The Edward S. Rogers Sr. Dept of Electrical and Computer Engineering
University of Toronto

ECE496Y Design Project Course – Final Report

Title:
Scalable Behavioral Simulation of Application-layer Peer-to-Peer Networking (Team 1)

Project I.D. # 0992002

Prepared by:	Sasitheran Shanmugarajah	shanmus@ecf.utoronto.ca
	Steven Wolfman	wolfma@ecf.utoronto.ca
	Rehan Siddiqui	siddiqr@ecf.utoronto.ca
Supervisor:	Prof. Baochun Li	
Section #:	4	
Section Coordinator:	Phil Anderson	
Date:	April 11, 2003	

EXECUTIVE SUMMARY

The purpose of this project was to build a software implementation of Freenet to simulate the behaviour of a scalable application-layer peer-to-peer network. The project consisted of implementing the Freenet functionality into an existing network simulator. All milestones and objectives were completed successfully.

Freenet is a type of decentralized peer-to-peer network which allows the distribution of uncensored information to its users. Anonymity of users is preserved by using Freenet's data and message transferring protocols which does not allow data to be traced back to its originators. Freenet also uses routing algorithms to dynamically relocate data over the network from areas where the data is in low demand to areas where it is in high demand. These features can revolutionise the way the Internet is used and could lead to more efficient and faster networks.

The project was implemented in C++ for the Linux operating system. The following features of Freenet were added to the existing simulator: message protocols, message and data routing algorithms, keys and searching algorithms, and management of data.

Analysis of our test simulations conducted with 10, 100, 500 and 1000 node topologies prove that the implementation accurately follows the Freenet protocol. Time analysis of the results show that large and complex topologies can be simulated within a seconds. These test results support our claim that this implementation of Freenet on the skeleton is correct, fast and scalable.

Contributions

Below is an outline of the contributions made by the authors Rehan Siddiqui, Sasitheran Shanmugarajah and Steven Wolfman towards the design project and final report.

Contribution made by Rehan Siddiqui to the design project are outlined in the table below:

Task	Individual(s) Responsible for the Task
Research into Freenet	Rehan Siddiqui (with Sasitheran Shanmugarajah and Steven Wolfman)
Research and Analysis of Network Skeleton Code	Rehan Siddiqui (with Sasitheran Shanmugarajah and Steven Wolfman)
Design Specifications	Rehan Siddiqui (with Sasitheran Shanmugarajah and Steven Wolfman)
Coding (Datastore and Routing Table)	Rehan Siddiqui
Testing Components - Data store and Routing Table	Rehan Siddiqui
Component Integration	Rehan Siddiqui (with Sasitheran Shanmugarajah and Steven Wolfman)
Integration Testing - Datastore and Routing Table	Rehan Siddiqui

Contribution made by Sasitheran Shanmugarajah to the design project are indicated below:

Task	Individual(s) Responsible for the Task
Research into Freenet	Sasitheran Shanmugarajah (with Rehan Siddiqui and Steven Wolfman)
Research and Analysis of Network Skeleton Code	Sasitheran Shanmugarajah (with Rehan Siddiqui and Steven Wolfman)
Design Specifications	Sasitheran Shanmugarajah (with Rehan Siddiqui and Steven Wolfman)
Coding (Message Handling Functions)	Sasitheran Shanmugarajah
Testing Components - Message Handling Functions	Sasitheran Shanmugarajah
Component Integration	Sasitheran Shanmugarajah (with Rehan Siddiqui and Steven Wolfman)
Integration Testing - Running Simulations	Sasitheran Shanmugarajah
Driver Program	Sasitheran Shanmugarajah

Contributions made by Steven Wolfman to the design project are indicated below:

Task	Individual(s) Responsible for the Task
Research into Freenet	Steven Wolfman (with Rehan Siddiqui and Sasitheran Shanmugarajah)
Research and Analysis of Network Skeleton Code	Steven Wolfman (with Rehan Siddiqui and Sasitheran Shanmugarajah)
Design Specifications	Steven Wolfman (with Rehan Siddiqui and Sasitheran Shanmugarajah)
Coding (Message Class and Processing)	Steven Wolfman
Testing Components - Message Class and Processing	Steven Wolfman
Component Integration	Steven Wolfman (with Rehan Siddiqui and Sasitheran Shanmugarajah)
Integration Testing - Running simulations	Steven Wolfman
Driver Program	Steven Wolfman

Contributions to the Final Report:

Section	Individual(s) Responsible for the Section
Cover Page	Rehan Siddiqui
Executive Summary	Rehan Siddiqui
Team Members Contribution	Rehan Siddiqui
Old Milestones	Sasitheran Shanmugarajah
Revised Milestones	Sasitheran Shanmugarajah
Table of Contents	Rehan Siddiqui
Acknowledgements	Steven Wolfman
Introduction	Rehan Siddiqui
Design	Steven Wolfman Sasitheran Shanmugarajah
Conclusion	Steven Wolfman Sasitheran Shanmugarajah
Appendices	Steven Wolfman Sasitheran Shanmugarajah
Formatting	Rehan Siddiqui Sasitheran Shanmugarajah
Editing	Steven Wolfman Sasitheran Shanmugarajah Rehan Siddiqui

TABLE OF CONTENTS

1. Acknowledgments

We would like to take this opportunity to thank our supervisor, Professor Li, for all his motivation and support throughout the project, for arranging the tutorial with the author of the skeleton code, Jiang Guo, and for giving us access to sim.ece to use for running our simulations.

We would also like to thank Jiang Guo for permitting us to use his code for the network skeleton, and for giving a tutorial on the skeleton code to help us use it.

2. Introduction

2.1 Background Information

Peer-to-Peer networking is a style of networking that allows computers to communicate directly with one another and thus allows the sharing of resources between these computers. In such a network, each computer acts as a client, someone who makes a request for a particular resource such as information, and as a server, someone who responds to requests for these resources. It is through this client/server relationship that large-scale peer-to-peer (P2P) networks such as Napster and Gnutella are able to function as massive virtual information storage and retrieval entities. Freenet is another such P2P network, and will be the focus of this project.

2.1.1 Freenet and its Unique Characteristics

Freenet allows for the distribution of information over a number of nodes (a node is simply a computer connected to the network), each of which must be running a software capable of communicating over a Freenet network [1]. The use of software here implies that the Freenet is an application-layer network. That is, Freenet is not cognizant of the underlying TCP/IP layer network that computers use today to connect to the World Wide Web for example. The implications of this are that nodes that may be connected within the Freenet may or may not be physically connected to each other.

A node can search for specific information over the network by sending a 'query request' message. This message would be propagated until the query is satisfied by another node possessing the information being sought. The answering node would then send the data requested to the query originator by routing it back through the network.

Freenet is different from other P2P networks in that it values anonymity highly [2]. Anonymity will ensure that information cannot be censored or denied, as no one will know who the originator of the information is. As such, the Freenet topology is designed to protect the privacy and security of each node in the network by ensuring that a receiving node will not be able to trace, with certainty, which other node is sending the data. This feature makes a Freenet network decentralized, that is, no one node can control or influence the network. The Freenet model also uses a routing algorithm to dynamically relocate data over the network from areas where the data is in low demand to areas where it is in high demand [2]. This is achieved by temporarily storing data on the nodes that lie on the routing path between the query originator and the replier.

2.2 Motivation/Rationale

The unique features of a Freenet network allow anonymity and an adaptive automatic distribution of data, making it an extremely promising and exciting technology that could revolutionize the way the Internet is currently being used. Since it is a very new technology that is in its early stages of development, Freenet's behaviour in large-scale networks is still undergoing much study and analysis. There are many questions still

unanswered, such as how the network would behave at a very large scale, for example using hundreds of thousands of nodes, and how the data would re-distribute itself on this structure.

In a real-life network many external factors can affect the performance of a Freenet topology such as the distance between nodes, the hops-to-live count of a message (i.e. the number of nodes a message can propagate through the network), and the connection speed and stability of each node. Controlling these factors in the real-world makes it difficult to conduct any useful or reliable studies. Therefore, it would be invaluable to be able to simulate Freenet over a network where all external variables can be controlled. This would allow the behaviour and topology to be studied and analyzed as well as improved to create a more efficient network design.

This project allows the achievement of this goal by implementing the Freenet topology on to the network simulator. The completed project is able to make head-to-head comparisons between Freenet and other P2P networks to see how each design copes with stress factors such as the number of nodes, or the flow of information for example. We can use these comparisons justify the feasibility of designs in terms of the efficiency of the routing protocols.

2.3 Project Objectives

As mentioned previously, Freenet allows for the transfer of information in a uniquely fashion that maximizes efficiency in time as well as security in the form of anonymity. This design project will cast the Freenet topology onto a network simulator and implement the routing algorithms specific to that topology. We will focus on the efficiency and scalability of the Freenet. However, we will not take into account Freenet's other aspects such as data security via encryption.

The project can be broken up into several objectives that should be met for the project to be successfully completed:

- The network simulation will be able to simulate a large-scale Freenet network (several thousand nodes).
- The simulated Freenet nodes will adhere to the messaging protocols of the Freenet network when communicating with each other.
- Each simulated Freenet node will route messages and data using the Freenet routing algorithm.
- When data is distributed over the Freenet network simulation, it will propagate adaptively over the nodes such that it is located in areas where it is in demand the most.
- The data stored on a simulated Freenet node will be managed according to the Freenet guidelines for managing data.

2.4 Literature Survey

There is an abundance of information relating to the Freenet that is readily available on the Internet. Much of this information comes from the Freenet website: http://freenetproject.org. These articles focused on the Freenet; its uses, advantages, and methods of implementation. To get an idea of Peer-to-Peer networking in general, we consulted [1].

2.4.1 A distributed Decentralized Information Storage and Retrieval System

Our initial inquiry into the Freenet led us to the paper written by the creator of Freenet, Ian Clarke, entitled "A Distributed Decentralized Information Storage and Retrieval System". This is a paper written by Clarke during his undergraduate studies at Edinborough, Scotland in 1999.

The paper describes the Freenet search algorithms as analogous to navigation in prehistoric society, where there existed no central government, no maps. Yet, Clarke argues, that people were still able to locate their destinations by taking advice from those they encountered along the journey. In other words, people would have more information about locations that were close to them, and a vague description of places far off. The Freenet is such a network that adapts to the requests of the users or nodes that are connected to the system. When a node makes a request to one of its neighbours, and if the neighbour does not have the information, the request is forwarded to another node

that is most likely to have it. This is repeated until the information is found. Once that happens, the information is relayed back the same path and each node in the path stores that information locally. Nodes are also more likely to receive requests that are similar to the information they store. Therefore nodes that are close to each other will ultimately store similar information since the same type of information keeps moving closer and closer to nodes that carry it. Minor subtleties such as what makes information A closer to B than C, for example, are also explained in the paper.

The architecture of the Freenet is also given in the paper. Essentially, there are four different message types that nodes use to communicate with each other: Data Request, Data Reply, Request Failed and Data Insert. The protocol is as follows: node A will send node B a "Data Request" message with a preset "hops to live" value in the message header. This value is basically the number of times the user requests their message to be forwarded. If node B does not have the information requested, it will forward the "Data Request" message to node C. This forwarded message will have its "hops-to-live" value decremented by one. This will repeat until the information is found or the "hops to live" field in the message header is 0. If the information is found, the, "Data Reply" message is sent back the same path to the originator of the "Data Request". If not, then a "Request Failed" message is sent back to the originator. The "Data Insert" message is used to add data to the network. When a node receives such a message, it locally stores the data and forwards the message to a node that will most likely have similar data. This is repeated until the "hops-to-live" counter is zero. We will be using this very same communication system to implement Freenet.

Finally, simulation setup, implementation and experimentation, and results are also discussed at length in the article. The simulated network and implementation of Freenet was coded in Java. One experiment of importance is the "Information Retrieval Time". The aim of this experiment was to determine how many requests are required to obtain information after the network adapts to a number of queries. The results of this experiment show that after about 800 queries, the network has adapted and stabilized so that to retrieve information thereafter, it requires about 10 requests on average. Moreover, this is regardless of the number of nodes, as experiments were conducted on 500, 600, 700, 800, and 900 node networks, all of which showed roughly the same result. This information will be useful to us in our own studies of the Freenet. In other words, since one of our goals is to achieve an optimized network we can use the simulation information in this article as a benchmark for our own simulation results.

2.4.2 The Freenet Protocol

http://www.firenze.linux.it/~marcoc/index.php?page=protocol

This online article describes in detail the specific protocol used by the current version of the Freenet. The way documents are organized is essentially a stack structure and to find documents rather quickly, a hash table is proposed.

We will implement the message format similar to what is detailed in this article. Message types are formatted such that they are broken into UTF-8 encoded lines. Instead of UTF-8 however, we will be using ASCII encoded messages. The first line is the message type. Then come a number of key-value headers with the key and value separated by a single '=' character. A typical example of such a header is given:

```
      DataReply
UniqueID=C24300FB7BEA06E3
Depth=a
HopsToLive=2c
Source=tcp/127.0.0.1:2386
DataLength=131
Data
 'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves
And the mome raths outgrabe.
```

Some common headers within message types and their description are also given, for example, UniqueID, HopsToLive, Depth, KeepAlive and Source. These headers are important for us during the implementation of message broadcasting within the network.

Also explained is the Cryptographic Link Layer. Although this is an integral part of the Freenet framework, it has been discarded from our design. The reason for this is that our intended purpose here is not to implement the dominant aspect of Freenet, namely anonymity, but rather to implement its adaptive nature.

2.4.3 Lecture at Stanford, by Ian Clarke, creator of Freenet

http://murl.microsoft.com/LectureDetails.asp?765

Ian Clarke here describes his motivation for Freenet. That is, for free speech to really be free and uncensored, anonymity is vital in order to avoid backlash or punishment. The system itself is described to be decentralized and scalable. Scalability was measured with simulations in fact. It was interesting to note that Clarke simulated up to 30,000 nodes, and on average, each request for information went through only 2.7 nodes to get a hit. This information will be helpful in our own analysis of simulating the Freenet within the skeleton network. Also, one fundamental requirement for Freenet to work is that people must request information. Without requests, simulations showed that the Freenet would drop information from the network. The more popular the information therefore, the more readily it will be available.

The way information is stored locally is also explained in the lecture. Clarke describes the data store essentially as a stack structure. Clarke draws the stack structure for his audience similar to figure 5.1.1. Each entry has either two or three fields. The first field is the key to which documents are matched. The second field is the address

from where the information came. The third field is the actual information or data. As can be seen from the diagram, below item 5, the information field is empty. This is because each node has a limit on the amount of information it can store. The question then arises as to which information each node should store. And the answer to this is the most requested information. The way it is implemented is that data that is requested more often is moved up in the stack and the least requested data is at the bottom. Therefore when a new piece of information is received, it is placed at the top of the stack and the entry at the bottom is flushed out. This is the implementation that we will be using for our own design.

Item#	Key	Location	Information
1	62548	128.100.1.34	Fsdfseafesafefse32rqj9p23ijr2
2	62589	142.168.0.2	23krj23890rn433kjn23j54nn4k3n
3	23456	24.23.15.16	23klrj3lk2n3kln2l3kj203jiklkn59
4	62454	64.34.255.3	9cxu7908gdrus9gp8jwer8fg9ewj
5	1002	66.92.3.3	9cn4n8n423843n2lkj38svdklpsjje
6	965	102.52.101.9	
7	687	201.59.64.65	
8	601	24.42.15.171	
9	201	24.153.23.66	
10	12	66.129.32.128	

Figure 5.1.1 – Sample datastore

Issues relating to the quality of service and efficiency of the Freenet were also raised. With respect to efficiency, certain information is stored locally at each node and it tends to move closer to where there is more demand for that information. Therefore,

there exists a trade-off between hard-disk space and bandwidth efficiency; an essentially good trade-off since hard disk space is cheaper than bandwidth. The quality of service of the Freenet is better than the World Wide Web. Denial-of-Service attacks are pointless since there is no central server and no individual node is essential for the system to work.

2.5 Report Outline

This report will cover the main aspects of our project after the initial research into Freenet. That is the design and implementation of the Freenet as well as the testing of the design.

The design can be divided into two main areas:

1. The Architecture of the Skeleton Network, and
2. The architecture of the Freenet

These two sections will cover the important aspects of each area as they pertain to our project design. Next, the report will examine the implementation of the above-mentioned design, and how the above two areas of design were integrated. Finally, we will present the test process and their results after the implementation of our design as well as our conclusions to this design project.

2.6 Milestone Evaluation

The milestones as drawn up in the project proposal (refer Appendix C) were revised mid-semester (refer to Appendix D). We completed the research into Freenet and the Network

Skeleton code as set on the original scheduled. The milestones we selected reflected our design approach. Once the initial research into the Freenet was complete, the software development cycle was split into two phases. The first phase focused on the specification, that is, how to extend the skeleton to Freenet. The second phase focused on implementing the specification and coding. The design project was split into 3 distinct parts so that members of the group could work on the project simultaneously without having to depend on each other's work.

We reached all our revised milestones on schedule as expected. This was mainly due to the fact the revised milestones better met our individual schedules compared to the old milestone list. Coding was finished by late March, and this report is our final milestone, completed on April 11th as expected.

3. Architecture of the Network Skeleton

The network skeleton we used to implement our simulator is essentially a program that contains the building blocks of a network to be used to construct any network topology with an arbitrary number of nodes. It was written by Jiang Guo and given to us by Professor Li to use for the implementation of our simulator.

In order to construct a network, the skeleton uses six main classes as network components. The first of these components is the basic element of any network, a node. Each node is assigned a unique id to identify it, as well as given a datastore and a routing table. In addition to these features, each node also has a message pool, which it uses to track the id of each message that is read by that node.

3.1 Datastore

The datastore is used to keep track of all the documents a node is holding. In order to do this, the datastore is implemented as a map, mapping the document id of each document stored at that node to a pointer to that document. The map stores this list in order sorted by document id so that it can quickly find any document requested. The datastore is assigned a default capacity of 20 documents, although this value can easily be changed.

3.2 Routing Table

Similar to the datastore, the skeleton's routing table is a map which maps a node id to a pointer to that node. Again, the values are ordered by node ids, and the routing table has a maximum capacity of 100 nodes.

A document is the main piece of "data" used by the skeleton. Although the name may sound confusing, the documents used with the skeleton are simply objects, not actual documents. Each document is assigned a single attribute, that being a unique identification number.

3.3 Messages

The most important object used by the skeleton is a message. Messages are used for all communication between nodes, and are the source of all activity in the network. In the skeleton, a message is implemented by assigning it a field for source and destination, as well as an id number. When a new message is created, only the id is assigned by the skeleton, and the source and destination are filled in by the sender just before a message is sent.

3.4 Network

Finally, the network class brings all the components together. First, the network is build by reading in all the nodes and documents, and assigning them as specified in the appropriate topology files. Once the network has been constructed, the run function is called. The run function starts running a simulation for a given number of rounds. In each round, the network starts by randomly generating zero or more events for each node. This is done by calling the generate_events() function in each node, which needs to be implemented depending on the type of events you want your simulation to have, but generally involves generating some messages and placing them in the inbox. The inbox is a queue in which all generated messages are placed. Next, the network processes all messages in the message inbox. The network processes the messages by taking one at a time out of the inbox and sending it to the proper node as indicated in its destination address. This is continued until all messages which started the round in the inbox have been processed, and then the simulation moves on to the next round. This process is continued for the specified number of rounds.

4. Architecture of Freenet

The Freenet is comprised of many components needing to be implemented in order to obtain an accurate simulation of its properties. This section describes these components of the Freenet that we have implemented, their interactions with each other, and some of the design choices made to help us implement them.

4.1 Keys

Keys are the most important part of the Freenet. Every piece of data on the network is described by a unique key. The Freenet assigns binary file keys to each file. These keys are obtained by applying a hash function to a string pertaining to that file. One of our more significant design choices was to not use this same method of obtaining keys, but to preserve the method used by the network skeleton of representing documents with short integers. In addition to keeping consistent with the skeleton, this choice made the routing functions much simpler to implement, while still following the Freenet's routing algorithm.

4.2 Datastore

A node's datastore is a table which stores a collection of documents at that node in a least recently used order (LRU). Whenever a new document is inserted at that node, or a request for a document passes through the node, that document is moved to the top of

the datastore. This process allows for documents which are unused or unwanted for long periods of time to fade away. To clarify, the datastore can be thought of a table with 3 columns. The first is the document key, the second is the original source of the data, and the third is the location of the actual data.

4.3 Routing Table

A routing table is another property of a node, and in the case of the Freenet, is very similar to the datastore. The one difference between the routing table and the datastore is that the routing table holds the original location of all data the node knows about, not just the data it is currently holding. That is, the routing table holds the key and original source of all the documents currently in the node's datastore as well as all previous documents which have been removed from the datastore due to lack of space, but still exist in the routing table.

4.4 Messages

Messages are the core of the Freenet. All activity that takes place on the network and all interaction between nodes is the result of messages being passed around. Our simulator preserves this use of messages by using a message system almost identical to that used by the Freenet. One small change was the omission of handshaking, as the simulator controls all nodes, it can guarantee that any nodes are connected according to the topology, and a handshake to establish a connection is not needed.

4.5 Message Types

Our simulator uses seven of the message types used in the Freenet. As mentioned above, we chose not to implement the Freenet's Request.Handshake and Reply.Handshake as they are not necessary for simulation of the protocol.

The first message type is REQUEST_INSERT. This message is used when a user wishes to upload a new piece of data to his or her node. The user generates a key for that data and sends a REQUEST_INSERT message to their node containing that key. The node then propagates this message through the network until the hops to live expire to verify that no known documents already exist with that key. Any node receiving a REQUEST_INSERT first checks its routing table and datastore to see if it knows of any data with the same key. If a key collision is found with a document in the datastore, the receiving node sends a SEND_DATA message back to the original node containing the data it had with that key. If no data that key exists in the datastore but the key is present in the routing table, a NOT_FOUND message is sent back to the original node, indicating that the selected key is already used. Finally, if no collision is detected, there are two options. If the hops to live of the original message have expired, a REPLY_INSERT message is send back, indicating that it is ok to use the selected key. If, however, the hops to live have not run out, the request is forwarded in the same manner as if it had been a search request.

If at any point in the above process a message reaches a dead end before the hops to live expire, the message backtracks to the first point where is can choose another direction and goes in the new direction. When this happens, a REQUEST_CONTINUE message is generated and sent back to the source of the original message letting it know that its message needed to backtrack, and it should allow for more time before receiving a response. Each node that generates a message requiring a response keeps a timer for how long to wait for the response to come back, so it must be notified to extend this timer if the message needs to extend its route. The timer is used in case messages get lost in the network or can't be delivered for any reason. Since each node tracks the messages it sends and the responses it is waiting for, this list can get big if lost messages are kept in the list forever. With the timer, a node forgets about a message and drops it from its list when the timer expires. If the message arrives after the timer expires, the node simply ignores it, causing the message to be killed.

After a node sends a REQUEST_INSERT, it waits for a REPLY_INSERT letting it know it is ok to insert the data. When it receives the reply, it generates a SEND_INSERT message containing the data, and forwards it upstream until it reaches the node that first generated the REPLY_INSERT. Now the data is on the network, and several other nodes near the original node also have the same data. This is important as no one can pinpoint the exact original source of the data.

The most important part of the Freenet is being able to search for data, and the algorithm used to find this data. When a user wishes to look for a document on the

Freenet, they sent a REQUEST_DATA message with the key of the document they are looking for. A node receiving a REQUEST_DATA message first checks its datastore for a document with the desired key. If it exists, the document is sent back towards the original requestor in a SEND_DATA message. If the key is found in the routing table, the node simply forwards the request message to the node listed in the table. However, if the data is not found in either list, instead of asking all neighbours to look for the data, the Freenet compares the key to other keys in its routing table and finds the closest match. It then forwards the request to the node where the closest match was found. The search is continued in this way until the key is found or the hops to live expire.

4.6 Routing Algorithm

The routing algorithm used by the Freenet and described in [2] is one if it's most important features. As opposed to most other peer-to-peer networks which search for data by broadcasting messages to all neighbours, the Freenet uses a unique method of trying to guess the single "best" neighbour who might have the data. This is done by comparing the key of the requested document with the keys in your routing table, and selecting the node with the closest match as the best node. If that node has already been visited, the next best match is selected. This will improve the efficiency of the network in the long run for two reasons as proven in [3]. First, nodes will tend to acquire groups of files with similar keys. This allows files in those groups to be easily found. Second, the routing table at each node will improve in finding various sets of keys. This is because of the first property. If a few surrounding nodes know you have files with

similar keys, you will get lots of request for other similar keys. This will result in your routing table being tuned to properly direct requests for those keys, or even you receiving those files yourself. In combination, these two properties create a very efficient algorithm which becomes more and more efficient as it is used more and more nodes learn about each other [3].

5. Implementation of the Freenet

This section of the report describes how the various components of Freenet were implemented into the skeleton.

Our aim was to ensure that the Freenet implementation did not modify any existing functionality in the skeleton. This was accomplished by making sure the design made efficient use of existing code, and that addition of any functions and parameters was done as a last resort.

5.1 Data

In a real Freenet topology, physical files are transferred from one node to another [2]. To represent files and data the skeleton's Document class was used. Each Document object is identified by a unique integer key and can be dynamically created during a simulation. The simulated nodes may then request or insert a Document using its key.

One of the project objectives is to ensure the implementation is scalable as Freenet is in real life [3]. We took this into consideration when designing how the program would handle data. To keep the simulation as fast and as efficient as possible, only one copy of a unique Document object is created and stored. A pointer to that object is then used to pass the data from node to node. So in effect, one copy of a file (in this case a Document object) is held in the simulation and its distribution over the network is simulated using pointers. Admittedly, this is very different from a real life topology,

where copies of files are created and transferred throughout the network, but a compromise had to be made in this situation. In reality, each host computer needs only to handle its own events and store its own documents, but in a simulator the CPU power and storage is restricted, as it must simulated thousands of nodes and their data. To offset these restrictions, the design we chose keeps the message size small requiring far less calculation and storage which may have slowed the simulation down. Valuable CPU time is not used in creating copies of these objects. Also in later versions of the skeleton, since pointers to general data are being passed, the code need not be changed to accommodate different types of data, for example another type of Document class.

5.2 Messages

The old skeleton Message class was inadequate for our needs as it consisted of only a message id. To simulate the various attributes of a Freenet message, a new class - FreenetMessage was added, which extended the Message class of the Skeleton. Since the new class inherited the attributes of the Message class, it would be more compatible with the existing skeleton functions.

The new parameters held by a FreenetMessage object are described below:

- Hops-to-live counter. This is set when the message is created by a node, and is then decremented by one at every node it passes until it equals zero at which point the message is discarded

- Message type: This variable identifies the type of the message. Nodes use this value to determine how to handle and respond to the message
- Depth counter: This value is incremented each time the message passes through a node, allowing a node to know how many hops a message has undergone
- Key: This short integer value holds the unique id of a document in a search request message
- Data pointer: This pointer points to the actual location of data held by a message. It is of type void so that any type of data may be held

With these parameters, Freenet Messages may be virtually simulated.

5.3 Nodes

Each node must have the ability to handle, process, and respond to these new message types. The Node class was extended adding the fn_process_message, which would take a Message, check its type, and then call the correct function to handle and respond to it.

To make the code easier to maintain and re-use later on, each message type was handled by separate functions. This modular approach to the design of our functions was used throughout the coding process.

Each of the message handling functions takes in a message object, and returns a new message which either contains a response, or is empty. The message handling functions are described in table 5.3.1 on the following page:

Function Name	Function Description
handle_reply_insert	Checks whether this node is the source of the insert request this message is replying to. If it is, then the document is sent upstream as an insertion into the network. If not, it forwards the message downstream to the source of the insert request.
handle_request_data	Checks whether this node has the document being requested. If it does, it is returned to the requestor. If not, the message is forwarded to the next most likely node to have the document.
handle_send_data	Copies the document being sent into the cache, and forwards the message upstream.
handle_not_found	Checks whether the Not_Found is replying to one of the messages sent out by this node. If it is, it will take the appropriate action to cancel the last request. If not the message is forwarded downstream to its target.
handle_request_continue	Forwards the request for data in another direction within the network.
handle_request_insert	Checks whether the Id of the document being inserted collides with any existing documents in this node's cache. If it does not, the message is forwarded to the next node until its hops-to-live is zero. If there is a collision, the document that was collided with is returned to the source of the Request Insert.
handle_send_insert	Copies the document being inserted into the node's cache, and checks if the message hops-to-live is zero. If it is not, the message is forwarded upstream.

Table 5.3.1 – Message Handling Functions

A significant function added to the Node class was getBestNode. This function accepts a document id and returns the id of node from the routing table that most probably has that document. This was done using Freenet's idea of closest keys, where closeness is defined as the number numerically closest to the key.

Two C++ Standard Template Language maps are used to keep record of the ids of messages that were created as well as forwarded by a node. When a response to a previous message reaches a node, the list of created messages allow the node to verify whether it is the source of the original message and can then take the appropriate action. If it is not the source, the message must be forwarded downstream and so the list of forwarded messages allows a node to look up which node passed the message to it before, and can then forward the reply to it.

The diagram on the following page (figure 5.3.1) explains this more clearly. Node A maintains two tables, that it uses to forward a message with Id = 432 from Node B to Node C. It checks to see whether it created this message, and since there is no Message id 432 in the Message Created Table, it knows to forward the message downstream to Node C, who must have originally forwarded the message being to replied to Node A.

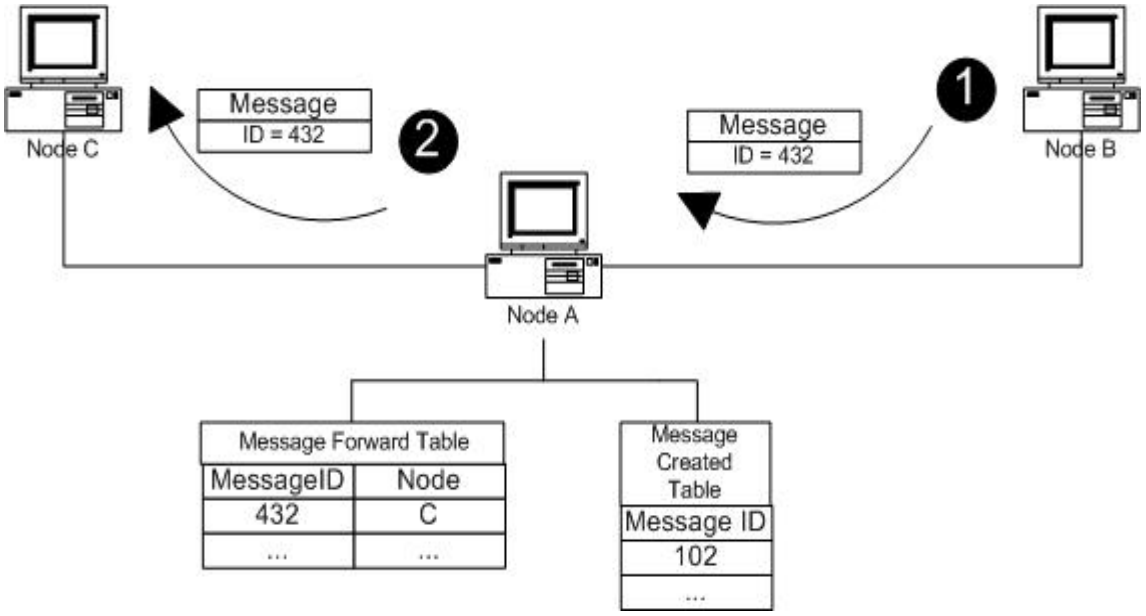

Figure 5.3.1 – Message Passing within the Freenet

5.4 Datastore

In a Freenet network, the files on a node and the IP addresses of other sources of the files are held in one table called a datastore. In the skeleton though, data is stored separately from other locations of the data, therefore using two separate tables. One table holds data hence called the skeleton datastore, and the other holds routing information hence called the routing table. To ensure that our implementation would be compatible with the previous functionality of the skeleton, we chose to implement the one Freenet Datastore using this two-table structure as well.

The routing table is used to hold the locations of documents not stored on that node. This information is held in the table by recording the id of the node that forwarded a Document. The implementation preserves the anonymity of the source of the data as it

is highly possible that the node that forwarded the document was not the original source of the document.

The Datastore class is used to simulate the storage medium on a node, and contains a list of Document ids being held on a node. A modified version of the Datastore class was used in our implementation for several reasons. A Freenet Datastore has to contain a list of data, sorted by most recently used. But the old skeleton Datastore uses a numerically ordered map to store data, which is inadequate to mimic the behaviour of a Freenet datastore. After much analysis, it was decided that rather than change the existing skeleton Datastore implementation, we would create a new Datastore class called Fn_Datastore. This would allow us to achieve the goal of not modifying any existing functionality but also allow us to implement the features we required as described in [2].

The new Datastore uses a C++ STL List object to store long integer values which identify the Documents held on a node. A node's Datastore is updated anytime a new Document is inserted or removed from the node, along with the routing table. This tight coupling of the two tables allows us to imitate the behavior of a Freenet datastore.

The algorithm used to update and replace documents in the Fn_Datastore and the routing table works as follows: the last requested document is placed on top of the list. The list has a user-defined limit, so only the first 20 Documents in the list are stored at any time. The routing table is used to keep a temporary record of other nodes where this document may be found. The diagram below (figure 5.4) shows this implementation:

Item#	Key	Location	Information
1	12548	128.100.1.34	Fsdfseafesafefse32rqj9p23ijr2
2	66589	142.168.0.2	23krj23890rn433kjn23j54nn4k3n
3	965	102.52.101.9	
4	687	201.59.64.65	

Freenet Datastore

KEY	NODE ID
12548	998891
66589	234333
965	883392
687	123422

FN_ROUTING TABLE

KEY	DATA POINTER
12548	0x122
66589	0x134

FN_DATASTORE

Figure 5.4.1 – Implementation of Datastore and Routing Table

In the above diagram, the functionality of the Freenet Datastore is carried out by two different tables as shown above. The arrows show where each column is mapped to in the new tables. In the simulator the IP addresses are represented by node Ids and a pointer to where the Document object is held is used instead of the actual information.

In the Network class a new function was added to generate random messages and insert them into the simulated network.

6. TESTING

The purpose of testing was to verify that our implementation of Freenet was correct. The correctness was tested using a driver program written exclusively for testing.

The tests were chosen and conducted to validate the following:

- Documents with unique keys can be inserted into the network
- Documents with keys that already existed are not allowed to be inserted into a network, provided that a collision occurred
- The messages are correctly routed toward the most likely nodes to have the requested document
- Requested Documents can be found and returned to the requesting node
- The correct replies are generated and forwarded for each type of message
- The performance of our simulated network was comparable to real life performance results

6.1 Running Simulations

A driver program was used to create random network topologies and events for the test simulations. The driver program accepts an integer value for the number of nodes in the network, and creates a random network topology and writes this topology into a file. This file can be read by the skeleton to build the topology for a simulation. The driver also creates random messages for every node in the generated network.

Two types of messages are created by the driver: the Request Insert message which allows a node to insert a Document into a network, and a Request Data message which sends a request to other nodes for a Document. Random Documents are chosen for insertions and requests each time. Some of the chosen Documents already exist in the network causing key collisions a feature our tests must verify, and some Documents are new allowing the insertion functionality to be verified.

Once inserted into the network, a message is distributed to the other nodes using the Freenet routing algorithm and may produce one or more replies. During the simulated rounds, each node writes messages into a log file, allowing the tester to see exactly what each node is doing, and where every message and Document is during any round.

After the simulations, the log files were analyzed to determine the paths various messages and documents took through the network. These results are described in more detail in the next section Test Results.

7. Test Results

Simulations were conducted for 10 nodes, 500 nodes and 1000 nodes. These samples numbers were chosen due to the large and complicated log files generated by the simulation. Analysis of the logs was done by hand making it a very slow and arduous process forcing us to restrict the number of nodes used in the topologies.

As the logs produced are more than 100 pages each, this section show cases only one log file for a network topology of 10 nodes. The full transcipt of this log is provided in Appendix B.

7.1 Simulation Log Files

Before viewing the log files, a brief description of what they contain and how to interpret the information is provided here.

The following virtual topology (refer to figure 7.1.1) made up of 10 nodes was used in one of our test simulations. Each node has a unique Id number that identifies it in the simulated network.

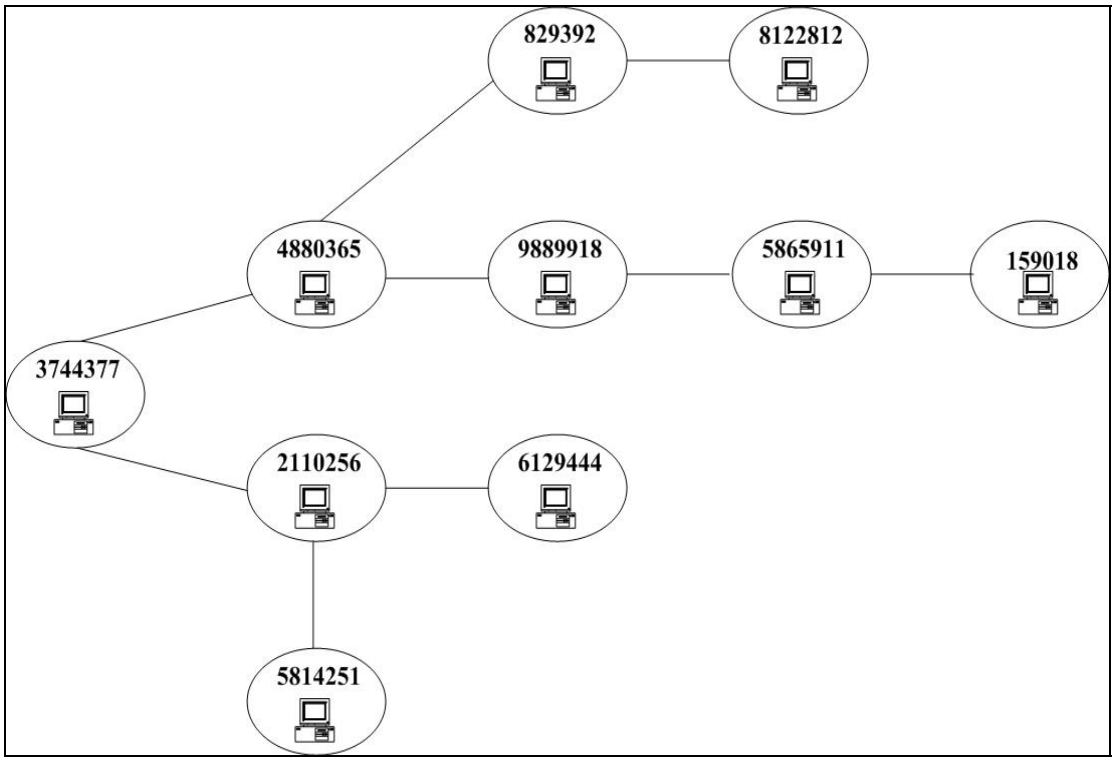

Figure 7.1.1 – Virtual Topology

A sample log file for the above topology is shown and described in more detail below. It has been split into sections so that the explanations will be clearer.

The first section of the log file describes the random messages created for each node by the driver program at the beginning of the simulation. As it is random, some nodes have no messages created for it and a 'No event generated' message is produced. Each line of the log file describes one generated event and the following information is displayed for each:

- the type of message created,
- the ids of any Documents involved in the event,
- the id of the node the current message has been created on.

This information can be seen below in the sample log file:

```
Message REQUEST_INSERT generated for doc 1004 by node 9889918
Message REQUEST_DATA generated for doc 1001 by node 8122812
No event generated by node 5865911in this round
Message REQUEST_DATA generated for doc 1004 by node 829392
Message REQUEST_INSERT generated for doc 9508535 by node 1596018
Message REQUEST_INSERT generated for doc 8825713 by node 5814251
Message REQUEST_INSERT generated for doc 109728 by node 6129444
No event generated by node 4880365in this round
No event generated by node 3744377in this round
Message REQUEST_INSERT generated for doc 1003 by node 2110256
```

After all the events have been generated, each round is simulated. The next section of the log file contains:

- the simulation Round number,
- the message id being currently processed from the global inbox,
- for each node, the message id and message type received
- for each node, the message type created in reply to a previous message

This may be seen below in the fragment from the sample log file below:

```
Current round is 0
NETWORK: Processing Message 0 from 9889918 to 4880365
NODE 4880365: Processing message 0 of type 4
NODE 4880365: Message type 4 put in Queue for 3744377
NETWORK: Processing Message 1 from 8122812 to 829392
NODE 829392: Processing message 1 of type 0
NODE 829392: Message type 0 put in Queue for 9889918
NETWORK: Processing Message 2 from 829392 to 8122812
NODE 8122812: Processing message 2 of type 0
NODE 8122812: Message type 3 put in Queue for 829392
NETWORK: Processing Message 3 from 1596018 to 5865911
NODE 5865911: Processing message 3 of type 4
NODE 5865911: Message type 4 put in Queue for 9889918
NETWORK: Processing Message 4 from 5814251 to 2110256
NODE 2110256: Processing message 4 of type 4
NODE 2110256: Message type 4 put in Queue for 6129444
NETWORK: Processing Message 5 from 6129444 to 2110256
NODE 2110256: Processing message 5 of type 4
NODE 2110256: Message type 2 put in Queue for 6129444
NETWORK: Processing Message 6 from 2110256 to 5814251
```

```
NODE 5814251: Processing message 6 of type 4
NODE 5814251: Message type 3 put in Queue for 2110256


Current round is 1
NETWORK: Processing Message 0 from 4880365 to 3744377
NODE 3744377: Processing message 0 of type 4
NODE 3744377: Message type 3 put in Queue for 4880365
...
```

These messages continue till all the rounds are over. As they are fairly long, the full transcripts of the test log for this simulations may be found in Appendix B.

7.2 Results of Log File Analysis

The responses for every message were verified by checking the status of the message at the end of all the rounds. This was done as the log files were too large to be analyzed line by line.

7.2.1 Responses to Request Data Messages

This message carries a request for a specific document by a node. There are three possible correct responses for this type of message, and an incorrect response [4].

1. Response 1: If a node storing this document receives the message, the reply message should contain the data
2. Response 2: If a node does not have the document, the message should be forwarded downstream to the requestor.

3. Response 3: If the document cannot be found and its hops-to-live counter is zero, the message should be discarded
4. Incorrect Response: Any response that is not 1, 2 or 3

Response 2 occurred 100% of the time, as it is the mechanism for forwarding the messages. Thus, table 7.2.1.1 shows the results of our simulations for Response 1 and 3:

Topology type	**Number of messages generated**	**% Occurrence of Response 1**	**% Occurrence of Response 3**	**% Occurrence of an incorrect response**
10 Nodes	9	100	0	0
100 Nodes	92	99	1	0
500 Nodes	420	79	21	0
1000 Nodes	932	77	23	0

Table 7.2.1.1 – Simulation Results

7.2.2 Responses to Request Insert Messages

This message tries to insert a document into the network. There are two scenarios that may occur with this message, producing different responses as based on [2]:

1. The document to be inserted does not already exist in the network. The correct response for this case would be a Send Data message indicating that the document can be inserted. An incorrect response is one which does not allow the document to be inserted.
2. The document to be inserted already exists on the network, in which case a collision occurs, that is, a node holding a document with the same id would respond that that id cannot be re-used. The correct response would be one that does not allow the document to be inserted. There is no incorrect response for this scenario as it Freenet will allow duplicate keys to be used, although it is frowned upon.

The tables below show the results of both these scenarios. When determining whether a correct or incorrect response has occurred, the scenario was taken into account as described above:

Results of Scenario 1:

Topology type	% Occurrence of A correct Response	% Occurrence of an Incorrect Response
10 Nodes	100	0
100 Nodes	100	0
500 Nodes	100	0
1000 Nodes	100	0

Table7.2.2.1 – Results of Scenario 1

Results of Scenario 2:

Topology type	Number of messages generated	% Occurrence of Correct Response
10 Nodes	9	100
100 Nodes	92	97
500 Nodes	420	78
1000 Nodes	932	76

Table7.2.2.2 – Results of Scenario 2

7.3 Simulation Times

The following are the times the simulations for a specific topology took:

- Average time to run 10 Node simulation: 0.09 seconds
- Average time to run 100 Node simulation: 0.37 seconds
- Average time to run 500 Node simulation: 0.86 seconds
- Average time to run 1000 Node simulation: 1.12 seconds

8. Problems encountered and solutions

One of the first problems we encountered when starting work on this project was with understanding the skeleton code we were given to work with. As our simulator needs to work with the skeleton, it is essential we have a good understanding of the code and how it works. The problem here however, is that the code doesn't work. While it does function correctly, it's just a skeleton, i.e., it doesn't do anything on its own. This problem was compounded by the fact that we were just learning C++, the language we would be writing the code in, and the language the skeleton was written in.

In addition to focusing more on learning C++, we worked on this problem by dividing the skeleton code up by class and going over each class thoroughly as a group. Once we had a basic understanding of the code, we attended a tutorial session hosted by the author of the code, Jiang Guo. This tutorial helped us to get a much better understanding of how the skeleton worked and answered some of the questions we had. We then reviewed the code several times, making sure we have a good understanding of what each class did. As our understanding of C++ improved and we looked over the code again and again, discussing it amongst ourselves each time, we started to get a better and better understanding of the skeleton and how to use it.

While this wasn't the most serious problem, it was a problem that needed to be dealt with, and done so as quickly as possible in order to get to work on planning how to

implement our simulator. Although we were able to overcome this problem without much difficulty, it may have been a factor in leading to our second problem.

The second and more serious problem we experienced was falling behind on our schedule. Because of the extra time needed to understand the skeleton code, we were late in starting the design for our simulator. This problem was compounded by the fact that we were nearing the end of the fall term, with heavy course workloads to finish before the end of the term, and upcoming exams to worry about. While we were originally planning to be putting the final touches on our design at this point, we found ourselves just starting the design and not having enough time to focus enough attention on it. This set us back even further, as we did not get far into the design until the winter break following our exams. By the time we got into the design, we found ourselves almost a month behind schedule.

While this problem can be significantly more detrimental to our project than the first, it is much easier to solve. Fortunately in planning our original schedule we allowed for some extra time at the end in case of such unexpected delays. In addition to this extra time, some extra work was put in on the project over the winter break and early in the new term while the course workload was still low. This extra work allowed us to finish out design specifications early in the new term, now only a couple weeks behind our original schedule. This time was then made up with some extra hours of coding over the spring break, and final testing was completed within the extra week we had allowed ourselves at the end of our schedule.

This problem may have been able to be avoided with better planning and more work early in the project however these were not the only causes. Several factors compounded to create this problem and make it as serious as it was. Fortunately we had allowed ourselves extra time at the end of the project to make up lost time, as well as allowing ample time in each phase of the project which permitted us to complete each phase faster than scheduled without compromising the quality of our code.

9. Conclusions

9.1 Discussion of the Test Results

The results of the simulation support our claim that this implementation of Freenet on the skeleton is correct, fast and scalable.

The responses produced in reply to the randomly generated Data Request messages match our expected results based on [3] and [4]. No incorrect responses were generated for this message in any of the topologies, that is, only responses from the expected three were produced by the simulation. If the simulator did not handle messages correctly it would have responded to it with an incorrect reply, or by discarding the message. All messages were accounted for at the end of the rounds and so no messages were incorrectly discarded. Therefore it may be concluded that no incorrect replies were detected and that these messages are processed and responded to correctly.

It is noticeable that in the 500 and 1000 node topologies, the percentage of successful document requests was 79% and 77%, respectively. The value is not 100% due to the value of the hops-to-live attribute of the simulator messages. The hops-to-live value is the number of nodes a message may pass through before it is discarded without any response, and can be set by a user before running a simulation. If this value is too low, then a message will not be able to traverse the whole network and thus may not find the node holding the document. During our testing, this value was set at half the size of the networks a typical value used by many real-life networks. This value was too low to

produce a 100% success rate as some nodes could not be reached by a message. It is still fairly high, even though half the network could not be traversed, and this is attributed to the fact that messages are routed toward the nodes most likely to have the document, supporting the claim that our implementation correctly follows the Freenet routing algorithm. The simulation would produce much higher results if run for more time with constant messages, allowing documents to distribute over the network. This would require a more complex driver program though, and is unnecessary to test our hypotheses.

The replies to randomly generated Request Insert messages also matched our expected results for both scenarios. In the first scenario for the Request Insert message, the document does not exist anywhere in the network. The expected response to this would be a Send Data message that indicated the document can be inserted into the network. The node should then insert the document, which is forwarded upstream. In all three network topologies 100% of this scenario was successful and the document was inserted. This demonstrates that these messages are processed and responded to correctly.

The second scenario is more complicated. Since the document to be inserted already exists on the network, we expected that most of the time a collision would occur, that is a node holding a document with the same id would respond that that id cannot be re-used. But since the hops-to-live attribute of the message was set to half the size of the network topology, we also expected that sometimes no collision would occur since not all nodes could be reached. This is exactly what our results show occurred in the larger 500 and 1000 node topologies. In the topology containing 500 Nodes, a collision occurred

78% of the time, and in the 1000 Node topology 76% or the requests resulted in a collision. This is a fairly high success rate considering that only half the topology can be traversed by the message. The fact that in the much smaller 100-Node topology collisions occurred 97% of the time supports the fact that the hops-to-live attribute has decreased the number of collisions in the larger topologies. The smaller the topology the greater the chance of a collision occurring. This was an expected outcome, and supports our claim that the implementation is correct.

The time taken to run all the simulations was in the range of 0.3 – 1.2 seconds. Most of this time was taken up due to outputting messages to the screen rather than processing the simulation. This shows that our implementation has met our objective of being fast and efficient. This speed allows the simulator to simulate much larger network topologies efficiently, making it scalable.

It can therefore be concluded that these test results support the claim that this implementation of our Freenet objectives is correct, fast, and scalable.

9.2 Next Steps

Now that the simulator has been completed and shown to work for small networks, the next logical step to take is to simulate larger networks with more nodes, and networks with nodes distributes in different topologies. These simulations can then be analyzed to determine the effect of the network topology on the Freenet's efficiency.

The simulator can also be used to look at the effects of changing various parameters. For example, the hops to live count on the messages can be increased to determine if each node can now find more data as its range has been increased, can find that data faster than before, or if the change simply causes more congestion in the network.

Finally, other protocols can be implemented using the same network skeleton and then compared against the Freenet using a common base for a fair comparison. The networks can be compared in terms of time to retrieve data, retrieving data in the fewest hops, or other such metrics to determine which networks are most efficient in which areas.

Appendix B: Simulation Logs

Log Files for 10 Node Topology

Message Hops-to-live = 5

```
Message REQUEST_INSERT generated for doc 1004 by node 9889918
Message REQUEST_DATA generated for doc 1001 by node 8122812
No event generated by node 5865911in this round
Message REQUEST_DATA generated for doc 1004 by node 829392
Message REQUEST_INSERT generated for doc 9508535 by node 1596018
Message REQUEST_INSERT generated for doc 8825713 by node 5814251
Message REQUEST_INSERT generated for doc 109728 by node 6129444
No event generated by node 4880365in this round
No event generated by node 3744377in this round
Message REQUEST_INSERT generated for doc 1003 by node 2110256

Current round is 0
NETWORK: Processing Message 0 from 9889918 to 4880365
NODE 4880365: Processing message 0 of type 4
NODE 4880365: Message type 4 put in Queue for 3744377
NETWORK: Processing Message 1 from 8122812 to 829392
NODE 829392: Processing message 1 of type 0
NODE 829392: Message type 0 put in Queue for 9889918
NETWORK: Processing Message 2 from 829392 to 8122812
NODE 8122812: Processing message 2 of type 0
NODE 8122812: Message type 3 put in Queue for 829392
NETWORK: Processing Message 3 from 1596018 to 5865911
NODE 5865911: Processing message 3 of type 4
NODE 5865911: Message type 4 put in Queue for 9889918
NETWORK: Processing Message 4 from 5814251 to 2110256
NODE 2110256: Processing message 4 of type 4
NODE 2110256: Message type 4 put in Queue for 6129444
NETWORK: Processing Message 5 from 6129444 to 2110256
NODE 2110256: Processing message 5 of type 4
NODE 2110256: Message type 2 put in Queue for 6129444
NETWORK: Processing Message 6 from 2110256 to 5814251
NODE 5814251: Processing message 6 of type 4
NODE 5814251: Message type 3 put in Queue for 2110256

Current round is 1
NETWORK: Processing Message 0 from 4880365 to 3744377
NODE 3744377: Processing message 0 of type 4
NODE 3744377: Message type 3 put in Queue for 4880365
NETWORK: Processing Message 1 from 829392 to 9889918
NODE 9889918: Processing message 1 of type 0
NODE 9889918: Message type 0 put in Queue for 4880365
NETWORK: Processing Message 2 from 8122812 to 829392
NODE 829392: Processing message 2 of type 3
NETWORK: Processing Message 3 from 5865911 to 9889918
NODE 9889918: Processing message 3 of type 4
NODE 9889918: Message type 2 put in Queue for 5865911
NETWORK: Processing Message 4 from 2110256 to 6129444
NODE 6129444: Processing message 4 of type 4
NODE 6129444: Message type 3 put in Queue for 2110256
NETWORK: Processing Message 5 from 2110256 to 6129444
NODE 6129444: Processing message 5 of type 2
NETWORK: Processing Message 6 from 5814251 to 2110256
NODE 2110256: Processing message 6 of type 3

Current round is 2
NETWORK: Processing Message 0 from 3744377 to 4880365
NODE 4880365: Processing message 0 of type 3
```

```
NODE 4880365: Message type 3 put in Queue for 9889918
NETWORK: Processing Message 1 from 9889918 to 4880365
NODE 4880365: Processing message 1 of type 0
NODE 4880365: Message type 0 put in Queue for 3744377
NETWORK: Processing Message 3 from 9889918 to 5865911
NODE 5865911: Processing message 3 of type 2
NODE 5865911: Message type 2 put in Queue for 1596018
NETWORK: Processing Message 4 from 6129444 to 2110256
NODE 2110256: Processing message 4 of type 3
NODE 2110256: Message type 0 put in Queue for 9889918

Current round is 3
NETWORK: Processing Message 0 from 3744377 to 9889918
NODE 9889918: Processing message 0 of type 3
NETWORK: Processing Message 1 from 4880365 to 3744377
NODE 3744377: Processing message 1 of type 0
NODE 3744377: Message type 2 put in Queue for 4880365
NETWORK: Processing Message 3 from 5865911 to 1596018
NODE 1596018: Processing message 3 of type 2
NETWORK: Processing Message 4 from 2110256 to 9889918
NODE 9889918: Processing message 4 of type 0
NODE 9889918: Message type 2 put in Queue for 2110256

Current round is 4
NETWORK: Processing Message 1 from 3744377 to 4880365
NODE 4880365: Processing message 1 of type 2
NODE 4880365: Message type 2 put in Queue for 9889918
NETWORK: Processing Message 4 from 9889918 to 2110256
NODE 2110256: Processing message 4 of type 2
NODE 2110256: Message type 2 put in Queue for 5814251

Current round is 5
NETWORK: Processing Message 1 from 4880365 to 9889918
NODE 9889918: Processing message 1 of type 2
NODE 9889918: Message type 2 put in Queue for 829392
NETWORK: Processing Message 4 from 2110256 to 5814251
NODE 5814251: Processing message 4 of type 2

Current round is 6
NETWORK: Processing Message 1 from 9889918 to 829392
NODE 829392: Processing message 1 of type 2
NODE 829392: Message type 2 put in Queue for 8122812

Current round is 7
NETWORK: Processing Message 1 from 829392 to 8122812
NODE 8122812: Processing message 1 of type 2

Current round is 8

Current round is 9

Current round is 10
```

Appendix C: Old Milestones

Milestone	Target Date for Completion	Responsibility
1) Technical Proposal	17th October, 2002	Steven Wolfman (1) Rehan Siddiqui (2) Sasi Shanmugarajah (3)
2) Research		
a) Freenet	Mid October	1, 2, 3
b) Network skeleton code (Simulator)	Mid November	1, 2, 3
3) Software Development		
a) Design Specifications:	Early December	1,2, 3
i) How to implement Freenet into the simulator		
ii) Design review and approval	Mid December	1, 2, 3
b) Coding:		
i) Query Protocols	Mid January	1
ii) Keys and Searching	Mid January	2
iii) Storing, retrieving and managing data	Mid January	3
iv) Message and Data routing	Late January	3
v) Component Integration	Early February	1, 2, 3
4) Testing		
a) Component Testing:		
i) Query Protocols	Late January	1
ii) Keys and Searching	Late January	2
iii) Storing, retrieval and managing data	Late January	2
iv) Message and data routing	Mid February	3
v) Component integration	Mid March	1, 2, 3
5) Documentation:		
a) Progress Report	January	1, 2, 3
b) Final Report	11th April, 2003	1, 2, 3

Appendix D: New Milestones

Milestone	Target Date for Completion	Responsibility
1) Technical Proposal	17th October, 2002	Steven Wolfman (1) Rehan Siddiqui (2) Sasi Shanmugarajah (3)
2) Research a) Freenet b) Network skeleton code (Simulator)	 Mid October Mid November	 1, 2, 3 1, 2, 3
3) Software Development a) Design Specifications: i) How to implement Freenet into the simulator ii) Design review and approval b) Coding: i) Query Protocols ii) Storing, retrieving and managing data iii) Message and Data routing iv) Run-time Engine, and Keys and Searching v) Component Integration	 Mid January Late January Early February Early February Early February Mid February Late February	 1,2, 3 1, 2, 3 1 2 3 3 1, 2, 3
4) Testing a) Component Testing: i) Query Protocols ii) Storing, retrieval and managing data iii) Message and data routing iv) Run-time Engine, and Keys and searching v) Component integration	 Early February Early February Early February Mid February Late March	 1 2 2 3 1, 2, 3
5) Documentation: a) Progress Report b) Final Report	 10th January, 2003 11th April, 2003	 1, 2, 3 1, 2, 3

List of References

[1] A. Langley, “Freenet” in Peer-to-Peer: Harnessing the Power of Disruptive Technologies, A. Oram, Ed., Sebastopol CA: O'Reilly and Associates, 2001. pp 123-132.

[2] I. Clarke, A Distributed Decentralized Information Storage and Retrieval System, unpublished report, Division of Informatics, University of Edinburgh, 1999.

[3] AmrZ.Kronfol, “FASD: A Fault-Tolerant, Adaptive, Scalable, Distributed Search Engine” [Online Article] Available at: http://www.freenetproject.org/kronfol_final_thesis.pdf

[4] A. Langley, “The Freenet Protocol” [Online Article] Available at: http://www.firenze.linux.it/~marcoc/index.php?page=protocol

[5] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong, “Freenet: A Distributed Anonymous Information Storage and Retrieval System” [Online Article] Available at: http://www.firenze.linux.it/~marcoc/index.php?page=icsi-revised

[6] I. Clarke, “The Freenet Project:A distributed decentralized information storage and retrieval system” [Online Lecture] [2001 Feb 14] Available at: http://murl.microsoft.com/LectureDetails.asp?765

[7] T. Hong, “Performance” in Peer-to-Peer: Harnessing the Power of Disruptive Technologies, A. Oram, Ed., Sebastopol CA: O'Reilly and Associates, 2001. pp 205-243.