# STUDY OF STEADY-STATES IN DISTRIBUTED DATA CACHING IN AD HOC NETWORKS

A Thesis by

Julinda Lyn Taylor

Bachelor of Science, Wichita State University, 2006

Submitted to the Department of Electrical Engineering and Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

May 2012

**STUDY OF STEADY-STATES IN DISTRIBUTED DATA CACHING IN AD HOC NETWORKS**

The following faculty members have examined the final copy of this thesis (or dissertation) for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science, with a major in Computer Science.

_____
Bin Tang, Committee Chair

_____
Prakash Ramanan,., Committee Member

_____
Esra Buyuktahtakin, Committee Member

## DEDICATION

To Lane, my loving husband, my family and friends, without whom, I would not have been able to complete this work.

## ACKNOWLEDGEMENTS

I would like to thank Dr. Tang for his tireless effort and patience. And, Dr. Yildrim for his contribution in Integer Linear Programming. I am grateful to my mentors and friends Keenan Jackson, Steve Copeland, Dave Lombard, and Tom Wallis. I would also like to thank my committee members: Dr. Tang, Dr. Ramanan, and Dr. Buyuktahtakin.

## ABSTRACT

There has been extensive research on cooperative distributed data caching in ad hoc networks. However, most of the work has focused on how to reduce the average delay of requests and improve the packet delivery ratio, etc; not much work has been done to study the steady-state status achieved by distributed caching algorithms. Information related to steady-state status includes the convergence time of the caching algorithms, the final data cache placement in the network, the stabilized cost performance, and the performance comparison of distributed caching algorithms with an optimal centralized caching solution. Previous theoretical results show that to minimize the average access cost in the network, the optimal number of replicas of each data object is proportional to the square root (or two-thirds) of the data's access frequency. In this work, we empirically show that the optimal replica number not only depends on the access frequencies of data, but also depends on the storage capacity of each node. We propose a heuristic model studying both cooperative, hybrid, and selfish caching steady-states in ad hoc networks. We formulate and solve the data caching problem optimally using integer linear programming (ILP) in order to validate our findings with regard to access frequency. We also provide empirical data regarding the steady-state cost of data based on the storage capacity of the nodes in the network. Via extensive ns-2 simulations [10], we gain some insight regarding the steady-states of distributed data caching.

# TABLE OF CONTENTS

## FIGURES

# CHAPTER 1

# INTRODUCTION

Wireless and ad hoc networks have increased in importance due to the explosion of wireless and mobile technology. Ad hoc networks have fluid topologies; nodes can change position, join, or leave the network entirely. The types of devices in the network might be a combination of mobile computers, mobile computing devices, or even embedded processors such as sensor nodes. Such networks are widely used for different tasks such as spontaneous meetings, collaboration on a project, or taking measurements over a large area (sensor networks).

## 1.1 Caching Strategies in Distributed Data Caching

Ad hoc networks have no central base. Information in the network travels across nodes in a multi-hop manner; data is requested from one part of the network and fulfilled by a potentially remote node. Caching can decrease the access latency of such requests, and, caching can decrease the resources needed to access data which is important in devices that have greatly reduced resources than traditional network devices. These features motivate the optimization of memory or cache space and the cost of data as it is accessed and moved within the network.

In order to save memory resources and cost for data access, data caching is used. Cooperative distributed data caching strategies have been widely studied as a method to increase the efficiency of a network by facilitating access to information (See the comprehensive Literature Review in Section 1.2). Traditionally, the advantages of caching data are decreased data access delay, improved reliability of data, and increased fault tolerance across the system. Current

distributed caching algorithms are heuristically based, even though some of them have endeavored to achieve performance bounds. The designed distributed algorithms have been constructed to either reduce the average query delay, the total number of wireless transmissions, and the query success ratio in the network [14, 20, 22], or to improve the data availability in the network [9,18].

In ad hoc networks, the data caching problem has its theoretical roots in the multi-facility location problem [1,2], which is known to be NP-hard. The primary question is: given an ad hoc network, the data access pattern (the probability that each node accesses each data item), the data size, the initial data distribution in the network, and the memory capacity of each node, how can the data be placed in caches such that the average access cost of each node is minimized? The optimal final cache placement has not been studied with any of the existing distributed data caching algorithms. There is no comparison of how the existing distributed data caching algorithms perform with regard to an optimal solution, or perform with regard to the non-cooperative selfish caching algorithm. There is also no heuristic data provided to demonstrate how different data caching algorithms perform with regard to storage space and access cost for data by nodes in the network.

### 1.2 Literature Review

Data caching and replication in ad hoc and sensor networks is an active research area. Recently, Du et al. [9] published a paper proposeing COOP, a novel cooperative caching scheme for on-demand data access applications in MANETs. The objective is to improve data availability and access efficiency by collaborating local resources of mobile nodes. Montanari et al. [18] use

probabilistic failure models to adaptively create and maintain a number of replicas of the data to provide data availability in sensor networks. Dimokas et al. [8] propose a new cache consistency and replacement policy in a wireless multimedia sensor networks, with the goal of latency minimization. Wu et al. [21] utilize the overhearing property of wireless communications for performance improvement of data caching in ad hoc networks. Fan et al. [11] design distributed caching heuristics, via which better performance can be achieved by detecting the variation of contentions to evaluate the benefit of selecting a node as cache node. Hara and Madria [14] are among the first to propose replica allocation methods in ad hoc networks, by taking into account the access frequency from mobile hosts to each data item and the status of the network connection. Yin and Cao [22] design and evaluate three simple distributed caching techniques, which we study in this thesis, Fiore et al. [12] design a cooperative caching scheme to create content diversity in ad hoc networks, so that a requesting user likely finds the desired information nearby. Zhao et al. [22] propose a novel asymmetric cooperative cache approach, where the data requests are transmitted to the cache layer on every node, but the data replies are only transmitted to the cache layer at the intermediate nodes that need to cache the data. Tang, Das, and Gupta [20], present a polynomial time centralized approximation algorithm to replicate data, which reduces the total data access delay to at least half of that obtained from the optimal solution. It also demonstrates a distributed caching technique derived from the centralized approximation algorithm, called benefit-based data caching technique, which is used in this thesis.

However, all the above work does not study the steady-state of distributed data caching, which is the topic of this thesis.

### 1.3 Contribution of this Thesis

This thesis asks the following questions: a) How to define and characterize steady-state of data caching in ad hoc networks, b) How does storage affect the efficiency of distributed caching algorithm performance, c) How do the distributed caching algorithms compare with an optimal caching solution in terms of average data access cost based on the cache placement in steady-state?

Optimality of data caching strategies have been studied in the context of mesh networks and peer-to-peer networks [6, 16, 17]. Jin and Wang [16] find that to minimize average cost in the networks, the optimal strategy replicates an object such that the number of its replicas is proportional to $p^{2/3}$ , where $p$ is the access probability of the object. The authors in [6, 17] show that the optimal replicated number of each object is proportional to the square root of its query probability. We refer to those two caching strategies as $p^{2/3}$ *caching* and $p^{1/2}$ *caching* in this thesis. Surprisingly, both works failed to show (theoretically and experimentally) how storage capacity affects the optimal number of replicas. We empirically show that the optimal replica number of a data item in a network not only depends on the access frequencies of data, but also depends on the storage capacity of each node.

On the other hand, the steady-state behaviors of cache replacement schemes such as LRU have been evaluated either requiring exponential time complexity or using approximated closed form expression [7, 13, 19], so in this thesis we use experimental heuristics to study the steady-states of distributed caching. Specifically, we study the steady-states of an existing cooperative data caching algorithm and a selfish data caching technique, and compare them to the optimal cache placement obtained using an integer linear programming approach. We utilize the cache placement in the dynamic caching process to calculate the average data access cost in both algorithms, and identify whether such cost gets stabilized when reaching steady-states.

We have three main observations. First, unlike previous work, we empirically show that the optimal replica number in a network not only depends on the access frequencies of data, but it also depends on the storage capacity of each node. Second, we show that a hybrid based caching strategy works optimally in scarcity situations and that a greedy based caching algorithm works optimally in a surplus situation. We provide some analysis of these findings via simulations. To our knowledge, this is the first work to empirically compare these algorithms. Finally, we show that selfish caching, which was shown to perform much worse than cooperative caching, performs comparably with cooperative caching in terms of the average data access cost based on the cache placement of the steady-states. We give some analysis of this phenomenon. We also study the steady-state efficiency of three data caching techniques: cooperative, selfish and a hybrid technique and evaluate their efficiency with regard to data storage. To the best of our knowledge, our work is the first one to formally formulate the data caching problem in ad hoc networks using ILP and solve it optimally.

Since this thesis focuses on the steady-states study of distributed data caching, and not on other important issues such as the mobility of ad hoc networks and data consistency, we make the following simplified assumptions. We assume the network is static. We also assume each node sends read-only queries and thus data consistency is not considered in this thesis.

### 1.4 Thesis Organization

The rest of the thesis is organized as follows: In Chapter 2, we present our network model and formulate the data caching problem using ILP. Chapter 3 presents the cooperative, hybrid, and selfish distributed caching schemes studied in this thesis. In Chapter 4 we propose the model and define the steady-states for the distributed data caching. We present and analyze the simulation results in Chapter 5. Chapter 6 concludes the thesis and recommends some future work. Appendix A contains source code for a program written to evaluate simulation results from ns-2. The program is used to calculate the total access cost of each node accessing each data across a given network topology and the data frequency at a given time.

# CHAPTER 2

# MODELS AND FORMULATIONS

## 2.1 The Network Model

A multi-hop ad hoc network can be represented as an undirected graph $G(V,E)$ where $V = \{1, 2, \ldots i, ..., |V|\}$ represents the nodes in the network, and $E$ is the set of weighted edges in the graph. Two network nodes that can communicate directly with each other are connected by an edge in the graph. There are multiple data items in the network and each is served by its source node. Each network node has limited memory and can cache multiple data items subject to its memory capacity limitation. The objective of the data caching problem is to minimize the total (or average) access cost. Below, we give a formal definition of the cache placement problem addressed in this thesis.

The set of data items in the network is $D = \{1, 2, \ldots, p, \ldots, |D|\}$, where data item $p \in D$ is originally produced at its source node $s_p$, where $s_p \in V$, (it is possible that a source node can have several data items). Source nodes always keep their original data; in addition, they still have access to a cache to store data from other source nodes. The size of data item $p$ is $s_p$ units. Node $i \in V$ has a memory capacity of $m_i$ units (for a source node $i$, $m_i$ is the available storage space after storing its original data). Two nodes can communicate directly with each other if they are within transmission range of each other. The edge weight may represent a link metric such as loss rate, distance, delay, or transmission bandwidth. In this thesis, the edge weight represents the bandwidth and we assume all the edges have the same bandwidth $B$. We use $a_{ip}$ to denote the access frequency with which a node $i$ requests the data item $p$, i.e., the number of times a node $i$

requesting the data item $p$ within some unit time. The transmission time of sending data item $p$ along any network edge is $s_p/B$. We use $d_{ij}$ to denote the number of transmissions to transmit a data item from node $i$ to $j$ and, thus it is equal to the number of edges of the shortest path between these two nodes. The *total data access cost* in ad hoc network before caching is the total transmission time spent by all the nodes to access all the initial copies of the data:

$$\sum_{i=1}^{|V|}\sum_{p=1}^{|D|} a_{ip} \times d_{iS_p} \times s_p/B.$$

Figure 1: Total time for each node to access all the data on the network, from the source node, without accessing any data from caches.

We allow $B = 1$ for the rest of the thesis; thus omitting the variable in our equations. The objective of the data caching problem is to minimize the total data access time by caching data items in the ad hoc network, under the storage constraint of nodes. In Section 2.3, we give a formal definition of the caching problem using integer linear programing (ILP).

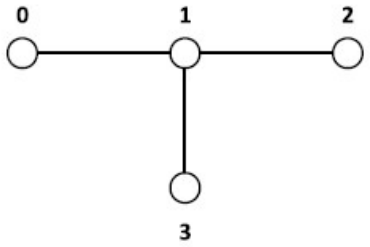


Figure 2: A simple ad hoc network illustrating the data caching problem. There are two data items (0 and 1) in the network– data 0 is at node 0, data 1 is at node 1.

A Simple Example

Figure 2 displays a simple example of a small network of four nodes (nodes 0, 1, 2, 3). There are two data items (0 and 1) in the network. Data item 0 is served from node 0, data item 1 is served from node 1. Every node has one storage capacity, i.e., it can store one data item. For all the nodes, their access frequency to data 0 is 10, access frequency to data 1 is 1. The optimal data placement is that node 0 caches a copy of data item1, while nodes 1, 2, and 3 each caches a copy of data item 0. The optimal total access cost is 2.

### 2.2 Steady-State Cost Investigation

For each node $i$, there is a storage capacity of $m_i$. If we assume that $m_i \geq |D|$, then the amount of memory available is *at least* large enough for each node to cache each data item $p$. In such a state, the average cost of the network will always converge to a steady-state of a 0 total access cost. When $m_i < |D|$, then the total access cost will not converge to a steady-state of 0. However, the total average cost may reach a stable, steady-state within the network depending on the distributed data caching strategy.

### 2.3 Integer Linear Programming Problem Formulation

Let $x_{ip} \in \{0, 1\}$ denote whether data item $p$ is cached in node $i$ , and let $y_{ijp} \in \{0, 1\}$ denote whether node $i$ accesses data item $p$ from node $j$. The ILP problem formulation is then given as follows:

Minimize (1):

$$\sum_{i \in V} \sum_{j \in V} \sum_{p \in D} a_{ip} \times d_{ij} \times y_{ijp} \times s_p, \qquad (1)$$

where

$$\sum_{j \in V} y_{ijp} = 1, \quad i \in V, p \in D \qquad (2)$$

$$x_{jp} - y_{ijp} \geq 0, \quad i, j \in V, p \in D \qquad (3)$$

$$\sum_{p \in D} s_p \times x_{jp} \leq m_j, \quad j \in V \qquad (4)$$

Figure 3: Integer linear programming minimization formulation.

Condition (2) guarantees that node *i* has access to data item *p* in the network. Condition (3) ensures that each data item can only be accessed from a node which stores that data. Condition (4) states that at node *j*, the total size of cached data cannot exceed node *j* 's memory capacity $m_j$ (i.e., memory constraint). The data caching problem is NP-hard, since the simpler multi-facility location problem is NP-hard [4, 5, 15].

# CHAPTER 3

# CACHING STRATEGIES

## 3.1 Cooperative, Selfish, and Hybrid Distributed Data Caching

In this section, we present first the data access models of distributed caching. We then discuss three representative algorithms of cooperative data caching and selfish data caching: benefit-based caching, LRU-based (selfish) caching, and hybrid based caching.

## 3.2 Data Access Model of Distributed Caching

The data access model includes the data access pattern, which indicates the popularity of the data. It also includes the data access interval which indicates the query traffic in the ad hoc network.

Data Access Pattern We consider the Zipf data access pattern. In this pattern, the data is ranked by their access popularity following the Zipf-like distribution [3, 23], where the access frequency of *ith* popular data item is $P_j = \frac{1/j^{\theta}}{\sum_{h=1}^{|D|} 1/h^{\theta}}$, represented by where $0 \leq \Theta \leq 1$. When $\Theta = 1$, the above distribution follows the strict Zipf distribution; while for $\Theta = 0$, it follows the uniform distribution, in which all $|D|$ data items are equally popular and accessed by each of the $|V|$ nodes.

Data Access Interval Each node in the network sends out a single stream of read-only queries, each of them requesting a data item following the above data access pattern. The data access

interval is the time interval between two consecutive queries and follows an exponential distribution with some mean value $T_q$.

For each node, before it sends out a data request, it always checks if the data item is already cached locally. If not, it sends out the request. After receiving the data item, whether it caches the data item into its local memory or not depends on whether the distributed caching scheme is cooperative, selfish, or hybrid.

### 3.3 Cooperative Data Caching [20]

Benefit-based caching is a cooperative data caching algorithm. Each node maintains a *nearest cache table* which records for each data item the closest cache node (including the source node) that has a copy of the data. If the node itself is a cache node of the data item (i.e., it is the closest cache node for this data item), it records the second-nearest cache node that has a copy of the data. By maintaining the accurate nearest cache table (please refer to [20] for its maintenance mechanism), each node can not only directly access the closest copy of the data item without searching for it, but can also make an intelligent local caching decision with the knowledge of the data copy information in its neighborhood. On the other hand, each node observes all the data request messages passing through it. These data request messages include the node's own data requests (being a data requester), the data request messages that the node forwards (being a relay node), and data requests it receives (being a cache node) of the requested data.

With the above nearest cache table and message observation, each node can constantly compute the *local benefit* of each data item. For each data item *p not* cached at node *i*, the node *i* calculates the local benefit gained by caching *p*. While for each data item *p* cached at node *i*, the node *i* computes the local benefit lost by removing it. In particular, the local benefit $B_{ip}$ of caching (or removing) *p* at node *i* is the reduction (or increase) in access cost given by

$$B_{ip} = t_{ip}\delta_p,$$

Figure 4: Benefit calculation for the access cost of data.

where $t_{ip}$ is the number of request messages observed by node *i* for data item *p*, and $\delta_i$ is the distance from *i* to the nearest node other than *i* that has the copy of the data item *p*. Using its nearest cache table, each node can compute the local benefit of data items in a localized manner using only local information.

Cache Replacement Policy When there is free memory space, a node caches the passing by or requested data item. Otherwise, the caching decision is based on the benefit calculation. If the benefit of caching the newly passing by or requested data item is larger than that of the cached data item with the smallest benefit, then cache replacement takes place.

### 3.4 Selfish Data Caching

In selfish distributed data caching, each node accesses the source node for each data item. It caches any passing by or requested data item if it has free memory space and uses a Least

Recently Used (LRU) policy for replacement of cached data. Each node's request can also be satisfied on the path to the source node if one of the intermediate nodes is a cache node of the requested data.

### 3.5 Hybrid Data Caching[22]

In hybrid distributed data caching, each node accesses the source node for each data item. It caches any passing-by or requested data item if it has free memory space. If free space is not available, or not enough, then the node will choose whether to cache the data or to cache the nearest path to the data based on the Time To Live (TTL) or the size of the data. If the TTL is small, then the node will attempt to cache the data since the path may soon become invalid. Then, the node will compare the data size to a threshold (600 bytes in this thesis) and cache the data if it is less than the threshold. If the data item is larger than the threshold size, then the node will cache the path to the data item. If there is not enough memory to cache the data item or its path, then the caching strategy will essentially follow a LRU policy with regard to replacement. The node will check for any data item that has expired and remove that data item first. If there is still not enough space, then the node will remove valid LRU data but keep the path to that data item.

# CHAPTER 4

# HEURISTICS, CONVERGENCE TIME, AND STEADY-STATE DEFINITION

## 4.1 Heuristics Model of Distributed Data Caching

At time t, let $M^t_p$ be the set of network nodes that stores a copy of data $p$. For ease of presentation, we assume the source node is also a cache node, i.e., $s_p \in M^t_p$. Given the cache placement of the network at time t, the total access cost of the network at time t is the summation of the access costs upon all the nodes, each of which goes to the nearest source or cache node to access each of the data items. Denote the total access cost of the network at time t as $T$(t), then

$$\tau(t) = \sum_{i=1}^{|V|} \sum_{p=1}^{|D|} a_{ip} \times min_{l \in M^t_p} d_{il} \times s_p.$$

Figure 5: Total access cost of the network at time t.

Figure 5 is similar to Figure 1, with one difference. Figure 1 is used to find a centralized optimal caching strategy to minimize the total (or average) access cost, while Figure 5 is used to calculate the total access cost based upon a specific cache placement resulting from a distributed caching strategy at a specific time.

## 4.2 Empirical Study of the Convergence Time

The distributed caching is in its steady-state at time $t_s$ if and only if for any time

$$\tau(t) = \sum_{i=1}^{|V|} \sum_{p=1}^{|D|} a_{ip} \times min_{l \in M^t_p} d_{il} \times s_p.$$

Figure 6: Definition of steady-state.

Therefore, whether a distributed caching scheme reaches its steady-state depends on the choice of $T_{th}$ . *Convergence time* is the time it takes for a caching scheme to reach its steady-state. Below we empirically study the convergence time of the above distributed caching algorithms. Figure 7 shows that the cost of data access in the network does not change very much. This is important because it is a way to determine whether the network has reached a convergence time.

We extract the cache placement at some moment of the caching process and calculate the total access cost of the network according to Figure 6. Figure 7 shows that the cooperative algorithms stabilize very well, while selfish caching does not. This can be explained by the different cache replacement policies adopted by each algorithm. Nodes only cache data which are beneficial (i.e., reducing the access cost in the network) in cooperative caching. In selfish caching, nodes adopt a LRU cache replacement policy, which always caches the new data item even if it is not a beneficial one. In an extreme case where each node's storage can only store one data item, in selfish caching, the node always stores the latest requested data which changes dynamically over time.

However, the cooperative strategy will stick with the beneficial data item and not replace it if the new requested data item is less beneficial. For the rest of the thesis, we adopt the simulation run time of 100,000 seconds to study the steady-states of both algorithms.

### 4.3 Comparison Between Cooperative Caching and Selfish Caching at Steady-States

In cooperative caching, nodes cache data items based on whether they are beneficial to the whole network. Meanwhile, since the selfish caching we study in this thesis uses LRU as its cache replacement policy, it eventually keeps its popular data items in its local memory, even though such data items could be available in its close neighborhood. Therefore, cooperative caching gives an average access cost which is smaller than that of selfish caching. Figure 9 shows the comparison of the average query delay of both cooperative and selfish caching by varying the mean query generate time from 3 to 40 seconds, under different storage capacities (50 and 100 data items, respectively). Here the mean query generation time is the time interval between two consecutive queries of each node. It shows that when node storage capacity is 100 data items, cooperative caching performs about twice as well than the selfish caching. The performance differential gets larger in more challenging scenarios, such as higher query traffic (query generate time is 3 seconds) and smaller node storage capacity (50 data items).

### 4.4 Empirical Study of Cost Over Time and Storage Capacity

Here we examine whether the total access cost of the network will reach 0 given $m_i \geq p$, where $m$ is the memory cache per node and $p$ is the number of data items. We do this using ns-2 to simulate network activity; then, the data are put through the program in order to calculate the cost of data access in the network. We examine the total access cost of the network where nodes are able to cache between 25% - 100% of the available data items in each of the nodes. We find that for each distributed data caching strategy, the overall cost across the network will reach 0 if each node has 100% storage capacity. In a memory rich or surplus environment the selfish

caching performs very well compared to the cooperative and hybrid caching strategies. This is because the selfish strategy automatically caches each data item as it passes through the node or as it is requested by the node rather than making any decisions regarding its size, TTL, or benefit to the node. The other caching strategies still perform their evaluation metrics and will still reach a steady-state of total access cost of 0 over time as shown in Figure 6.

In more challenging environments, where there is only enough space to cache a fraction of the data, the hybrid strategy outperforms the other two strategies. The combination of caching the path to data and small data items is more efficient then the benefit based strategy. Both the benefit based strategy and the selfish caching strategy will only cache full data items. The selfish caching strategy keeps replacing full data items as they pass through the node or are requested by the node.

# CHAPTER 5

# DISCUSSION AND RESULTS

## 5.1 Performance Evaluation

We demonstrate, through simulations, the performance of our designed cache placement algorithms over randomly generated network topologies. We first compare the relative quality of the cooperative, selfish and hybrid caching schemes using the ns-2 simulator [10]. Using those results, we use a program written for this thesis (Appendix A) to determine total access cost and data frequency.

## 5.2 Simulation Configuration for Total Access Cost and Mean Generating Time

We simulated our algorithms on a network of 40 randomly placed nodes in an area of 1000 x1000$m^2$. Five topologies were created using a utility that is packaged with the ns2 simulator named *setdest*. This utility takes the number of nodes, size of the network in meters, and mobility of nodes, then generates a topology with those parameters. The topologies must be checked for full connectivity before they are used. The transmission range for two directly communicating nodes in the ns2 simulator is 250 meters. The wireless bandwidth is 2 Mb/s. In our network, there are 100 data items, each item is 750 bytes. These total cost values were computed by first running simulations in ns-2. Then, in a program written in C for this thesis (see Appendix), the total access cost to the network for each node to access each data item is computed.

There are two randomly placed source nodes $S_0$ and $S_1$ where $S_0$ stores the data items with even IDs and $S_1$ stores the data items with odd IDs. Each client node in the network sends out a single stream of read-only queries. Each query is essentially a request for a data item following the Zipf distribution. We choose to be 0.8 based on real web trace studies [3]. Data item 0 has an access frequency of 10, while other data items' access frequencies can be calculated easily using the Zipf function. In our cooperative caching scheme, the query is forwarded to the nearest cache (based on the nearest-cache table entry). In the selfish scheme, the query is forwarded to the server unless the data item is available locally; if the query encounters a node with the requested data item cached, then the query is answered by the encountered node itself. In the hybrid scheme, the query is forwarded to the server unless the data cache path has been cached.

Total Access Cost in Steady-States Figure 8 shows the total access cost comparison between ILP optimal, cooperative, and selfish caching, by varying the storage capacity of nodes. The total access cost is based on the cache placement at the steady-states. In this Figure, each data point represents an average of five runs on different network topologies, and the error bars indicate the 95% confidence interval (we only show confidence intervals in Figure 8 since other Figures are clear to our claim).

We make the following observations. First, it shows that the selfish caching performs comparably with the cooperative one. This seems contradictory with Figure 7, which shows that cooperative caching performs about twice as well as the selfish caching. This is because for the selfish caching scheme, the data access is usually not from the closest cache since each node is

not aware of other close by cache nodes, while in the total access cost calculation, both selfish and cooperative caching assumes data access from the closest cache. This demonstrates one important difference between the theoretical root (multi-facility location problem) and the distributed implementation of the caching schemes. It also shows that both strategies perform 25% worse than optimal ILP solution. This is as expected. When storage capacity is small (50), the cooperative strategy does perform better than the selfish strategy, showing its superiority to selfish caching at challenging scenarios. However, their performance difference is only around 15%, which is much smaller than the two-times difference if we use the average delay shown in Figure 9.

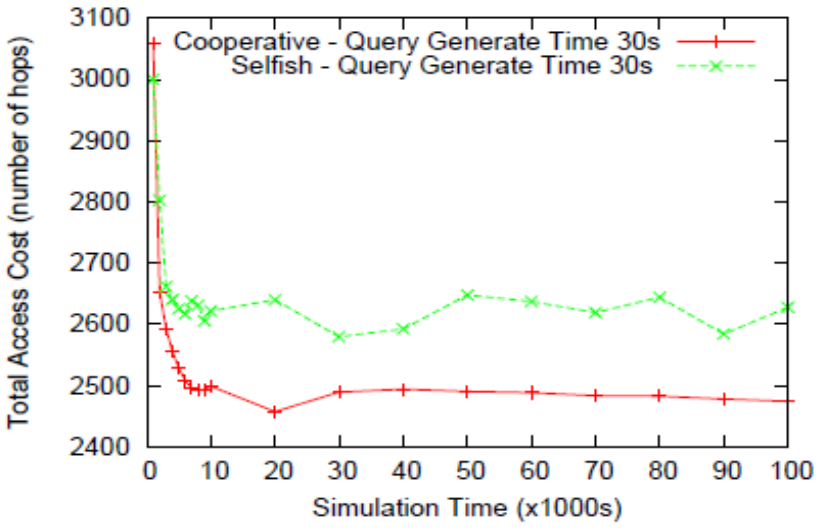


Figure 7: Convergence time of cooperative and selfish caching.

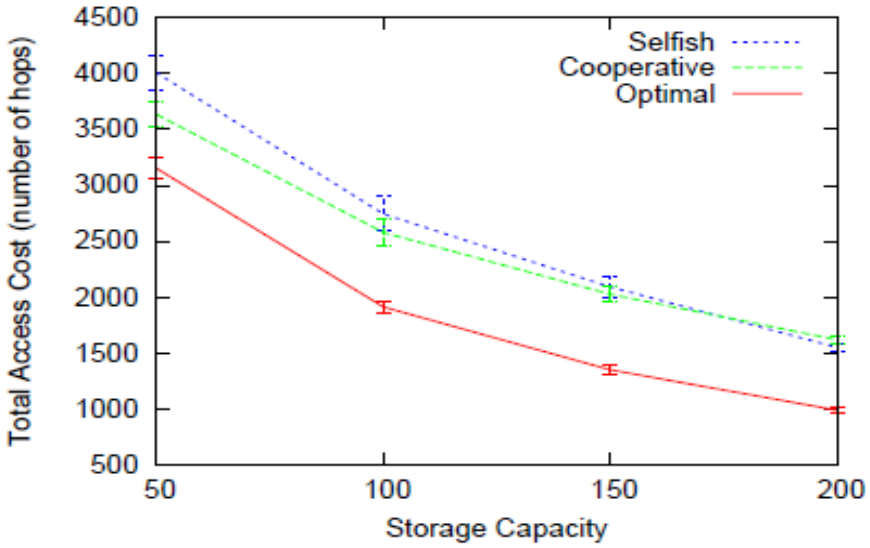


Figure 8: Total access cost comparison based on the cache placement obtained at steady-states. Here, the query generate time is 30s.

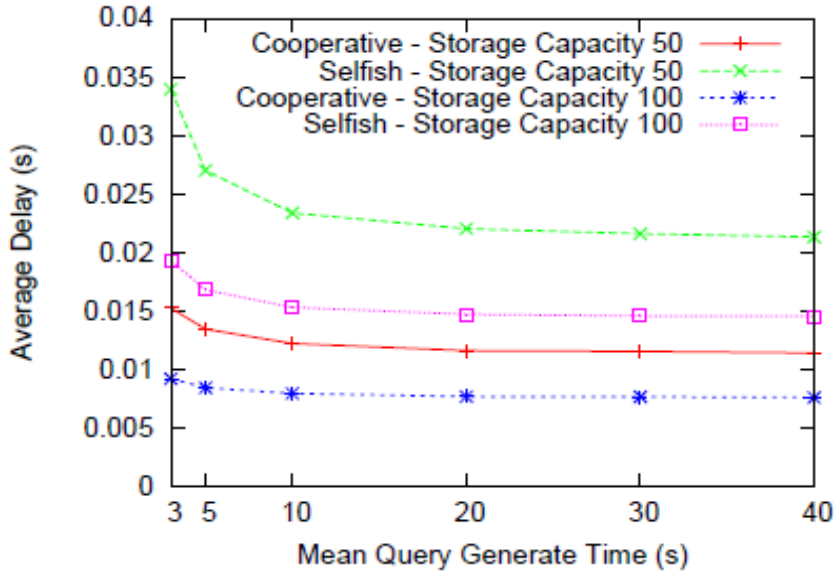


Figure 9: Average delay comparison between cooperative and selfish caching.

### 5.3 Number of Copies of Data Items in Steady-States

Next we study the number of copies of data items in steady-state caching, and compare them with the previous results, viz. $p^{2/3}$caching and $p^{1/2}$caching [6, 16, 17]. Figure 10 and 11 show the varying of number of copies of data items with respect to their popularity rankings with storage capacities of 100 data items and 200 data items respectively. These values were computed by first running simulations in ns-2. Then, in a program written in C for this thesis (see Appendix A), the frequency of the data item is counted. In both Figures, we only show the number of copies for some representative data items: 0, 10, 50, 100, and 300.

We make the following observations

- First, when the storage capacity is small (100 data items), both cooperative and selfish schemes correspond well with the ILP optimal calculation. However, when the storage capacity increases (say, to 200 data items in Figure 11), their performance difference becomes obvious. In both cases, it seems that selfish caching performs a little better than cooperative caching.

- Second, both $p^{2/3}$ and $p^{1/2}$ caching perform worse compared with the optimal solutions, and in a large capacity case, the performance difference is even more notable, since both results do not take into account the effect of storage capacity upon the optimal replica numbers.

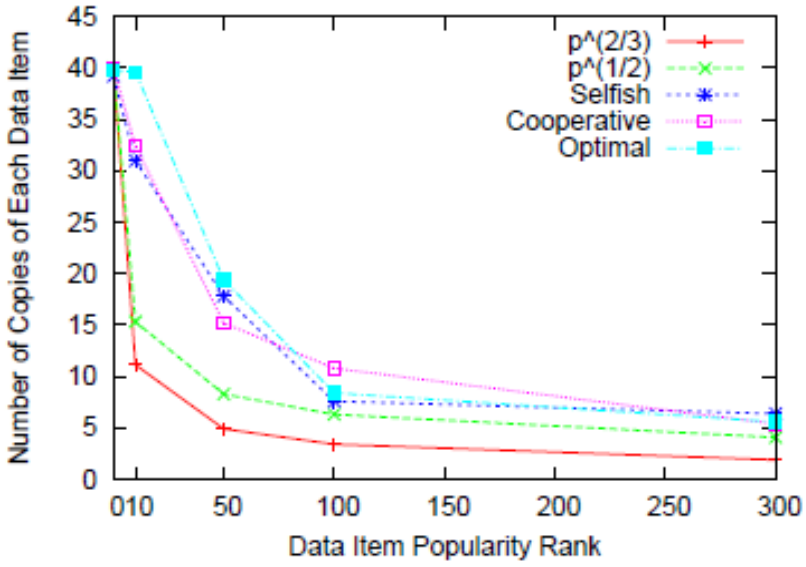


Figure 10: Number of replicas for data with different access frequencies in steady-states. Here, storage capacity of 100 data items is considered.

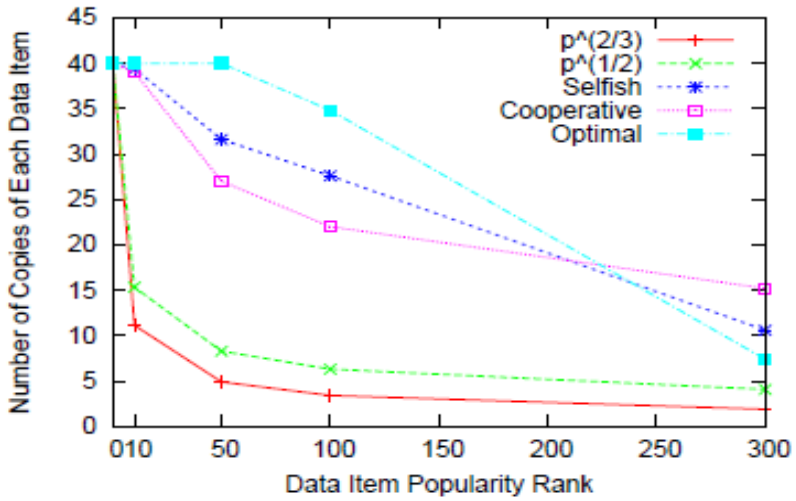


Figure 11: Number of replicas for data with different access frequencies in steady-states. Here, storage capacity of 200 data items is considered.

### 5.4 Simulation Setup and Evaluation for Cost and Storage Simulations

We simulated our algorithms on a network of 40 randomly placed nodes in an area of 1000 x1000m$^2$. The transmission range for two directly communicating nodes in the ns-2 simulator was 250 meters. The wireless bandwidth was 2 Mb/s. In our network, there were 100 data items, each generated randomly between 500 – 1000 (750 bytes averagely). These values were computed by first running simulations in ns-2. Then, in a program written in C for this thesis (see Appendix A), the total cost of each node to access every data item is computed.

There are two randomly placed source nodes $S_0$ and $S_1$ where $S_0$ stores the data items with even IDs and $S_1$ stores the data items with odd IDs. Each client node in the network sends out a single stream of read-only queries. Each query is essentially a request for a data item following the Zipf distribution. We choose to be 0.8 based on real web trace studies [3]. Data item 0 has access frequency of 10, while other data's access frequency can be calculated easily using the Zipf function. In our cooperative caching scheme, the query is forwarded to the nearest cache (based on the nearest-cache table entry). In the selfish scheme, the query is forwarded to the server unless the data item is available locally; if the query encounters a node with the requested data item cached, then the query is answered by the encountered node itself. In the hybrid scheme, the query is forwarded to the server unless the data cache path has been cached.

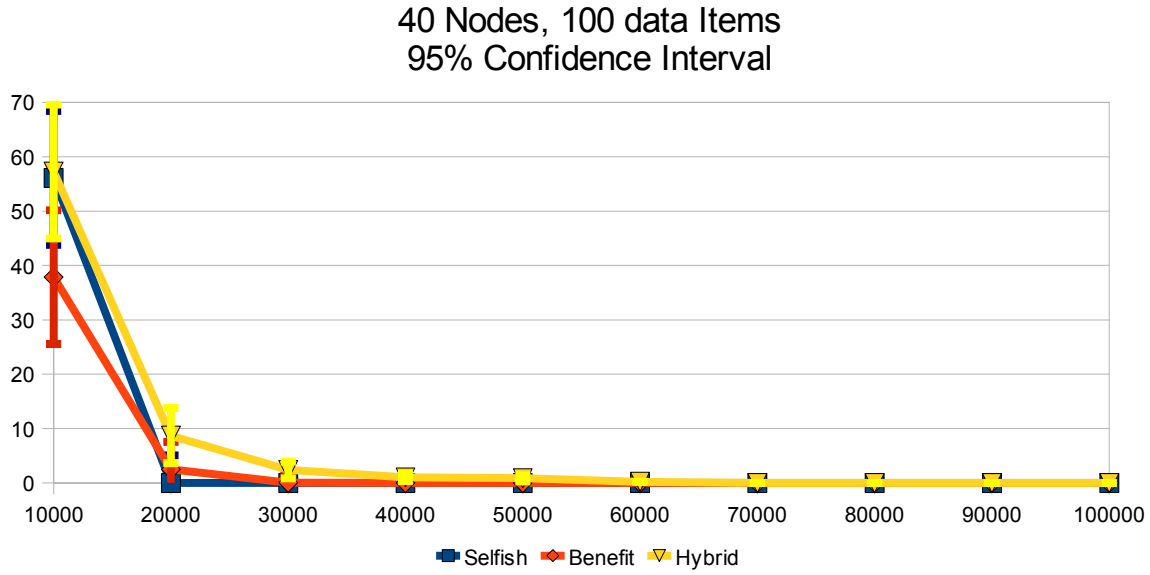


Figure 12: Total average cost calculated.

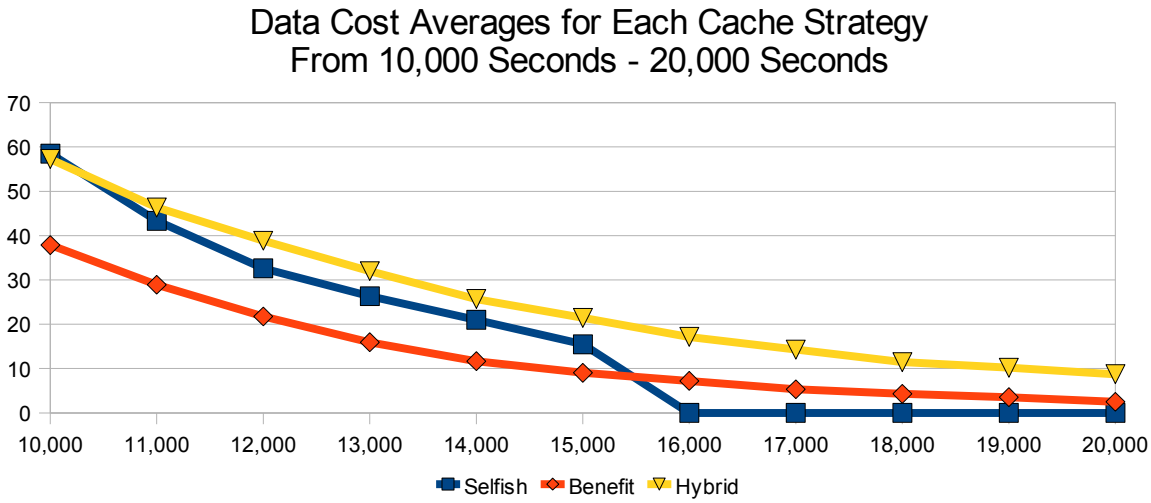


Figure 13: Detail look at cost as the selfish strategy reaches a steady-state.

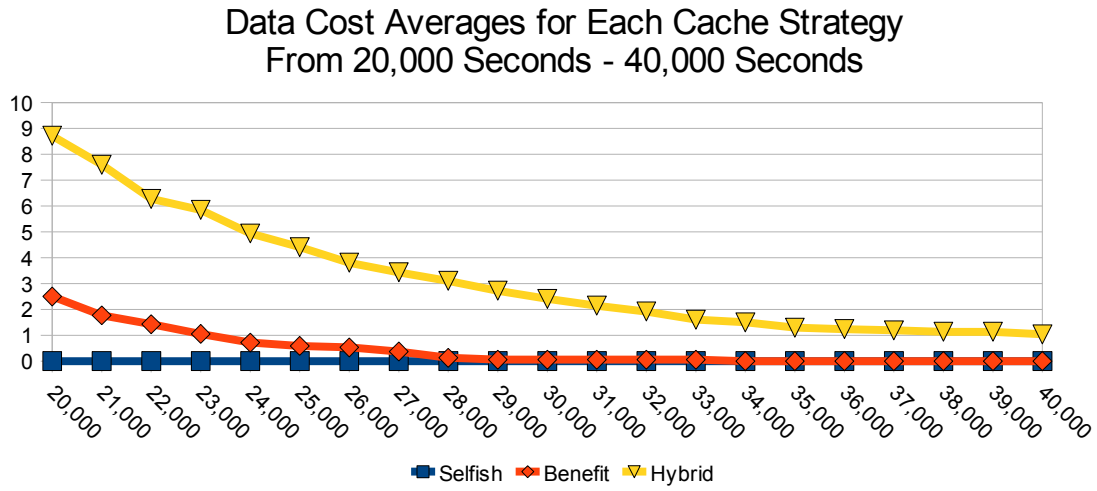


Figure 14: Detail look at cost as the benefit strategy reaches a steady-state.

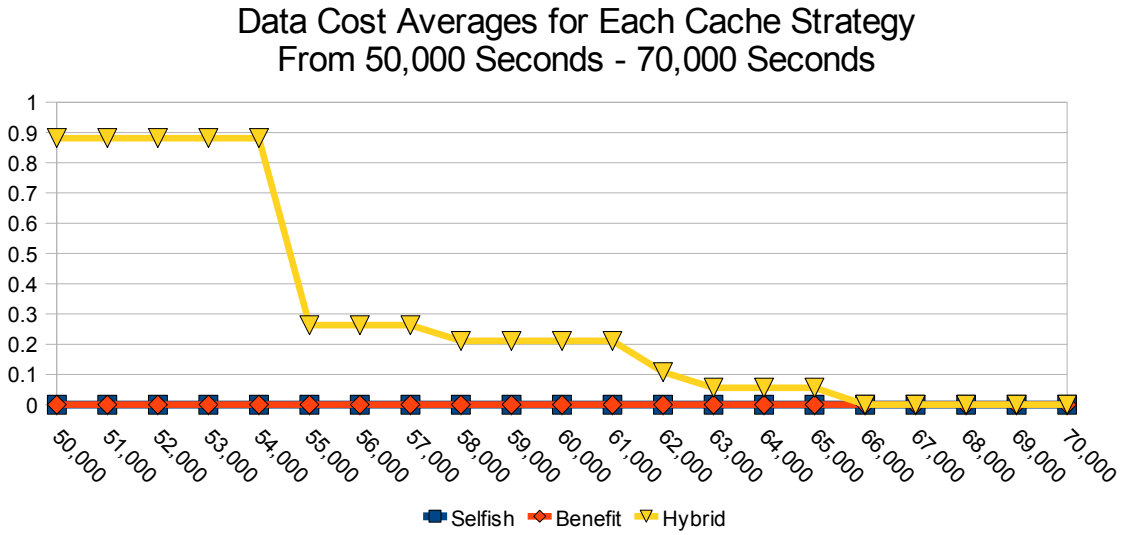


Figure 15: Detail look at cost as the hybrid strategy reaches a steady-state.

As shown in Figures 12-15, the selfish caching strategy reaches the zero cost state much earlier than the hybrid strategy, and the benefit based strategy is between the other two strategies. Figures 16-19, demonstrate that the cost values for the hybrid strategy become smaller than the selfish strategy when the memory capacity of each node is a fraction of the number of data items available. As the storage becomes smaller, the hybrid and benefit based strategies are able to assess lower access costs than the selfish strategy.

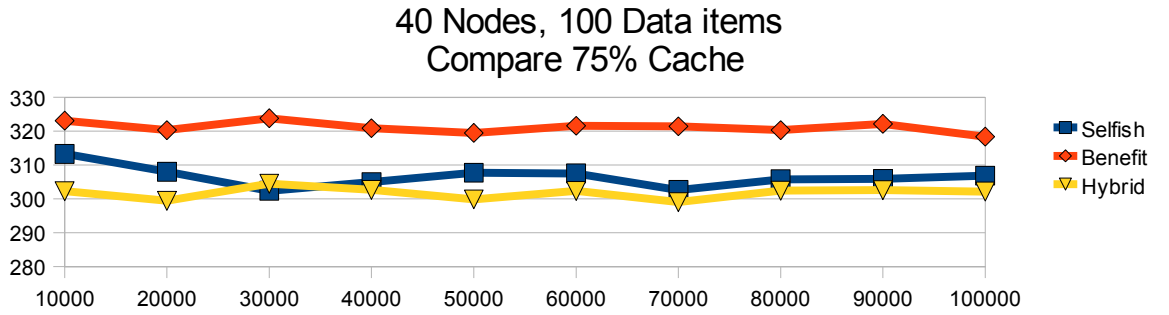


Figure 16: Total access cost at 75% cache availability.

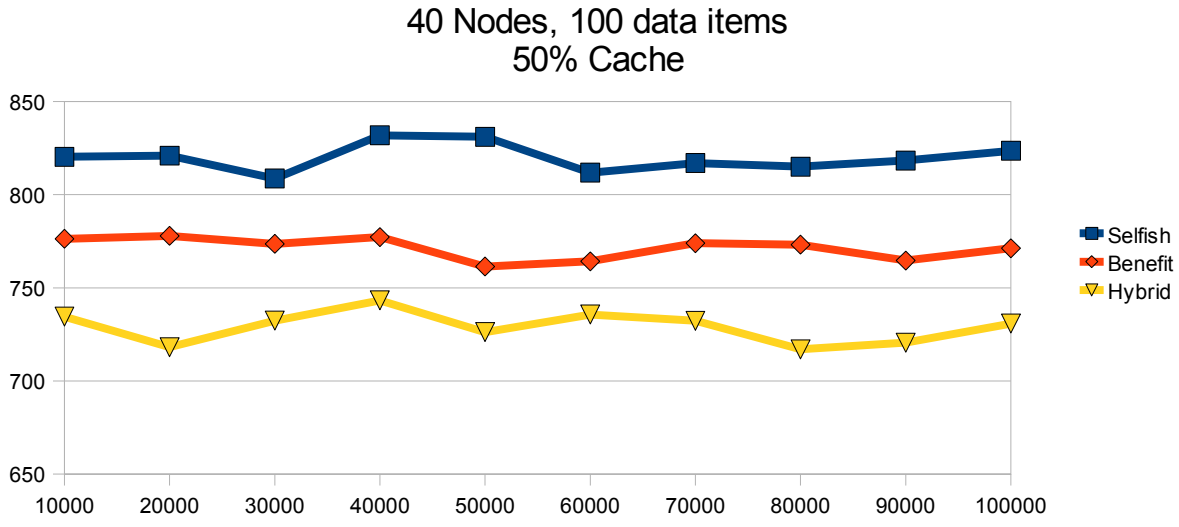


Figure 17: Total access cost at 50% cache availability.

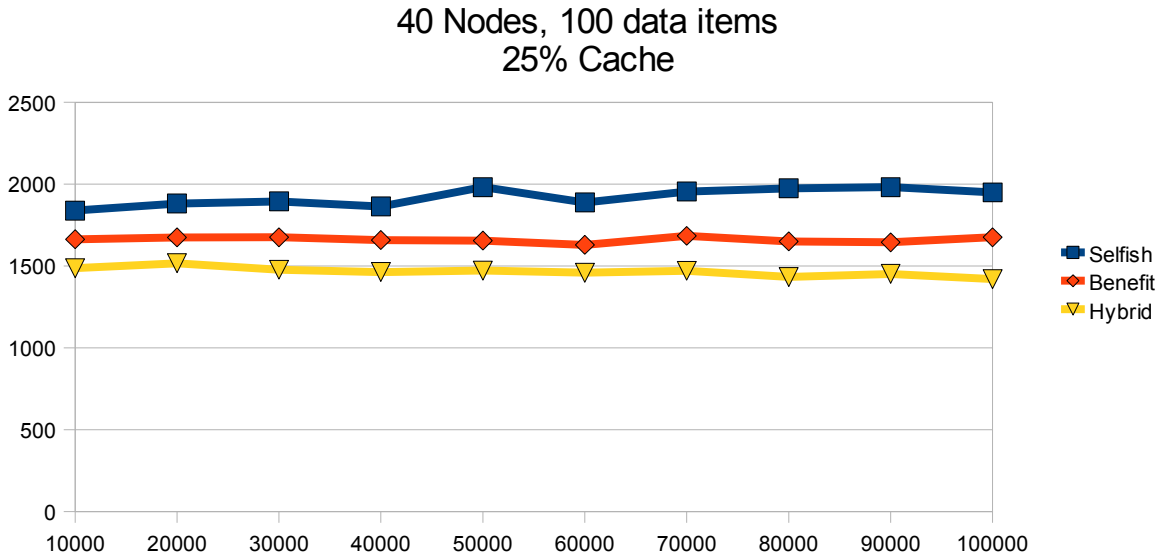


Figure 18: Total access cost at 25% cache availability.

# CHAPTER 6

# CONCLUSION AND FUTURE WORK

## 6.1 Conclusion

In this thesis, we studied the steady-states of distributed data caching in wireless ad hoc networks. Unlike previous results showing the superiority of cooperative caching to the selfish caching, this work compares three different distributed data strategies and how the storage capacity of each node in the network can affect the total access cost across the network. This thesis also demonstrates that the average data access cost, based on the cache placement at steady-states, is comparable for the selfish, cooperative, and hybrid caching. We also formulated and solved the data caching problem optimally using integer linear programming and compared the optimal solution with the distributed caching techniques. We have shown empirically that the optimal replica number not only depends on the access frequencies of data items, but also depends on the storage capacity of each node.

## 6.2 Future Work

Thus far, there has been no research, empirically or theoretically, which shows a correlation between the number of copies of any particular data item and the storage capacity of the nodes in the network. In our work, we have shown that there is a correlation between the overall cost of accessing data in the network and the number or frequency of data items available in the networks nodes. In the future, we would like to use our empirical data in order to find a mathematical definition of the relationship we have displayed in this thesis. We would also like to study different network configurations to examine the scalability of the models. And, we

would like to consider a multi-phasic Integer Linear Programming approach to recalculate the optimal solution.

# REFERENCES

REFERENCES

[1] I. Baev. Approximation algorithms for data placement problems. SIAM J. Comput., 38(4):1411–1429, 2008.

[2] I. Baev and R. Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *Proc. of ACM-SIAM SODA*, 2001.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proc. of INFOCOM*, 1999.

[4] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *Proc. of IEEE FOCS* 1999.

[5] F.A. Chudak and D. Shmoys. Improved approximation algorithms for a capacitated facility location problem. *Lecture Notes in Computer Science*, 1610, 1999.

[6] E. Cohen and S. Shenkar. Replication strategies in unstructured peer-to-peer networks. In *Proc. of ACM SIGCOMM*, 2002.

[7] Asit Dan and Dan Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *Proc. of ACM SIGMETRICS,* 1990.

[8] N. Dimokasa, D. Katsarosb, and Y. Manolopoulosa. Cache consistency in wireless multimedia sensor networks. *Ad Hoc Networks*, 8(2):214–240, 2010.

[9] Yu Du, Sandeep K. S. Gupta, and Georgios Varsamopoulos. Improving on-demand data access efficiency in manets with cooperative caching. *Ad Hoc Networks*, 7(3):579–598, 2009.

[10] K. Fall and K. Varadhan (Eds.). The *ns manual.* available from http://www-mash.cs.berkeley.edu/ns/, cited: March 15, 2011.

[11] Xiaopeng Fan, Jiannong Cao, and Weigang Wu;. Contention-aware data caching in wireless multihop ad hoc networks. In *Proc. of IEEE 6th International Conference on Mobile Adhoc and Sensor Systems (MASS '09),* pages 1–9, 2009.

[12] Marco Fiore, Francesco Mininni, Claudio Casetti, and Carla-Fabiana Chiasserini. To cache or not to cache. In *Proc. of IEEE INFOCOM 2009.*

[13] Philippe Flajolet, Daniele Gardy, and Loys Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics,* 39:207–229, 1992.

[14] Takahiro Hara and Sanjay K. Madria. Data replication for improving data accessibility in ad hoc networks. *IEEE Transactions on Mobile Computing*, 5(11):1515–1532, 2006.

[15] K. Jain and V. V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *Journal of the ACM*, 48(2):274 – 296, 2001.

[16] Shudong Jin and Limin Wang. Content and service replication strategies in multi-hop wireless mesh networks. In *Proc. of the ACM MSWiM 2005*, pages 79–86.

[17] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proc. of the 16th international conference on Supercomputing* (ICS '02), pages 84–95, 2002.

[18] Mirko Montanari, Riccardo Crepaldi, Indranil Gupta, and Robin Kravets. Using failure models for controlling data availability in wireless sensor networks. In *Proc. of IEEE Infocom Minisymposium*, 2009.

[19] David Starobinski and David N. C. Tse. Probabilistic methods for web caching. *Performance Evaluation*, 46(2-3):125–137, 2001.

[20] Bin Tang, Samir Das, and Himanshu Gupta. Benefit-based data caching in ad hoc networks. *IEEE Transactions on Mobile Computing*, 7(3):289–304, 2008.

[21] Weigang Wu, Jiannong Cao, and Xiaopeng Fan. Overhearing-aided data caching in wireless ad hoc networks. In Proc. of the 2009 29th *IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW '09),* pages 137–144, 2009.

[22] Liangzhong Yin and Guohong Cao. Supporting cooperative caching in ad hoc networks. *IEEE Transactions on Mobile Computing*, 5(1):77–89, 2006.

[23] G. K. *Zipf. Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison-Wesley, 1949.

# APPENDIX

## APPENDIX

## Source Code

**utils.h**

```
#ifndef UTILS_H
#define UTILS_H


#define mSetNull(a)    (memset(a, '\0', sizeof(a)))
#define mSetZero(a)    (memset(a, 0, sizeof(a)))


typedef enum flag_types
  {T_FILE, N_FILE, B_FILE}
  FLAG_TYPES;

typedef struct flag_type
{
  char *flag;
  char *name;
  char *desc;
}FLAG_TYPE;


void rm_nl(char *buf);
void usage();

#endif
```

**utils.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

//local libraries
#include "prog_defs.h"
#include "utils.h"


void rm_nl(char *buf)
{
  while(*buf != '\n' && *buf != '\0')
```

```
    buf++;

  *buf = '\0';
  return;
}

void usage()
{
  int i;
  FLAG_TYPE flag_table[] =
    {
      {"-t", "topology file", "Specify a topology file name"},
      {"-f", "simulation file", "Specify a simulation data file name"},
      {NULL, NULL, NULL}
    };

  printf("\nProper usage\n");
  for( i = 0; flag_table[i].flag; i++)
    printf("[%s %s] ", flag_table[i].flag, flag_table[i].name);

  printf("\n\n");

  for( i=0; flag_table[i].flag; i++ )
    printf(" %3s %-20s %s\n", flag_table[i].flag, flag_table[i].name,
              flag_table[i].desc);

  return;
}
```

**prog_defs.h**

```
#ifndef PROG_DEFS_H
#define PROG_DEFS_H
#include <stdio.h>
#include <stdbool.h>

#define STRLEN 200
#define MAX_CACHE 100
#define MAX_DATA 100
#define MAX_NODES 40
#define HYBRID true
#define PACKET_SIZE 20

typedef struct hop_table
```

```
{
  int table[MAX_NODES][MAX_NODES];
}HOP_TABLE;

typedef struct node_table
{
  int node;
  int cache[MAX_CACHE];
  double cost[MAX_CACHE];
  int size[MAX_CACHE];
}NODE_TABLE;

typedef struct data_table
{
  int data;
  int nodes[MAX_NODES];
}DATA_TABLE;

typedef struct cost_type
{
  double cost[MAX_DATA];
}COST_TABLE;

typedef struct frequency
{
  int freq[MAX_DATA];
}FREQ_TABLE;

typedef struct access_table
{
  int id;
  double zipf;
}ACCESS_TABLE;

HOP_TABLE* createHopTable(void);
HOP_TABLE* deleteHopTable(HOP_TABLE *table);
void initNodes(NODE_TABLE *node);
void initData(DATA_TABLE *data);
void initCost(COST_TABLE *cost);
void initFreq(FREQ_TABLE *freq);
bool processHopTable( FILE *top_file, HOP_TABLE *table);
bool processNodeFile(FILE *infile, NODE_TABLE *node_table,
                     DATA_TABLE *data_Table);
```

```
bool processAccessFile(FILE *access_file, ACCESS_TABLE *access_table);
void parseHopField(char *arg, char *command, int *origin,
                   int *dest, int *hops);
void parseNodes(char *buf, char *command, int *node, int *data, int *size);
void parseAccessFile( char *buf, int *id, double *zipf);
void printHop(HOP_TABLE *table);
void printNodeTable(NODE_TABLE *table);
void printDataTable(DATA_TABLE *table);
void printCost(NODE_TABLE *node_table, ACCESS_TABLE *access_table);
void totalAccessCost(ACCESS_TABLE *access_table);
void printAccessTable(ACCESS_TABLE *access_table);
void printFrequency( FREQ_TABLE * freq, NODE_TABLE *node_table);
void calculateNodeCost(HOP_TABLE *hop_table, DATA_TABLE *data_table,
                       NODE_TABLE *node_table, ACCESS_TABLE *access_table);
void accessFrequency(FREQ_TABLE *freq_table,DATA_TABLE *data_table);
void totalcost(NODE_TABLE *node_table);
void printHybrid(NODE_TABLE *node_table);

#endif
```

**topology.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>
#include "prog_defs.h"
#include "utils.h"

int main(int argc, char *argv[])
{

  extern char *optarg;
  extern int optind;
  int c;
  HOP_TABLE *hop_table = createHopTable();
  NODE_TABLE node_table[MAX_NODES];
  DATA_TABLE data_table[MAX_DATA];
  COST_TABLE cost_table;
  FREQ_TABLE freq_table;
  ACCESS_TABLE access_table[MAX_DATA];

  initNodes(node_table);
```

```
initData(data_table);
initCost(&cost_table);
initFreq(&freq_table);

FILE *top_file = NULL;
FILE *sim_file = NULL;
FILE *access_file = NULL;

//no arguments
while(( c = getopt(argc, argv, "t:f:") )!=EOF)
  {
   switch(c)
        {
        case 't':
         if(!optarg)
           usage();

         else
           {
            top_file = fopen( optarg, "r");
            if( !top_file)
                 {
                  printf("Error opening %s\n", optarg);
                  printf("Attempting to use default file\n");
                  top_file = fopen( "/home/julie/curtop/topology/40_nodes_files/40_5", "r");
                  if(!top_file)
                    {
                     printf("Error opening default file. Exiting.\n");
                     exit(1);
                    }
                 }
           }
         break;
        case 'f':
         if(!optarg)
           usage();
         else
           {
            sim_file = fopen(optarg, "r");
            if(!sim_file)
                 {
                  printf("Error opening file %s\n", optarg);
                  printf("Attempting to use default file\n");
```

```
                    sim_file = fopen("node_print_benefit.txt", "r");
                    if(!sim_file)
                      {
                       printf("Error opening default file.  Exiting\n");
                       exit(1);
                      }
                   }
              }//else
             break;
           default:
             usage();

           }//switch
    }//while
// no arguments
  if( argc == 1)
    {
     printf("Using default topology file\n");
     top_file = fopen( "/home/julie/curtop/topology/40_nodes_files/40_1", "r");
     if( !top_file)
          {
           printf("Unable to open default file\n");
           usage();
           exit(1);
          }
    }
  char filename[200] = {'\0'};
  int time = 10000;
  while( time <= 20000)
    {
     initNodes(node_table);
     initData(data_table);
     initCost(&cost_table);
     initFreq(&freq_table);

     sprintf(filename, "/home/julie/curtop/cost_tests/safe/hybrid/random/100/40_1/%dsecs.txt",
time);
     sim_file = fopen(filename, "r");
     if(!sim_file)
          {
           printf("Unable to open simulation file\n");
           usage();
           exit(1);
```

```
        }
    access_file = fopen("/home/julie/curtop/topology/zipf_test.txt", "r");

    if( !access_file)
        {
          printf("Unable to open access file\n");
          usage();
          exit(1);
        }
    //Process files
    processHopTable( top_file, hop_table);
    processNodeFile(sim_file, node_table, data_table);
    processAccessFile(access_file, access_table);
    calculateNodeCost(hop_table, data_table,node_table, access_table);
    accessFrequency(&freq_table, data_table);

    printf("Time is: %d    ", time);
    totalcost(node_table);
    time += 1000;
    if(sim_file)
    {
        fclose(sim_file);
        sim_file = NULL;
    }
  }//while
 //clean up and exit

 if(top_file)
   fclose(top_file);
 if(sim_file)
   fclose(sim_file);
 if(access_file)
   fclose(access_file);
 if(hop_table)
   deleteHopTable(hop_table);
 exit(0);

}

//Constructors/destructors

HOP_TABLE* createHopTable()
{
```

```
  HOP_TABLE* table = (HOP_TABLE*)calloc( 1, sizeof(HOP_TABLE));

  if(!table)
    {
     printf("Unable to allocate memory for topology table.  Exiting.\n");
     exit(2);
    }
  return table;
}

HOP_TABLE *deleteHopTable(HOP_TABLE *table)
{
  if(table)
    free(table);
  return NULL;
}

void initNodes(NODE_TABLE *nodes)
{
  int i=0, j=0;
  for( i=0; i < MAX_NODES; i++ )
    {
     nodes[i].node = -1;
     for( j= 0; j < MAX_CACHE; j++ )
          nodes[i].cache[j] = -1;
     for(j=0; j<MAX_CACHE; j++)
          nodes[i].cost[j] = -1;
     for(j=0; j<MAX_CACHE; j++)
          nodes[i].size[j] = -1;
    }
  return;
}

void initData(DATA_TABLE *data)
{
  int i, j;
  for( i=0; i<MAX_DATA; i++ )
    {
     data[i].data = -1;
     for(j=0; j<MAX_NODES; j++)
          data[i].nodes[j] = -1;
    }
  return;
```

```
}

void initCost(COST_TABLE *cost)
{
  int i;
  for(i=0; i< MAX_DATA; i++)
    {
      cost->cost[i] = 0;
    }
}

void initFreq(FREQ_TABLE *freq)
{
  int i;
  for(i=0; i<MAX_DATA; i++)
    {
      freq->freq[i] = 0;
    }
}
bool processHopTable( FILE *topfile, HOP_TABLE *table)
{
  char buf[STRLEN];
  char command[STRLEN];
  //clear out buffers
  mSetNull(buf);
  mSetNull(command);
  int origin, dest, hops;

  //read in and process data
  while( fgets( buf, STRLEN, topfile))
    {
      rm_nl(buf);
      parseHopField(buf, command, &origin, &dest, &hops);
      if(!strcmp(command, "god"))
            {
              if( origin == dest )
                table->table[origin][dest] = 0;
              else
                {
                  table->table[origin][dest] = hops;
                  table->table[dest][origin] = hops;
                }
            }
```

```
    mSetNull(buf);
    mSetNull(command);
   }
  return true;
}

bool processNodeFile(FILE *infile, NODE_TABLE *node_table, DATA_TABLE *data_table)
{
 char buf[STRLEN];
  char command[STRLEN];
  //clear out buffers
  mSetNull(buf);
  mSetNull(command);
  int previous_node = -1;
  int current_node = -1;
  int current_data= -1;
  int current_size = -1;
  int position=0;
  int i=0, j=0;
  //read in and process data

  while( fgets( buf, STRLEN, infile))
    {
     rm_nl(buf);
     parseNodes(buf, command, &current_node, &current_data, &current_size);

     if( current_node >= 0 && current_node > previous_node)
          {
           node_table[position++].node = current_node;
           i=0;
           previous_node = current_node;
          }
     if(current_node >= 0 && current_data >= 0 )
          {
           node_table[current_node].cache[current_data] = current_data;
           node_table[current_node].size[current_data] = current_size;
           //make data table
           data_table[current_data].data = current_data;
           for(j=0; j<MAX_NODES;)
             {
              if(data_table[current_data].nodes[j] < 0)
                   {
                    data_table[current_data].nodes[j++] = current_node;
```

```
                    break;
                  }
              else
                  j++;
            }
          }
      //clean buffers
      mSetNull(buf);
      mSetNull(command);

    }
    return true;
}

 bool processAccessFile(FILE *access_file, ACCESS_TABLE *access_table)
 {
   char buf[STRLEN];
   int id =-1;
   double zipf=0;
   mSetNull(buf);
   int i=0;
   while(fgets(buf,STRLEN, access_file))
     {
      rm_nl(buf);
      parseAccessFile(buf, &id, &zipf);
      if( id == 0 )
           i = 0;
      if ( id >= 0 && i < MAX_DATA)
           {
            access_table[i].id = id;
            access_table[i++].zipf = zipf;
           }
     }
   return true;
 }

void parseHopField( char *arg, char *command, int *origin, int *dest
                        , int *hops)
{
  int i=0;
  char buf[STRLEN];
  mSetNull(buf);
  //get past the $
```

```
  arg++;
  while( *arg != '_' && *arg != '\0')
    command[i++] = *arg++;

  if( !strcmp(command, "god"))
    {
      while( !isdigit(*arg))
           arg++;
      i = 0;

      while( !isspace(*arg) && *arg != '\0')
           {
            buf[i++] = *arg;
            arg++;
           }
      *origin = atoi(buf);
      //get past the white space
      arg++;
      mSetNull(buf);
      i = 0;
      while(!isspace(*arg) && *arg != '\0' )
           {
            buf[i++] = *arg;
            arg++;
           }
      arg++;
      *dest = atoi(buf);
      i = 0;
      mSetNull(buf);
      while(!isspace(*arg) && *arg != '\0')
           {
            buf[i++] = *arg;
            arg++;
           }
      *hops = atoi(buf);
    }
  return;
}

void parseNodes( char *arg, char *command, int *node, int *data, int *size)
{
  int i=0;
  char buf[STRLEN];
  mSetNull(buf);

  while( !isspace(*arg) && *arg != '\0')
```

```
   command[i++] = *arg++;

  if(!strcmp( command, "Current") || *arg == '\0')
    {
          *node = -1;
          *data = -1;
     return;
    }
  if(!strcmp(command, "Node"))
    {
     i=0;
     arg+=4;
     while( *arg != ',' && *arg != '\0')
          buf[i++] = *arg++;
     *node = atoi(buf);

     return;
    }
  if(!strcmp(command, "ID"))
    {//reset
     mSetNull(buf);
     i=0;
     //skip space
     arg++;
     while(*arg != ',' && *arg != '\0')
          buf[i++] = *arg++;
     *data = atoi(buf);
     //Read the size put in for hybrid cache
     //skip comma
     mSetNull(buf);
     arg += 7;
     i=0;
     while(*arg != '\0')
          buf[i++] = *arg++;
     *size = atoi(buf);
     return;
    }
  return;
}

void parseAccessFile(char *arg, int *id, double *zipf)
{
  char buf[STRLEN];
```

```
  mSetNull(buf);
  int i=0;

  if(arg[0] == '0')
    {
     buf[0] = *arg;
     *id = atoi(buf);
     *zipf = 10;
     return;
    }

  while(!isspace(*arg) && *arg != '\0')
    buf[i++] = *arg++;

  if(!strcmp(buf,"Probability"))
    {
     mSetNull(buf);
     i = 0;
     arg +=6;
     while(isdigit(*arg) && *arg!=':')
          buf[i++] = *arg++;
     *id = atoi(buf);
     while(isspace(*arg))
          arg++;
     mSetNull(buf);
     i=0;
     arg +=2;
     *zipf = atof(arg);
    }
  return;
}

void printHop(HOP_TABLE *table)
{
  int i, j, k=0;

  for(i=0; i<MAX_NODES; i++ )
    {
     for(j=0; j<MAX_NODES; j++, k++)
     {
          if(k == 10)
          {
             printf("\nNode %d", i);
```

```
            k=0;
          }
          printf(" %3d ", table->table[i][j]);
     }
     printf("\n");
   }

  return;
}

void printNodeTable(NODE_TABLE *table)
{
  int i, j;
  double sum = 0;
  for( i=0; i<MAX_NODES; i++)
    {
     printf("\nNode id %d\n", table[i].node);
     for(j=0; j<MAX_CACHE; j++)
          {
           // if( table[i].cache[j] >= 0 )
            printf("Data ID %d Size is %d cost is %lf\n", table[i].cache[j], table[i].size[j],
                    table[i].cost[j]);
                    sum += table[i].cost[j];

          }
          printf("Node %d cost is %lf\n", i, sum);
      sum = 0;
   }
          printf("\n\n\n");
  return;
}

void printDataTable(DATA_TABLE *table)
{
  int i,j, num;
  for(i=0; i< MAX_DATA; i++ )
    {
     num = 0;
     printf("ID is %d\n", table[i].data);

     for(j=0; j<MAX_NODES; j++ )
          {
           if(table[i].nodes[j] >=0)
```

```
            {
             printf("Cached nodes are %d\n", table[i].nodes[j]);
             num++;
            }
          }
     printf("%d nodes cached\n", num);
     printf("\n\n");
   }
  return;
}

void printCost(NODE_TABLE *node_table, ACCESS_TABLE *access_table)
{
  int i,j;
  double node_cost = 0;
  int cached = 0, uncached = 0;
  double cached_savings = 0;

  for(i=0; i< MAX_NODES; i++)
    {
     for( j=0; j<MAX_DATA; j++)
          {
           if( node_table[i].cost[j] > 0 )
             {
             printf("Node %d Id %d Cost is %.5lf Size is %d\n",
                      i,j, node_table[i].cost[j], node_table[i].size[j]);
              node_cost += node_table[i].cost[j];
              uncached++;

             }
             else
             {
              // printf("Node %d Id %d Cost is %.5lf Size is %d\n",
          //             i,j, access_table[j].zipf, node_table[i].size[j]);
                        cached++;
                        cached_savings += access_table[j].zipf;
             }
          }
    // printf("Total Node cost for node %d is %lf\n", i, node_cost);
  //  printf("Total Node cost for cached node %d is %.5lf\n", node_table[i].node,
cached_savings);
    printf("Uncached = %d Cached = %d\n", uncached, cached);
    node_cost = 0;
```

```
      uncached = 0;
      cached = 0;
    }
    printf("Total node cost %.5lf is %.5lf\n", node_cost, cached_savings);
}

void printFrequency( FREQ_TABLE * freq, NODE_TABLE *node_table)
{
  int i;
  int count = 0;
  for(i=0; i<MAX_DATA; i++)
  {
    printf("Data %d is %d frequent\n", i, freq->freq[i]);
      count += freq->freq[i];
  }
  printf("Total data cached %d\n", count);
  return;
}
void printAccessTable(ACCESS_TABLE * table)
{
  int i;
  for( i = 0; i < MAX_DATA; i++ )
    printf("Access Table %lf\n", table[i].zipf);
  return;
}
void calculateNodeCost(HOP_TABLE *hop_table, DATA_TABLE *data_table,
                       NODE_TABLE *node_table,  ACCESS_TABLE *access_table)
{
  int current_node;
  int current_data;
  int target_node;
  double hop_cost = -1;
  double access_cost;
  int hops;
  //find first data
  for( current_node=0; current_node < MAX_NODES; current_node++ )
    {
     for(current_data =0; current_data < MAX_DATA; current_data++ )
          {
           //find the access cost
           access_cost = access_table[current_data].zipf;
           for( target_node = 0; target_node < MAX_NODES; target_node++)
             {
```

```
            if(node_table[current_node].size[current_data] == PACKET_SIZE)
            {
                //printf("Node %d has cached data %d at 20\n", current_node, current_data);
            }
            else if(node_table[current_node].cache[current_data] >= 0
                 && node_table[current_node].size[current_data] !=
                                                    PACKET_SIZE)
                {
                 node_table[current_node].cost[current_data] = 0;
                }
            else if ( hop_table->table[current_node][target_node] > 0 )
                {

                 if( data_table[current_node].nodes[target_node] >= 0)
                 {
                 hops = hop_table->table[current_node][target_node];
                 hop_cost = hops * access_cost;
                 if (node_table[current_node].cost[current_data] < 0 ||
                node_table[current_node].cost[current_data] > hop_cost)
                   node_table[current_node].cost[current_data] = hop_cost;
                 }
                }
            else
            //we don't have a node
                {
                 if(current_data == 0 || current_data % 2 == 0)
                   hops = hop_table->table[current_node][0];
                 else
                   hops = hop_table->table[current_node][1];

                 hop_cost = hops * access_cost;


                if( node_table[current_node].cost[current_data] < 0 ||
node_table[current_node].cost[current_data] > hop_cost)
                {
                 node_table[current_node].cost[current_data] = hop_cost;
                }
                }
            if(node_table[current_node].cost[current_data] == 0 )
                break;

        }
```

```
        }//for
    }
  return;
}

void accessFrequency(FREQ_TABLE *freq, DATA_TABLE *data_table)
{
  int data_item;
  int nodes;

  for( data_item = 0 ; data_item < MAX_DATA; data_item++)
    {
     //source node
     //   if(!HYBRID)
          freq->freq[data_item]++;
     for(nodes = 0; nodes < MAX_NODES; nodes++)
          {
           if( data_table[data_item].nodes[nodes] >= 0 )
             {

               freq->freq[data_item]++;
             }


           /* else
              printf("Node %d is not caching %d value is %d\n",
                      nodes, data_item, data_table[data_item].nodes[nodes]);
           */
          }
    }
  return;

}

void totalcost(NODE_TABLE *node_table)
{
  int node, data;
  double cost=0;
  int count = 0;


  for(node = 0; node < MAX_NODES; node++)
    {
```

```
    for(data = 0; data < MAX_CACHE; data ++ )
        {
         if( node_table[node].cost[data] >= 0 )
           {
            cost += node_table[node].cost[data];
            if (node_table[node].cost[data] == 0)
                count++;
           }
            else
                count++;
        }
    count=0;
   }
  printf("Total cost %lf\n", cost);

}

void printHybrid(NODE_TABLE *node_table)
{
          int sumCache = 0, sumPath = 0, totalcache = 0;
          int sizeOver600=0;
          double percentOfPath;
          int i, j;

          for(i=0; i<MAX_NODES; i++)
          {
                    for( j=0; j<MAX_DATA; j++ )
                    {
                              if(node_table[i].size[j] == PACKET_SIZE)
                                        sumPath++;
                              if( node_table[i].size[j] >= 0 &&
                                        node_table[i].size[j] > 500)
                                                  sumCache++;
                              if(node_table[i].size[j] > 600 )
                                        sizeOver600++;

                    }

          }
          totalcache = MAX_CACHE * MAX_NODES;

 printf("Cache %d, Path %d, Total %d\n", sumCache/MAX_NODES, sumPath/MAX_NODES,
(sumCache + sumPath)/MAX_NODES);
```

```
percentOfPath = (double)sumPath/totalcache;
printf("Percent of Nodes caching a path %lf\n", percentOfPath);

printf("Cached nodes over 600 size %d, percent %lf\n", sizeOver600/40,
(double)sizeOver600/totalcache);
}
```