2.42	5.81	2	15	2.15
	Group 3	4.96	0	15	2.48	5.93	2	15	2.12
	Group 4	4.99	0	16	2.44	5.70	2	16	2.06
	Group 5	4.98	0	15	2.46	5.96	2	15	2.14
	Group 6	5.02	0	15	2.46	5.93	2	14	1.95
	Group 7	4.93	0	16	2.47	5.89	2	16	2.02
	Group 8	4.92	0	15	2.48	5.95	2	15	2.16
	Group 9	5.03	0	15	2.47	6.04	2	15	2.03

Table 5.36: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 5,000 node network.

5,000 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	82410	14439	2055	65%	11%
	Group 2	82569	13164	2172	65%	11%
	Group 3	82822	12189	2087	66%	11%
	Group 4	82923	13081	2194	66%	11%
	Group 5	81746	12581	2039	66%	11%
	Group 6	82754	13133	2166	65%	11%
	Group 7	82879	12917	2277	67%	11%
	Group 8	82761	11766	2233	66%	11%
	Group 9	80426	12377	2048	65%	11%

Table 5.37: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 5,000 node network.

5,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	5.49	0	18	2.63	6.27	2	18	2.16
	Group 2	5.42	0	18	2.62	6.22	2	18	2.25
	Group 3	5.42	0	18	2.61	6.23	2	18	2.28
	Group 4	5.43	0	18	2.62	6.25	2	18	2.39
	Group 5	5.45	0	18	2.64	6.35	2	17	2.30
	Group 6	5.43	0	17	2.64	6.38	2	17	2.27
	Group 7	5.43	0	18	2.63	6.40	2	18	2.28
	Group 8	5.37	0	17	2.66	6.33	2	17	2.33
	Group 9	5.43	0	18	2.63	6.42	2	18	2.22

Table 5.38: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 10,000 node network.

10,000 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	82410	15137	1137	43%	11%
	Group 2	82569	15247	1287	43%	11%
	Group 3	82823	14210	1128	43%	11%
	Group 4	82923	14651	1325	43%	11%
	Group 5	82794	14510	1263	43%	11%
	Group 6	82755	14809	1255	43%	11%
	Group 7	82879	14475	1253	43%	11%
	Group 8	82761	13844	1234	42%	11%
	Group 9	82529	15311	1180	42%	11%

Table 5.39: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 10,000 node network.

10,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	5.64	0	19	2.70	6.40	2	19	2.34
	Group 2	5.61	0	20	2.70	6.38	2	19	2.43
	Group 3	5.60	0	19	2.66	6.39	2	19	2.33
	Group 4	5.61	0	19	2.69	6.37	2	19	2.41
	Group 5	5.61	0	20	2.69	6.45	2	20	2.37
	Group 6	5.68	0	20	2.69	6.46	2	19	2.37
	Group 7	5.65	0	19	2.68	6.54	2	19	2.32
	Group 8	5.62	0	20	2.66	6.48	2	20	2.28
	Group 9	5.63	0	20	2.67	6.49	2	20	2.30

5.7.4. Case 4: Introducing Group Churn

For this case, the experiment was set up exactly as in Case 1 but the churn feather was switched on. The simulator ran for 5,000 simulation seconds before REMOVE events (which consequently lead to deletion of entries in the group ID finger table) were scheduled. This allowed the network to settle before introducing churn for the rest of the simulation run that lasted for 10,000 simulation seconds. Also for this case, in addition to Chord and DGD, results for static group assignment (SGA) as described in [30] was also analyzed.

As expected, the trend in performance with increased network size is similar to that observed in Case 1 (see Table 5.2 to 5.11 and Table 5.40 to 5.7.4.10). The main difference noted is that DGD with churn recorded degraded performance compared to DGD without churn when resolving queries using cache data (see Figure 5.25), but performance using non-cache data for both DGD with and without churn is comparable (see Figure 5.24).

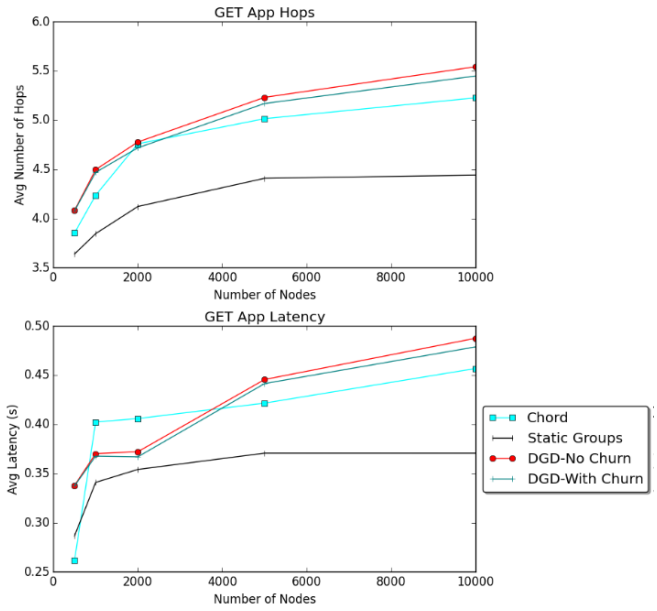

Figure 5.24: Average cost of query resolution based on non-cache data for varying number of nodes in Chord and DGD with churn, DGD without churn and static group assignment.

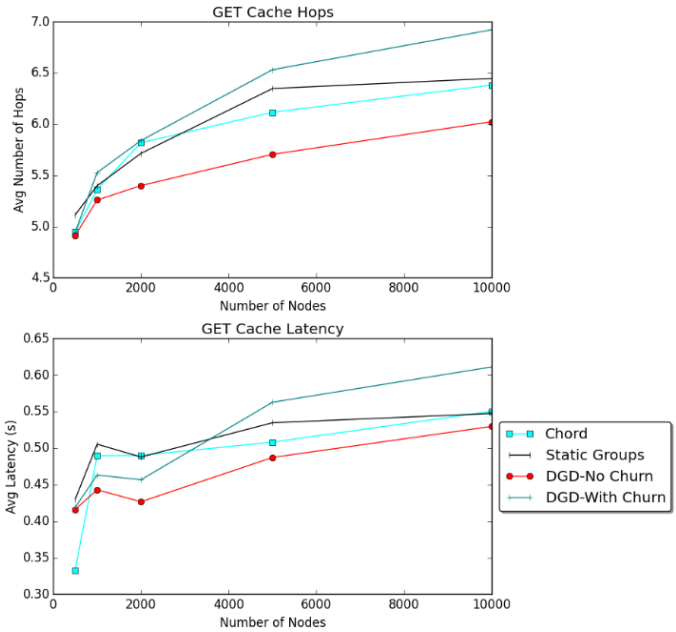

Figure 5.25: Average cost of query resolution based on cache data for varying number of nodes in Chord and DGD with churn, DGD without churn and static group assignment.

As previously explained, we expect that resolving queries using non-cache data in DGD will result in slightly degraded performance because it is a hybrid of Chord and "smart flooding." From Figure 5.24, we observe that the effects of churn on query resolution performance using non-cache data is almost negligible, however, this is not the case for queries resolved using cache data (see Figure 5.25). One consistent and important observation made in all cases discussed where the communities are asymmetrical, is that approximately 90% of all the queries propagated are resolved using application data. Based on this knowledge, we can conclude that the effects of the churn introduced easily went unnoticed when resolving queries using non-cache data, because the numbers queries affected were not enough to cause any significant effect on the overall performance. Conversely, with only 10% of the queries being resolved using cache data, the effects of churn are very evident and we see that query resolution using cache data worsened by approximately one hop.

Table 5.40: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 500 node network.

500 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	116368	7705	1500	100%	46%
	Group 2	115229	6877	1500	100%	46%
	Group 3	2919	150	204	95%	1%
	Group 4	2985	174	212	92%	1%
	Group 5	2885	114	223	93%	1%
	Group 6	2908	132	257	93%	1%
	Group 7	2826	122	197	94%	1%
	Group 8	2957	172	194	92%	1%
	Group 9	3053	130	217	90%	1%

Table 5.41: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 500 node network.

500 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.02	0	12	2.26	4.83	2	12	1.79
	Group 2	4.01	0	12	2.29	4.90	2	12	1.78
	Group 3	4.81	0	11	1.81	5.99	2	11	2.25
	Group 4	4.96	0	12	1.64	5.22	2	11	1.90
	Group 5	4.80	0	13	1.81	5.52	2	11	2.10
	Group 6	4.80	0	11	1.78	5.28	2	11	1.89
	Group 7	4.91	0	11	1.71	5.38	2	11	2.10
	Group 8	4.94	0	12	1.69	5.72	2	11	1.88
	Group 9	4.74	0	11	1.82	5.95	2	11	2.25

Table 5.42: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 1,000 node network.

1,000 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	248335	20997	3000	100%	46%
	Group 2	247293	21952	3000	100%	46%
	Group 3	6857	340	261	74%	1%
	Group 4	6905	466	316	75%	1%
	Group 5	6612	351	292	75%	1%
	Group 6	6708	529	283	75%	1%
	Group 7	6932	768	319	75%	1%
	Group 8	6969	564	279	73%	1%
	Group 9	6830	379	249	74%	1%

Table 5.43: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 1,000 node network.

1,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.41	0	13	2.46	5.15	2	13	1.92
	Group 2	4.42	0	13	2.46	5.29	2	13	1.93
	Group 3	5.27	0	13	1.86	5.84	2	13	1.82
	Group 4	5.35	0	10	1.74	5.52	2	9	1.70
	Group 5	5.23	0	11	1.83	5.58	2	11	1.73
	Group 6	5.35	0	13	1.89	5.92	2	13	1.86
	Group 7	5.47	0	13	1.78	5.79	2	13	1.69
	Group 8	5.50	0	12	1.80	5.56	2	12	1.76
	Group 9	5.33	0	10	1.88	5.67	2	9	1.60

Table 5.44: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 2,000 node network.

2,000 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	515072	53904	5991	100%	45%
	Group 2	509440	51832	5990	100%	45%
	Group 3	15772	1015	325	53%	1%
	Group 4	15685	2008	399	52%	1%
	Group 5	15830	1300	376	52%	1%
	Group 6	15951	1488	343	51%	1%
	Group 7	16079	1683	375	52%	1%
	Group 8	15757	2131	331	52%	1%
	Group 9	16019	1567	256	53%	1%

Table 5.45: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 2,000 node network.

2,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	4.66	0	15	2.54	5.31	2	15	2.21
	Group 2	4.70	0	16	2.54	5.40	2	16	2.18
	Group 3	5.53	0	13	1.89	5.89	2	13	1.75
	Group 4	5.74	0	13	1.79	5.73	2	9	1.51
	Group 5	5.57	0	13	1.91	5.82	2	10	1.73
	Group 6	5.55	0	14	1.88	5.63	2	14	1.73
	Group 7	5.64	0	14	1.84	5.83	2	9	1.60
	Group 8	5.83	0	11	1.79	5.98	2	10	1.64
	Group 9	5.59	0	11	1.84	5.85	2	11	1.67

Table 5.46: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 5,000 node network.

5,000 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	1331723	199671	14417	99%	44%
	Group 2	1302451	191502	14380	99%	43%
	Group 3	50929	7843	384	31%	2%
	Group 4	51382	10111	380	33%	2%
	Group 5	51363	7369	445	31%	2%
	Group 6	51458	8639	436	31%	2%
	Group 7	50971	9259	393	32%	2%
	Group 8	50654	10726	410	32%	2%
	Group 9	51019	7966	413	33%	2%

Table 5.47: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 5,000 node network.

5,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	5.11	0	18	2.70	5.55	2	18	2.58
	Group 2	5.13	0	18	2.71	5.68	2	18	2.65
	Group 3	6.03	0	12	2.05	6.29	2	10	1.75
	Group 4	5.96	0	12	2.07	6.20	2	11	1.79
	Group 5	6.05	0	16	2.13	6.75	2	16	2.22
	Group 6	6.02	0	12	2.06	6.48	2	12	1.91
	Group 7	6.05	0	12	2.04	6.19	2	11	1.88
	Group 8	6.09	0	12	2.06	6.26	2	11	1.73
	Group 9	6.00	0	13	2.09	6.55	2	12	2.00

Table 5.48: Summary of number of queries resolved using application data and cache data in DGD; number of links formed within each group and node membership per group for a 10,000 node network.

10,000 Nodes						
		App Count	**Cache Count**	**# of Links**	**% of Mem. Nodes**	**%age of Queries**
	Group 1	2725108	524221	26533	92%	43%
	Group 2	2643437	508973	26657	92%	42%
	Group 3	134554	27997	452	24%	2%
	Group 4	133428	37898	379	26%	2%
	Group 5	134463	32446	497	24%	2%
	Group 6	137248	34632	539	24%	2%
	Group 7	137135	29405	646	25%	2%
	Group 8	135358	38284	383	25%	2%
	Group 9	134967	28501	445	25%	2%

Table 5.49: Summary of average, minimum, maximum and standard deviation of hops needed to resolve queries for a 10,000 node network.

10,000 Nodes									
		App Mean	**App Min**	**App Max**	**App Std Dev**	**Cache Mean**	**Cache Min**	**Cache Max**	**Cache Std Dev**
	Group 1	5.40	0	20	2.80	5.91	2	20	2.95
	Group 2	5.44	0	20	2.78	5.93	2	20	2.88
	Group 3	6.19	0	16	2.19	6.65	2	16	2.00
	Group 4	6.19	0	13	2.18	6.38	2	11	1.98
	Group 5	6.13	0	13	2.26	6.55	2	12	1.96
	Group 6	6.18	0	13	2.20	6.55	2	11	1.92
	Group 7	6.13	0	14	2.22	6.54	2	12	1.92
	Group 8	6.26	0	13	2.15	6.77	2	12	1.93
	Group 9	6.21	0	13	2.24	6.49	2	12	1.98

Another important observation made in the results presented in Case 4 is that, DGD without churn outperforms SGA by approximately half a hop when cache data is used (see Figure 5.4.7.2). However, the latter case outperforms DGD by approximately one and half hops when using non-cache data to resolve queries (see Figure 5.4.7.2). The significant improvement observed in SGA could be attributed to the fact that forwarding of queries is not only based on community identification, but also the location of the next hop in the network. DGD's better performance when resolving queries with cache data could be attributed to the fact that more accurate community membership decisions are made based on a node's personal interest and that a node can be a member of multiple communities at the same time. Given that majority of the queries in the system are resolved using non-cache data, this explains why SGA has the best overall system performance when compared to Chord and DGD.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1. Summary and Conclusion

P2P applications have increased and evolved over time and with the ever increasing number of users in P2P networks, maintaining performance expectations becomes increasingly difficult. Solutions such as caching have been proposed, however, commonly used caching schemes only favor resources that are popular relative to the entire network. The reality is that query requests in P2P networks are characterized by the Zipf's law which loosely translates to; there are there are few popular queries and many unpopular queries in the network. With this type of distribution, already existing caching schemes such as PopCache and Beehive are not very effective because only a small subset of nodes in the network can benefit from them.

Communities (grouping based on similar interests) is a natural occurring phenomenon in P2P networks. With an aim to optimize caching, a Community Based Caching (CBC) algorithm is proposed in [29] and confirms that caching based on community interests indeed improves lookup performance in P2P networks. However, the simulation network used to confirm results in [29] contained nodes were limited to belonging to only one group and where community assignment was done in a centralized manner. In addition, once a node joined a community, it could not change its membership. As discussed in Chapter 1 and 2, community membership is something that should happen dynamically over time and nodes allowed to be members of multiple communities at the same time.

In this thesis, we focus of developing a decentralized dynamic community formation algorithm that is defined in Chapter 4. Keys with group identification information appended to

them are used to aid in group discovery. The proposed key generation algorithm would be something that is known by all nodes in the network and therefore all nodes would be able to extrapolation vital group ID information and search of potential group members when applicable. Details of simulation implementation are covered in Chapter 5 and also cases presented to characterize and determine DGD's effect on CBC are analyzed. Findings show that even though DGD improved lookup performance when using CBC, this is only the case when there is no churn in the network. The work presented in has not only began paved the way for decentralized and dynamic group discovery in P2P networks, but it has also shown there is great potential to improve lookup performance in networks that are constantly growing with robust solutions.

6.2. Future Work

An important observation made in Chapter 5 is that static group assignment (SGA) as presented in [29] resulted in significant improvement when resolving queries using non-cache data. This was attributed to the fact that forwarding decisions were not only based on communities, but also location of the next hop in the network. Similarly, in an attempt to improve DGD's performance, replacing "smart flooding" with logic where the distance of the next hop is considered in addition to group membership, could potentially aid in improving DGD's overall performance.

Another observation made in Chapter 5 is that churn only affected query resolution using cache data. In order to find DGD's optimal region of performance, it would be desirable to find a consistent way of measuring churn in the network and then varying the churn rate and observing its effect on DGD and CBC.

The dynamic group discovery algorithm proposed is dependent on each node knowing exactly how many nodes are present in the network at any given time. This information is used to determine how many attempts should be made at resolving a query in the group level before resorting to using the underlying algorithm. When testing the algorithm in Chapter 5, the number of nodes present in the network was always know. However, in a real network, nodes are constantly joining and leaving the network and in order for the algorithm to work, nodes need to find a way to accurately determine the number of total nodes present.

DGD is also dependent on maintaining an additional group ID finger table. Knowing that resources are finite, it becomes increasingly desirable to find ways of finding the optimal group ID finger table size and also finding optimal ways of adding entries into the table when a group member is discovered. For example, if each node checked its already existing finger table to see how far the potential new group ID finger entry is located relative to other fingers that already exists, then this may be one way of optimizing entries in the group ID finger table.

Additionally, testing DGD's performance when used on top of other types of structured P2P networks such as Pastry and Content Address Networks could be done to ensure DGD's portability and robustness.

REFERENCES

[1] Bitcoin, (2014, Dec 20), "Bitcoin" [Online]. Available: https://bitcoin.org/.

[2] L. A Adamic and B. A. Huberman, (2014, Feb 2), "Zipf's law and the Internet" [Online]. Available: http://www.hpl.hp.com/research/idl/papers/ranking/adamicglottometrics.pdf.

[3] Ernesto, (2014, Oct 30), "Popcorn Time Under 'Legal Investigation,' Switches Domain" [Online], Available: https://torrentfreak.com/popcorn-time-legal-investigation-switches-domain-141014/.

[4] I. Stocia et al., "Chord: a scalable peer-to-peer protocol for internet applications," in IEEE/ACM Transactions on Networking, vol. 11, no. 1, Feb. 003, pp.17-32.

[5] Tomnod, (2014, March 11),"Tomnod" [Online]. Available: http://www.tomnod.com/.

[6] K. L. Painter, (2014, March 11)," Digital Globe lunches crowdsource search for missing Malaysian plane" [Online]. Available: http://www.denverpost.com/business/ci_25315311/digitalglobe-launches-crowdsource-search-missing-malaysian-plane.

[7] R. Wilson, (2014, Feb 10), "Trending on Twitter: A look at Algorithms Behind Trending Topics*"* [Online]. Available: http://www.ignitesocialmedia.com/twitter-marketing/trending-on-twitter-a-look-at-algorithms-behind-trending-topics/.

[8] Higgs, P. Brocken, "Symmetries and the masses of gauge bosons." Phys. Rev. Lett. 13, 508 (1964).

[9] M. De Domenico, A. Lima and M. Musolesi, "The Anatomy of a Scientific Rumor" in Nature, vol. 3, no. 10, 2013. [Online]. Available: http://dx.doi.org/10.1038/srep02980.

[10] K. Liu et al., "Client-side Web Mining for Community Formation in Peer-to-Peer Environments," in The ACM SIGKDD Exploration Newsletter, vol. 8, no. 2, pp. 11-20, December 2006.

[11] A. Anagnostopoulos, R. Kumara and M. Mahadian, "Influence and Correlation in Social Networks," in Proceedings of the 14th ACM SIGKDD international conference on knowledge discovery and data mining, pp. 7-15, New York, USA, 2008.

[12] D. Centrola et al., "Homophily, Cultural Drift, and the Co-Evolution of Cultural Groups," in Journal of Conflict Resolution, vol. 51, no. 6, pp. 905-929, December 2007.

[13] E. Bakshy et al., "The Role of Social Networks in Information Diffusion." In Proceedings of the 21st international conference on World Wide Web, pp. 519-528, New York, USA, 2012.

[14] M. Girvan and M. E. J. Newman, "Community Structure in Social and Biological Networks," in Proceedings of the National Academy of Sciences of the United States of America, vol. 99, no. 12, pp. 7821-7826, USA, June 2002.

[15] M. E. J. Newman, "Modularity and Community Structure in Networks," in Proceedings of the National Academy of Science of the United States of America, vol. 103, no. 23, 8577-8582, USA, June 2006.

[16] Y. Ahn, J. P. Bagrow and S. Lehmann, "Link Communities Reveal Multiscale Complexity in Networks," in Nature, vol. 466, no. 7307, pp. 761-764, August 2010.

[17] J. Yang and J. Leskovec, "Defining and Evaluating Network Communities Based on Ground Truth," in Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics, article no. 3, New York, USA, August 2012.

[18] S. L. Feld, "The focused organization of social ties," in American Journal of Sociology, vol. 86, no. 5, pp. 1015-1035, March 1981.

[19] G. Simmel, "Conflict and the web of group affiliations." in Place of publication: Free Press, New York 1964.

[20] J. Yang and J. Leskovec, "Community-Affiliation Graph Model for Overlapping Network Community Detection," in 2012 IEEE 12th International Conference on Data Mining, pp. 1170-1175. 2012.

[21] G. Flake, S. Lawrence, and C. Giles, "Efficient Identification of Web Communities," in Proceedings of the sixth ACM SIGKDD international conference on knowledge discovery and data mining, pp. 150-150, New York, USA, 2000.

[22] J. Vassileva, R. Cheng, L. Sun and W. Han, "Stimulating User Participation in a File-Sharing P2P System Supporting University Classes," in P2P Journal, July 2004.

[23] Y. Wang and J. Vassileva, "Trust-Based Community Formation in Peer-to-Peer File Sharing Networks," in Proceedings of the IEEE/WIC/ACM International Conference on Wen Intelligence, pp. 341-348, Washington, DC, USA, 2004.

[24] K. Liu et al., "Client-side Web Mining for Community Formation in Peer-to-Peer Environments," in ACM SIGKDD Explorations Newsletter, vol. 8, no. 2, pp. 11-20, New York, USA, December 2006.

[25] T. Das, S. Nandi and N. Ganguly, "Community Formation and Search in P2P: A Robust Self-Adjusting Algorithm," in Proceedings of the first international conference on COMmunication, Systems And NETworks, pp. 1-8, New Jersey, USA, 2009.

[26] M. Khambatti, K. Ryu and P. Dasgupta, "Peer-to-Peer Communities: Formation and Discovery," in P2P Journal, 2002.

[27] M. Granvovetter, "The Strength of Weak Ties," in American Journal of Sociology, vol. 78, no. 6, pp. 1360-1380, USA, May 1973.

[28] M. G. Madiseh, M. L. McGuire and S. W. Neville, "Secret Key Generation within Peer-to-Peer Network Overlays," in P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference, pp. 156-163, Victoria, BC, November 2012.

[29] H. M. N. D. Bandara and A. P. Jaysumana, "Community-Based Caching in Enhanced Lookup Performance in P2P Systems," in Parallel and Distributed Systems, IEEE Transactions, vol. 24, no. 9, pp. 1752-1762, USA, September 2013.

[30] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," Proceedings of the IFIP/ACM International Conference on Distributed System Platforms (Middleware), pp. 329-350, London, UK, November 2001.

[31] S. Ratnasamy et al., "A Scalable Content-Addressable Network," in Proceedings of the 2001 conference on Applications, technologies, architectures and protocols for computer communications, pp. 161-172, New York, USA, August 2001.

[32] M. Kelaskar et al., "A Study of Discovery Mechanisms of Peer-to-Peer Applications," in Cluster Computing and the Grid, 2002 IEEE/ACM International Symposium, pp. 444, May 2002.

[33] V. K. Paruchuri, A. Durresi and R. Jail, "Optimized Flooding Protocol for Ad hoc Network," unpublished.

[34] I. Baumagart, B Heep and S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework," in IEEE Global Internet Symposium, pp. 70-84, May 2007.

[35] S. A. Baset and H. G. Schulzrinne, "An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol," in Proceedings of the 25th IEEE International Conference on Computing Communications, pp. 1-11, Barcelona, Spain, April 2006.

[36] N. Christin, "Peer-to-Peer Networks: Interdisciplinary Challenges for Interconnected Systems," in Information Assurance and Security Ethics in Complex Systems: An Interdisciplinary Perspective, 2010.

[37] V. Ramasubramanian and E. G. Sirer, "Beehive: O(1) Lookup Performance for Power-law Query Distributions in Peer-to-Peer Overlays," in Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation, vol. 1, pp. 8, Berkeley, California, USA, May 2004.

[38] W. Rao et. al., "Optimal Proactive Caching in Peer-to-Peer Network: Analysis and Application," in Proceedings of the 16th ACM conference on information and knowledge management. pp. 663-672, New York, USA. November 2007.

[39] G. Camarillo, (2014, Feb 2), "Peer-to-Peer (P2P) Architecture: Definition, Taxonomies, Examples and Applicability," [Online]. Available: https://tools.ietf.org/html/rfc5694

[40] L. A Adamic, (2014, Feb 2), "Zipf's, Power-laws and Pareto – a ranking tutorial," [Online]. Available: http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html.

[41] Wikipedia: BitTorrent. Available: http://en.wikipedia.org/wiki/BitTorrent.

[42] Wikipedia: The Pirate Bay. Available: http://en.wikipedia.org/wiki/The_Pirate_Bay.

[43] (2015, March 1) "Data, data everywhere" in The Economist. Available: http://www.economist.com/node/15557443.

[44] Internet World Status. Available: http://www.internetworldstats.com/stats.htm.

[45] Multi-Agent Distributed Mobile and Ubiquitous Computing Lab (2014, December 12). Available at: http://madmuc.usask.ca/peer-motivation.htm

APPENDIX A

A.1. Oversim Simulation Introduction

Oversim [36], is an event driven and flexible network simulator framework which was chosen to simulate and analyze DGD performance. This sections includes routines, written in C++ programming language, which illustrate how the simulator was used to achieve desired results. Some of the functional implementation included in this section include; establishing group interest, forwarding messages based on acquired group interest, handling find group member requests and replies and finally, deleting group members from the group ID finger table.

A.2. Establishing and Acquiring Group Interest

Per the algorithm described in Chapter 4, a node's initial basis for group interest is established when it joins the network and is based on personal interests. The routine called to set this up is highlighted in the `void MyChord::addGroupIdFinger(…)` function below. Nodes were also able to acquire interest at runtime depending on what type of queries being processed. Interest acquisition was handled in the `NodeVector* MyChord::findNode(…)` routine and the snippet of code with this logic is presented below.

```
void MyChord::addGroupIdFinger(const OverlayKey& key, const NodeHandle& sourceNode)
{
    int groupID = 0;

    groupID = getgroupID(key);

    bool isAlreadyMemeberOfGroup =  fingerTable->groupExists(groupID);
    /**
        Adding node to finger table vector.
        NOTE: 'sourceNode' when this function is called must not be
              another node in the network
    **/
    bool addSuccess = fingerTable->addGroupIdFinger(groupID,sourceNode);

    if(!addSuccess)
        throw cRuntimeError("MyChord::addGroupIdFinger: Problem with
                                adding a groupID finger");

    if (groupID >= 100 && groupID < 200)
    {   if(!isAlreadyMemeberOfGroup)
        {
            addToDynamicGroupIdInterestList(groupID);
            groupID_1_stats++; // Record statistics
            RECORD_STATS(globalStatistics->recordOutVector("MyChord:
                        Group 1 Size", groupID_1_stats));
            RECORD_STATS(globalStatistics->recordOutVector("MyChord:
                            Group 1 Address", thisNode.getKey().hash()));
        }
    }
    else if (groupID >= 200 && groupID < 300)
    {
        if(!isAlreadyMemeberOfGroup)
        {
            addToDynamicGroupIdInterestList(groupID);
            groupID_2_stats++;
            RECORD_STATS(globalStatistics->recordOutVector("MyChord:
                        Group 2 Size", groupID_2_stats));
            RECORD_STATS(globalStatistics->recordOutVector("MyChord:
                            Group 2 Address", thisNode.getKey().hash()));
        }
    }

    ...

    else if (groupID >= 900 && groupID < 1000)
    {
        addToDynamicGroupIdInterestList(groupID);
        if(!isAlreadyMemeberOfGroup)
        {
            groupID_9_stats++;
            RECORD_STATS(globalStatistics->recordOutVector("MyChord:
                            Group 9 Size", groupID_9_stats));
            RECORD_STATS(globalStatistics->recordOutVector("MyChord:
                        Group 9 Address", thisNode.getKey().hash()));
        }
    }
}
```

```
NodeVector* MyChord::findNode(const OverlayKey& key, int numRedundantNodes,
                              int numSiblings, BaseOverlayMessage* msg)
{

    ...

    if (findGroupMembers && (msg->getEncapsulatedPacket() != NULL))
    {
        BaseOverlayMessage* innerMsg =
              static_cast<BaseOverlayMessage*>(msg->getEncapsulatedPacket());

        if(    (innerMsg->getEncapsulatedPacket() != NULL)
            && (innerMsg->getType() == APPDATA))
        {
            MyDHTGetMessage* dhtGetMsg =
          dynamic_cast<MyDHTGetMessage*>(innerMsg->getEncapsulatedPacket());

            // Message received is a valid DHT GET message
            if(dhtGetMsg != NULL)
            {
                bool isNexHopSet = false;
                // Checking to see if we can add this node to
                // our groupID finger table
                if( useDynamicGroupAlloc && dhtGetMsg != NULL )
                {
                    bool sendFindGroupRequest = false;
                    // Extract group ID information
                    int dynamiceGroupID = getdynamiceGroupID(key);
                    // Check if interest in group ID exists
                    if(fingerTable->groupRangeExists(dynamiceGroupID))
                    {
                        if (dhtGetMsg->getMyHopCount() < maxGETHops)
                        {
                            GroupMap* groupVect =
                              fingerTable->
                              getGroupIdFingers(dynamiceGroupID, thisNode);

                            // We have finger to node in same group
                            if (groupVect->size() != 0)
                            {
                                // Set next hop
                                nextHop = new NodeVector();
                                for(unsigned int i=0;
                                    i < groupVect->size(); i++)
                                {
                                    // If it's not mine
                                    if(   groupVect->at(i).sourceNode
                                       != thisNode)
                                    {
                                        groupVect->
                                        at(i).sourceNode.setPort(1024);
                                        nextHop->
                                        push_back(groupVect->
                                                  at(i).sourceNode);
                                        isNexHopSet = true;
                                    }
                                }
                                nextHop->downsizeTo(numRedundantNodes);
                            }
                            delete groupVect;

                            // Is max number of fingers reached for group?
                            && Is max entries for table reached?
                            if (   (!fingerTable->
                                    isMaxFingersReached(dynamiceGroupID,
                                             maxFingersToGroups, thisNode)
                                   )
```

```
                        && (fingerTable->getGroupSize() <
                            (unsigned)maxGroupIdTableSize)
                           )
                    {
                        sendFindGroupRequest = true;
                    }
                }
            }//if(groupFingerTable->groupExists(dynamiceGroupID))

            // Build and send 'find group member' request
            if (sendFindGroupRequest)
            {
                MyFindGroupMemberMessage* findMemberMsg =
                   new MyFindGroupMemberMessage("FINDGROUPMEMBER");
                findMemberMsg->setCommand(FINDGROUPMEMBER);
                findMemberMsg->setOriginalSrcNode(getThisNode());
                findMemberMsg->setGroupId(dynamiceGroupID);
                findMemberMsg->setHopCount(0);
                sendMessageToUDP(successorList->
                                 getSuccessor().nodeHandle,
                                 findMemberMsg
                                 );
            }

            if(isNexHopSet)
                return nextHop;
        }//if( useDynamicGroupAlloc && dhtGetMsg != NULL )

        // groupID == -1 means that it is invalid
        if(!isNexHopSet && useDynamicGroupAlloc)
            nextHop = closestPreceedingNode(key, -1);
        else
            nextHop = closestPreceedingNode(key, -1, true);
    }
    else
        nextHop = closestPreceedingNode(key, -1);
}
else
    nextHop = closestPreceedingNode(key, -1);
}

...

}
```

A.3. Handling 'Find Group Member' Requests and Replies

Nodes discovered other group members in the network with the help of special 'Find Group Member' messages. This code snippets below, i.e., `void MyChord::handleFindGroupMemberRequest(…)` and `void MyChord::handleFindGroupMemberReply(…)`, show how each node processes 'Find Group Member' messages to achieve the primary goal, which is to discover as many group members as possible, all while being cognizant of the availability of limited resources.

```
void MyChord::handleFindGroupMemberRequest(MyChordMessage* chordMsg)
{
    MyFindGroupMemberMessage* findGroupMsg =
        check_and_cast<MyFindGroupMemberMessage*>(chordMsg);

    if(useDynamicGroupAlloc)
    {
        GroupMap* groupVect = fingerTable->getGroupIdFingers(
                                              findGroupMsg->getGroupId(),
                                              thisNode);

        if ((groupVect->size() == 0) &&
            (findGroupMsg->getHopCount() < (maxFindGroupMemberHops - 1)))
        {
            //// If no exact match, forward message to next node
            MyFindGroupMemberMessage* forwardMsg =
                new MyFindGroupMemberMessage("FINDGROUPMEMBER");
            forwardMsg->setCommand(FINDGROUPMEMBER);
            forwardMsg->setOriginalSrcNode(
                                    findGroupMsg->getOriginalSrcNode());
            forwardMsg->setGroupId(findGroupMsg->getGroupId());
            forwardMsg->setHopCount(findGroupMsg->getHopCount() + 1);
            sendMessageToUDP(successorList->
                                getSuccessor().nodeHandle, forwardMsg);
            return;
        }
        else if (groupVect->size() != 0)
        {
            MyFindGroupMemberReply* replyMsg = new
                            MyFindGroupMemberReply("FINDGROUPMEMBERREPLY");
            replyMsg->setCommand(FINDGROUPMEMBERREPLY);
            replyMsg->setGroupId(findGroupMsg->getGroupId());
            replyMsg->setSucNodeArraySize(groupVect->size());
            for(unsigned int k = 0; k < groupVect->size(); k++)
                replyMsg->setSucNode(k, groupVect->at(k).sourceNode);
            sendMessageToUDP(findGroupMsg->getOriginalSrcNode(), replyMsg);
            return;
        }
    }
}
```

```
void MyChord::handleFindGroupMemberReply(MyChordMessage* chordMsg)
{
    MyFindGroupMemberReply* findGroupReply =
        check_and_cast<MyFindGroupMemberReply*>(chordMsg);

    if(useDynamicGroupAlloc)
    {
        for(unsigned int i = 0;
            i < findGroupReply->getSucNodeArraySize(); i++)
        {
            // Is max number of fingers reached for group?
            if ((!fingerTable->isMaxFingersReached(findGroupReply->
                                                    getGroupId(),
                                        maxFingersToGroups, thisNode)))
            {
                bool add = false;
                // Is max entries for table reached?
                if (fingerTable->getGroupSize() >
                    (unsigned)maxGroupIdTableSize)
                {
                    // Need to decide what node to delete
                    add = fingerTable->
                             sepcialDeleteFromGroupFingerTable( thisNode );
                }
                else // max not reached
                    add = true;

                if (add)
                {
                    const NodeHandle& groupSourceNode =
                    fingerTable->getGroupPair(findGroupReply->getGroupId(),
                                       findGroupReply->getSucNode(i));
                    // If entry doesn't exist then add it
                    if(groupSourceNode.isUnspecified())
                    {
                        if (findGroupReply->
                                  getSucNode(i).isUnspecified())//Not given
                        {
                            std::cout
                                << "MyChord::handleFindGroupMemberReply()::
                                   Node at index "
                                << i << " is unspecified" << endl;
                            continue;
                        }
                        bool addSuccess = fingerTable->
                                         addGroupIdFinger(
                                            findGroupReply->getGroupId(),
                                          findGroupReply->getSucNode(i)
                                         );
                        if ( !addSuccess )
                            throw cRuntimeError(
                                    "MyChord::handleFindGroupMemberReply():
                                     Problem occurred when adding key");
                    }
                }
            }
        }
        return;
    }
    else if ( ... )
    {
        ...
    }
}
```

A.4. Deleting Group Members from Finger Table

As part of maintaining the group ID finger table, once a node leaves the network and consequently, is no longer a member of any group, any entries of such nodes are deleted from the group ID finger table. To simulate this, a special DHT remove message was used and the code snippet below highlights the logic used to delete entries from the group ID finger table.

```
NodeVector* MyChord::findNode(const OverlayKey& key,
                              int numRedundantNodes,
                              int numSiblings,
                              BaseOverlayMessage* msg)
{    ...

    if (useDynamicGroupAlloc)
    {
        MyDHTRemoveMessage * dhtRemoveMsg = NULL;
        if(msg->getEncapsulatedPacket() != NULL)
        {
            BaseOverlayMessage* innerMsg = static_cast<BaseOverlayMessage*>
                                        (msg->getEncapsulatedPacket());
            if(  (innerMsg->getEncapsulatedPacket() != NULL)
               &&(innerMsg->getType() == APPDATA))
                dhtRemoveMsg = dynamic_cast<MyDHTRemoveMessage*>
                                    (innerMsg->getEncapsulatedPacket());

            // If message is a remove message
            if(dhtRemoveMsg != NULL)
            {
                // Checking to see if exact match exists
                if(fingerTable->groupExists(dhtRemoveMsg->getGroupId()))
                {
                    // Get fingers with exact group
                    GroupMap* groupVect = fingerTable->
                              getGroupIdFingers(dhtRemoveMsg->getGroupId(),
                                                          thisNode);
                    // We have entry in table
                    if (groupVect->size() != 0)
                    {
                        // Loop through vector requesting to delete items
                        // that don't belong to me
                        bool status;
                        for(unsigned int i=0; i < groupVect->size(); i++)
                        {
                            if(groupVect->at(i).sourceNode != thisNode)
                            {
                                status = fingerTable->deleteGroupIdFinger(
                                              dhtRemoveMsg->getGroupId(),
                                            groupVect->at(i).sourceNode);
                                if (!status)
                                    throw cRuntimeError("MyChord::findNode:
                                            problem with deleting node...");
                                GroupMap* groupVect2 = fingerTable->
                                                    getRangeGroupIdFingers(
                                              dhtRemoveMsg->getGroupId(),
                                                          thisNode);
                            }
                        }
                    }
```

```
                    delete groupVect;
                }

                if (dhtRemoveMsg->getCurHops() <
                    dhtRemoveMsg->getMaxHops())
                {
                    dhtRemoveMsg->setCurHops(dhtRemoveMsg->getCurHops()
                                             + 1);
                    // forward message to all neighbours in groupID
                    // finger table

                    nextHop = new NodeVector();
                    GroupMap* groupVect = fingerTable->
                         getRangeGroupIdFingers(dhtRemoveMsg->getGroupId(),
                                                thisNode);

                    if (groupVect != 0)
                    {
                        for(unsigned int i=0; i < groupVect->size(); i++)
                        {
                            groupVect->at(i).sourceNode.setPort(1024);
                            nextHop->push_back(groupVect->
                                                    at(i).sourceNode);
                        }
                    }
                    else
                    {
                        for (uint32_t i = 0;
                             i < successorList->getSize(); i++)
                            nextHop->push_back(successorList->
                                               getSuccessor(i).nodeHandle);
                    }

                    delete groupVect;
                    nextHop->downsizeTo(numRedundantNodes);
                    return nextHop;
                }
                else
                {
                    dhtRemoveMsg->setCurHops(dhtRemoveMsg->getCurHops()
                                             + 1);
                    nextHop = new NodeVector();
                    nextHop->push_back(thisNode);
                    nextHop->downsizeTo(numSiblings); //Reduce if too much
                    return nextHop;
                }

            }
        }
    }

    ...
}
```

LIST OF ABBREVIATIONS

DHT – Distributed Hash Tables

P2P – Peer-to-Peer

DGD – Dynamic Group Discovery

DCD – Dynamic Community Discovery

SGA – Static Group Assignment

CBC – Community Based Caching

Mem. – Member

Min – Minimum

Max – Maximum

Std – Standard

Dev – Deviation