

CDHT: A Clustered DHT Communication Scheme for Serverless (P2P) Networks

Wei Ye

Faculty of Information Technology

Monash University

Melbourne, Victoria, Australia



Thesis submitted in fulfillment of the requirement for the degree of Master
of Information Technology (by Research)

July 2007

Abstract

Serverless Peer-to-Peer (P2P) networks are becoming more and more influential on the Internet and in the computing world. These P2P systems have many advantages over their client-server counterparts, such as high availability, low maintenance cost and fault tolerance. They also make it possible to harness massive idle computing and storage resources at the edges of the Net.

Two P2P models have emerged to achieve these benefits: the unstructured model and the structured model. The unstructured model typically forwards a query message to a selected number of nodes based on local knowledge. The structured P2P model is known as the Distributed Hash Table (DHT). In DHT networks, nodes and content/files are associated with a hash key as nodeId or contentId/fileId, respectively. The query message is routed through the overlay network to the node responsible for that key.

Both the structured and the unstructured model have strengths and weaknesses. The structured model is favoured in research but not in real applications. It's argued that the unstructured model is more suitable for the real world. It's rather difficult to determine which of the two models provides the overall better solution. In contrast to many researchers simply favoring one and against the other, we provide a consolidated approach of the two models, such that the unstructured model's network-organising flexibility can complement the structured one's preciseness and the structured model's globally oriented search can complement the unstructured model's locally oriented search. As a way forward, we would like to see if the structured model could absorb the strengths of the unstructured model without giving up on its advantages. Hot spots and handling dynamic network environments are the inherent weaknesses of the DHT-based P2P networks. The solution lies in isolating the DHT overlay from the network uncertainty and seeking an efficient and complementary lookup scheme outside the DHT.

A Clustered DHT (CDHT) communication scheme is presented in this thesis. It addresses some of the fundamental problems of DHT by combining the structured model with the unstructured one. This combination leads to a new content location scheme which balances network traffic and improves the lookup efficiency for popular content.

CDHT has a hybrid-overlay architecture that forms the DHT overlay based on the Serverless Layer clusters. The architecture delegates the handling of the network uncertainty to the network-aware Serverless Layer; thus a more reliable network and content structure is constructed as the foundation of the DHT overlay. Heuristic Content Location (HCL) is proposed to exploit network locality and content popularity for balancing network traffic, autonomously, while avoiding hot spots and improving on lookup performance for popular content location. HCL works alongside the DHT lookup pattern in the CDHT architecture. The experimental results show CDHT's superiority over DHT in handling dynamic network conditions, avoiding hot spots and improving efficiency in locating popular content.

As the network has become an integral part of the computing environment, the case for having a distributed network storage has subsequently become stronger. With redundant copies stored anywhere across a network, a P2P storage system would enable files to be accessed nomadically and persistently. Such a storage system can also reduce maintenance cost to the minimum level. Serverless Network Storage (SNS) is a Serverless (P2P) Storage application based on CDHT. SNS has a four-layered architecture. The layer separation of application functions from the commonly used networking functions allows a better control over the design and management of the application. Also, the functionality of the application is vastly improved by the segregation. SNS has demonstrated better performance in locating popular content than the peer DHT-based systems. The distributed storage is made to appear as a single large network drive, with all the main features of a conventional file system being made available to the client nodes.

The results of the research connected to this thesis have been published in three papers; "Distributed Network File Storage for a Serverless (P2P) Network" published in the Proceedings of the IEEE International Conference on Networks (ICON) 2003 in Sydney Australia; "CDHT: The Design of A Clustered DHT Routing Scheme for Serverless (P2P) Networks" published in the Proceedings of IEEE ICON 2004 in Singapore and "CDHT: A Clustered DHT Communication Scheme for Serverless (P2P) Networks" published in the Proceedings of IEEE ICON 2005 in Malaysia.

To Ling

Acknowledgement

This work would never have come to fruition without the guidance, support and advice of Dr Asad Khan. His original blueprint on Serverless P2P network inspired the research into CDHT and SNS. He always set high-standard requirements to achieve high-quality work. His ongoing assistance and advice ensured that this work would have every possible chance of achieving its aim. I am especially grateful for his advice that ensured the acceptance for publication of the papers related to CDHT and SNS mentioned above.

I also acknowledge Prof Elizebath Kendall for her valuable advice and the review she provided for CDHT and SNS. Many thanks are also extended to Professor Balasubramaniam Srinivasan for his support for this work. Professor Srinivasan became my associate supervisor in the late period of my candidature. It is a great pity that I could not have had more time to benefit from his advice.

Finally, I would like to extend my thanks to the Faculty of Information Technology, and especially the former School of Network Computing. It is an honour to have studied and worked in such a great environment, as well as meeting such wonderful and helpful friends.

Declaration

To the best of my knowledge, this thesis contains no material which has been accepted for the award of any publication or any degree in any university except where reference is cited.

Wei Ye

Faculty of Information Technology

Monash University

Melbourne, Victoria

Australia

Contents

Abstract	2
Acknowledgement	5
Declaration	6
Chapter 1 Introduction	8
1.1 Research Motivation and Contribution	9
1.2 Thesis Overview	11
Chapter 2 Characteristics of Unstructured P2P Model	12
2.1 Nature and history of Unstructured P2P Overlay	13
2.2 Unstructured P2P Overlays based on Power-law or Small-world network model	18
Chapter 3 Characteristics Of Structured P2P Model	24
3.1 DHT Organization	24
3.2 Various DHT Networks	25
Chapter 4 Unstructured VS Structured P2P Model	32
4.1 Content request of P2P networks	33
4.2 Comparison between structured and unstructured model	36
4.3 Our approach and research motivation	42
Chapter 5 CDHT Communication Scheme	44
5.1 Serverless Layer	45
5.2 Heuristic Content Location Scheme	54
5.3 CDHT Architecture	60
5.4 Experimental Results	64
Chapter 6 Serverless P2P Network Storage	71
6.1 Related Work	72
6.2 Serverless (P2P) Network Storage	74
Chapter 7 Conclusion	83
Reference	86
Appendix	93

Chapter 1 Introduction

Since the advent of the client/server model, the Internet has experienced a revolutionary change with an explosive increase in the number of PCs and edge devices. On the one hand, the nature of the client/server model made a small number of well-known servers remain under high demand and become hubs of the Internet; on the other hand, a large number of clients remained under utilized. Parallel with the change on the Internet, Moore's Law had been effective, resulting in exponential growth of computing power on individual PCs. There was thus a huge untapped resource potential on the Internet, including CPU power, storage capacity, and bandwidth waiting to be utilized. The client/server model failed to leverage that unused potential, whilst constantly pursuing upgrades on servers and the network to meet increasing high access demand and avoid network congestion.

Coupled with the increased costs and overhead of maintaining and providing services through the client/server model, a new paradigm "peer-to-peer networking" has gained precedence and consequently enormous attention in the last few years. This technology aims to harness the unused capacity and potential of the large user base existing on the Internet.

Peer-to-Peer (P2P) technology has been widespread since the music exchange online project Napster. Although Napster was server based, the server was only used for directory services and the core of the technology was P2P based. It was regarded as the typical representative of the first-generation P2P networks, thus its technical advantages and disadvantages have been widely studied in-depth. The overlay simplicity made Napster highly resilient to rapid network changes e.g. nodes entry and exit. The content location (routing) scheme in Napster was via message flooding or broadcasting. This tended to result in lookup inefficiency and network congestion.

Two different approaches, the unstructured model and the structured model, have been taken to tackle the message-flooding issue and improve upon the scalability in P2P networks. The unstructured model retains overlay simplicity, as well as employs certain network models such as Random Walks [1] [2] and Small World [3] [4] to make P2P

networks scalable. The structured model is referred to as Distributed Hash Table (DHT) based networks. DHT effectively cures the scalability issue but imposes non-trivial organizational and maintenance overheads on the network. Its lookup scheme tends to cause load imbalance since some nodes receive more routing traffic or query load than others. DHT based networks, such as Pastry [5], Tapestry [6], Chord [7] and CAN [8], are favoured research topics; in the real world, however, there are not many DHT based P2P application. Popular P2P networks such as Gnutella [9] and Freenet [10] are essentially unstructured networks.

1.1 Research Motivation and Contribution

It is important to understand the nature of P2P networks in order to understand the differences between structured and unstructured approaches. It has been stated that information request on the Internet follows Zipf's Law [11] [12] and the Internet is a network made up of heterogeneous and transient nodes [13]. These observations also relate to the popular content sharing applications which are invariably based upon unstructured P2P networks. The structured P2P approach is less favoured in real world applications since its content-requesting scheme often leads to load balancing issues and the approach is not resilient to dynamic network conditions (e.g. nodes joining and leaving the network arbitrarily). These two inherent weaknesses of the structured model, as duly noted by the proponents of the structured model [14], explain why the unstructured P2P network model is still generally favoured among popular applications.

Both the structured and the unstructured model have strengths and weaknesses. It's rather difficult to determine which of the two models would provide the overall better solution. In contrast to many researchers simply favoring one and against the other, we provide a consolidated approach of the two models, such that the unstructured model's network-organising flexibility can complement the structured one's preciseness and the structured model's globally oriented search can complement the unstructured model's locally oriented search. As a way forward, we would like to see if the structured model could absorb the strengths of the unstructured model without giving up on its advantages.

In this thesis, a Clustered DHT (CDHT) communication scheme is presented. CDHT combines the structured and the unstructured P2P network models in way that enables these two competing approaches to complement each other.

The design of CDHT was proposed in [16] with some initial experimental results. The full CDHT communication scheme has been proposed in [17] with more comprehensive experimental results. By absorbing the unstructured model as a part of its design, the goals and contribution of CDHT are:

1. To establish a more reliable network and content distribution structure for the DHT overlay so that it can tolerate dynamic network conditions.
2. To devise a load balanced architecture.
3. To further enhance the superior content location capability of DHT based structured networks.

Heuristic Content Location (HCL) is proposed to exploit network locality and content popularity by adaptively balancing network traffic, avoiding traffic hot spots, and improving lookup performance for popular content location. HCL works alongside the DHT lookup scheme in the CDHT architecture.

CDHT is based around P2P networks with no fixed nodes being assumed in any part of the network. Such networks are also known as Serverless networks and their architecture is described in [19] [20].

The work on CDHT was undertaken in conjunction with research and development of SNS [18] - a Serverless (pure P2P) network file storage application. SNS aims at providing more versatility and mobility than its peer applications based on the pure DHTs such as CFS [21] and PAST [22]. In addition, SNS has the ability to avoid hot spots.

1.2 Thesis Overview

This thesis begins with a comprehensive characteristics analysis of the P2P unstructured and structured models, in Chapter 2 and Chapter 3 respectively. Chapter 4 analyzes content request pattern of P2P networks and reveals how the pattern relates to the two P2P models. Chapter 4 further compares the advantages and disadvantages of the two P2P models. Chapter 5 presents the architectural design of CDHT and discusses how Serverless Layer, DHT and HCL work together to form a CDHT network and make it function. Experimental results are presented in Chapter 5. In Chapter 6, a CDHT based Serverless Network Storage application – SNS is presented. Finally, the conclusion and potential future work is presented in Chapter 7.

Chapter 2

Characteristics of Unstructured P2P Model

The Internet's ancestor ARPANET was a small peer-to-peer based network. Over the last decade, Internet has evolved into a massive web comprised of millions of PCs, with the domination of the client/server model. With the advent of AIM/ICQ, Napster [23], SETI@home [25], Gnutella [9], Freenet [24] and a number of similarly influential networks, P2P was re-born. The "new" P2P networks have the same organizational philosophy as the "old" one did – decentralization. What make the "new" P2P paradigm distinctive from its ancestor are features such as variable connectivity, non-machine-centric addressing scheme, and network autonomy.

Up until 1994, Internet had one basic model of connectivity. Permanent IP addresses were assigned to machines that were assumed always on, and connected. A change to IP address was assumed to be abnormal and could take days to propagate the network. In the early days of the Internet, PCs had not been designed to be part of it. Subsequently, the invention of Mosaic made it possible for a PC to run a Web browser. A PC was connected to the Internet via a modem and, as PCs entered and left the network cloud frequently and unpredictably, connectivity remained inconsistent and varied.

The explosive increase in PC numbers on the Net resulted in the use of the dynamic IP addressing scheme which provided an immediate solution to the insufficiency of IP addresses. In doing so, the resources on the PCs (i.e. the clients with dynamic addresses) were ignored. This seemed reasonable, in the early days, since PCs did not carry significant amounts of processing and storage resources. But, over time, as hardware and software improved, the aggregate unused resources over the Net started to look enormous and attractive. However, the machine-centric addressing scheme still remained the major obstacle in accessing these resources. To break the barrier, AIM employed the people-centric addressing scheme and Napster the content-centric scheme. Their success heralded the emergence of a "new" breed of P2P networks that would break the domination of the client/server model on the Net.

To handle variable connectivity and the continuing increase in user numbers, a modern P2P network has to have good scalability, as well as the ability to self-detect, self-organize, and self-heal. In other words, it has to be autonomous. Network autonomy makes a P2P system remain workable, without human intervention, despite random network changes.

Variable connectivity, non-machine-centric addressing schemes, and network autonomy are common features of “new” P2P networks regardless of these being unstructured or structured. The following sections will introduce the characteristics of the unstructured and the structured models as well as further investigate how valuable P2P can be to the Net and what changes it has brought, or it is bringing, to the Net.

2.1 Nature and history of Unstructured P2P Overlay

Tens of millions of interconnected PCs became “dark matter of the Internet”. SETI@home is one of pioneer projects which utilized the massive distributed computation resources on the Net.

SETI is a scientific research area whose goal is to detect intelligent life signal in the universe [25]. Served from the Space Science Lab at U. C. Berkeley, SETI@home runs as a screensaver, which is indeed the client program, using “spare cycles” from more than one million PCs to process radio telescope data in search of signs of extraterrestrial intelligence. A screensaver only does its work when its host computer is idle. The client program sets up an Internet connection to the SETI@home data distribution server, obtains a work unit, and closes the connection. It then processes the data; this may take from an hour to several days, depending on the speed of the computer. When it’s finished, the client re-connects to the server, sends back the results, and gets a new work unit. Every few minutes the program writes a “checkpoint” file to disk, so that it can pick up where it left off in case the user turns off the computer.

At the first glance, SETI@home is not at all peer-to-peer. It uses an old-style, asymmetric, client/server architecture. Millions of independent clients download their

work data and upload the computed results to the central repository at the Space Science Lab. The clients don't peer with one another in any way.

But look a little deeper, and something new emerges: the clients are active participants, not just passive "browsers". What's more, the project uses the massive redundancy of computing resources to work around problems such as network reliability and availability.

SETI@home shows how the domination of the client/server model on the Internet can be broken. Millions of PCs are able to participate and contribute their resources. A few other projects, such as Napster, Gnutella, and Freenet with more peer-to-peer features, emerged at the same time or shortly after SETI@home. The common feature amongst these is that they employ a broadcasting like routing scheme to enable direct communication amongst peers. Those systems are often termed "unstructured" because of their communication pattern.

2.1.1 Gnutella

Gnutella, which is a protocol instead of software, started the decentralized peer-to-peer revolution. It reshaped the conventional approach to network applications and proved that fully decentralised P2P technology is feasible and viable. There are a number of Gnutella-compatible software written with different programming languages and run on different platforms. These software can communicate with one another.

Gnutella built an application-level network upon the Internet. The underlying structure of the Internet is completely hidden from users - who care only about the desired content name rather than IP or host addresses. Gnutella is fully decentralized in contrast to Napster's semi-decentralization. Napster users join the network by connecting to one of its servers whilst Gnutella users do so by connecting to any host on the network. In contrast to Napster, Gnutella has no directory service or any sort of central control.

Distributed intelligence

Gnutella is a distributed, real-time information retrieval system. Functionally, Gnutella differs from other similar systems in its distributed intelligence. A query issued within the network may result in different forms of responses. That is because each Gnutella-compliant software can interpret the query in its own manner and meaning. Some software looks inside of the files. Others may look at merely the filenames. Some interprets multiword queries as conjunctions whilst others as dejunctions. The Gnutella protocol acts as the interface which allows data access across heterogeneous information stores.

Message broadcasting

An application-level network is built on top of the Internet in Gnutella. There is no persistent connection or circuit in the virtual network. Gnutella utilizes the TCP based broadcasting to transmit messages over the network.

Each message is assigned a unique identifier (UUID). Every time a message is delivered or originated, its UUID is stored on the hosts it passes through. If the same message is received twice, it is not retransmitted. Each message has a Time-to-live (TTL) value. Typically, a message starts with a TTL of 7. When it passes from host to host, the TTL is decremented. When the TTL reaches 0, it is not transmitted again.

Utilizing message broadcasting is highly scalable and effective. However, considering bandwidth consumption, working this way is even worse than Napster. Broadcasting forwards a message to a set of neighboring hosts, which in turn forward to other nodes. As the system grows, searching and data retrieval becomes more and more cumbersome. A local network is easily congested if the number of Gnutella users within it reaches a certain value.

2.1.2 Freenet

Freenet is a decentralized P2P file system. It stores, distributes, and retrieves files across the Internet. It is scalable and robust; however, the most significant feature of it is

good privacy. It prevents censorship of documents and ensures anonymity for users. Similar to Gnutella, Freenet treats all its nodes equal to one another. This prevents any single point of failure or control. Requests are messages that can be forwarded through many different nodes. Messages time out after passing through a certain number of nodes.

When a reply is issued in response to a request, the reply is passed back through the request-forwarding path. Each node along the path may cache the reply locally, such that it can respond immediately to any further requests for the same file. This means that the more popular a file is, the easier it can be found.

The request and response scheme (routing scheme) ensures that a node can only identify the neighbouring node(s) along the routing path, rather than the node which initiated the request. This way Freenet achieves good anonymity for users. Theoretically a powerful user can still perform traffic analysis of the whole network to track down the requestor. In order to do so, however, it requires the control of a significant number of nodes. That control tends to be too extensive and thus incredible to effect.

Keys and documents

In order to prevent false documents from spreading on Freenet, various types of keys are utilized to index documents. Each key can be treated as an array of bytes and routed over the network. The closeness algorithm routes a key to its represented document.

Freenet defines a general Uniform Resource Indicator (URI) in the form “*free:keytype@data*”. Each key type has its own interpretation of the *data* part of the URI. There are a few key types such as Content Hash Keys (CHS), Keyword Signed Keys (KSK), and Signature Verification Keys (SVK). Each key type applies different security and encryption scheme.

All documents stored in Freenet are encrypted before insertion. The key is either random and distributed by the requestor along with URI, or based on data unknown to the operating node. A node thus cannot tell what data is contained in a document and cannot stop itself from caching or forwarding document that it may otherwise object to.

Store and request data

Each Freenet node maintains its own data store, which associates keys to addresses and/or documents. A data store is in effect a stack that applies the Least Recent Used (LRU) replacement policy. A document reference may move up or down in its stack based on its popularity. Documents that are requested more often are moved up in the stack, displacing the less requested ones. Bigger and distant documents are at a disadvantage to be kept in the stack.

A popular document may be cached locally with its key pointing to the local copy whilst an unpopular document has no local copy but only an address for the key to point to. A node looks for its local data first when it receives a request for a key. If it has the key, the request is replied immediately. If not, the node searches its data store to find the key closest to the requested one. The node referenced by the closest key is the one that the request is forwarded to.

The closeness algorithm is the cornerstone of Freenet's routing. Each node makes its own decision based on the algorithm to decide where the next message destination is. The leverage of the key closeness algorithm makes Freenet semi-structured P2P model.

Freenet's routing scheme, based on the key closeness algorithm, is much better than the Gnutell-like broadcasting scheme in terms of scalability and avoiding traffic congestion. On the other hand, the local decision of individual nodes is crucially dependent on the data store records, which are updated by the LRU cache replacement. LRU can result in significant lookup inefficiency when the number of files stored in the network is high [27].

2.1.3 Jabber

Jabber [71] [72] is a set of streaming XML protocols and technologies that enable any two entities on the Internet to exchange messages in close to real time. Unlike most instant messaging protocols, Jabber has a decentralized architecture. It is similar to email; anyone can run their own Jabber server and there is no central master server. Every user on the network has a unique *Jabber ID (JID)*. To avoid the need for a central server with

a list of Ids, a JID is structured like an email address with a username and a DNS address for the server where that user resides separated by an at sign (@). A Jabber server forwards a message to its destination server by reading the DNS address of JID.

Jabber can be viewed having a hybrid communication scheme. At the level of Jabber client and server communication, it follows the pattern of a client-server model. At the level of inter-server communication, it is indeed a peer-to-peer model. This hybrid communication scheme can be used to bridge different IM systems such as MSN messenger and AOL instant messenger. The “bridge” is known as a gateway. Any Jabber user can “register” with one of these gateways by providing the information needed to log on to that network, and can then communicate with users of that network as if they were Jabber users. In other words, any Jabber client can access a network to which such a gateway exists, without the need for the client to have direct access to the Internet.

2.2 Unstructured P2P Overlays based on Power-law or Small-world network model

The unstructured P2P model, such as Gnutella and Freenet, typically locates content with Time-to-live (TTL) controlled message-flooding mechanism. This enables the unstructured model to have significant advantage in network simplicity i.e. easy network organization and self-maintenance. However, message-flooding mechanism may cause scalability and performance disadvantages, as indicated in the previous section.

Barabasi and Albert analyzed the properties of power-law networks and how they are generated in [28]. Such networks have a common property known as scale-free power-law distribution. This feature was found to be a consequence of two generic mechanisms: (a) networks expand continuously by the addition of new vertices, and (b) new vertices attach preferentially to sites that are already well connected. The power-law distribution reflects the existence of a few nodes with very high degrees and many others with low degrees of connectivity. The feature of power-law distribution was not found in standard random graphs [28].

Recent study of Gnutella [26] and Freenet [27] showed that power-law distribution could be utilized to improve their scalability and lookup efficiency.

2.2.1 Improving Gnutella scalability by utilizing power-law distribution

Random walk is a well known technique, where a query message is forwarded to a number of randomly chosen neighboring nodes at each time. The message is called a “walker”. When the number of walkers increases, the lookup latency decreases. However, more walkers generate more query load. When the number of walks is big enough, increasing it further yields little reduction in the number of hops, but significantly intensifies the message traffic.

According to Adamic et al. [2], the number of hops (search pathlength) to reach the whole network graph in random walks tends to be

$$S \sim \ln^2(N) \quad (1)$$

where N is the size of the network graph.

The expression (1) indicates that pure random walk has lookup efficiency disadvantage when network size keeps increasing. The study in [2] suggests that a better scaling is achievable by intentionally choosing high degree nodes. A walker can select nodes by degree sequence, visiting the node with the highest degree, followed by a node of the next highest degree, etc. Random search biasing towards high degree nodes follows power-law distribution, in which the search pathlength is required as

$$S \sim N^{2-4/\tau} \quad (2)$$

In the limit $\tau \rightarrow 2$, the search pathlength achieves a scale-free effect. The degree distribution is a power-law form

$$p(x) \sim x^{-\tau} \quad (3)$$

It’s worthwhile to question the reasonableness of the limit $\tau \rightarrow 2$ fitting for network graph growth. What does the Internet look like? Are there any topological properties that don’t change over time? M. Faloutsos et al discovered power-law characteristics of the Internet topology in [29] despite Internet’s apparent randomness. Their work showed that

the Internet backbone has a power-law distribution with exponent values between 2.15 and 2.2. Other work [28] further showed that web page hyperlink on the World Wide Web (WWW) has a power-law distribution with exponent of 2.1. Those works indicate that development of some large networks is governed by scale-free power-law distribution. The exponent value close to 2 characterizes the common feature of computing network growth. Other networks may have different exponent value range. For example, the power grid of the western United States, which does not serve as a message passing network, has an exponent $\tau \rightarrow 4$ [28].

Based on the work [28], [29] and [2], search in a random network could be made more efficient by biasing random walks towards high-degree nodes. Biasing query towards high-degree nodes will furthermore reduce the load on each node. Hence accumulatively it improves the scalability of the whole network.

Gnutella utilized an unstructured overlay network which is pure decentralized. It imposes no constraint on file placement. From the network's perspective, the major advantage of Gnutella lies in the simplicity of its structure that minimizes the overhead of network organization and self-maintenance. The simplicity achieves significant robustness in relation to nodes failure. However, the cost of the simplicity is Gnutella's unscalable routing scheme which floods each query throughout the network.

Following [2] Y. Chawathe et al proposed an improved model of Gnutella in [26]. Their model biases random walks towards high-degree nodes while retaining network simplicity. It recognizes heterogeneity in peer bandwidth and incorporates that heterogeneity into its design. The overall simulation result shows that the improved model achieves much better scalability than the original Gnutella network. .

2.2.2 Improve Freenet performance by utilizing Small-world model

In 1967, an experiment was conducted by Harvard professor Stanley Milgram, involving some 160 randomly chosen volunteers living in Nebraska. These participants were each asked to pass a letter from them to a target person in Boston using only intermediaries known to one another on a first-name basis. In other words, each volunteer

would only pass their letter to a friend whom is thought possibly closest to the target. Surprisingly 42 letters reached the target with an average number of just 5.5 intermediaries.

Such a low number, compared to the then-U.S. population of 200 millions, demonstrated for the first-time *Small-world* effect. Small-world phenomenon indicates that under certain circumstances message passing with local information knowledge can be efficient. That leads to two important properties of a Small-world network (a) a low average search pathlength between two randomly chosen nodes and (b) a high clustering coefficient.

D. J. Watts and S. H. Strogatz demonstrated in [3] that a Small-world network can be constructed. They began by considering a *regular graph* which consists of a ring of n vertices, each of which is connected to its nearest k neighbors. In the regular graph, the search pathlength is $n/2k$. Since all links are contained within local neighborhoods, a regular graph is highly clustered. For k neighbors of a given vertex, the total number of possible connections among them is $k*(k-1)/2$. Since the number of neighbors is $k-1$, the value of clustering coefficient is $(k-1)/(k*(k-1)/2) = 2/k$. If n is 2048 and k is 4, the pathlength is 256 and the clustering coefficient is 0.5.

The opposite of the completely ordered regular graph is *random graph*, in which vertices are connected to each other at random. It turns out that for large random graphs, the search pathlength is approximately $\log n/\log k$, whilst the clustering coefficient is k/n . If n is 2048 and k is 4, the pathlength is 6 and the clustering coefficient is 0.0019. A random graph has much better pathlength than a regular graph. In contrast to a regular graph being highly clustered, a random graph has low clustering coefficient.

Most real-world networks lie somewhere in between strictly regular and fully random. In order to explore intermediate cases, Watts and Strogatz used a clever trick to study the in-between region. Starting with a 1000-node regular graph with k equal to 10, they “rewired” it by taking each edge in turn and, with probability p , moving it to connect to a different, randomly chosen vertex. When p is 0, the regular graph remains unchanged; when p is 1, a random graph results. Surprisingly, they found that with larger p , clustering remains high but pathlength drops drastically as shown in Figure 2.1. At a p

value of 0.01, the graph shows a hybrid characteristic. Its clustering coefficient still looks like that of the regular graph. However, its pathlength has nearly dropped to the random graph level. Watts and Strogatz dubbed the combination of high local clustering and short global pathlengths Small-world graph.

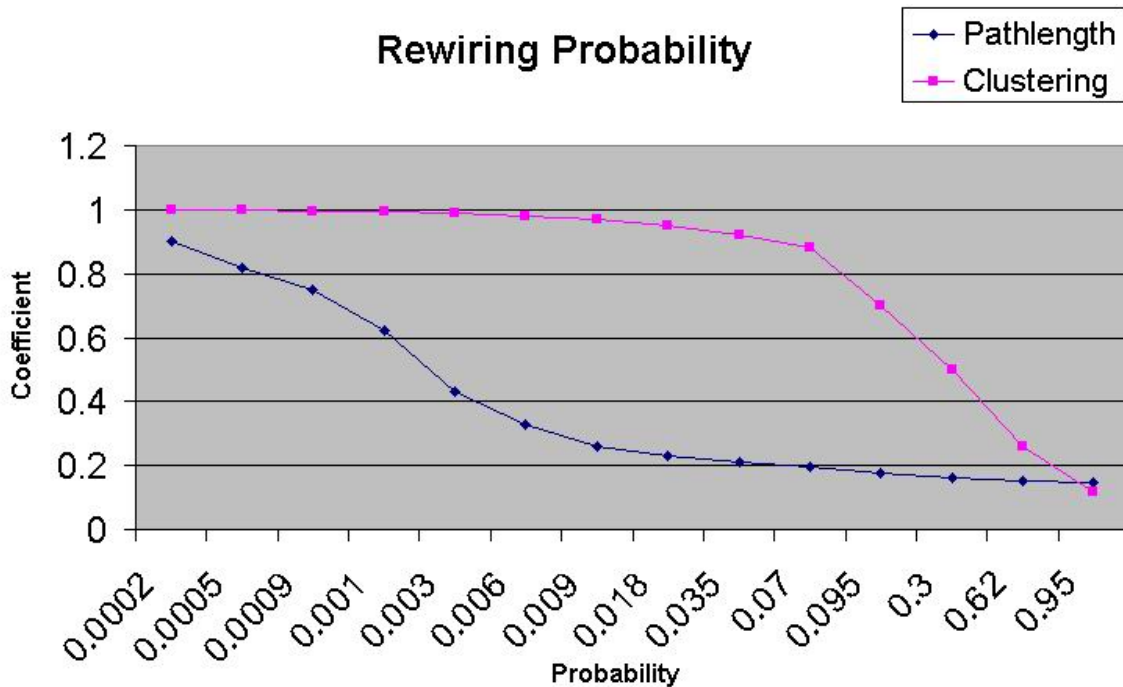


Figure 2. 1 Coefficient ratio of Pathlength and Clustering

Small-world can be utilized to improve lookup performance of the unstructured model. H. Zhang and A. Goel in [27] found that the hit ratio, which is the likelihood of finding data associated with a key within a fixed number of network hops, in Freenet is crucially dependant on the local cache policy used to manage the cache of data and the routing table. Freenet exploits Least Recent Used (LRU) cache replacement policy which can result in significantly low ratio under high load (i.e. when the number of files stored in the network is high). Zhang and Goel proposed an enhanced-clustering cache replacement scheme. It forces the routing tables to be formed in a Small-world fashion such that each node has most of its pointers to its local neighbours (clustering) as well as some randomly chosen distant nodes.

Zhang and Goel performed a number of simulations to compare their model to Freenet. The result demonstrates a significantly higher hit ratio and a smaller number of search pathlength. They even developed an idealized model of Freenet under the proposed cache

policy. The idealized model achieves the pathlength as $O(\log^2 N)$. Given a typical structured P2P network such as CAN or Chord achieving pathlength of $O(\log N)$, it is surprising that an unstructured network can be made close to the efficiency of the structured ones.

Small-world can even be useful to improve lookup performance of the structured model. A Chord based Small-world network was proposed by K. Hui et al in [30]. The proposed network achieves even better performance than Chord. It provides an efficient way to replicate popular and dynamic contents. It has some similarity to Heuristic Content Location (HCL) proposed in this thesis based on CDHT; but it differs from HCL in its biasing towards the nodes with high capacity whilst HCL biases towards the nodes with popular content. The details of HCL and more to Hui's model will be discussed in the later sections.

Chapter 3

Characteristics Of Structured P2P Model

Distributed Hash Table (DHT) networks, such as Pastry, Tapestry, Chord and CAN, have been under the spotlight in the P2P area. In these DHT networks, nodes and content/files are associated with a hash key as nodeId or contentId/fileId, respectively. Each node is required to store certain pieces of content and their keys. It also keeps a dynamic routing table, in order to route the query message that is the content key. The query message is routed through the overlay network to the node responsible for that key.

DHT was originally motivated by unstructured P2P overlays, in order to achieve better efficiency and scalability. DHT is often referred to as structured P2P overlay. DHT networks are fully decentralized and can scale up to thousands or millions of nodes. The DHT infrastructure can be used to build more complex network services, such as distributed file systems, instant messaging, and content distribution network.

3.1 DHT Organization

A DHT network is built on an abstraction called *keyspace*. A keyspace partitioning scheme allocates nodeIds that belong to the keyspace to the participating nodes to form a structured DHT overlay. A nodeId is assigned randomly when a node joins the system such that the resulting set of nodeIds is uniformly distributed across the keyspace.

The keyspace and its partitioning scheme may vary from network to network. Chord uses a circular 160 bits ID space. CAN employs a *d-dimensional* space. In Pastry, keys/fileIds and nodeIds are 128 bits in length and can be thought of as a sequence of digits in base 16. Tapestry utilizes a keyspace scheme similar to Pastry.

In a typical DHT network, a file to be published acquires a unique hash ID by converting the filename or data using certain hash function such as SHA-1 [31]. The

DHT network sends the file to a node or nodes whose `nodeId(s)` are closest to the `fileId` in the keyspace.

A file can be looked up in the following steps. The originating node generates a hash key based on the filename or its data using the same hash algorithm as the one for file publishing. The key is thus identical to the `fileId`. A *lookup(key)* query is then issued by the originating node. Any node receiving the query must forward it to a node whose ID is “closer” to the key until the node that stores the file is reached. This process is in effect the DHT routing scheme, which may vary in different networks such as Pastry, Tapestry, Chord, and CAN.

3.2 Various DHT Networks

Different DHT networks may have different routing schemes based on their own keyspace and partitioning algorithms. All DHT networks share some variant of its most essential property - for any key k , the node either owns k or has a link to a node that is *closer* to k in terms of the keyspace distance discussed in Section 3.1. When routing a message to the owner of the key k , each node receiving the message forwards it to the neighbour whose ID is closest to k . When there is no such neighbor, then the message must have arrived at the closest node, which is the owner of k .

Two key constraints on the DHT implementation are (a) to guarantee that the maximum number of hops in any route (route length) is low, so that requests complete quickly; and (b) to guarantee that the maximum number of routing entries of any node (maximum node degree) is low, so that maintenance overhead is not excessive. Of course, having shorter routes requires higher maximum degree/routing entries on individual nodes. There are various combinations of the maximum degree and the route length, to achieve good lookup efficiency and to keep low maintenance cost. The most common choice of the combinations is: *Degree* $O(\log n)$, *route length* $O(\log n)$, where n is the number of nodes in the DHT. This choice is not quite optimal in terms of degree/route length tradeoff, but such topologies allow more flexibility in choice of

neighbors. That flexibility can be utilized to avoid routing hot spots and to exploit network locality.

Chord

Each node in Chord [7] [32] maintains a routing table called *finger table*. In a Chord node n , the entries of the finger table will be those nodes matching a rule according to $n + 2^0, n + 2^1, n + 2^2, \dots, n + 2^m$ where 2^m defines the maximum number of nodes supported by the network. The fingers increase according to the power of two. Hence, each step cuts the distance to the target by half, resulting in a lookup pathlength of $O(\log n)$.

Each node in Chord maintains a *predecessor pointer* for joining the network. A node's predecessor pointer contains the Chord identifier and IP address of the immediate predecessor of that node, and can be used to walk counterclockwise around the identifier circle. Chord must perform three tasks when a node n joins the network: (a). Initialize the predecessor pointer and fingers of node n ; (b). Update the fingers and predecessor pointers of existing nodes to reflect the addition of n ; (c). Notify the higher layer software so that it can transfer values associated with keys that node n is now responsible for.

When a node n fails, nodes whose finger tables include n must find n 's successor. In addition, the failure of n must not be allowed to disrupt queries that are in progress as the system is re-stabilizing. The key step in failure recovery is maintaining correct successor pointers, since in the worst case finding predecessor can make progress using only successors. To help achieve this, each Chord node maintains a *successor-list* of its r nearest successors on the Chord ring. If node n notices that its successor has failed, it replaces that node with the first live entry in its successor list. The successor-list mechanism may also help higher layer software replicate data. An application using Chord might store replicas of the data associated with a key at the k nodes succeeding the key. The fact that a Chord node keeps track of its r successors means that it can inform the higher layer software to propagate new replicas when successors come and go.

Based on the above mechanism, Chord achieves a good balance between lookup efficiency and robustness. The use of successor-list and predecessor pointer ensures the correctness of lookup when nodes join and leave/fail randomly. However, when all successors of a node fail at the same time, Chord cannot guarantee the effective repairing of the network. This scenario has little chance to occur when a network is stable; it does, however, have implication in a highly dynamic network where a large number of mobile users join, leave or change location frequently.

Content Addressable Network (CAN)

CAN [8] uses a d -dimensional Cartesian coordinate space to implement the DHT abstraction. This coordinate space is completely logical and bears no relation to any physical coordinate system. The entire coordinate space is dynamically partitioned among all the nodes in the system such that every node “owns” its distinct zone within the overall space

The (key, value) pairs are stored in and retrieved from the virtual coordinate space. The key of a pair is first mapped to a point in the space using a hash function. The (key, value) pair is then stored at the node owning the zone that the point belongs to. The retrieval process follows the same steps as the storing process. A node that retrieves the value maps the key (by applying the same hash function) to the storing point. If the originating node does not own the storing point, the request will be routed to the node whose zone contains the storing point.

When a new node joins the network, it must be allocated its own portion of the coordinate space. In order to achieve that, the new node first randomly chooses a point in the space and issues a request to that point to join the network. The request is routed through the CAN until the occupant node whose zone contains the destination point receives it. The occupant node splits its zone in half and assigns one half to the new node. The occupant node also notifies the new node of the IP addresses of its new neighbours. When receiving the IP addresses, the new node constructs its own routing table, and announces itself to its neighbours for them to update their routing tables.

When a node leaves the network, its zone will be taken over by one of its neighbours. That taking over neighbour usually owns a current zone that is the smallest amongst all the neighbours of the leaving node. The takeover may produce a valid new zone; otherwise, the neighbour will handle both zones temporarily until the background zone re-assignment algorithm re-allocates the zones. Similar to Chord, as discussed in Section 3.2.1, simultaneous multiple node failures may result in the fragmentation of the coordinate space in a CAN network. The recovery of the network will depend on whether the zone re-assignment algorithm is able to repair the node failures and re-organize the coordinate space efficiently.

CAN proposes a variety of techniques that reduce the lookup latency. For example, each node measures the network round-trip time (RTT) to each of its neighbors. Upon receiving a lookup message, a node forwards the message to the neighbor with the maximum ratio of the progress towards the destination node in the coordinate space to the RTT.

Pastry

Pastry [5] is a prefix-based DHT routing structure. Pastry assigns a 128-bit `nodeId` to each participating node. The `nodeId` indicates the position of a node in the circular ID space. The assignment of a `nodeId` is completely random such that the Pastry nodes have a uniform distribution in the ID space.

Each Pastry node maintains a *routing table*. A routing table contains $\log_2 N$ rows and each of the rows has $2^b - 1$ entries. Each entry at row n of the routing table refers to a node whose `nodeId` shares the `nodeId` of the present node in the first n digits; the $n + 1$ th digit of the present `nodeId` is different from the entry `nodeId`'s same digit, which has $2^b - 1$ possible values. It is understandable that the more entries a routing table has, the shorter is the routing pathlength. However, more entries in the routing table results in more network maintenance overhead. The choice of b represents a tradeoff between the routing table size and the lookup efficiency. The typical value of b is 4.

Each Pastry node also maintains a *leaf set* and a *neighbourhood set*. When a node receives a search request, the node will first look for the destination nodeId in its leaf set. If that nodeId is found in its leaf set, the request will be forwarded directly to the node with that nodeId. The search process finishes there. If the destination nodeId is not in the leaf set, the prefix-based routing table will decide where to forward the request next. The neighbourhood set is used to maintain the properties of network locality such that a request has more possibility to be forwarded to a node which has closer routing distance to the present node.

In the Pastry routing process, each step cuts the routing distance by a factor of 2^b . It results in reaching the destination node in $\log_2 N$ steps. If any of the routing steps involves a leaf set, the routing pathlength will be even shorter than $\log_2 N$. There is a very small possibility that one node on the routing path does not exist or fails such that the request can't reach its target. In this case, similar to other DHT structures, Pastry will have to repair the routing tables and leaf sets when coming across an unreachable node.

Tapestry

Tapestry [6] is very similar to Pastry but differs in its approach to mapping keys to nodes based on suffix instead of prefix. Tapestry is based on a lookup scheme developed by Plaxton et al. [33].

Plaxton Data Structure

Plaxton et al. present in [33] a distributed network data structure for object location and message routing. Plaxton data structure guarantees a limited delivery time, by combining location of an object with routing to its location. *However, Plaxton makes the assumption that the Plaxton data structure is static, without node changing or object insertions and deletions.*

In the Plaxton data structure, a node N has a *neighbour map* with multiple levels. Each level represents a matching suffix up to a digit position in the ID space. Each level of the neighbour map contains a number of entries close to the local node in a pattern such as

the following: for example, the 7th entry of the third level for node 6389 is the node closest to 6389 in network distance that ends in 789, where i th entry in the j th level is the closest node that ends in “ i ”+suffix($N, j-1$).

In the routing process, a node receiving the query message shares a suffix of at least n digits with the destination nodeId. When the receiving node decides the next node to route the message to, it looks at the $n + 1$ th level of its neighbour map to find the entry with the next digit matching the destination nodeId. This routing mechanism ensures the routing path length less than $\text{Log}_b N$, where N as the network size and b as the number of entries at each routing level. It further yields the size of a neighbour map as $b \cdot \text{Log}_b N$.

Tapestry Routing

Based on the Plaxton data structure, Tapestry generally follows Plaxton’s routing scheme. However, Tapestry differs from the Plaxton structure in keeping secondary links in the routing table for the purpose of fault tolerance. The routing table size in Tapestry is thus $c \cdot b \cdot \text{Log}_b N$, where c is the number of the secondary links.

In Tapestry, an object is published by routing a publish message towards its *root node* whose nodeId is closest to the ID of the object. Each node along the routing path stores a pointer to that object and its root node. When an object is being located, a request message is routed towards the root node of the object. The nodes along the routing path check their pointer cache. If there is a pointer mapping to that object, the request will be redirected to the mapping location. Otherwise, the request will be forwarded on until the root node is reached.

Tapestry tries to take advantage of *network proximity* or *network locality* in its routing scheme. The local nodes in the network topology are preferred to the remote nodes for receiving query messages. When an object is requested, its local copies have more chances to be retrieved than the original object itself and its remote copies. Network locality reduces the lookup latency and network overhead. However, the utilization of network locality has cost in terms of greater structural complexity, which is a major weakness of Tapestry. Also, since built upon the Plaxton data structure, the original Tapestry network does not support dynamic joining and leaving of nodes. The attempt to

support dynamic network conditions would make Tapestry even more complex to implement.

Chapter 4

Unstructured VS Structured P2P Model

DHTs effectively cure the scalability problem of P2P networks; on the other hand, they cause some side effects that need to be addressed [14]. These side effects include:

- A. *Resilience to failures or suitability in a highly dynamic network environment.* Generally all DHT algorithms have been analyzed under static conditions. P2P networks are extremely transient. It's of high importance that routing is able to function with as many nodes failing at the same time without notification.
- B. *Routing hot spots.* Hot spots are caused by content popularity or routing traffic.
- C. *Proximity Routing.* Various DHT algorithms employ different mechanisms to implement proximity routing.
- D. *Indexing and keyword search.* All DHTs retrieve data based on a unique identifier. Distributed indexing and keyword search could be made possible by adding an extra layer on top of DHT; however, how to effectively handle this without adding too much complexity and reducing efficiency remains an open question.

Given the apparent advantages and disadvantages of the unstructured and the structured models, many researchers simply favor one and against the other. Nevertheless, the nature or characteristics of P2P networks needs to be explored in order to further investigate the differences between the two models. A study done by S. Sariou et al. [13] to characterizes the two most popular P2P file sharing systems, namely Napster and Gnutella. The data collected in their measurement includes the distributions of bottleneck bandwidths, latencies between the number of shared files per peer, the distribution of peers across DNS domains, and the lifetime of the peers in the systems, i.e. how frequently peers connect to the systems and how long they remain connected. The results indicate that there is a significant amount of heterogeneity in terms of bandwidth, latency, availability, and the degree of sharing within these models. Apart from their

heterogeneity, the key characteristic of P2P networks is the content request pattern, which affects the fitness of the structured and the unstructured models.

The following sections will describe the pattern of content request within P2P networks and then compare the two models in this regard.

4.1 Content request of P2P networks

In order to study the content request pattern of P2P networks, it's important to begin with the Internet. It's been stated that information request on the Internet follows Zipf's law [11] [12].

4.1.1 What is Zipf's law

Zipf's law is named after the Harvard linguistic Professor George Kingsley Zipf (1902 – 1950) who sought to determine the frequency of use in English text of most common words. Zipf illustrated his law with an analysis of James Joyce's Ulysses. "He showed that the tenth most frequent word occurred 2,653 times, the hundredth most frequent word occurred 265 times, the two hundredth word occurred 133 times, and so on. Zipf found that the rank of the word multiplied by the frequency of the word equals a constant that is approximately 26,500".

The above example of where Zipf's law applies, to frequency of word occurrence is in English texts. As implicated by Zipf's law, the nature of communication is such that it is more efficient to place emphasis on using shorter words. Hence the most common words tend to be short and appear often. There are other examples, such as populations of cities and income of companies, which follow Zipf's law.

Zipf's law typically holds well when the "objects" themselves have a property (such as length or size), which is modelled by an exponential distribution, or other skewed distribution that places restrictions on how often "larger" objects can occur. Mathematically, Zipf's law states that the size of the i 'th largest occurrence of the object/event is inversely proportional to it's rank i :

$$f_i \propto 1 / i^\alpha \quad (i = 1, \dots, N)$$

where α is close to unity, and N is the number of distinct occurrences of the event. The simplest case of Zipf's law is a "1/ f function". Given a set of Zipfian distributed frequencies, sorted from most common to least common, the second most common frequency will occur $1/2$ as often as the first. The third most common frequency will occur $1/3$ as often as the first. The n^{th} most common frequency will occur $1/n$ as often as the first.

4.1.2 Zipf's law and the Internet

L. A. Adamic and B. A. Huberman demonstrated in [11] that Zipf's law governs many features of the Internet. The Internet is comprised of networks at many levels, and some of the most exciting consequences of Zipf's law have been discovered in this area.

The World Wide Web is a network of interconnected web pages. There are tens of thousands of sites on the World Wide Web, but not many are well known to all visitors. Measurements on the World Wide Web, by Admic in [35] and by Jeong et al. in [36], shows that the ranked distribution of visitors to web sites follows Zipf's law.

The Internet backbone is a physical network used to transmit data, including web pages, in the form of packets, from one location to another. The scale free degree distribution of the Internet backbone, shown in [37] by Faloutsos et al., implies that some nodes in the network maintain a large number of connections (proportional to the total size of the network), while most nodes have just one or two connections.

While the traditional Erdős-Rényi model [38] has a Poisson node degree distribution, the above networks approximately follow a Zipf or scale-free degree distribution $p(k) \sim k^{-\tau}$,

where k is the node degree, and τ is the scale-free exponent. To account for these observations, new random graph growth models have been developed that rely on the preferential attachment in [39] by Erdős et al.

A problem confronting Internet service providers (ISP) is to support rapidly growing web traffic while maintaining quality of service in the form of fast response time for file requests. Caching has two advantages. First, since the requests are served immediately from the cache, the response time can be significantly faster than contacting the original server. Second, caching conserves bandwidth by avoiding redundant transfers along remote Internet links. As any cache would have a finite size, it is impossible for the cache to store all the files users are requesting. Several studies [12] [40] have found the popularity of files requested follows a Zipf distribution. Hence, by applying Zipf's law, the cache need only store the most frequently requested files in order to satisfy a large fraction of users requests.

4.1.3 Zipf's law and P2P networks

P2P technology, notably P2P file sharing applications, has gained an unprecedented popularity in the past few years. Gnutella became one of most popular file sharing systems since Napster, with more than one million users at any point in time [41]. Gnutella like other P2P systems significantly contributes to network traffic on the Internet [42]. The surging impact of P2P systems leads to an interesting question: what is the content request pattern in these systems?

S. Zhao, D. Stutzbach and R. Rejaie presented their work in [43] to investigate the characteristics of Gnutella including the query pattern. They developed a fast P2P crawler to capture around 40 snapshots of files available in Gnutella with more than 100 million distinct files in each snapshot. One of the results indicated by the snapshots is that file popularity mostly follows a Zipf distribution. The Zipf distribution of file popularity implies that a small number of files are extremely popular and these files thus have many copies across the network; while many other files only have a few copies available among participating peers.

S. Sen and J. Wang analyzed the characteristics of P2P traffic in [44]. They utilized a passive measurement approach and measured P2P systems at several levels of granularity: IP address, network prefix, and Autonomous Systems (AS). They extracted and analyzed 800 million flow-level records collected at multiple border routers across the ISP's network over a period of 3 months. Their study focused on three popular P2P systems – FastTrack [45], Gnutella [9], and DirectConnect [46]. They had 4 major observations, (i) All three systems exhibit significant increases in both traffic volume and number of users, even across consecutive months. The traffic volume generated by individual hosts is extremely variable – less than 10 of the IPs contribute around 99 of the total traffic volume. (ii) The P2P traffic distributions for traffic volume, connectivity, and average bandwidths usage are extremely skewed. (iii) All three P2P systems exhibit a high level of system dynamics – only a small fraction of hosts are persistent over long time periods. (iv) The fraction of P2P traffic contributed by each network prefix remains relatively unchanged and much more stable than the corresponding distribution of either Web traffic or overall traffic over the time period of one month. Even though their work [44] couldn't prove that P2P traffic *precisely* follows Zipf's distribution, it still indicated that popular content contribute to most of the traffic. In other words, the P2P content query pattern matches the one of the web traffic.

4.2 Comparison between structured and unstructured model

The preceding section discussed the finding that information requests on the Internet and popular P2P networks generally follow Zipf's law. It was indicated that the Internet and P2P networks are made up of heterogeneous and transient nodes. These observations also relate to the popular content sharing applications which are mainly based upon unstructured P2P networks.

DHT's popularity in research is in sharp contrasts to its slow adoption in real applications. The reason being that DHT based content-requesting scheme often leads to load balancing issues and the approach is not resilient to dynamic network conditions (e.g. nodes joining and leaving the network, arbitrarily). These two inherent weaknesses of the structured model, as duly noted by the proponents of the structured model [13], explain

why the unstructured P2P network model is still generally favoured among popular applications. This section will compare and contrast the main characteristics of the two models in order to reveal their strength and weaknesses.

4.2.1 Lookup efficiency

The structured model can lookup a piece of content in $O(\log N)$ steps. Section 2.2 argued that the power-law distribution or the Small-world model can be utilized to improve the performance of the unstructured P2P networks such as Gnutella and Freenet. The performance enhancement is achieved by using the random walk with a bias towards certain nodes. The pathlength in such cases can be expressed as $\log_k N$, where k is the network degree. Since lookup pathlength in the unstructured model can be radically reduced, the unstructured networks utilising the above approach can in theory be made to match the performance of the structured networks.

The Small-world like approach can not only significantly improve the performance of the unstructured P2P networks; it also can be utilized to improve the performance of the structured ones. K. Hui, J. Lui and D. Yau [30] proposed a Small-world protocol which is built on the top of Chord.

In their proposed network, there are two types of node, namely *head nodes* and *inner nodes*, and two types of link, namely *long links* and *cluster links*. Long links connect two different nodes from different clusters while cluster links connect two different nodes in the same cluster. Each cluster has *one* head node, which has at most long links and cluster links to all the nodes within its cluster. An inner node has a link to the head node within its cluster and cluster links to some of the nodes within its cluster. With the above configuration, an inner node, say i , can communicate with a target node, say j , within its cluster either by a cluster link (provided i and j are connected), or node i can send a message to its head node, and then the head node forwards the message to node j using a cluster link. For communicating with a target node in a different cluster, node i has to first send the message to its head node, and then the head node sends the message using the long link which is the closest to the target node j . The message may arrive at some

nodes which are not within the same cluster as node j . The procedure is then repeated until the message is transferred to some node within the same cluster as node j .

The object lookup process proceeds in two phases. In phase one, a node asks its cluster neighboring nodes if they contain the target object. If any of these cluster neighbors replies positively, then the lookup process is completed. Otherwise, the object lookup process continues and phase two begins. In phase two, the node first checks its own status within its cluster. If it is the head node, it forwards the lookup request to its long-link neighbor which is at the closest distance to the target object. On the other hand, if it is an inner node, it forwards the lookup request to its head node within the same cluster, and then the head node will recursively continue the object lookup process as described above. For example, when the long-link neighbor receives the object lookup request, it checks whether it is the owner of the object. If not, it will act as if it is the node initiating the object lookup, and repeat the data lookup process. The lookup process continues in these two phases until the data object is found.

Hui et al. in [30] conducted a number of experiments to compare the performance of their proposed small world network with that of other structured P2P networks such as Chord. The results showed that their approach can achieve a lower object lookup latency and can effectively satisfy a large number of users requesting a popular data object. Their approach however, didn't provide enough evidence to indicate the effectiveness and efficiency of unpopular content (e.g. a object with a single copy in the network) location.

4.2.2 Handling highly dynamic network environments

In a pure DHT-based network, the building block and the application level communication unit is an individual node. Each node has to maintain a routing table and also a list of numerically neighbouring nodes, e.g. a leaf set in Pastry. The entry or the exit of a node thus has the potential to influence the entire network; entailing uncertainty in results and higher data migration cost within dynamic networks.

The simplicity of the unstructured model avoids the overlay construction cost, thus the network maintenance overhead is reduced to its minimum level. The measured activity in

Gnutella and Napster indicates that the median up-time for a node is 60 minutes [42]. As analyzed by Y. Chawathe in [26], for large systems of, for instance, 100,000 nodes, this implies a churn rate of over 1600 nodes coming and going per minute. Churn causes little problem for Gnutella and other systems that employ unstructured overlay networks as long as a peer doesn't become disconnected by the loss of all of its neighbors, and even in that case the peer can merely repeat the bootstrap procedure to re-join the network. In contrast, churn does cause significant overhead for DHTs. In order to preserve the efficiency and correctness of routing, most DHTs require $O(\log n)$ repair operations after each failure. *Graceless* failures, where a node fails without informing its neighbors beforehand and transferring the relevant state, require more time and work in DHTs to (a) discover the failure and (b) replicate the lost data or pointers. If the churn rate is too high, the overhead caused by these repair operations can become substantial.

4.2.3 Load balancing

Content popularity or routing traffic impacts the query load. Both models have the load balancing problem but the causes are different.

In the structured model, owing to the query patterns of DHTs, the nodes with popular content and their numeric neighbours tend to be overloaded with query traffic. DHTs address this problem through its caching mechanism. For instance, PAST [22] caches popular content on the nodes along the query path to reduce the access latency. However cached copies may remain under utilised owing to the fact that other nodes may still follow their (separate) query paths to the content.

Key distribution imbalance is another factor that causes the load-balancing problem in the structured model [47]. DHT networks such as Pastry store a piece of content on a randomly chosen node with the `nodeId` numerically closer to the `contentId`. However, current distributed hash algorithms cannot guarantee the even mapping from `contentId` to `nodeId`; thus certain nodes may receive more content than the average. A. Rao et al. proposed *virtual servers* in [47] to tackle the load balancing issue caused by key distribution imbalance. A virtual server looks like a single peer to the underlying DHT, but each physical node can be responsible for more than one virtual server. For example,

in Chord, each virtual server is responsible for a contiguous region of the identifier space but a node can own non-contiguous portions of the ring by having multiple servers. The key advantage of splitting load into virtual servers is to move a virtual server from any node to any other node in the system. However, the use of virtual servers imposes significant overhead to processing power, memory, and especially bandwidth. Also, it's rather difficult to apply this mechanism in a dynamic network environment having a large number of transient nodes with limited computing resources.

In the unstructured model, nodes with high network degrees, i.e. with more global links, act as the shortcuts or bridges for most of queries. Therefore, they are likely to get overloaded. The experiment conducted by Saroiu et al. in [42] showed, that if sufficient number of shortcut nodes get overloaded or maliciously attacked, the entire network may be “shattered” into a number of discontinued fragments.

4.2.4 Suitable content search pattern

As discussed in the preceding sections, information request on popular P2P networks follows Zipf's law. It leads to the expectation that most queries in the popular P2P file-sharing systems are for relatively well-replicated files, which are called *hay* by Y. Chawathe et al. in [26]. Hay like content is in contrast to the files called *needle*, which have only one single copy existing in the network.

Some experiments in [26] gathered traces of queries and download requests using an instrumented Gnutella client. It is noted that most of the requests correspond to files that have a large number of available replicas. For example, half of the requests were for files with more than 100 replicas, and approximately 80% of that half of requests were for files with more than 80 replicas. Their work further indicated that Gnutella could easily find well-replicated files. Thus, if most searches are for hay, not needles, then Gnutella's lack of *exact recall* is not a significant disadvantage. DHTs have exact recall, allowing a file to be found, even if there is only a single copy of that file in the system. Therefore, DHTs are suitable for needle search.

The suitability of hay or needle search is indeed decided by the search nature of the structured and unstructured model. The mapping nature of search within a structured network makes no distinction between hay and needle contents. The unstructured model is more suitable for hay search since it does not mandate the connection between nodes and content.

Keyword based search is inherently difficult within a structured network due to its numeric ID mapping nature. Attempts to add an extra indexing to the DHT for keyword based search [48] [49], inevitably add more network overhead to the already non-trivial overlay maintenance cost. This is only further complicated by the additional caching algorithms to avoid overloading nodes that store the index for popular keywords. In contrast, *Keyword based search* is inherently easy within an unstructured network. Gnutella and other similar unstructured systems effortlessly support keyword searches and other complex queries since all such searches are executed locally on a node-by-node basis.

The comparison between the unstructured and the structured models can be summarized as follows. It is evident that both the structured and the unstructured models can be efficient under specific conditions. The unstructured model is generally superior to the structured one with respect to the network maintenance cost. Search within the structured model is globally oriented, thus more accurate than the unstructured one. Search within the unstructured model is locally oriented; it therefore can exploit the content knowledge within a local network. Hence, in terms of unpopular content search, the structured model is better than the unstructured one that cannot guarantee the search result. In terms of popular content search, however, the unstructured model is better than the structured one in search efficiency and load balancing.

4.3 Our approach and research motivation

Both the structured and the unstructured model have strengths and weaknesses. It's rather difficult to determine which of the two models would provide the overall better solution. In contrast to many researchers simply favoring one and against the other, we provide a consolidated approach of the two models, such that the unstructured model's network-organising flexibility can complement the structured one's preciseness and the structured model's globally oriented search can complement the unstructured model's locally oriented search. As a way forward, we would like to see if the structured model could absorb the strengths of the unstructured model without relinquishing on its advantages.

In the following chapter we will present a Clustered DHT (CDHT) communication scheme [16] [17], which combines the structured and the unstructured P2P network models in such a way that these two competing approaches complement each other. The goals of CDHT are,

1. To establish a more reliable network and content structure for the DHT overlay so that it can tolerate dynamic network conditions.
2. To devise a load balanced architecture.
3. To further enhance the superior content location capability of DHT based structured networks.

Heuristic Content Location (HCL) is proposed to exploit network locality and content popularity for adaptively balancing network traffic, avoiding traffic *hot spots*, and improving lookup performance for popular content location. HCL works alongside the DHT lookup pattern in the CDHT architecture

Project JXTA [15] assumes that consumer- and small devices- based P2P network is more likely to be highly unpredictable as peers will have a high churn rate. That assumption is closer to the random walker approach [49] of the unstructured model. In a highly fluctuating and unpredictable environment, the cost of maintaining a consistent distributed index is likely to outweigh the advantages of having one, as most of the time may be spent on updating indices (i.e. index trashing). Project JXTA proposes a hybrid

approach that uses a loosely-consistent DHT rendezvous walker that combines the use of DHT to locate content and with a limited range walker resolves the DHT inconsistency problem within the dynamic rendezvous network.

The design of CDHT/HCL has some similarity to Project JXTA since they both recognize the advantages and disadvantages of the two models and take a hybrid approach to achieve a better solution. It's anticipated that the combination of the two models would fundamentally resolve DHT's inherent weaknesses and make it more fit for real world applications.

Chapter 5

Clustered DHT (CDHT) Communication Scheme For Serverless (P2P) Networks

The dynamic aspects DHT P2P systems would need to be adequately addressed before the mobile devices could gain access to consistent and ubiquitous content. Also, a better solution than virtual nodes is needed to address hot spots issue such that DHT could sustain in a highly dynamic network with lots of transient/short-term mobile devices.

The Serverless (P2P) network architecture we envision should lead to a network which is not only as scalable as the pure DHT-based systems, but it is also more resilient to dynamic network conditions. The network should be able to achieve load balancing autonomously. To achieve these objectives a Serverless (P2P) clustered DHT routing scheme (CDHT) is presented in this chapter, which has a hybrid-overlay structure. CDHT delegates the network-aware Serverless Layer [19] [20] the task of forming the clusters of nodes on an ad hoc basis. It then entrusts the DHT overlay with the task of connecting these clusters to form a global network.

CDHT provides an architecture to make DHT suitable for dynamic network conditions. In terms of load balancing, HCL based on CDHT can eliminate both content and routing hot spots; it can thus be viewed as an alternative to the pure DHT based solutions. However, unlike the pure DHT based solutions, HCL has the ability to work in highly heterogeneous and dynamic network environments.

5.1 Serverless Layer

Server based models have been used to provide services such as e-mail, ftp, and web. However, with an unprecedented increase in the Internet user base, maintaining servers for hosting Internet applications/services has become an expensive exercise. A high server access rate introduces a bottleneck in the host's processing power as well as in its share of the network bandwidth. The solutions to this problem have included the use of high-end servers, more bandwidth, fault-tolerant mechanisms, and distributed load-balancing techniques. Further growth in demand has seen the development of regional hosting infrastructure services; however, implementation of these solutions comes at a significant cost and many enterprises are finding it difficult to keep up with the costs and effort required to re-engineer the application serving architecture from being server-centric to being essentially *serverless*.

A design framework for an Internet-wide *serverless* architecture is proposed by Khan in [19]. It provides an infrastructure upon which services could be effectively hosted without requiring any central server(s). The fundamental characteristics of such a system would be service discovery within the Internet, coordination, and organisation of the participating systems into manageable clusters, and efficient communications. An algorithm for a cluster formation is proposed in [20]; based on the serverless architecture. That algorithm is introduced in this chapter from the perspective of an individual entity or a node acting as the fundamental building block of a P2P network. Once these building blocks are in place, they may establish contact and form a *serverless* P2P network.

5.1.1 Serverless Applications

For traditional server-centric systems, an increase in the number of clients would degrade the performance and would necessitate measures to cope with the growth [19] [20]. On the other hand, as a serverless application derives the resources from its user base. Hence, an increase in the number of peers for such an application would enhance performance and replenish its resource base. Applications that are suitable to be implemented in the serverless mode would include: content distribution and mobile data

storage services, messaging and communication applications, and distributed lookup and naming services. These applications scale well in a distributed manner and can operate with minimal centralised control. However, applications that need a high degree of centralised control and consistency, such as a DBMS, may not be feasible for such a serverless system.

At the core of the *serverless* architecture is the concept that it is neither concentrated in a single location nor does it have a central controlling entity. Participating entities called “*peers*” may join or leave on the fly. A peer may take charge of a particular session, in order to coordinate and maintain synchronisation, and then go offline. Another participating peer will be able to take its place without a significant disruption or degradation to the overall service of the network.

Because of the ad hoc nature of this architecture, one of the basic challenges is the dynamic discovery of other peers hosting the instances of the application. In the current context of the Internet, instances would activate and deactivate arbitrarily. Further more the dynamic nature of host IP addresses [51] [52] and the use of Network Address Translation (NAT) techniques [53] makes it very difficult for a host to maintain a persistent connection with other peers.

5.1.2 Clustering Scheme

The clustering scheme presented by Senaratna and Khan in [20] is derived from the clustering techniques used in wireless mobile networks [54]. According to this scheme the entire population of the nodes in a wireless system is grouped into clusters. Each cluster is assigned a clusterhead; the nodes may communicate with the clusterhead and possibly with each other. Some nodes would be part of two or more clusters based on their ability to communicate with multiple clusterheads. Such nodes are named as the *gateways*. The *gateways* act to enable inter-cluster routing.

The clustering technique may use two alternative distributed clustering algorithms. The “lowest-ID algorithm” elects the node with the lowest identification as its clusterhead. A node that can hear more than one clusterhead becomes a *gateway* node and a node that is

neither a clusterhead nor a *gateway* is treated as an ordinary node. The “highest connectivity cluster algorithm” requires that each node broadcast its list of nodes that it can hear from; including itself. The node that is the most highly connected will be elected as the clusterhead. In the case of a tie, the lowest ID would prevail.

Wireless networks are broadcast based. The discovery of other nodes as well as clustering and electing clusterheads is done using broadcast. However the Internet in its current state is primarily unicast. Broadcasts can only be used within specific local area networks and do not work with point-to-point protocols, for e.g. PPP. Multicast networks exist, but these are not widely populated. Also it’s worth a mention that the initial multicast systems were devised for streaming data, such as audio and video within an enterprise [55]. Later, a wide-area multicasting architecture was proposed which required the use of *mrouters* for carrying multicast packets over unicast networks [56]. Many contemporary routers do not support such features.

In view of the above, the proposed implementation would have to be evolved within the confines of the unicast aspect of the Internet. In this regard it’s envisaged that a clustering approach would provide features of the broadcast based communications, within a unicast environment, without inordinately affecting the bandwidth resources. A *serverless* P2P cluster will split into multiple clusters once the size of the cluster has grown beyond a pre-defined limit. Each cluster will in-turn elect a router node [19] [20] called a *leader node*. A leader node would co-ordinate communication within the cluster as well as providing the message routing service between the clusters.

After splitting and having elected the leaders, the clusters act as independent entities and all communications between these are routed through their respective leaders. New nodes may join these clusters but no change to the leader node would be made unless the current leader node goes offline. The cluster may continue to grow until its size limit is reached, it will then split recursively; creating a community/neighbourhood of clusters.

Since a node in a neighbourhood may appear and disappear in an unpredictable manner, the persistence of a leader node is guaranteed by always having a list of nodes actively contending for the leadership role.

Election of the Leader Node

A cluster once formed will elect a leader node. Lowest id or highest connectivity criteria used in wireless networks [54] presents effective ways of achieving this. However, it is possible for a node satisfying the above criteria to have limited resources; hampering its ability to act as a leader node.

In the serverless approach, each node will calculate a rating value. The rating would be communicated to all other nodes at the time of discovery and the neighbourhood node updates. The node with the highest rating within the cluster is elected as the leader. In the event of a tie, the lowest id rule applies. The node having the second highest rating will act in a leadership contention role and thus will assume the role of the leader if the current leader goes offline.

The rating for a node is calculated on the basis of its suitability for leadership. The characteristics that are assessed include processor speed, memory, and network connection speed (the available bandwidth). This prevents the nodes with lower resources or with congested network connections from being elected. The rating may include an adjustment factor for the time served as the leader in respect to the total uptime for that node. This ensures fairness in electing the leader. For this scheme to be implemented, the historical data relating to the node's uptime must be kept.

Communication within a Cluster

A cluster is logically viewed as a ring-like structure, arranged in an ascending order based on the node ids. However, a node is allowed to directly communicate with another node within the same cluster or another cluster, provided that it has prior knowledge about the remote node.

A node in a cluster records its successor and predecessor nodes. Cluster-wide communications occur along the ring. A node receiving a cluster-wide update will forward it to its successor node. Update messages have a Time To Live (TTL) value set. The initial value of the TTL is one less than the size of the cluster. A node will decrement the TTL by one at each hop. When a node receives an update with a TTL of zero; update will not be forwarded to the next node.

Each cluster-wide communication has a sequence number and a node will send an acknowledgement to its predecessor node after receiving such a message. If a node does not receive an acknowledgement it will initiate a discovery procedure to inquire whether the node is available. The message will be re-transmitted if it is available, otherwise a dropout procedure will be initiated.

The leader node will handle joins and dropouts within the cluster. A new node wanting to join the cluster is directed to the leader. The leader adds it to the cluster map and initiates a cluster-wide map update. The new node is inserted into the ring according to the node's ID.

Dropouts are handled in a similar manner. A node periodically sends ping messages to its successor and predecessor nodes. If it doesn't receive a reply after 3 repeated ping messages, it initiates a dropout sequence, during which, it will notify the cluster-leader. The cluster leader will initiate a new map update if it finds that the node is no longer available.

Forming and Managing Clusters

A peer starting up will go through several stages before it may actively participate in a cluster. Figure 5- 1 depicts the state transition diagram for a typical peer node in a *serverless* network.

A starting-up peer will enter the *Online* state. In this state it has no knowledge of the other instances and it is in a discovery state. It enters the *Socialised* state once it discovers other peers. Once it knows of other active peer(s), the peer will solicit joining the cluster and enter the *Clustered* state i.e. after being accepted into the cluster. It may acquire the *Leader* state if the cluster elects the peer as the leader node for the cluster.

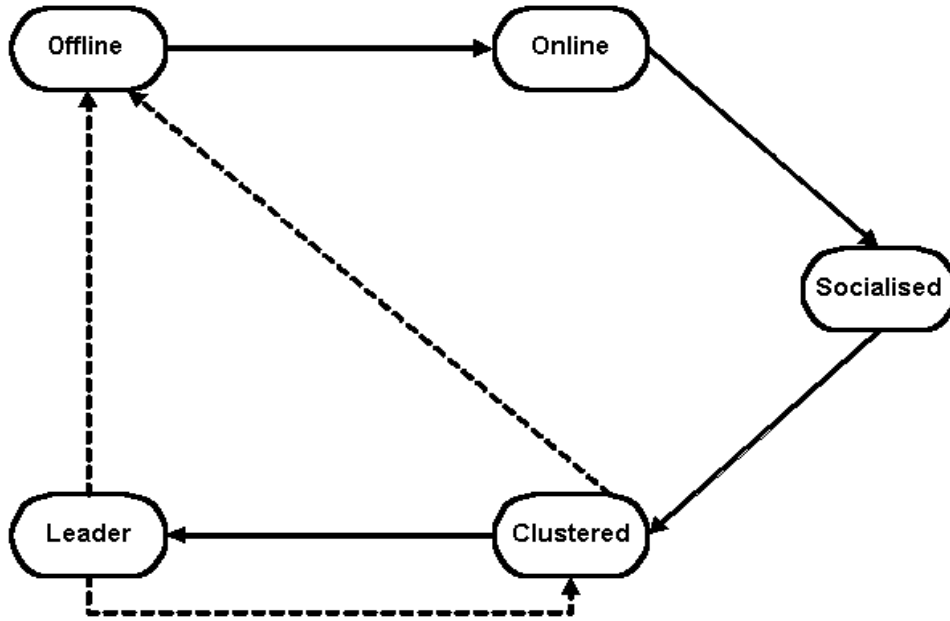


Figure 5- 1 State Transition Diagram for a peer node[20]

Algorithms Used in Peer Coordination and Management

At start-up, a peer will search for and interact with other peer instances in order to join the *serverless* cluster. The initial algorithm for such a peer is shown below in Figure 5- 2. The peer, discovered after the start-up, could be in any one of the states described in Figure 5- 1. The discovered peer will reply in a manner, that commensurate with the state it currently is in, to assist the newly initialised peer with joining a cluster.

```

Set State to ONLINE
Search for other instances
If a peer is discovered
  Set state to SOCIALISED
  Case state (state of the discovered peer)
  ONLINE/SOCIALISED: Form a cluster and elect leader based on
                      the leader selection criteria
                      If elected as leader
                        Set state to LEADER
                      Else
                        Set state to CLUSTERED
                      End if
  CLUSTERED: Request for leader node
              Request for cluster membership
              If accepted
                Set state to CLUSTERED
              End if
  LEADER: Request cluster membership
           If accepted
             Set the state to CLUSTERED
           End if
End if

```

Figure 5- 2 Initial Clustering Algorithm [20]

The start-up peer may find a peer that is either in the ONLINE or SOCIALISED states. In this scenario both the peers shall initiate the sequence to form a cluster. The 3-way sequence is depicted in Figure 5- 3 (a). Peer A sends a *discover query (D_QUERY)* to Peer B. Since Peer B is not a member of a cluster at this stage, it indicates its willingness to form a cluster, by sending a *discover reply (D_REPLY)*. Once A receives the reply it may accept by sending a *discover accept (D_ACCEPT)* message and form a new cluster, with Peer B. A and B will now enter the *Clustered* state. Based on the leader election criterion specified above, the newly formed cluster will elect a leader, and the leader node will enter the *Leader* state.

If B was already a member of a cluster, it will redirect A to the cluster leader upon receiving the query from A. The scenario is depicted in Figure 5- 3 (b).

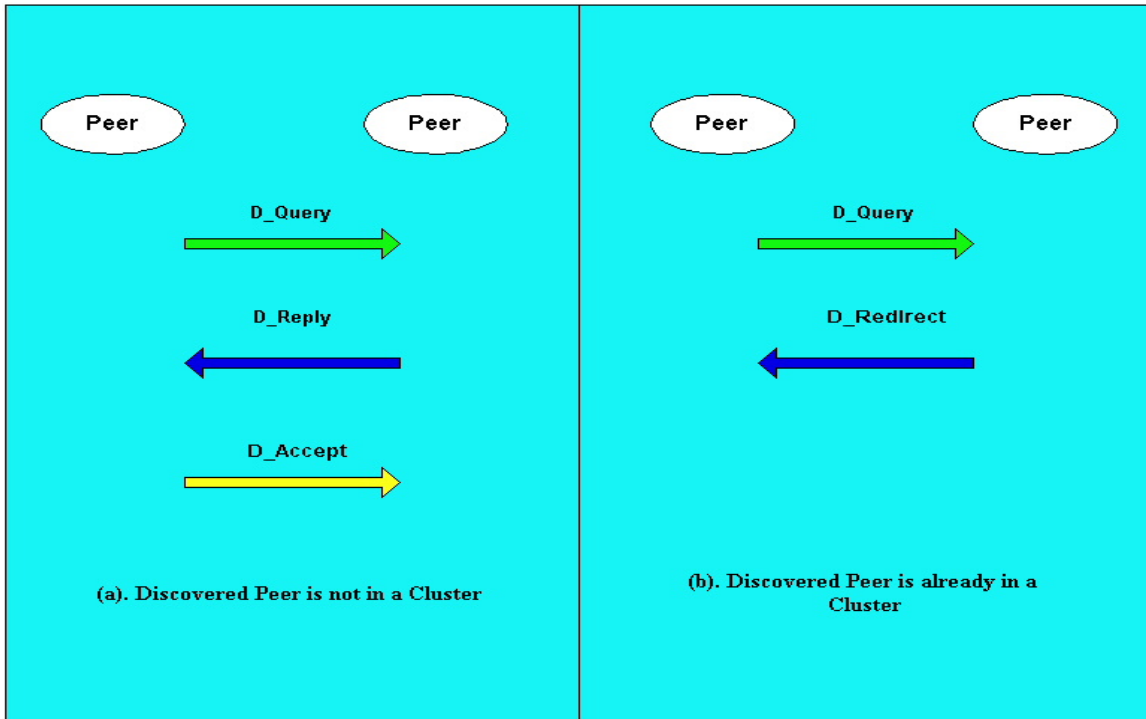


Figure 5- 3 Exchange of messages between newly discovered [20]

Once clusters are formed, any new peer going online may join any of these clusters in the order of its discovering the other active peers. The leader node will handle the joins and leaves and will act to synchronise the cluster after any such change. A peer wishing to join a cluster must request the cluster leader. The leader may then decide to either accept the request or to re-direct the peer to another leader. The decision will depend on the current size of the cluster and the configuration options.

Once a cluster reaches a maximum size, it needs to split itself. The splits are initiated by the leader node. The leader node will send a list of neighbouring leader nodes to all its cluster nodes and then initiate the cluster split sequence. A normal cluster node, in the above instance will first receive a list of neighbouring leader nodes followed by a split initiation message.

The leader node list is distributed to enable a newly elected leader to maintain contacts with other clusters. Each peer in the cluster will split its cluster map and re-create the cluster using the appropriate sub-map - the one the peer belong to now. If it is elected as the leader of the newborn cluster; it will activate the leader-node list and inform the

neighbouring leader nodes about the change. Otherwise it will discard the leader list and continue as a cluster node.

Serverless peer-to-peer networks show considerable potential towards harnessing the untapped resources within millions of computers connected to the Internet. The ultimate goal would be to have intelligent and autonomous entities, meeting, forming alliances, and working together towards a common objective without much need for the human intervention.

The prototype in [20] shows that the initial cluster formation and organisation could well work within the distributed framework of autonomous instances of the application. Coordinating without the intervention of a central entity, the instances were able to form the clusters, elect a leader, and communicate within the neighbourhood. The election of the leader took place unambiguously, whereby each entity in the neighbourhood knew exactly who the leader node was in its case at all times.

In the preliminary implementation, the framework was kept as simple as possible and the communications was kept to a minimum. However, owing to the ad-hoc nature of the architecture, extra communications will be necessary to support various application serving requirements.

CDHT is built on top of the Serverless Layer. The Serverless Layer can sustain highly dynamic configurational changes within a cluster; hence, it deals with network uncertainty in CDHT.

It may be noted that under a conventional DHT scheme routing would not remain efficient in a network where the nodeIds keep changing – leading to rapid outdateding of the nodal routing tables. CDHT addresses this issue by implementing the following features in its architecture:

1. Network stability – CDHT tackles network uncertainty by localising the influence of changes to the network configuration through the Serverless layer. Hence the resultant DHT is produced by a more stable and well-connected overlay - i.e. the clusters;

2. Content stability – The content storage and management unit is a cluster instead of an individual node. For instance, when a piece of content is saved into a cluster, the cluster may replicate the copies on different nodes within itself. Doing so would ensure that the content remains highly available at all times.
3. Consistent connection between clusters and content – This aspect is derived from the content stability condition. The consistent connection may be used to achieve load balancing and to further enhance the DHT lookup performance for popular content.

M. Ripeanu et al [57] have shown that a mismatch between P2P overlay topology and Internet infrastructure has the potential to critically impact the performance of the network. CDHT alleviates the risk of the mismatch since the Serverless Layer maintains the current status of the underlying physical network.

5.2 Heuristic Content Location Scheme

Heuristic Content Location (HCL) is proposed to exploit network locality and content popularity for balancing network traffic autonomously, while avoiding *hot spots* and improving on lookup performance for popular content location. HCL works alongside the DHT lookup pattern in the CDHT architecture. Closely related work to HCL is described by K. Sripanidkulchai et al in [58]. It makes use of *interest-based shortcuts* to enhance the performance in Gnutella-like networks [8] - message flooding is used for content location.

The consistent connection between content and its storage unit is the premise of the HCL design. We developed a prototype for the HCL simulation with the assumption that the network is stable. A random network graph is generated with 1,024 nodes and each node is assigned a random hash ID. Initially, we assumed that each node only had the knowledge of its immediate neighbours.

At each time, we randomly selected a different node as the source and another as the destination. The source node checks its routing table first. If there is no route to the desired node available, it then broadcasts a request to its neighbours. This process repeats itself through the network until the request reaches the destination. Upon receiving the first request, the destination node sends back a reply to the source node following the reverse of the first successful path of the request. The reply includes its ID and address. All the nodes along the way add this information into their own routing tables. The size of each routing table is limited to 10. Least-Recently-Used is the replacement policy; repeated entries are omitted. A simplified scenario is illustrated as Figure 5- 4.

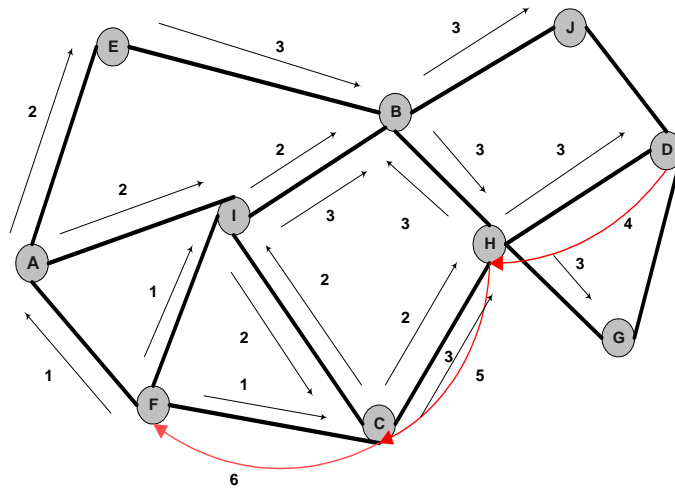


Figure 5- 4 The Process of HCL

In this figure, suppose Node F wants to find Node D. It checks its routing table and finds that there is no route listed for Node D. It then broadcasts a request to its neighbours. After three steps, the request first reaches Node D through Node C and then Node H. Node D returns a reply with its ID and address by tracing the first successful path. Node H, C, and F receive the reply and add the route into their own routing tables. Afterwards if Node A wants to find Node D as well, we can see the routing steps are only two in this case and the broadcast is necessary only once - Node A benefits from the knowledge of its neighbour Node F. The numbers in Figure

5- 4 represent the broadcast levels and some nodes such as Node I may receive the request from the source nodes at different broadcast levels.

In our first experiment, we simply repeated the above process. The result is shown in Figure 5- 5. The results indicate that the average number of hops to reach the destination decreases exponentially with respect to the number of the trials.

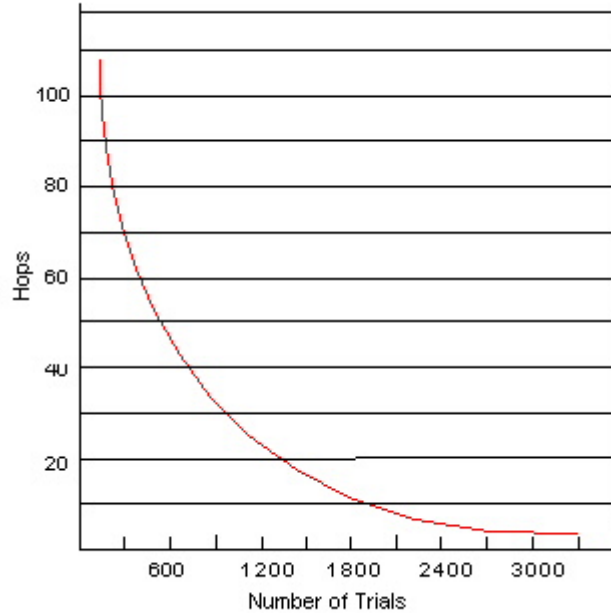


Figure 5- 5 Hops variation with the time increase

In our second experiment, we restored the initial state on each node and randomly inserted 1,024 pieces of data into the network. We looked up arbitrary data from a randomly selected node each time. Once a piece of data is found, a data pointer is added into the nodes along the reversed first successful path. The data pointer relates the content to its storing node. Similarly, broadcast is stopped if a data pointer is found during the request forwarding. The results were very close and similar to the first experiment.

From the experiments we conclude that knowledge of neighbouring nodes can significantly reduce the hops and the content location time. The more active a network and the more popular the content, the heuristic mechanism works better. Additionally, finding content from the neighbouring nodes makes the routing path match more closely

to the underlying physical network topology. As long as a network is randomly formed and the content is randomly inserted and looked up, results similar to the above experiments may be expected.

A large number of simulations and experiments will be required to conclusively prove the superior abilities of CHDT and HCL for handling dynamic network environments and achieving load balancing autonomously. In this regard it is decided to separate the experimentation into several stages. The first stage is to verify the trade-off between the HCL lookup and the pure DHT lookup. The verification is presented as follows.

In our experiment, we assume that the network is stable and compare the HCL lookup with the Pastry lookup. It's important to point out that HCL is able to work with other DHTs too. Since Pastry and other DHTs do not recognise content popularity, we assume that all the content in the experiment has the same popularity in order to maintain consistency. FreePastry [58] is used as the simulator, but we add HCL-related facilities into it such as the *Content Pointer Cache*. FreePastry assigns a hash nodeId to each node. We simply assume that each nodeId is a piece of content stored on the local node. Thus the content is dispersed across the network randomly. Similar to the preceding experiments, we generate a random network graph with 1,024 nodes.

Pastry is characterized by the routing capability of reaching the destination in less than $\lceil \log_{2^b} N \rceil$ steps. N equals to 1,024 here; we give the parameter b the value 2. In this case, the size of the neighbourhood set m is $2 * 2^b = 8$; the size of the leaf set l is $2^b = 4$; and the size of the routing table r is $\log_{2^b} N * (2^b - 1) = 15$, according to [5]. We specify the size of the Content Pointer Cache in any node to be the same as the size of the routing table. At the beginning, the Content Pointer Cache has no entry.

For each test we randomly select a pair of nodes as the source and the destination. Pastry lookup is started first. Upon a successful lookup, a content pointer is added into the Content Pointer Caches on the nodes along the reverse of the lookup path. Then a HCL lookup is started on another randomly chosen pair of nodes. At each step, HCL randomly selects no more than two nodes from the neighbourhood set of Pastry as the target nodes, and then broadcasts the request to them until the final destination node is

reached. In this experiment, we only count the pathlength of the HCL lookup process and no content pointer is recorded into the Content Pointer Caches on the nodes along the reverse of the lookup path. Doing so ensures that the knowledge in the Content Pointer Caches comes from the Pastry lookup. We repeated the above process 6,000 times and calculated the average pathlength for Pastry and HCL respectively for after every 500 trials. The result is illustrated in Figure 5- 6.

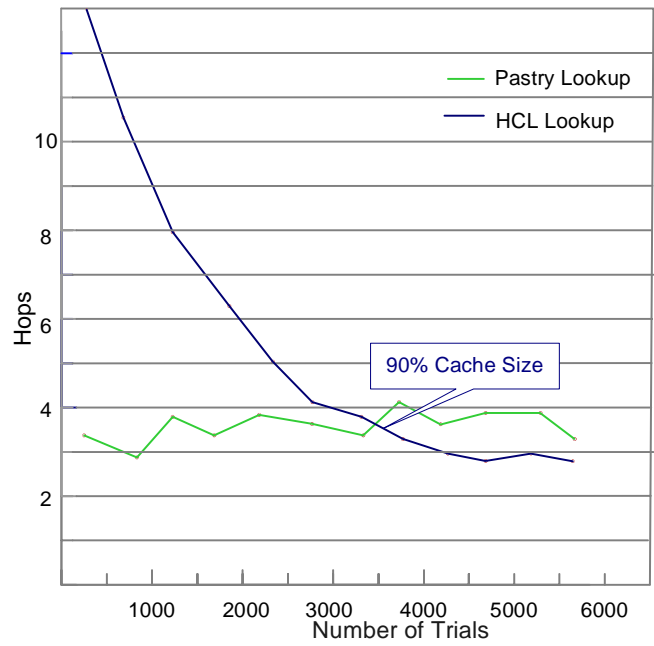


Figure 5- 6 The comparison between Pastry and HCL lookup

The average pathlength of the Pastry lookup fluctuates between 3 and 4. We can see that Pastry works without being affected by network activeness. HCL contrasts with it. The pathlength of the HCL lookup decreases exponentially with respect to the number of the trials. In the end, the average pathlength of the HCL lookup is less than the result of the Pastry lookup. Benefiting from the accumulation of the content knowledge and the network activeness, the value of the average pathlength of the HCL lookup is eventually close to the value of $\log_r N$ where r is the size of the Content Pointer Cache.

When the nodes on a network are outnumbered by the content, the entries in the Content Pointer Cache represent the popular content pointers. It indicates that the popular content can be looked up more efficiently compared to the less popular ones with HCL.

The initial stage of the HCL lookup process is a period of “dark” time during which the lookup performance is low. In the implementation, to circumvent that period, we could keep HCL inactive until the network activeness reaches a certain degree e.g. the number of the cache entries is not less than 90% of the cache size. In addition, we could use the value of $\log_r N$ or $(\log_r N - 1)$ as the TTL value of a HCL lookup operation in the implementations. If a HCL lookup expires without getting a response, the DHT lookup is then called.

The result of the simulation demonstrates that HCL can achieve better lookup efficiency for popular content within an Autonomous Region. If the content can't be located with HCL in the local Autonomous Region, the DHT lookup is then initiated to the extent of the entire network. HCL can thus be viewed as a complementary and efficient lookup pattern working alongside the DHT lookup patterns. The simulation also implies the effectiveness of the HCL lookup depends on a relatively stable network and content structure, which is contributed by the Serverless Layer tackling the network uncertainty in the CDHT architecture.

5.3 CDHT Architecture

In CDHT, each cluster in the Serverless Layer selects a single node as the *Content Rendezvous* (CR) node according to selection criteria such as processing power, bandwidth, storage capacity, etc. Each CR has a *Content Repository* to record the hash IDs of the stored content in that cluster and the addresses of the storing nodes. It's assumed that the stored content has at least two copies - one each on a different node in the local cluster. The number of copies may vary under different implementations. Each CR also has a *Content Pointer Cache* to record the known contentIds and the address of their storing CR nodes for the HCL function. Content information passing through the local CR node is recorded in the Cache following the Least-Recently-Used replacement policy.

CRs in the network form a DHT overlay. Other (ordinary) nodes don't participate in the DHT activities. In other words, any DHT routing table only has the entries of the CR nodes. The hybrid overlay structure is illustrated as Figure 5- 7. The clouds represent the Serverless Layer clusters, on which the DHT overlay is constructed.

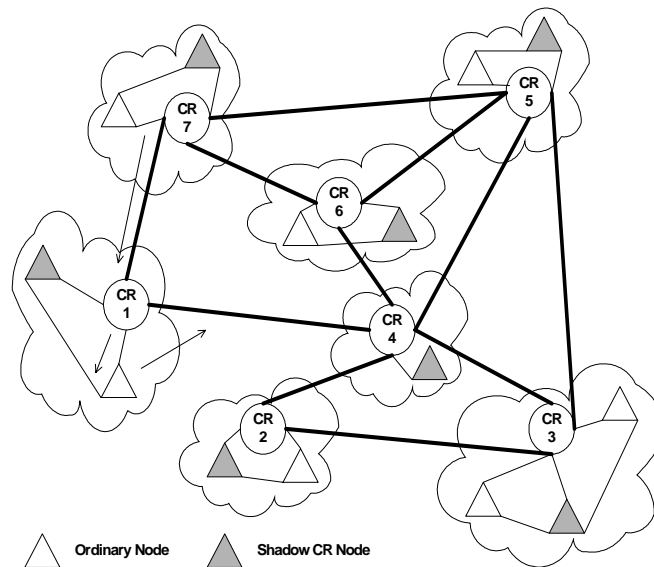


Figure 5- 7 The hybrid overlay structure of a CDHT network

Every ordinary node within a Serverless Layer cluster keeps some information the same with its CR node, e.g. the first row of the DHT routing table (not necessarily the

entire table, because an ordinary node is only involved in the first step of the DHT lookup process). The CR node also keeps a set of neighbouring CR nodes. The CR node periodically synchronises this information with other nodes within the same cluster. At the start of the content lookup operation, a HCL query is initiated. The query is broadcast to the CR node of the local cluster and also to a certain number of neighbouring CR nodes. If the HCL query doesn't result in a response in a specified period of time, the DHT lookup function is then called upon. A time to live (TTL) value is specified by the originating node.

Any request from a cluster to the DHT overlay labels the CR node of that cluster as the source node. The response is returned to the CR node and the CR node forwards the response to the originating node. The path of the request and response is summarily illustrated in Figure 5- 7. A CR node acts as the incoming gateway and the content location activity coordinator within its local cluster. Network resilience is vastly improved owing to the ordinary nodes being shielded from DHT overlay operations. These operations only occur within the CR nodes.

Each cluster has a *Shadow Content Rendezvous* node. The Shadow CR node keeps the same routing and content knowledge as the CR node. If the CR node goes offline, the Shadow CR assumes the role and a new Shadow CR is selected (in other words any node can assume the role of the CR). The leader node within a Serverless Layer cluster acting also as the CR could simplify the structure of the cluster, but doing so would make the leader node a hot spot. When the load on a CR exceeds a certain amount, the Shadow CR joins the DHT overlay with its own nodeId and shares the load burden with the CR node. A Serverless Layer cluster maintains an optimum number of nodes required for minimising inter-cluster and intra-cluster communication overheads [19]. This prevents the CR from becoming a server-like super node or a hot spot.

Ordinary Node Joining and Leaving

Dynamic node change in a cluster does not impact other clusters unless it fails to meet the minimum requirement of the cluster formation [19].

A new node joining the cluster would convey its characteristic, e.g. available storage space and processing power, to the CR node. The CR node may later assign certain

content to it for storage. The CR node would record the contentId and the address of the node into its Content Repository as its *save* operation. The node is then ready to receive and respond to the content queries from the CR node. The content stored on a node may become unavailable when the node goes off-line. In this case, acting on the information in the Content Repository, the CR node simply recovers the lost content from other storing nodes and copies it to an available node. It then updates its Content Repository accordingly; or waits for a new node to join for the content replication process to be completed. The impact of node joining or leaving is thus limited to the (Serverless Layer managed) local cluster.

Content Rendezvous Node Change

In the event of a CR node going offline, its Shadow CR node takes over the role and becomes the new CR. It keeps the same information as the old CR node, including the nodeId of the old CR node. The nodeId now needs to point to a different IP address and the new CR node has to inform the DHT overlay of the address change. It does so by sending messages to all its known nodes, e.g. the nodes in the routing table, the leaf set, and the neighbourhood set as defined in the Pastry overlay. Besides the known nodes, there may also be some CR nodes, unknown to the new CR node, relying on the old nodeId. Their records are passively updated as follows. Upon connection failure, these CR nodes simply remove the old nodeId and replace it with a new one. Since the new CR remains in the same network position as the old CR, it is very likely to be re-selected [60]. Thus content stability, with respect to the content search paths, is generally maintained under this mechanism.

Load Balancing of Content Rendezvous Nodes

A CR node must participate in the DHT overlay and coordinate the lookup activities of the nodes from its local cluster. It is important to prevent the CR from becoming a hot spot. The Serverless Layer maintains an optimal number of nodes within the cluster. A load control algorithm is used to balance the load between the CR and the Shadow CR node.

The capacity of a node is assumed as the number of queries it can handle per time unit. The capacity ratio of a node is the percentage of the even value of query numbers within

a certain period versus the capacity. The CR node and its Shadow CR node in a cluster periodically broadcast their capacity ratios to the nodes in the cluster. The capacity ratio may be quantified as

$$r = \sum_{i=1}^n q_i / (c*n)$$

where r is the capacity ratio, q_i is the query number in the i^{th} time unit, c is the capacity, n is the total number of time units that a certain period has.

It's known that the DHT query pattern tends to result in a single response. As introduced in the preceding discussion, a HCL query employs multicast. It thus may result in multiple responses. The load control algorithm is to share the query load of HCL and DHT on a CR and its Shadow CR node. An ordinary node can differentiate the capacity status on the CR and the Shadow CR node based on the received ratio messages. An ordinary node can understand the following four capacity statuses:

- a. The CR node can handle HCL queries;
- b. The CR node can handle DHT queries and the Shadow CR node is used to share the query load on the CR node;
- c. The CR node is overloaded and the Shadow CR node is used as the replacement;
- d. The Shadow CR node is overloaded too.

The load severity sequence of the four statuses is a<b<c<d. The ordinary nodes react accordingly. The algorithm is implemented on an ordinary node as follows.

```

For any given capacity status x
If x <= a:
    Initiate both HCL and DHT query in the name of the CR node
If a < x and x <= b:
    Initiate DHT query in the name of the CR node but initiate HCL query in the name of a
    randomly selected node, which could be the CR or the Shadow CR node. If the capacity
    ratio of the CR node is 70%, the possibility of choosing the CR node is 1 - 70% = 30%.
If b < x <= c and c < x <= d:
    Initiate all queries in the name of the Shadow CR node
If x > d:
    1) Initiate DHT query in the name of a randomly selected node, which could be the CR or
    the Shadow CR node, with an equal probability.

```

2) Do not initiate the HCL query; instead, initiate the neighbouring queries within the local cluster. The neighbouring query refers to the network-degree based query in the unstructured P2P model.

The neighbouring queries would not be sent to the CR and the Shadow CR node even if they were included in the neighbouring node list/routing table of the local node.

If $x > d$ and x lasts for a long period or appears frequently:

Request the Serverless Layer to initiate the cluster sub-division

5.4 Experimental Results

CDHT with HCL is superior to the pure structured model in the three aspects: *a.* improves the lookup performance for popular content; *b.* improves the network resilience to handle highly dynamic network environments; *c.* improves the load balancing. Data in support of *a* was provided in Section 5.2. In this section, we provide the experimental results in support of *b* and *c*.

DHT and CDHT model were implemented using the Java implementation of NS-2 – JNS 1.7 [61]. NS-2 [62] stands for Network Simulator version 2. It is a discrete event simulator targeted as network research. NS-2 provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless networks.

JNS is a project originating from UCL. The latest version available from sourceforge is version 1.7. This version adds multicasting, multithreading and dynamic scheduling. In addition JNS now supports simulation of "real" Java network applications, as there is a `fake.net.MulticastSocket` and a `DynamicScheduler` which will transform any network calls made from a Java program into schedulable events on JNS. This allows programmers to test real-world implementations of protocols rather than just scripts. JNS allows developers of networking protocols to simulate their protocols in a controlled environment. The simulator then produces a trace file (same format as NAM [64] trace files) which can be viewed in a network animator such as Javis [63].

A. The comparison of lookup success rate between DHT and CDHT within highly dynamic networks

Two network models were built as DHT and CDHT model respectively with JNS 1.7. Each network was assigned 16,384 ($=4^7$) nodes for these simulations. It was assumed

that during a specific period certain numbers of nodes would leave the network without notification. These nodes were selected randomly. It was also assumed that the length of the period was short enough such that the routing tables in the DHT nodes could not be recovered promptly. CR recovery is a process in CDHT network where the Shadow CR node takes over the responsibility of the CR node automatically in the event of CR node leaving the network. CR recovery is very likely to be successful since it's a local activity whereas the DHT routing table recovery is a global activity in terms of the involved network extent.

The simulations were performed with 1600, 2400 and 4800 randomly selected nodes leaving the network in the DHT model. Each simulation comprised 100 lookups (where each of the lookup could be either a success or a failure). The successful lookup rate was calculated with the number of the successful lookup divided by the total lookups. The same procedure was repeated for the CDHT model. For the sake of comparison with DHT, it was assumed that the successful CR recovery rate was 90% in the CDHT model. The comparison of the two models' successful lookup rate is illustrated in Figure 5- 8. It can be seen that CDHT provides better lookup rates for these dynamic network simulations.

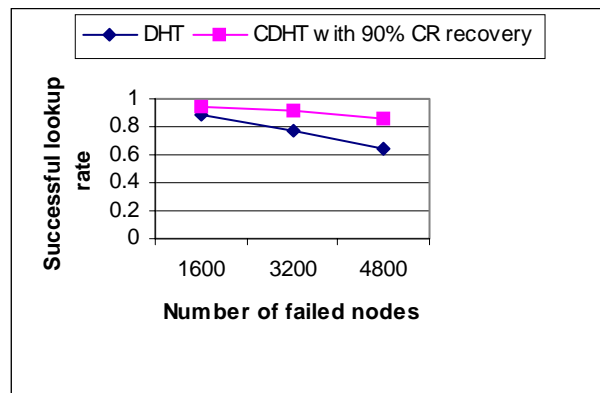


Figure 5- 8 A comparison of lookup success rate between DHT and CDHT

In order to fully verify the characteristics of the CDHT model, it's necessary to assess the impact of the CR recovery on CDHT lookup rate. Simulations were performed with 80%, 50% and 0% CR recovery rate respectively when 1600 random nodes left the

network. Each simulation comprised 100 lookups. These simulations were repeated with 3200 and 4800 node leaving the network.

The comparison of HCL lookup success rate with various CR recovery rates is illustrated in Figure 5- 9. The model with 80% CR recovery rate has a much better lookup rate than the other two. It can be concluded that CR recovery strategy is crucial to the effectiveness of CDHT.

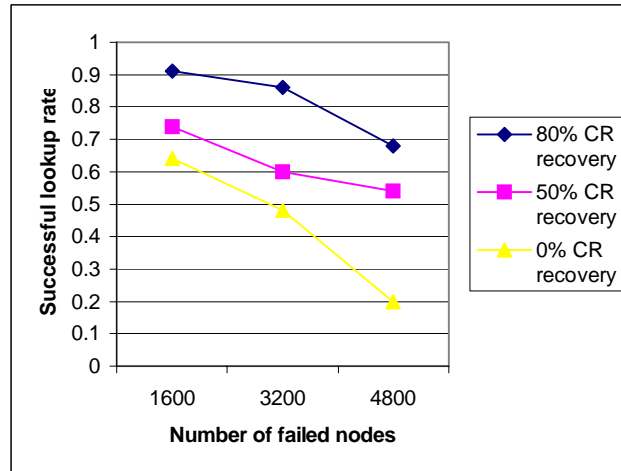


Figure 5- 9 A comparison of CDHT/HCL lookup success rate with various CR recovery rates

A part of CDHT network trace made visible by Jarvis 2.0 is illustrated in Figure 5- 10. JNS 1.7 produced the Jarvis trace through the above simulation.

Javis is a packet flow and network animator. It is a Java based version of NAM [64]. NAM is a Network Animator, widely used with NS-2 simulator. NAM accepts ns traces files, and presents the animation of the simulation based on them. The results from various network simulators can be captured and fed into Javis after a run has completed. Javis can show a network topology, nodes locations, links between nodes, demonstrate the flow of messages send upon links, topology evolution over time, etc.

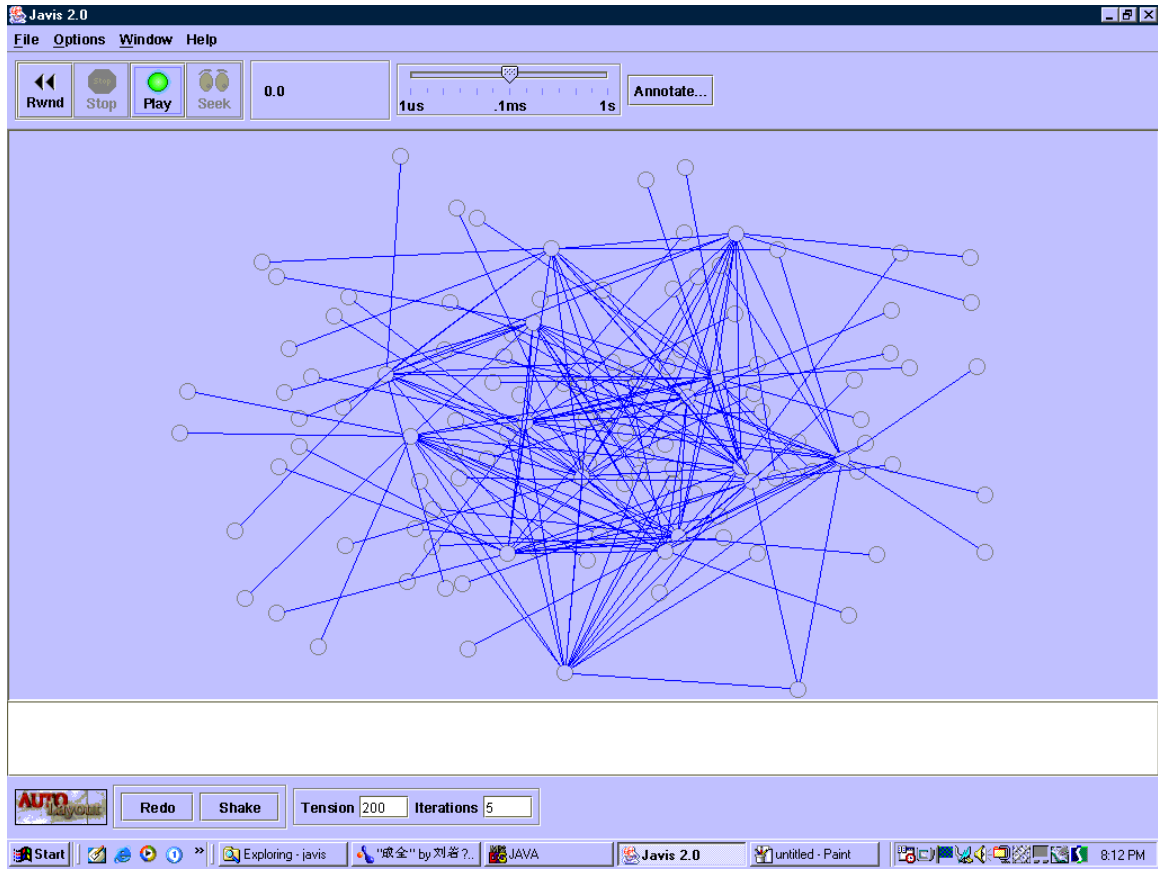


Figure 5- 10 A part of CDHT network trace generated by Jarvis

B. The comparison of query load caused by content popularity

DHT and CDHT network models with 1,024 nodes were established respectively using JNS-1.7. For the sake of simplicity, a nodeId was assumed to be the content stored on the host node. It was also assumed that the nodeId equivalent to value 1,024 was one of popular content. Hence the 1,024th node was forced to become the hot spot.

The popular content was set to 5% of the number of total content. According to the nature of the DHT lookup, the nodes with nodeIds numerically close to the 1,024th node tend to become hot spots as well. In the CDHT network model, HCL lookup scatters the query load into a number of Autonomous Regions such that the load concentrating on the 1,024th node and its numerical neighbours is considerably reduced. It is worth pointing out that the percentage of the number of Popular Content Request (PCR) over the number of the overall content request is an important factor influencing the HCL lookup results.

One hundred trials were performed for the DHT lookup and the HCL lookup respectively with 10%, 30% and 50% PCR. The varying PCR values impacted the *content-appeared* time values, in the Content Pointer Cache, on each node. In each trial a node was selected randomly to look up the popular content i.e. nodeId 1,024. After these 100 trials, the numbers of lookup request passing through each node were accumulated using the nodeId field.

The results depicted in Figure 5- 11 demonstrate that the HCL lookup is able to significantly balance the query load caused by content popularity on the hot spot and its numerical neighbours. The simulation also demonstrates that the load balancing of HCL lookups increases with higher values of PCR.

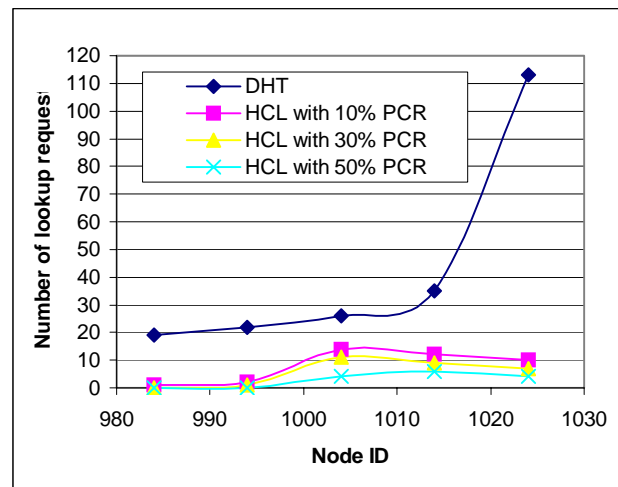


Figure 5- 11 A comparison of query load caused by content popularity

C. The comparison of query load caused by key distribution imbalance

DHT networks such as Pastry store a piece of content on a chosen node with the nodeId numerically closer to the contentId. However, current distributed hash algorithms cannot guarantee the even mapping from contentId to nodeId [47]; thus certain nodes may receive more content than the average. These nodes would sustain more routing traffic than other nodes. A. Rao et al proposed *virtual servers* to tackle the load balancing issue caused by key distribution imbalance. However, the use of virtual servers imposes significant overhead to processing power, memory, and especially bandwidth. Also it's

rather difficult to apply this mechanism in a dynamic network environment having a large number of transient nodes with limited computing resources.

In this experiment, the DHT and CDHT network models with 1,024 nodes were established. The nodeId equivalent to value 1,024 was again chosen as the one of most popular content. Ten nodeIds numerically closer to this nodeId were selected at random but were assigned the same IP address. It assured that the routing traffic passing these ten nodeIds would pass through the same node. One hundred trials were performed for DHT and HCL respectively using various PCR values. In each trial, a random node was selected to look up the 1,024th node. The number of lookup request passing the routing hot spot were added up.

Figure 5- 12 depicts the results of the comparison for different lookup types. It demonstrates that the HCL lookup can considerably reduce the routing overload, compared to the DHT lookup. Again, higher PCR value resulted in better load balancing performance.

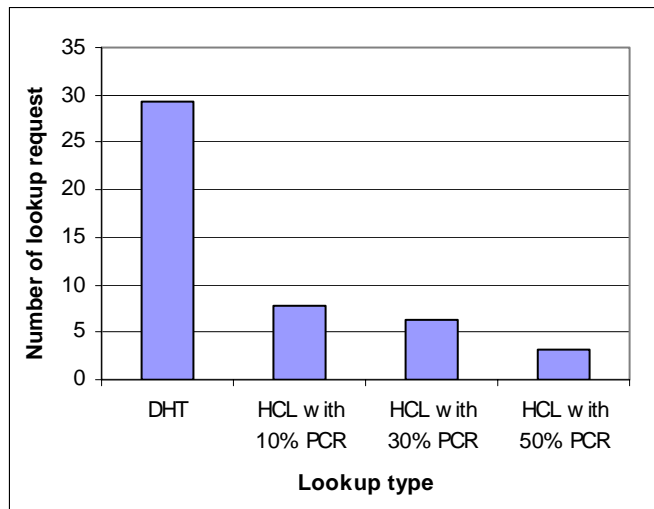


Figure 5- 12 A comparison of query load caused by routing traffic

The experiments were performed by building related facilities based on JNS. The example code is showed in Appendix.

Chapter 6

Serverless P2P Network Storage

The client-server model makes the server as the hub of network activity. A failure at the server may impact a large user base. In relation to the storage area, a question has been often posed by the Internet users: can their stored file be retrieved from any laptop or PDA? Meanwhile, enterprise user who needs to get files from the network would like to access their files even when the file server is down. These questions may be answered by taking a different perspective to the existing client-server model where the network services are viewed as server centric (rather than network centric).

The network has become an integral part of the computing environment; the case for having a distributed network storage has also become stronger. With redundant copies stored anywhere across network, files may be accessed nomadically and persistently. Moreover, no extra investment such as in the form of a Storage Area Network (SAN) is needed, since the proposed system makes use of the existing computing and network resources within the participating nodes. As long as the network operates, this distributed system keeps operating as well. Backing up of the files also becomes generally unnecessary in this case.

Serverless Network Storage (SNS), based on the CDHT Serverless (P2P) network model, is proposed as a large-scale Serverless (P2P) storage application to provide reliable access to the users' files within a collaborative P2P network. The application also provides features such as high availability, data persistence, access control, mobility, and scalability.

6.1 Related Work

The distributed enterprise storage - Network Attached Storage (NAS) or Storage Area Network (SAN), or Fabric Attached Storage (FAS) as the convergence of the NAS and SAN - has been developed in spite it being generally accepted by the industry that issues, such as storage infrastructure transformation, managing storage system, and the huge costs of installation and maintenance still need to be adequately addressed.

There are some research projects in the P2P networking area that have been initiated to provide persistent network storage. These systems take advantage of underutilized resources and existing network infrastructure and do not require huge investment in modifying or creating an entirely separate networking infrastructure. More importantly, these tend to be self-managed systems with little of reliance on human intervention.

Intermemory [65] is such a self-organizing archival storage service built on Internet. It uses an enhanced-DNS lookup scheme to locate file fragments. Farsite [66] targets the WAN and a distributed directory service is used to locate content/file.

OceanStore [67] is aimed to provide a global and persistent transactional storage utility. It makes use of Byzantine update commitment, m-of-n data coding, and introspective replica management to realize the maintenance-free storage [67]. Generally it could be viewed as a distributed database. Tapestry [6] is the underlying substrate responsible for routing and object locating. Its price is in the complexity.

PAST is a large-scale, Internet-based storage utility. It is based on top of Pastry [5], a peer-to-peer routing and *content locating* scheme. Tapestry and Pastry have some similarity to the work of Plaxton scheme [33]. While CFS shares some common features with them, and similarly it is built on Chord [7], another peer-to-peer routing and content locating protocol. Based on the above description, it can be seen that SNS is similar to OceanStore, PAST, and CFS, since all of them are built on an underlying peer-to-peer networking protocol. From the perspective of system functionality, PAST and CFS are closer to SNS however SNS provides added levels of functionality and generally provides a more efficient scheme for the core *content locating* functions such as message routing and file/content search and locate operations through a well defined layered architecture.

The layered architecture also makes it possible for SNS to be developed as a framework for network centric application.

6.1.1 Content Locating

Content locating is the term used for the routing of the query and response messages, through a P2P network, for the purpose of retrieving a stored piece of information (within the network).

In PAST, each file and node has a unique distributed identifier. Pastry, the underlying P2P network, uses *fileId* as the destination address to send a request message. The file is retrieved from the nearest available node with a file replica. CFS is a block-oriented network storage scheme. It supports similar functions as PAST with the difference that the file is divided into standard sized blocks for replication over the network. Chord, the underlying P2P network in this case, implements a hash-like operation that maps the *blockId* to the *serverId* in order to reassemble the file from the replicated blocks.

Both Pastry and Chord need to exchange messages among the nodes every time a file lookup operation is performed. SNS on the other hand does not require message exchanges for every file lookup operation

6.1.2 System functionality

Limited by the topology, which relies upon the underlying peer-to-peer protocol to locate content, a particular relationship must be established between *fileId/blockId* and *nodeId* in PAST and CFS. These systems have to rely upon on this relationship entirely to perform operations, thus a file's other attributes are ignored. Therefore, functionality of the storage is substantially undermined. For instance, in PAST system, to retrieve a file, a client must know its complex *fileId*, which is rather inconvenient from the end client's standpoint. Additionally, since there is no ways to know file's other attributes such as file type or client-defined description, hence searching for file using such attributes is not possible.

There is no explicit delete operation available within CFS [21]; only insert and lookup operations are supported. Data is stored for an agreed interval of time. The file system provides a read-only access to the clients. The publisher may however update the contents.

PAST [22] allows insert, lookup, and reclaim operations. The reclaim operation clears the storage occupied by the replicas belonging to a specific *fileId*. However, the reclaim operation does not guarantee the actual removal of the file and its replicas from the system.

SNS aims to provide superior internal mechanisms for *content location* and storage operations. These mechanisms will be described in the following sections.

6.2 Serverless (P2P) Network Storage

A distributed/network file storage system has significant advantages over a traditional file storage system. These include geographically limitless access, eternal existence of a file, making use of underutilized resources, and self managing without authorized administrators. Only Serverless/P2P systems have the competence of fulfilling the advantages under heterogeneous and dynamic network conditions.

For the purposes of this research it is hypothesized that a Serverless Network Storage (SNS) system with layered architecture design can provide a persistent and scalable network file storage that has the main features of a conventional file system, provides a much greater level of data access (from fixed and roaming devices), and data redundancy without extensive and continual investments in the system infrastructure. The development of an SNS prototype constitutes the main contribution to the body of knowledge in the field.

6.2.1 Overview

SNS is designed to be a persistent, scalable, and secure network storage. In contrast to PAST or CFS, which focus on the Internet usage, SNS also caters for LAN requirements

by providing features such as disk quota management and access control. However the true strength of SNS, as a viable alternative to the traditional client-server network storage models, is envisaged through its layered design, which readily embraces features such as performance, scalability, data persistence, and security within a purely serverless P2P environment.

Essentially, each SNS node is hosted on a P2P networking layer as shown in Figure 6-1. Each of the top three SNS layers relies on this lower layer for functionality and/or connectivity. The lowest layer, i.e. the Serverless Layer, is part of the P2P network and thus has no application specific intelligence. The Serverless Layer interacts with the TCP/IP Layer to create and manage a logical network. The overall architectural design of the Serverless Layer is discussed in [19]. The implementation of a Serverless Layer is described in [20]. In order to make this paper self-contained, a brief explanation is given of the terms *cluster* and the *leader node* referred to in this paper. A *cluster* is defined in the Serverless Layer, as a community of nodes that have a close relationship within a certain locale [20]. When a *cluster* increases in size, it's subdivided into smaller sized *clusters*. Each *cluster* has a *leader* and a *shadow leader*. The *leader node* in a *cluster* coordinates communications within the *cluster* and acts as the gateway for communicating with other *clusters*.

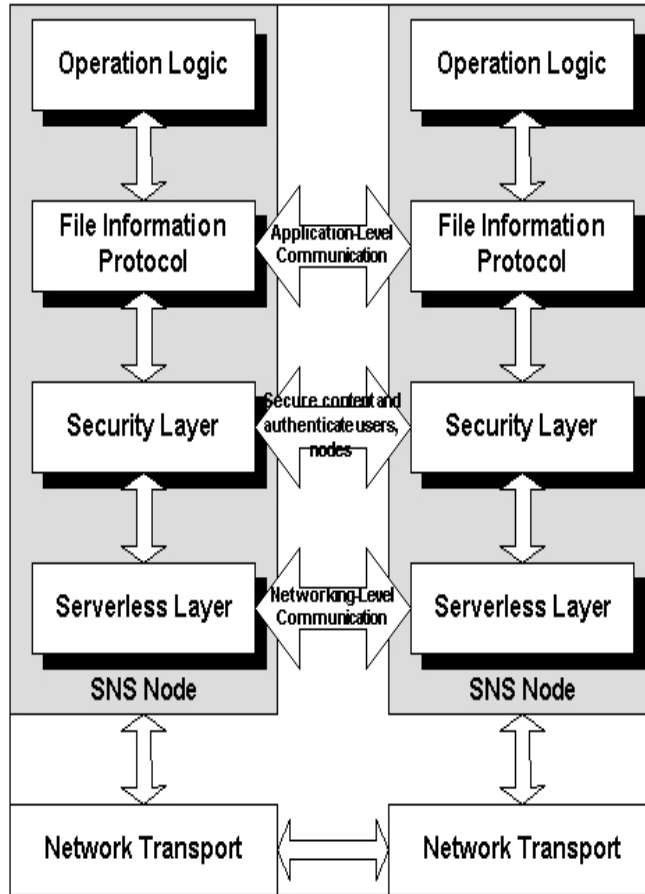


Figure 6- 1 The layered architecture of SNS

The terms of *home cluster*, *shadow home cluster*, *relative cluster*, and *stranger cluster* are defined within the Operation Logic and FIP Layers of the SNS and hence are notionally different from the *clusters* defined within the Serverless Layer. However in most cases a Serverless *cluster* would be the same as the SNS cluster.

The four SNS clustering terms represent the relationship between a file and its storage nodes. The (SNS) cluster that contains the node from where the file is saved from, into the network, is termed as the *home cluster* for the file. A cluster must meet the minimum storage capacity per node requirement to become the *home cluster* for the file. The Allocation Algorithm decides whether a cluster is qualified to become the *home cluster*. Each file can have only one *home cluster*. The Allocation Algorithm computes the number of replicas of the file, which are to be stored within the *home cluster*. A cluster, which is not the *home cluster*, but assigned to maintain r copies of the file, is called a *relative cluster*. The number of replicas in a *home cluster* is kept more than r , where r is a

fixed number computed by the Allocation Algorithm. The rationale behind storing greater than r replicas in the *home cluster* is based on the assumption that it's more likely that a client node would access the stored file from a node belonging to the *home cluster* of the file. The clusters that maintain no replicas of the file are termed as the *stranger clusters*. One of the *relative clusters* can assume the role of the *home cluster* in the case of the *home cluster's* capacity per node decreasing to a point where it can no longer meet the minimum requirement of being a *home cluster*. The *relative cluster* is renamed as the *shadow home cluster* in this case and the number of replicas in this *shadow home cluster* is increased from r to the value set for the *home clusters*. File replicas may now be stored within the under utilised nodes of the newly created *shadow home cluster*. Thus the role of a cluster, and the nodes within the cluster, may change dynamically with a large number of nodes joining or leaving a cluster.

6.2.2 File Information Protocol

File Information Protocol (FIP) is the application-level protocol responsible for updating and synchronising the property-repositories using XML-formatted messages. FIP maintains two property-repositories on each node, pertaining to every other node within the *cluster*. The first one is called the *File Property Repository*, which records all the attributes of the files with their *home cluster* being the same as the *local cluster*; it also includes the addresses of their replicas. The second one is called the *Storage Property Repository*, which records the storage provided by the nodes of the *local cluster* and the files stored within these nodes. FIP also maintains a repository on the *leader node* and the *shadow leader node* in every *cluster*, called the *Outsider File Property Repository*, comprising a list of the files stored in the local cluster whose *home cluster* is not the local cluster.

There are several message formats defined in FIP, such as Saved File Message, Deleted File Message, Search Query Message, Search Response Message, and Access Grant Message. The Saved File Message or the Deleted File Message is used to update and synchronise property repositories, among nodes and clusters, after a *save* or a *delete* file function has been executed. Search Query, and Search Response Messages are used for

content location. Access Grant Message is used by the *grant* function to bestow access rights on other clients.

6.2.3 Operation Logic

This section lists the Operation Logic operations. It also introduces the Allocation Algorithm, which coordinates between the Operation Logic and FIP Layers.

At present SNS supports a very basic set of operations for accessing the distributed file storage:

- ❑ *Save*. The client/user sets a priority level for the file by specifying an integer value for the priority variable *a*. The value of *a*, determines the number of nodes within the *home cluster* and the *relative clusters* that would be chosen to store the replicas. The Saved File Message is used to synchronise this operation among clusters. Bigger the value of *a*, greater the number of replicas a client may have for a certain file.
- ❑ *Retrieve*. The operation retrieves the nominated file from the system to the destination node. The file is retrieved by name and it is copied to the specified directory/path on the node.
- ❑ *Delete*. This operation deletes all the replicas of the file from the system. The Deleted File Message is used to synchronise this operation among clusters.
- ❑ *Listall*. This operation lists all the files saved by a certain client within the system. Only the file owner has the right to perform this operation.
- ❑ *Search*. Any client may search and/or retrieve a file from the system subject to the permission set by the file owner.
- ❑ *Grant*. Only the file owner is eligible to perform the grant operation for the file. At present read, delete and full-control rights are supported.

In order to reduce the network traffic, SNS requires that the nearest node gets the highest priority for storing or retrieving a file. Starting with the local cluster, SNS searches the nearby clusters to find the node(s) with the file replica(s). This process requires the Operation Logic Layer to work in close coordination with the FIP Layer.

6.2.4 Allocation Algorithm

Systems such as PAST would replicate a file on the nodes with *nodeIds* being numerically closest to the *fileId*. If one of these nodes cannot accommodate the replica, it then chooses one of nodes closest to itself for replication [22]. If however the available storage capacities at the destination nodes were known before hand, then there would be no need to redirect the file(s) to their nearest neighbours for replication. Doing so would save time and reduce network overheads.

SNS uses the Allocation Algorithm to select the nodes for file replication. At the node-level it determines the nodes within the *home cluster* and the *relative clusters* and at the cluster-level it determines the number of *relative clusters* that would be required in this regard. It also balances the number of replicas across the network when the number of nodes within the cluster varies or a cluster ceases to exist.

The *home cluster* nodes apply the Allocation Algorithm by referring to their respective *Local Storage Property Repositories*. Hence in most cases no message exchange is required among the nodes at this stage. The *relative clusters* and *stranger clusters* nodes apply the Algorithm by communicating with their *leader nodes*. It is left up to the *leader nodes* of these clusters to decide, which of their nodes would keep a copy of the file. The *Outsider File Property Repository*, on each *leader node*, is updated accordingly.

A SNS system may be viewed as a federation of nodes and clusters. The Allocation Algorithm plays the role of coordinator within the federation. Each cluster is only responsible for its own inhabitants i.e. the files. There are two types of inhabitants within a SNS federation. The first type is a file whose *home cluster* is the same as its local cluster and the second type is a migrant file whose *home cluster* is not the same as its local cluster. In other words, every file stored in SNS has two types of replicas. Clusters participate in the federation management by exchanging XML formatted messages. Normally, an internal change and/or instability within a cluster do not affect other clusters; unless the role of cluster changes. The Allocation Algorithm thus provides application resilience and persistence of data within an uncertain networking environment. It also attempts to minimise the number of messages required for inter-cluster coordination.

6.2.5 User Quota Management and Caching

The aggregate file size, which includes the original file size and the sum of all its replicas, is debited against the client's quota when a file is saved into the system. The delete operation adds the aggregate file size to the client's quota. The number of replicas may be changed autonomously, by each client node, according to the file availability considerations and the available network storage.

The Allocation Algorithm needs to fetch the desired file from a certain network distance in case of the request originating from a *relative cluster* or a *stranger cluster* node. The inter-cluster communication is conducted through the *cluster leader nodes* in this case. The *leader node* caches the file locally or it selects another node within its cluster for this purpose and assigns a Time-to-Live (TTL) value for the cached copy. The TTL value is adjusted on the fly according to the access rate of the file. The popular files thus may have more cached copies than the less frequently requested files. It may be pointed out that the cached copies of file do not affect the number of the file replicas defined under the Allocation Algorithm.

6.2.6 SNS Implementation

Based on the Serverless Layer's current functionality, a SNS prototype has been developed in Java. Several basic functions of the FIP and the Operation Logic layers such as *save*, *retrieve*, and *listall*, have been implemented in this prototype. Also, a very basic collaboration among the Operation Logic, FIP, and the Serverless Layer has been realized through the implementation of the Allocation Algorithm.

SNS prototype can be run on PCs and Java-enabled handheld devices such as PDA. It enables PCs and PDAs to form an ad hoc network and communicate with one another autonomously, in order to save and share files. Figure 6- 2 and Figure 6- 3 show the interfaces of SNS prototype for PC and PDA, respectively.

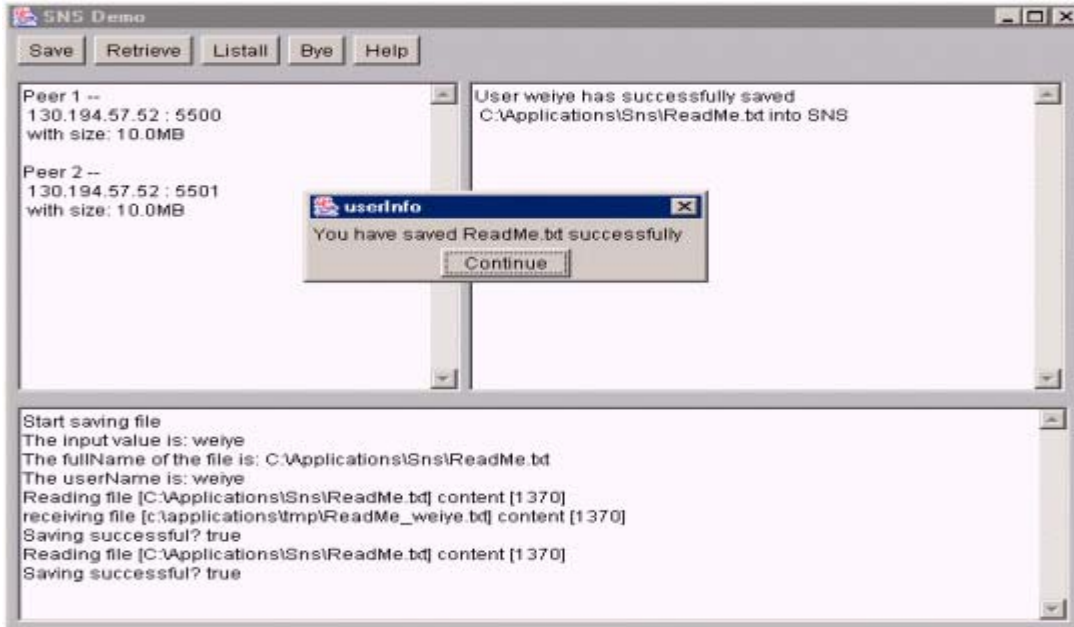


Figure 6- 2 PC interface of SNS implementation

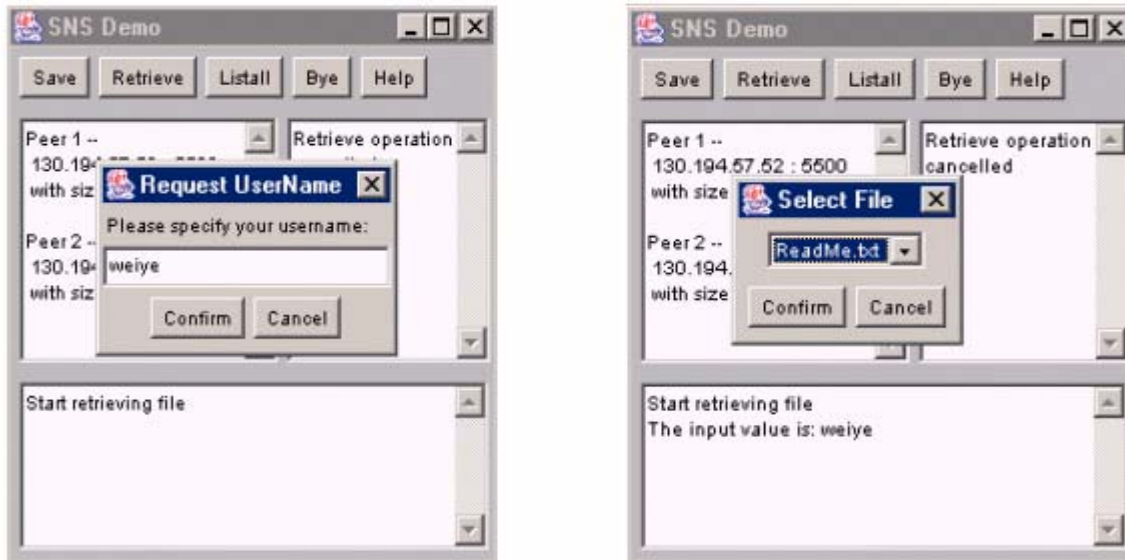


Figure 6- 3 PDA interface of SNS implementation

CoolCampus [68] is a research project of Monash University. CoolCampus aims to investigate the form, function and role of pervasive computing environments involving advanced technologies such as wireless communication, wearable and handheld computers, smart spaces, smartcards, etc., in empowering university students and staff in

their day to day activities. SNS was involved as part of CoolCampus project to implement a distributed Serverless storage application within wireless networks [69].

Chapter 7 Conclusion

The argument over which of the P2P models, whether structured or unstructured, has persisted for some time. Most researchers simply have a preference for one rather than the other. The first step in our research was to thoroughly investigate these two models. This investigation involved a comprehensive comparison of those two models.

Chapter 2 reviews the history and the development of the unstructured P2P model. A number of projects, such as Gnutella, Freenet and Jabber, were investigated in the chapter to reveal their common features. The major weakness of the unstructured model is the message-broadcasting routing scheme, which causes a number of issues such as performance, scalability and network congestion.

Certain network models such as Power-law and Small-world can be utilized to improve the performance of the unstructured model. Chapter 2 discusses the Power-law distribution, which can be used to improve Gnutella scalability. The chapter also discusses the concept of a Small-world network that improves the performance of Freenet.

Chapter 3 looks into the characteristics of the structured model. In this model, a DHT network is built on an abstract called keyspace. However, the keyspace and its partitioning scheme may vary from network to network. That variation leads to different routing schemes. This chapter investigates the structures and routing schemes of Chord, CAN, Pastry and Tapestry, as well as related techniques such as Plaxton data structure. The analysis indicates DHT's advantages on scalability and lookup efficiency, as well as the disadvantages such as network organization complexity.

Chapter 4 first analyses the content request pattern on the Internet. The literature indicates that content request follows Zipf's law on the Internet. The chapter further discusses the finding that content request on popular P2P systems generally follows Zipf's law as well. It is noted that the Internet and P2P networks are made up of heterogeneous transient nodes.

Based on the knowledge of network infrastructure, content request pattern and the characteristic analysis in Chapter 2 and 3, Chapter 4 compares the structured and

unstructured models in more detail, focusing on characteristics such as lookup efficiency, handling highly dynamic network environments, load balancing and content search patterns. The comparisons show that the unstructured model can look up content as efficiently as the structured one. Added to this, the unstructured model is generally superior to the structured one with respect to network maintenance cost while search within the structured model is globally oriented, thus more accurate than the unstructured one. The comparisons indicate that both the structured and the unstructured model have strengths and weaknesses. Consequently, it is rather difficult to determine which of them would provide the overall better solution. This leads us to the CDHT communication scheme, which combines the structured and the unstructured models in such a way that these two competing approaches complement each other.

The Serverless (P2P) network architecture we envision leads to a network which is not only as scalable as the pure DHT-based systems, but it is also more resilient to dynamic network conditions. The network should be able to achieve load balancing autonomously. To achieve these objectives a Serverless (P2P) clustered DHT routing scheme (CDHT), which has a hybrid-overlay structure, is presented in Chapter 5. CDHT delegates to the network-aware Serverless Layer the task of forming the clusters of nodes on an ad hoc basis. The Serverless Layer effectively forms clusters, elects cluster leader nodes and enables intra- and inter- cluster communication. It can sustain highly dynamic configurational changes within a cluster; hence, it deals with network uncertainty in CDHT, which is built on top of Server Layer.

Heuristic Content Location (HCL) exploits network locality and content popularity for balancing network traffic autonomously while avoiding hot spots and improving on lookup performance for popular content location. HCL works alongside the DHT lookup pattern in the CDHT architecture. The simulation results demonstrate that HCL can achieve better lookup efficiency for popular content than DHT. The simulation also implies that the effectiveness of the HCL lookup depends on a relatively stable network and content structure. This is contributed by the Serverless Layer tackling the network uncertainty in the CDHT architecture.

In CDHT, each cluster in the Serverless Layer selects a single node as the Content Rendezvous (CR) node according to selection criteria such as processing power, bandwidth, storage capacity, etc. Each CR node has a Content Repository and a Content Pointer Cache for efficient content location. CRs of Serverless Layer clusters form a DHT overlay. Other (ordinary) nodes do not participate in the DHT activities. CDHT implements specific algorithms to deal with rapid network change and CR replacement, as well as to balance load on CR nodes.

The experimental results show that CDHT with HCL is superior to the pure structured model in three aspects: it improves the lookup performance for popular content; it improves the network's resilience to handling highly dynamic network environments and improves load balancing.

A Serverless P2P file storage system has significant advantages over a traditional client-server system such as geographically limitless access, eternal existence of a file, making use of underutilized resources, and self managing without authorized administrators. Chapter 6 first investigates a number of related DHT-based storage systems, especially focusing on content location and system functionality.

SNS is further presented in Chapter 6. Based on CDHT, SNS has a layered architecture to enable a persistent, scalable and secure network storage. Compared to the peer systems, SNS provides better performance in locating popular content and provides richer system functions. A Java-based prototype was implemented on PC and PDA. Any such devices running the prototype can form an autonomous network, store and retrieve files, as well as contribute disk space. It is envisaged that SNS would evolve as a distributed database modelled on the P2P computing paradigm; hence, the relevant functionalities such as indexing and transaction would be within the scope of future research.

Graph Neuron (GN) [70] is a highly-scalable P2P associative memory system capable of handling concurrent streams of inputs by processing them and matching them with the historical data (available within the network). The message routing and nearest GN location would be facilitated by CDHT. This is also being investigated as part of our ongoing research in wireless sensor networks and net-centric pattern recognition algorithms.

Reference

- [1] A. Barabasi and R. Albert, “Emergence of Scaling in Random Networks”, *Science*, Vol 286, pp. 509, Oct 1999.
- [2] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman, “ Search in power-law networks”, *Physical Review E*, Vol 64, 2001.
- [3] D. J. Watts, “Small-worlds: The dynamics of networks between Order and Randomness”, *Princeton University Press*, 1999.
- [4] J. Kleinberg, “Navigation in a small-world”, *Nature*, 406, pp. 845, 2000.
- [5] A. Rowstron and P. Druschel, “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems”, *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.
- [6] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. “Tapestry: An infrastructure for fault-resilient wide-area location and routing”, *TechnicalReport UCB/CSD-01-1141*, U. C. Berkeley, April 2001.
- [7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for Internet applications,” *Proc. ACM Special Interest Group on Data Communication 2001*, pp. 149-160, San Diego, CA, Aug. 2001.
- [8] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A Scalable Content-Addressable Network,” *Proc. ACM Special Interest Group on Data Communication 2001*, Aug. 2001.
- [9] The Gnutella protocol specification, 2000.
<http://dss.clip2.com/GnutellaProtocol04.pdf>
- [10] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, “Freenet: A distributed anonymous information storage and retrieval system in designing privacy

enhancing technologies”, *International Workshop on Design Issues in Anonymity and Unobservability, LNCS 2009, 2001*.

- [11] L. A. Adamic, B. A. Huberman, “Zipf’s law and the internet”, *Glottometrics 3* (2002).
- [12] C. R. Cunha, A. Bestavros, and M. E. Crovella, “ Characteristics of WWW client-based traces”, *Technical Report TR-95-010, Boston University, April 1995*.
- [13] S. Saroiu, P. K. Gummadi and S. D. Gribble, “A measurement study of Peer-to-Peer file sharing systems”, *Proc. Multimedia Computing and Networking*, pp. 156-170, 2002.
- [14] S. Ratnasamy, S. Shenker, and I. Stoica, “Routing Algorithms for DHTs: Some Open Questions”, *Proc. International Peer-to-Peer Workshop*, pp. 161-172, 2002.
- [15] B. Traversat, M. Abdelaziz, and E. Pouyoul, “Project JXTA: A loosely-consistent DHT Rendezvous Walker”, http://www.jxta.org/project/www/white_papers.html
- [16] W. Ye, A. I. Khan and E. A. Kendall, “CDHT: The design of a Clustered DHT Routing Scheme for Serverless (P2P) Networks”, *Proc. IEEE The 12th International Conference on Networks*, pp 351-356, Nov. 2004, Singapore.
- [17] W. Ye, A. I. Khan, E. A. Kendall, and C. T. Yeo, “CDHT: A Clustered DHT Communication Scheme for Serverless (P2P) Networks”, *Proc. IEEE The 13th International Conference on Networks*, pp 684-691, Nov. 2005, Malaysia.
- [18] W. Ye, A. I. Khan and E. A. Kendall, “Distributed network file storage for a Serverless (p2p) network”, *Proc. IEEE The 11th International Conference on Networks*, pp. 343-347, Sep. 2003, Sydney, Australia.
- [19] A. I. Khan and R. Spindler, “A blueprint for building serverless application on the net,” Available at <http://au.arxiv.org/abs/cs.DC/0107009>
- [20] S. N. Senaratna and A. I. Khan, “An autonomous clustering algorithm for Serverless peer-to-peer systems,” *Proc. The 6th International Conference on High Performance Computing in Asia Pacific Region*, pp. 371-378, 2002. Bangalore, India.

- [21] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," *Proc. ACM Symposium on Operating Systems Principles 2001*, Banff, Canada, Oct. 2001.
- [22] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," *Proc. ACM Symposium on Operating Systems Principles 2001*, Banff, Canada, Oct.2001.
- [23] C. Shirky. Napster. A chapter in "Peer-to-Peer: Harnessing the power of disruptive technologies", pp. 21-37, edited by Andy Oram, *O'Reilly and Assoc*, 2001.
- [24] I. Clarke, S. G. Miller, T. W. Hong, O. Sandberg and B. Wiley, "Protecting free expression online with Freenet", *IEEE Internet Computing*, pp. 40-49, Jan/Feb 2002.
- [25] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, Nov. 2002, Vol. 45 No. 11, pp. 56-61.
- [26] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making Gnutella-like P2P systems scalable", *ACM Special Interest Group on Data Communication 2003*, pp. 407-418.
- [27] H. Zhang and A. Goel, "Using the small-world model to improve Freenet performance", *IEEE Infocomm 2002*, pp. 1228-1237.
- [28] A-L. Barabasi and R. Albert, "Emergence of Scaling in Random Networks", *Science*, Vol 286, pp. 509-512, Oct 1999.
- [29] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On Power-law relationships of the Internet topology", *ACM Special Interest Group on Data Communication 1999*, pp. 251-262.
- [30] K. Hui, J. Lui, D. Yau, "Small World Overlay P2P Networks", *International Work on Quality of Service 2004*.
- [31] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/NIST, National Technical Information Service, *Springfield, VA, Apr. 1995*.

- [32] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in P2P systems", *Communications Of the ACM*, Vol 46. No2, pp. 43-48, February 2003.
- [33] C. G. Plaxton, R. Rajaraman, and A.W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, Vol 32, pp. 241–280, 1999.
- [34] Potter, William Gray, " 'Of Making Many Books There is No End': Bibliometrics and Libraries." *The Journal of Academic Librarianship*, Vol 14, pp. 238a-238c, September 1988.
- [35] L. A. Adamic, "The Small World Web", *Proceedings of ECDL'99. Lecture Notes in Computer Science 1696*, pp. 443-452, Berlin: Springer.
- [36] H. Jeong, R. Albert and A.L.Barabasi, "Diameter of the World Wide Web". *Nature* 1999 401, 130.
- [37] M. Faloutsos, P. Faloutsos and C. Faloutsos, "On Power-Law Relationships of the Internet Topology", *Proceedings of ACM Special Interest Group on Data Communication 1999*, pp. 251-262.
- [38] P. Erdős and A. Rényi, "On the Evolution of Random Graphs", *Publications of the Mathematical Institute of the Hungarian Academy of Sciences* 5, pp. 17-61.
- [39] R. Albert and A. L. Barabasi, "Statistical mechanics of complex networks", *Review of Modern Physics* 74, pp. 47-94.
- [40] L. Breslau et al., "Web Caching and Zipf-like Distributions: evidence and implications", *Proceedings of INFOCOM'99*, pp. 126-134.
- [41] Slyck.com, <http://www.slyck.com>, 2005
- [42] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy, "An analysis of Internet content delivery systems," in *Symposium on Operating Systems Design and Implementation*, pp. 315–327, 2002.

- [43] S. Zhao, D. Stutzbach and R. Rejaie, “Characterizing Files in the Modern Gnutella Network: A Measurement Study”, *Proceeding of Multimedia Computing and Networking 2006*.
- [44] S. Sen and J. Wang, “Analyzing peer-to-peer traffic across large networks”, in *Networking, IEEE/ACM Transactions*, April 2004
- [45] KaZaA.com, <http://www.kazaa.com>.
- [46] Direct Connect, <http://www.neo-modus.com>.
- [47] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, I. Stoica, “Load Balancing in Structured P2P Systems”, *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS) 2003*, pp. 219.
- [48] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, I. Stoica, “Complex Queries in DHT-based peer-to-peer networks”, *1st International Workshop on Peer-to-Peer Systems (IPTPS) 2002, LNCS 2429*.
- [49] J. Li et al, “On the feasibility of Peer-to-Peer Web indexing and search”, *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS) 2003*, pp207-215.
- [50] Q. Lv et al, “Search and Replication in unstructured peer-to-peer networks”, in *International Conference on Supercomputing*, pp. 84–95, ACM Press. New York, USA, 2002.
- [51] R. Droms, “Dynamic Host Configuration Protocol”, *IETF, March 1997, RFC2131*.
- [52] C. Rigney, A. Rubens, W. Simpson, and S. Willens, “Remote Authentication Dial In User Service”, *IETF, April 1997, RFC2138*.
- [53] P. Srisuresh, and M. Holdrege, “IP Network Address Translator (NAT) Terminology and Considerations”, *IETF, August 1999, RFC2663*.
- [54] M. Gerla, and J. T. Tsai, “Multicluster, Mobile, Multimedia Radio Network”, *ACM–Baltzer Journal of Wireless Networks, No. 1, (1995), 255-265*.
- [55] S. E. Deering, “A Multicast Extension to the Internet Protocol”, *IETF, August 1989, RFC 1112*.

- [56] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei, "An Architecture for Wide-Area Multicast Routing", *Special Interest Group on Data Communication 1994*, pp. 126-135, London UK, August 1994.
- [57] M. Ripeanu, A. Iamnitchi, and I. Foster, "Mapping the Gnutella Network", *The IEEE Internet Computing Journal*, pp. 40-49, Jan/Feb 2002.
- [58] K. Sripanidkulchai, B. Maggs, H. Zhang, "Efficient content location using Interest-Based Locality in Peer-to-Peer systems", *Proc. Infocomm*, 2003.
- [59] FreePastry, <http://www.cs.rice.edu/CS/Systems/Pastry/FreePastry>
- [60] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron, "Exploiting network proximity in peer-to-peer overlay networks", *Technical report MSR-TR-2002-82*.
- [61] JNS1.7, <http://jns.sourceforge.net/>
- [62] NS-2, <http://www.isi.edu/nsnam/ns/>
- [63] JAVIS, <http://www.cs.technion.ac.il/~gabik/Javis4Swans.html>
- [64] NAM, http://warriors.eecs.umich.edu/viz_tools/nam.html.
- [65] A. Goldberg and P. Yianilos, "Towards an archival Intermemory", *Proc. IEEE Int'l Conf. ADL'98*, pp. 147-156, IEEE Computer Soc. Press, Los Alamitos, California, 1998.
- [66] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs", *Proc. ACM SIGMETRICS 2000*, pp. 34-43.
- [67] J. Kubiawicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao, "Oceanstore: An architecture for global-scale persistent store", *Proc. ACM Architectural Support for Programming Languages and Operating Systems 2000*, Cambridge, MA.
- [68] Monash University, CoolCampus Project, <http://www.infotech.monash.edu.au/promotion/coolcampus/index.html>

- [69] SNS and CoolCampus Project,
<http://www.infotech.monash.edu.au/promotion/coolcampus/projects/projects-peertopeer.html>
- [70] A. I. Khan, “A peer-to-peer associate memory network for intelligent information systems”, *Proc. The 13th Australian Conference on Information Systems, volume 1, 2002.*
- [71] B. Pourebrahimi, K.L.M. Bertels, S. Vassiliadis, “A Survey of Peer-to-Peer Networks”, *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, ProRisc 2005, November 2005 (BibTeX).
- [72] Jabber, <http://www.jabber.org>

Appendix A. Code example

The following is the example of code to build a DHT network model based on JNS. It is used to test the lookup success rate of DHT.

```
/**
 * Test_DHT.java
 *
 * Copyright (c) 2007. First created on 29-May-04
 *
 * @author Wei Ye
 */
package jns;

import jns.command.Command;
import jns.command.StopCommand;
import jns.element.*;
import jns.trace.JavisTrace;
import jns.trace.Trace;
import jns.util.Debug;
import jns.util.IPAddr;
import jns.util.Protocols;
import jns.util.Preferences;
import jns.util.NodeIDManager;
import jns.dynamic.PacketSender;

import java.io.IOException;
import java.io.FileOutputStream;
import java.util.Enumeration;
import java.util.Vector;

/**
 * This class builds and tests DHT protocol.
 */
public class Test_DHT
{
    private static Node[] node;

    private static int network_size;

    private static Simulator sim;

    private static FileOutputStream fos;

    private static Debug debug;

    public Test_DHT () {
        node = new Node[Preferences.Node_Number];
    }
}
```

```

sim = Simulator.getInstance();
network_size = Preferences.Node_Number;
}

public static void main(String args[])
{

Test_DHT testDHT = new Test_DHT();
if(args.length == 3)
    debug = new Debug("Test_DHT-" + args[2] + ".log");
else
    debug = new Debug("Test_DHT.log");

//Create the nodes, and attach with tracing to the simulator
NodeIDManager idManager = NodeIDManager.getInstance();
idManager.setIDSet();
for(int i=0; i<Preferences.Node_Number; i++) {
    String id = idManager.retrieveSequenceID(i);
    node[i] = new Node(id);
    sim.attach(node[i]);
}

for(int i=0; i<Preferences.Node_Number; i++) {
    String id = idManager.retrieveSequenceID(i);
    //Get routing entries
    Vector routingTable = (Vector) idManager.getRoutingTable(id);
    Enumeration e = routingTable.elements();
    String[] entryName = new String[Preferences.Routing_Entry_Number];
    Node[] routingEntry = new Node[Preferences.Routing_Entry_Number];
    Interface[] localface = new Interface[Preferences.Routing_Entry_Number];
    Interface[] entryface = new Interface[Preferences.Routing_Entry_Number];
    int j = 0;
    while(e.hasMoreElements()) {
        entryName[j] = (String) e.nextElement();
        routingEntry[j] = new Node(entryName[j]);

        Vector existingElements = (Vector) sim.getExistingElements();
        Enumeration s = existingElements.elements();
        while(s.hasMoreElements()) {
            Node curnode = (Node) s.nextElement();
            String name = curnode.getName();
            if(name.equals(entryName[j])) {
                routingEntry[j] = curnode;
                int[] entrynum = (int[]) idManager.getIPAddressNumber(name);
                int[] localnum = (int[]) idManager.getIPAddressNumber(id);
                entryface[j] = new DuplexInterface(new IPAddr(entrynum[0], entrynum[1],
entrynum[2], entrynum[3]));
            }
        }
    }
}

```

```

        localface[j] = new DuplexInterface(new IPAddr(localnum[0], localnum[1],
localnum[2], localnum[3]));
        node[i].attach(localface[j]);
        sim.attach(localface[j]);
        curnode.attach(entryface[j]);
        sim.attach(entryface[j]);

        Link a = new DuplexLink(500000, 0.008);
        entryface[j].attach(a, true);
        localface[j].attach(a, true);
        sim.attach(a);

        node[i].addRoute(new IPAddr(entrynum[0], entrynum[1], entrynum[2], entrynum[3]),
new IPAddr(255, 255, 255, 0), localface[j]);
        curnode.addRoute(new IPAddr(localnum[0], localnum[1], localnum[2], localnum[3]),
new IPAddr(255, 255, 255, 0), entryface[j]);

        break;
    }
}
j++;
}

//Get leaf entries
Vector leafSet = (Vector) idManager.getLeafSet(id);
Enumeration l = leafSet.elements();
String[] leafName = new String[Preferences.Leaf_Entry_Number];
Node[] leafEntry = new Node[Preferences.Leaf_Entry_Number];
Interface[] localface1 = new Interface[Preferences.Leaf_Entry_Number];
Interface[] leafface = new Interface[Preferences.Leaf_Entry_Number];
int k = 0;
while(l.hasMoreElements()) {
    leafName[k] = (String) l.nextElement();
    leafEntry[k] = new Node(leafName[k]);

    Vector existingElements = (Vector) sim.getExistingElements();
    Enumeration s = existingElements.elements();
    while(s.hasMoreElements()) {
        Node curnode = (Node) s.nextElement();
        String name = curnode.getName();
        if(name.equals(leafName[k])) {
            int[] leafnum = (int[]) idManager.getIPAddressNumber(name);
            int[] localnum = (int[]) idManager.getIPAddressNumber(id);
            leafface[k] = new DuplexInterface(new IPAddr(leafnum[0], leafnum[1], leafnum[2],
leafnum[3]));
            localface1[k] = new DuplexInterface(new IPAddr(localnum[0], localnum[1],
localnum[2], localnum[3]));

```

```

        node[i].attach(localface1[k]);
        sim.attach(localface1[k]);
        curnode.attach(leafface[k]);
        sim.attach(leafface[k]);

        Link a = new DuplexLink(500000, 0.008);
        leafface[k].attach(a, true);
        localface1[k].attach(a, true);
        sim.attach(a);

        node[i].addRoute(new IPAddr(leafnum[0], leafnum[1], leafnum[2], leafnum[3]), new
IPAddr(255, 255, 255, 0), localface1[k]);
        curnode.addRoute(new IPAddr(localnum[0], localnum[1], localnum[2], localnum[3]),
new IPAddr(255, 255, 255, 0), leafface[k]);

        break;
    }
}
k++;
}

try {
    int source_num = (int) Integer.parseInt(args[0]);
    int des_num = (int) Integer.parseInt(args[1]);
    if(args.length == 3) {
        int dept_nodes_num = (int) Integer.parseInt(args[2]);
        testDHT.removeDHTNodes(dept_nodes_num);
    }
    testDHT.lookup(source_num, des_num);
} catch(NumberFormatException nfe) {
    nfe.printStackTrace();
}

sim.schedule(new StopCommand(1));
}

private void lookup(int source_num, int des_num) {
    NodeIDManager idManager = NodeIDManager.getInstance();
    new Thread(sim).start();
    Node source = (Node) node[source_num];
    Node dest = (Node) node[des_num];
    String dest_id = dest.getName();
    String dest_ip = idManager.getIPAddress(dest_id);
    byte[] data = (byte[]) dest_ip.getBytes();
    int[] dest_num = idManager.getIPNumberFromString(dest_ip);
    IPAddr dest_ip_addr = new IPAddr(dest_num[0], dest_num[1], dest_num[2], dest_num[3]);
    String source_id = source.getName();

```

```

String sec_dest_id = idManager.getNextDest(source_id, dest_ip_addr);
String sec_dest = idManager.getIPAddress(sec_dest_id);
int[] sec_dest_num = idManager.getIPNumberFromString(sec_dest);
IPAddr sec_dest_ip = new IPAddr(sec_dest_num[0], sec_dest_num[1], sec_dest_num[2],
sec_dest_num[3]);
IPHandler.countLookup();
sim.schedule(new PacketSender(source.getIPHandler(), sec_dest_ip, 0.1,data));
}

private void removeDHTNodes(int num) {
    int n = 0;
    for(int i=0; i<num; i++) {
        float m = (float) Math.random() * network_size;
        n = Math.round(m);
        if(n == network_size)
            n = network_size - 1;
        Node r_node = (Node) node[n];
        sim.detach(r_node);
        network_size --;
    }
}

private static String getNodeRemoved(IPAddr ipAddr) {
    String removedNode = null;
    boolean removed = false;
    Vector v = sim.getRemovedElements();
    Enumeration e = v.elements();
    while(e.hasMoreElements()) {
        Node curNode = (Node) e.nextElement();
        removed = curNode.hasIPAddress(ipAddr);
        if(removed)
            return curNode.getName();
    }
    return removedNode;
}
}
}

```

The following is the example of code to build a CDHT network model based on JNS. It is used to test the lookup success rate of CDHT/HCL with various CR recovery rates.

```

/**
 * Test_CDHT.java
 *
 * Copyright (c) 2007. First created on 5-June-04
 *
 * @author Wei Ye

```

```

*
*/
package jns;

import jns.command.Command;
import jns.command.StopCommand;
import jns.element.*;
import jns.trace.JavisTrace;
import jns.trace.Trace;
import jns.util.Debug;
import jns.util.IPAddr;
import jns.util.Protocols;
import jns.util.Preferences;
import jns.util.NodeIDManager;
import jns.dynamic.PacketSender;

import java.io.IOException;
import java.io.FileOutputStream;
import java.util.Enumeration;
import java.util.Vector;

/**
 * This class builds and tests CDHT protocol.
 */
public class Test_CDHT
{
    private static Node[] node;

    private static Node[] cnode;

    private static int dht_network_size;

    private static int cdht_network_size;

    private static Simulator sim;

    private static FileOutputStream fos;

    private static Debug debug;

    public Test_CDHT () {
        node = new Node[Preferences.Node_Number/4];
        cnode = new Node[Preferences.Node_Number*3/4];
        sim = Simulator.getInstance();
        dht_network_size = Preferences.Node_Number/4;
        cdht_network_size = Preferences.Node_Number*3/4;
    }
}

```

```

public static void main(String args[]) {
    Test_CDHT testCDHT = new Test_CDHT();
    if(args.length == 2)
        debug = new Debug("Test_CDHT.log");
    else if(args.length == 4)
        debug = new Debug("Test_CDHT-" + args[2] + "-" + args[3] + ".log");

    //Create the nodes for the DHT layer, and attach with tracing to the simulator
    NodeIDManager idManager = NodeIDManager.getInstance();
    idManager.setIDSet();
    idManager.differentiateContent();
    idManager.setIDSet();
    for(int i=0; i<Preferences.Node_Number/4; i++) {
        String id = idManager.retrieveSequenceID(i);
        node[i] = new Node(id);
        node[i].fillContentPointerCache();
        sim.attach(node[i]);
    }

    String id = "";
    for(int i=0; i<Preferences.Node_Number/4; i++) {
        id = idManager.retrieveSequenceID(i);
        //Get routing entries
        Vector routingTable = (Vector) idManager.getRoutingTable(id);
        Enumeration e = routingTable.elements();
        String[] entryName = new String[Preferences.Routing_Entry_Number];
        Node[] routingEntry = new Node[Preferences.Routing_Entry_Number];
        Interface[] localface = new Interface[Preferences.Routing_Entry_Number];
        Interface[] entryface = new Interface[Preferences.Routing_Entry_Number];
        int j = 0;
        while(e.hasMoreElements()) {
            entryName[j] = (String) e.nextElement();
            routingEntry[j] = new Node(entryName[j]);

            Vector existingElements = (Vector) sim.getExistingElements();
            Enumeration s = existingElements.elements();
            while(s.hasMoreElements()) {
                Node curnode = (Node) s.nextElement();
                String name = curnode.getName();
                if(name.equals(entryName[j])) {
                    routingEntry[j] = curnode;
                    int[] entrynum = (int[]) idManager.getIPAddressNumber(name);
                    int[] localnum = (int[]) idManager.getIPAddressNumber(id);
                    entryface[j] = new DuplexInterface(new IPAddr(entrynum[0], entrynum[1],
entrynum[2], entrynum[3]));
                }
            }
        }
    }
}

```

```

        localface[j] = new DuplexInterface(new IPAddr(localnum[0], localnum[1],
localnum[2], localnum[3]));
        node[i].attach(localface[j]);
        sim.attach(localface[j]);
        curnode.attach(entryface[j]);
        sim.attach(entryface[j]);

        Link a = new DuplexLink(500000, 0.008);
        entryface[j].attach(a, true);
        localface[j].attach(a, true);
        sim.attach(a);

        node[i].addRoute(new IPAddr(entrynum[0], entrynum[1], entrynum[2],
entrynum[3]), new IPAddr(255, 255, 255, 0), localface[j]);
        curnode.addRoute(new IPAddr(localnum[0], localnum[1], localnum[2],
localnum[3]), new IPAddr(255, 255, 255, 0), entryface[j]);

        break;
    }
}
j++;
}
//Get leaf entries
Vector leafSet = (Vector) idManager.getLeafSet(id);
Enumeration l = leafSet.elements();
String[] leafName = new String[Preferences.Leaf_Entry_Number];
Node[] leafEntry = new Node[Preferences.Leaf_Entry_Number];
Interface[] localface1 = new Interface[Preferences.Leaf_Entry_Number];
Interface[] leafface = new Interface[Preferences.Leaf_Entry_Number];
int k = 0;
while(l.hasMoreElements()) {
    leafName[k] = (String) l.nextElement();
    leafEntry[k] = new Node(leafName[k]);

    Vector existingElements = (Vector) sim.getExistingElements();
    Enumeration s = existingElements.elements();
    while(s.hasMoreElements()) {
        Node curnode = (Node) s.nextElement();
        String name = curnode.getName();
        if(name.equals(leafName[k])) {
            int[] leafnum = (int[]) idManager.getIPAddressNumber(name);
            int[] localnum = (int[]) idManager.getIPAddressNumber(id);
            leafface[k] = new DuplexInterface(new IPAddr(leafnum[0], leafnum[1], leafnum[2],
leafnum[3]));
            localface1[k] = new DuplexInterface(new IPAddr(localnum[0], localnum[1],
localnum[2], localnum[3]));
            node[i].attach(localface1[k]);

```

```

sim.attach(localface1[k]);
sim.attach(leafface[k]);

Link a = new DuplexLink(500000, 0.008);
leafface[k].attach(a, true);
localface1[k].attach(a, true);
sim.attach(a);

node[j].addRoute(new IPAddr(leafnum[0], leafnum[1], leafnum[2], leafnum[3]), new
IPAddr(255, 255, 255, 0), localface1[k]);
curnode.addRoute(new IPAddr(localnum[0], localnum[1], localnum[2],
localnum[3]), new IPAddr(255, 255, 255, 0), leafface[k]);

break;
}
}
k++;
}
}

```

```

int c = Preferences.Node_Number/4;
for(int i=0; i<c; i++) {
id = idManager.retrieveSequenceID(i);
Vector existingElements = (Vector) sim.getExistingElements();
Node dht_node = (Node) existingElements.get(i);
cnode[i] = new Node(Integer.toString(i));
cnode[i+c] = new Node(Integer.toString(i+c));
sim.attach(cnode[i]);
sim.attach(cnode[i+c]);

IPAddr[] cipAddr = new IPAddr[3];
int starting_subnet = Preferences.Starting_Subnet;
for(int sub_length=128; sub_length<=c*3; sub_length=sub_length+128) {
int cur_subnet = starting_subnet + sub_length/128;
int cur_ip = 0;
if(i >= 128 && i < 256)
cur_ip = i - 128;
else
cur_ip = i;
if(i>=sub_length - 128 && i<sub_length) {
cipAddr[0] = new IPAddr(192,168,cur_subnet, cur_ip);
}
if(i+c>=sub_length - 128 && i+c<sub_length) {
cipAddr[1] = new IPAddr(192,168,cur_subnet, cur_ip);
}
if(i+2*c>=sub_length - 128 && i+2*c<sub_length) {
cipAddr[2] = new IPAddr(192,168,cur_subnet, cur_ip);
}
}
}

```

```

    }
}

int[] ipnum = idManager.getIPAddressNumber(id);
Interface[] dht_iface = new Interface[3];
Interface[] cdht_iface = new Interface[3];
for(int k=0; k<3; k++) {
    cdht_iface[k] = new DuplexInterface(cipAddr[k]);
    dht_iface[k] = new DuplexInterface(new IPAddr(ipnum[0], ipnum[1], ipnum[2],
ipnum[3]));
    dht_node.attach(dht_iface[k]);
}
cnode[i].attach(cdht_iface[0]);
cnode[i+c].attach(cdht_iface[1]);
cnode[i+2*c].attach(cdht_iface[2]);
for(int k=0; k<3; k++) {
    sim.attach(cdht_iface[k]);
    sim.attach(dht_iface[k]);
}

for(int k=0; k<3; k++) {
    Link a = new DuplexLink(500000, 0.008);
    cdht_iface[k].attach(a, true);
    dht_iface[k].attach(a, true);
    sim.attach(a);
    dht_node.addRoute(cipAddr[k], new IPAddr(255, 255, 255, 0), dht_iface[k]);
}
cnode[i].addDefaultRoute(cdht_iface[0]);
cnode[i+c].addDefaultRoute(cdht_iface[1]);
cnode[i+2*c].addDefaultRoute(cdht_iface[2]);
}

try {
    int source_num = (int) Integer.parseInt(args[0]);
    int des_num = (int) Integer.parseInt(args[1]);
    if(args.length == 4) {
        int dept_nodes_num = (int) Integer.parseInt(args[2]);
        double rv_failed_rate = (double) Double.parseDouble(args[3]);
        double perc = (double) dept_nodes_num/Preferences.Node_Number;
        int dept_dhtnodes_num = (int) (dht_network_size * perc * rv_failed_rate);
        int dept_cdhtnodes_num = 0;
        dept_cdhtnodes_num = dept_nodes_num - dept_dhtnodes_num;
        testCDHT.removeDHTNodes(dept_dhtnodes_num);
        testCDHT.removeCDHTNodes(dept_cdhtnodes_num);
    }
    testCDHT.lookup(source_num, des_num);
} catch(NumberFormatException nfe) {

```

```

        nfe.printStackTrace();
    }
    sim.schedule(new StopCommand(1));
}

private void lookup(int source_num, int des_num) {
    NodeIDManager idManager = NodeIDManager.getInstance();
    new Thread(sim).start();
    Node source = (Node) cnode[source_num];
    Node dest = (Node) node[des_num];
    String dest_id = dest.getName();
    String dest_ip = idManager.getIPAddress(dest_id);
    byte[] data = (byte[]) dest_ip.getBytes();
    int[] dest_num = idManager.getIPNumberFromString(dest_ip);
    IPAddr dest_ip_addr = new IPAddr(dest_num[0], dest_num[1], dest_num[2],
dest_num[3]);
    IPHandler sipHandler = (IPHandler) source.getIPHandler();
    IPAddr sipAddr = (IPAddr) sipHandler.getAddress();
    String source_ip = sipAddr.toString();
    String source_id = source.getName();
    IPHandler.countLookup();
    sim.schedule(new PacketSender(source.getIPHandler(), dest_ip_addr, 0.1, data));
}

private void removeDHTNodes(int num) {
    int n = 0;
    for(int i=0; i<num; i++) {
        float m = (float) Math.random() * dht_network_size;
        n = Math.round(m);
        if(n == dht_network_size)
            n = dht_network_size - 1;
        Node r_node = (Node) node[n];
        sim.detach(r_node);
        dht_network_size--;
    }
}

private void removeCDHTNodes(int num) {
    int n = 0;
    for(int i=0; i<num; i++) {
        float m = (float) Math.random() * cdht_network_size;
        n = Math.round(m);
        if(n == cdht_network_size)
            n = cdht_network_size - 1;
        Node r_node = (Node) cnode[n];
        sim.detach(r_node);
        cdht_network_size--;
    }
}

```

```

    }
}

private static String getNodeRemoved(IPAddr ipAddr) {
    String removedNode = null;
    boolean removed = false;
    Vector v = sim.getRemovedElements();
    Enumeration e = v.elements();
    while(e.hasMoreElements())
    {
        Node curnode = (Node) e.nextElement();
        removed = curnode.hasIPAddress(ipAddr);
        if(removed)
            return curnode.getName();
    }
    return removedNode;
}
}
}

```

The following is the example of code based on JNS to test load balancing in DHT and CDHT.

```

/**
 * Test_LB.java
 *
 * Copyright (c) 2007. First created in July 2004
 *
 * @author Wei Ye
 *
 */
package jns;

import jns.command.Command;
import jns.command.StopCommand;
import jns.element.*;
import jns.trace.JavisTrace;
import jns.trace.Trace;
import jns.util.Debug;
import jns.util.IPAddr;
import jns.util.Protocols;
import jns.util.Preferences;
import jns.util.NodeIDManager;
import jns.dynamic.PacketSender;

import java.io.IOException;
import java.io.File;

```

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.StringTokenizer;
import java.util.Vector;

/**
 * This class tests Load Balancing with DHT and HCL lookup patterns respectively.
 */
public class Test_LB
{
    private static Node[] node;

    private static Node[] cnode;

    private static int dht_network_size;

    private static int cdht_network_size;

    private static Simulator sim;

    private static FileOutputStream fos;

    private static Debug debug;

    public static String lookup_pattern;

    public static Test_LB testLB;

    public Test_LB () {
        node = new Node[Preferences.Node_Number/4];
        cnode = new Node[Preferences.Node_Number*3/4];
        sim = Simulator.getInstance();
        dht_network_size = Preferences.Node_Number/4;
        cdht_network_size = Preferences.Node_Number*3/4;
    }

    public static Test_LB getInstance() {
        if(testLB == null)
            testLB = new Test_LB();
        return testLB;
    }

    public static void main(String args[])
    {

```

```

Test_LB testLB = getInstance();
if(args.length == 2)
    debug = new Debug("Test_LB.log");
else if(args.length == 3)
    debug = new Debug("Test_LB-" + args[2] + ".log");

//Create the nodes for the DHT layer, and attach with tracing to the simulator
NodeIDManager idManager = NodeIDManager.getInstance();
idManager.setIDSet();
idManager.differentiateContent();
idManager.setIDSet();
Hashtable cp_cache_table = importContentPointerCache();
for(int i=0; i<Preferences.Node_Number/4; i++) {
    String id = idManager.retrieveSequenceID(i);
    node[i] = new Node(id);
    if(cp_cache_table.size() == 0)
        node[i].fillContentPointerCache();
    else {
        Vector cp_cache = (Vector) cp_cache_table.get(id);
        node[i].setContentPointerCache(cp_cache);
    }
    sim.attach(node[i]);
}

String id = "";
for(int i=0; i<Preferences.Node_Number/4; i++) {
    id = idManager.retrieveSequenceID(i);
    //Get routing entries
    Vector routingTable = (Vector) idManager.getRoutingTable(id);
    Enumeration e = routingTable.elements();
    String[] entryName = new String[Preferences.Routing_Entry_Number];
    Node[] routingEntry = new Node[Preferences.Routing_Entry_Number];
    Interface[] localface = new Interface[Preferences.Routing_Entry_Number];
    Interface[] entryface = new Interface[Preferences.Routing_Entry_Number];
    int j = 0;
    while(e.hasMoreElements()) {
        entryName[j] = (String) e.nextElement();
        Vector existingElements = (Vector) sim.getExistingElements();
        Enumeration s = existingElements.elements();
        while(s.hasMoreElements()) {
            Node curnode = (Node) s.nextElement();
            String name = curnode.getName();
            if(name.equals(entryName[j])) {
                routingEntry[j] = curnode;
                int[] entrynum = (int[]) idManager.getIPAddressNumber(name);
                int[] localnum = (int[]) idManager.getIPAddressNumber(id);
            }
        }
        j++;
    }
}

```

```

        entryface[j] = new DuplexInterface(new IPAddr(etrynum[0], etrynum[1],
etrynum[2], etrynum[3]));
        localface[j] = new DuplexInterface(new IPAddr(localnum[0], localnum[1],
localnum[2], localnum[3]));
        node[j].attach(localface[j]);
        sim.attach(localface[j]);
        curnode.attach(entryface[j]);
        sim.attach(entryface[j]);

        Link a = new DuplexLink(500000, 0.008);
        entryface[j].attach(a, true);
        localface[j].attach(a, true);
        sim.attach(a);

        node[j].addRoute(new IPAddr(etrynum[0], etrynum[1], etrynum[2],
etrynum[3]), new IPAddr(255, 255, 255, 0), localface[j]);
        curnode.addRoute(new IPAddr(localnum[0], localnum[1], localnum[2],
localnum[3]), new IPAddr(255, 255, 255, 0), entryface[j]);

        break;
    }
}
j++;
}
//Get leaf entries
Vector leafSet = (Vector) idManager.getLeafSet(id);
Enumeration l = leafSet.elements();
String[] leafName = new String[Preferences.Leaf_Entry_Number];
Node[] leafEntry = new Node[Preferences.Leaf_Entry_Number];
Interface[] localface1 = new Interface[Preferences.Leaf_Entry_Number];
Interface[] leafface = new Interface[Preferences.Leaf_Entry_Number];
int k = 0;
while(l.hasMoreElements()) {
    leafName[k] = (String) l.nextElement();
    leafEntry[k] = new Node(leafName[k]);

    Vector existingElements = (Vector) sim.getExistingElements();
    Enumeration s = existingElements.elements();
    while(s.hasMoreElements()) {
        Node curnode = (Node) s.nextElement();
        if(name.equals(leafName[k])) {
            leafEntry[k] = curnode;
            int[] leafnum = (int[]) idManager.getIPAddressNumber(name);
            int[] localnum = (int[]) idManager.getIPAddressNumber(id);
            leafface[k] = new DuplexInterface(new IPAddr(leafnum[0], leafnum[1], leafnum[2],
leafnum[3]));

```

```

        localface1[k] = new DuplexInterface(new IPAddr(localnum[0], localnum[1],
localnum[2], localnum[3]));
        node[i].attach(localface1[k]);
        sim.attach(localface1[k]);
        currnode.attach(leafface[k]);
        sim.attach(leafface[k]);

        Link a = new DuplexLink(500000, 0.008);
        leafface[k].attach(a, true);
        localface1[k].attach(a, true);
        sim.attach(a);
        node[i].addRoute(new IPAddr(leafnum[0], leafnum[1], leafnum[2], leafnum[3]), new
IPAddr(255, 255, 255, 0), localface1[k]);
        break;
    }
}
k++;
}
}

```

```

int c = Preferences.Node_Number/4;
for(int i=0; i<c; i++) {
    Vector existingElements = (Vector) sim.getExistingElements();
    Node dht_node = (Node) existingElements.get(i);
    String id1 = idManager.retrieveSequenceID(i+c);
    String id2 = idManager.retrieveSequenceID(i+2*c);
    String id3 = idManager.retrieveSequenceID(i+3*c);
    cnode[i] = new Node(id1);
    cnode[i+c] = new Node(id2);
    cnode[i+2*c] = new Node(id3);
    sim.attach(cnode[i]);
    sim.attach(cnode[i+c]);
    sim.attach(cnode[i+2*c]);

    IPAddr[] cipAddr = new IPAddr[3];
    int starting_subnet = Preferences.Starting_Subnet;
    for(int sub_length=128; sub_length<=c*3; sub_length=sub_length+128) {
        int cur_subnet = starting_subnet + sub_length/128;
        int cur_ip = 0;
        if(i >= 128 && i < 256)
            cur_ip = i - 128;
        else
            cur_ip = i;
        if(i>=sub_length - 128 && i<sub_length) {
            cipAddr[0] = new IPAddr(192, 168, cur_subnet, cur_ip);
        }
        if(i+c>=sub_length - 128 && i+c<sub_length) {

```

```

        cipAddr[1] = new IPAddr(192,168,cur_subnet, cur_ip);
    }
    if(i+2*c>=sub_length - 128 && i+2*c<sub_length) {
        cipAddr[2] = new IPAddr(192,168,cur_subnet, cur_ip);
    }
}

int[] ipnum = idManager.getIPAddressNumber(id);
Interface[] dht_iface = new Interface[3];
Interface[] cdht_iface = new Interface[3];
for(int k=0; k<3; k++) {
    cdht_iface[k] = new DuplexInterface(cipAddr[k]);
    dht_iface[k] = new DuplexInterface(new IPAddr(ipnum[0], ipnum[1], ipnum[2],
ipnum[3]));
    dht_node.attach(dht_iface[k]);
}
cnode[i].attach(cdht_iface[0]);
cnode[i+c].attach(cdht_iface[1]);
for(int k=0; k<3; k++) {
    sim.attach(cdht_iface[k]);
    sim.attach(dht_iface[k]);
}

for(int k=0; k<3; k++) {
    Link a = new DuplexLink(500000, 0.008);
    cdht_iface[k].attach(a, true);
    dht_iface[k].attach(a, true);
    sim.attach(a);
    dht_node.addRoute(cipAddr[k], new IPAddr(255, 255, 255, 0), dht_iface[k]);
}
cnode[i].addDefaultRoute(cdht_iface[0]);
cnode[i+c].addDefaultRoute(cdht_iface[1]);
cnode[i+2*c].addDefaultRoute(cdht_iface[2]);
}

try {
    int source_num = (int) Integer.parseInt(args[0]);
    int des_num = (int) Integer.parseInt(args[1]);
    if(args.length == 3) {
        if(args[2].equals(Preferences.DHT_Lookup_Pattern)) {
            lookup_pattern = Preferences.DHT_Lookup_Pattern;
            testLB.DHTlookup(source_num, des_num);
        }
        else if(args[2].equals(Preferences.HCL_Lookup_Pattern)) {
            lookup_pattern = Preferences.HCL_Lookup_Pattern;
            testLB.HCLlookup(source_num, des_num);
        }
    }
}

```

```

    }
} catch(NumberFormatException nfe) {
    nfe.printStackTrace();
}
}
sim.schedule(new StopCommand(1));
}

public static Hashtable importContentPointerCache() {
    Hashtable cp_cache_table = new Hashtable();
    importCache:
    try{
        File file = new File(Preferences.CP_Cache_File);
        if(!file.exists())
            break importCache;
        FileInputStream fis = new FileInputStream(file);
        byte[] filecontent = new byte[(int) file.length()];
        fis.read(filecontent);
        fis.close();

        String cache_content = new String(filecontent);
        cache_content.trim();
        StringTokenizer st1 = new StringTokenizer(cache_content, "@");
        while(st1.hasMoreTokens()) {
            String cache = st1.nextToken();
            if(cache.length() == 1)
                continue;
            StringTokenizer st2 = new StringTokenizer(cache,
Preferences.Separator);
            while(st2.hasMoreTokens()) {
                String node_id = st2.nextToken();
                String node_cache = st2.nextToken();
                Vector cp_cache = new Vector();
                StringTokenizer st3 = new StringTokenizer(node_cache, ";");
                while(st3.hasMoreElements()) {
                    String cp_string = st3.nextToken();
                    ContentPointer cp = new ContentPointer();
                    StringTokenizer st4 = new StringTokenizer(cp_string,
";");

                    while(st4.hasMoreTokens()) {
                        String cp_id = st4.nextToken();
                        cp.setContentID(cp_id);
                    }
                    cp_cache.add(cp);
                }
                cp_cache_table.put(node_id, cp_cache);
            }
        }
    }
}
}

```

```

    } catch(IOException ioe) {
        ioe.printStackTrace();
    }
    return cp_cache_table;
}

public static void exportContentPointerCache() {
    try {
        FileOutputStream fos = new FileOutputStream(Preferences.CP_Cache_File, false);
        String msg = "";
        for(int i=0; i<node.length; i++) {
            msg = node[i].getName() + Preferences.Separator;
            Vector cp_cache = node[i].getContentPointerCache();
            Enumeration e = cp_cache.elements();
            while(e.hasMoreElements()) {
                ContentPointer cp = (ContentPointer) e.nextElement();
                msg = msg + cp.getContentID() + "," + cp.getAccessRate() + ",";
            }
            msg = msg + "@";
            fos.write(msg.getBytes());
            fos.write((new String("\n")).getBytes());
        }
        fos.close();
    } catch(IOException ioe) {
        ioe.printStackTrace();
    }
}

private void HCLlookup(int source_num, int des_num) {}

private void DHTlookup(int source_num, int des_num) {}

private void removeDHTNodes(int num) {}

private void removeLBNodes(int num) {}

private static String getNodeRemoved(IPAddr ipAddr) {}

public Node[] getDHTNodes() {}

public Node getDHTNode(String nodeId) {}

```