Stony Brook University

Data Caching in Ad Hoc and Sensor Networks

A Dissertation Presented

by

Bin Tang

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2007

Stony Brook University

The Graduate School

Bin Tang

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation.

Himanshu Gupta -- Dissertation Advisor
Assistant Professor of Computer Science

Samir Das – Chair person of Defense
Associate Professor of Computer Science

Jie Gao -- Dissertation Committee Member
Assistant Professor of Computer Science

Xin Wang -- Outside Member
Assistant Professor of Electrical and Computer Engineering

This dissertation is accepted by the Graduate School

Lawrence Martin

Dean of the Graduate School

Abstract of the Dissertation

Data Caching in Ad Hoc and Sensor Networks

by

Bin Tang

Doctor of Philosophy
in
Computer Science
Stony Brook University
2007

One of the key components of many emerging networks such as ad hoc and sensor networks is data storage/caching. It requires optimized placement of data for efficient access, even when the users of the data are geographically distributed, mobile or have very limited computing and communication capacities. Meanwhile, ad hoc and sensor networks are resource constrained in terms of battery power, memory capacity, etc. In this dissertation work, we address the data caching problem in ad hoc and sensor networks under different constraints.

First, we consider the cache placement problem of minimizing total data access cost in ad hoc networks with multiple data items and nodes with limited memory capacity. The above optimization problem is known to be NP-hard. Defining *benefit* as the reduction in total access cost, we present a polynomial-time centralized approximation algorithm that provably delivers a solution whose benefit is at least one-fourth (one-half for uniform-size data items) of the optimal benefit. The approximation algorithm is amenable to localized distributed implementation, which is shown via simulations to perform close to the approximation algorithm. Our distributed algorithm naturally extends to networks with mobile nodes. We simulate our distributed algorithm using a network simulator (*ns2*), and demonstrate that it significantly outperforms another existing caching technique (by Yin and Cao [56]) in all important performance metrics. The performance differential is particularly large in more challenging scenarios, such as higher access frequency and smaller memory.

Second, we address an optimization problem that arises in the context of cache placement in sensor networks. In particular, we consider the cache placement problem where the goal is to determine a set of nodes in the network to cache/store the given data item, such that the overall communication cost incurred in accessing the item is minimized, under the constraint that the total

communication cost in updating the selected caches is less than a given constant. The update cost constraint is based on the observation that, the updates always originate from the server node, and hence, the server node and the surrounding nodes bear most of the communication cost incurred in updating. Therefore, there is a need to constrain the total update cost incurred in the network, to prolong the lifetime of the server node and the nodes around it – and hence, possibly of the sensor network. In our network model, there is a single server (containing the original copy of the data item) and multiple client nodes (that wish to access the data item). For various settings of the problem, we design optimal, near-optimal, heuristic-based, and distributed algorithms, and evaluate their performance through simulations on randomly generated sensor networks.

Third, we consider the problem of caching a data item in a network wherein the data item is read as well as updated by other nodes and there is a limit on the number of cache nodes allowed. More formally, given a network graph, the read/write frequencies to the data item by each node, and the cost of caching the data item at each node, the problem addressed in this work is to select a set of P nodes to cache the data item such that the sum of the reading, writing (using an optimal Steiner tree), and storage cost is minimized. or networks with a tree topology, we design an optimal dynamic programming algorithm . For the general graph topology, where the problem is NP-complete, we present a centralized heuristic and its distributed implementation. Through extensive simulations in general graphs, we show that the centralized heuristic performs very close to the exponential optimal algorithm for small networks, and for larger networks, the distributed implementation and the dynamic programming algorithm on an appropriately extracted tree perform quite close to the centralized heuristic.

To My Wife, Yutian Chen
And To My Parents

Contents

List of Figures

Acknowledgements

First and foremost I want to thank Dr. Himanshu Gupta, my doctoral advisor, for his inspiration and encouragement. Dr. Gupta taught me how to think clearly and express clearly, which are two most important things I learned, among many others. His support and patience helped me overcome many difficult times of my dissertation research. I hope that one day I would become as good an advisor to my students as Dr. Gupta has been to me.

I am also deeply grateful to Dr. Samir Das. From the day one I joined the WINGS lab, Dr. Das has been actively involved with my research. His insight and comment directly instilled into it and greatly enhanced its quality. I am also very grateful to Dr. Jie Gao. Her insightful comments and constructive suggestions were always thought-provoking, and her encouragement to me will always be appreciated. I would like to thank Dr. Xin Wang for being my committee member and helpful comment on my dissertation. I also appreciate Dr. Jennifer Wong for her enthusiastic help and great suggestions.

I own a special thank you to Dr. Shiyong Lu, a Stony Brook alumnus and a faculty member of Computer Science in Wayne State University. He's been a big brother to me, encouraging and helping me all along the way. Thank you, Shiyong! I also extend my gratitude to my friends in the WINGS lab: Xianjin, Vishnu, Shweta, Anand, Zongheng, Ritesh, Anand Prabhu, Mahmoud. Thank you for making my time in the lab such fun and enjoyable.

I would like to thank deep from my heart my friends from academic advising center for their strong support. I am also indebted to Dr. Miriam Rafailovich, Dr. Michael Kifer and Dr. Yuanyuan Yang for their guidance in the early stage of my Ph.D. study. I am very grateful to Mr. Brian Tria, system administrator and Ms. Kathy Germana, assistant to the Chair, for their long time support.

Most importantly, none of this would have been possible without the love and support of my wife Yutian Chen. All these years, she has been my source of comfort and strength when I was in those lowest moments at Stony Brook. Words can never express enough my appreciation for the love she has given me. Finally, I would like to express my heart-felt gratitude to my parents and my brother, whose unwavering faith and confidence in me is what drives me forward to make this far, and to continue on the road ahead.

Chapter 1

Introduction

Ad hoc networks are multihop wireless networks of small computing devices without the intervening of infrastructure. The computing devices could be conventional computers (e.g., PDA, laptop, or PC) running interactive software applications, or backbone routing platforms without any interactive use, or even embedded processors interfaced with sensors/actuators that directly interface with the physical environment. Sensor networks are wireless ad hoc networks which consist of sensor nodes with short-range radios and limited onboard processing capability, forming a multi-hop network of irregular topology.

Ad hoc networks require minimum setup and administration cost, thus find tremendous use in a wide range of applications. For example, on one end, MANETs or *mobile ad hoc networks* [39] have been considered for impromptu conferencing and various tactical applications such as law enforcement, search-and-rescue, military, and remote explorations. On the other end, wireless networks of microcontroller-based sensor/actuator nodes have been considered for *sensor networking* and *robot swarms*. Applications of sensor networking are plentiful [3] – from habitat or environment monitoring to remote surveillance. Recently, *mesh networks* of wireless capable routers are also under active study for applications such as community or enterprise networking [2].

However, both ad hoc networks and sensor networks face challenges. First, the nodes are usually powered by small batteries, making energy efficiency a critical design goal. Second, the scarcity of wireless bandwidth seriously affect the efficient operation of ad hoc and sensor networks. Third, the nodes in such networks usually have very limited memory capacity and processing power.

Caching has been a widely used effective technique in the web environment [8, 10, 43] and peer-to-peer networks [19, 25, 38] to alleviate problems such as server overloading, delayed respond time, and inadequate bandwidth. However, relatively less work has been done on the cache placement problem in the specific context of ad hoc networks and sensor networks. In this dissertation

work, we specifically address the challenges faced by ad hoc and sensor networks and study how caching can improve the functionalities of such networks. We have developed a paradigm of data caching techniques to support effective data access in ad hoc networks. We also focus on designing caching techniques to conserve energy in the network by caching data items at selected sensor nodes in a sensor network.

In particular, we address the data caching problem in ad hoc and sensor networks under different constraints: update cost constraint [49, 51] , memory capacity constraint [50] and number of allowable caches constraint [24]. We have developed optimal, near optimal centralized algorithms in either tree networks or general networks. We also present their distributed implementations. The thesis is organized as follows. In Chapter 2, we address the data caching in ad hoc networks with memory constraint. In Chapter 3, we address the data caching under update cost constraint in sensor networks. We study the data caching under number constraint in Chapter 4. In Chapter 5 we conclude by discussing the future work.

Chapter 2

Data Caching Under Memory Constraint

2.1 Introduction

Ad hoc networks are multihop wireless networks of small computing devices with wireless interfaces. The computing devices could be conventional computers (e.g., PDA, laptop, or PC) or backbone routing platforms, or even embedded processors such as sensor nodes. The problem of optimal placement of caches to reduce overall cost of accessing data is motivated by the following two defining characteristics of ad hoc networks. Firstly, the ad hoc networks are multihop networks without a central base station. Thus, remote access of information typically occurs via multi-hop routing, which can greatly benefit from caching to reduce access latency. Secondly, the network is generally resource constrained in terms of channel bandwidth or battery power in the nodes. Caching helps in reducing communication, which results in savings in bandwidth as well as battery energy. The problem of cache placement is particularly challenging when each network node has limited memory to cache data items.

In this paper, our focus is on developing efficient caching techniques in ad hoc networks with memory limitations. Research into data storage, access, and dissemination techniques in ad hoc networks is not new. In particular, these mechanisms have been investigated in connection with sensor networking [29, 44], peer-to-peer networks [1, 31], mesh networks [32], world wide web [43], and even more general ad hoc networks [25, 56]. However, the presented approaches have so far been somewhat "ad hoc" and empirically-based, without any strong analytical foundation. In contrast, the theory literature abounds in analytical studies into the optimality properties of caching and replica allocation problems (see, for example, [6]). However, distributed implementations

of these techniques and their performances in complex network settings have not been investigated. Its even unclear whether these techniques are amenable to efficient distributed implementations. Our goal in this paper is to develop an approach that is both analytically tractable with a provable performance bound in a centralized setting, and is also amenable to a natural distributed implementation.

In our network model, there are multiple data items; each data item has a server, and a set of clients that wish to access the data item at a given frequency. Each node carefully chooses data items to cache in its limited memory to minimize the overall access cost. Essentially, in this article, we develop efficient strategies to select data items to cache at each node. In particular, we develop two algorithms – a centralized approximation algorithm which delivers a 4-approximation (2-approximation for uniform-size data items) solution, and a localized distributed algorithm which is based on the approximation algorithm and can handle mobility of nodes and dynamic traffic conditions. Using simulations, we show that the distributed algorithm performs very close to the approximation algorithm. Finally, we show through extensive experiments on *ns-2* [21] that our proposed distributed algorithm performs much better than prior approach over a broad range of parameter values. Ours is the first work to present a distributed implementation based on an approximation algorithm for the general problem of cache placement of multiple data items under memory constraint.

The rest of the paper is organized as follows. In Section 2.2, we formally define the cache placement problem addressed in this paper, and present an overview of the related work. In Section 2.3, we present our designed centralized approximation and distributed algorithms. Section 4.5 presents simulation results. We end with concluding remarks in Section 3.6.

2.2 Cache Placement Problem

In this section, we formally define the cache placement problem addressed in our article, and discuss related work.

A multi-hop ad hoc network can be represented as an undirected graph $G(V, E)$ where the set of vertices 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. The edge weight may represent a link metric such as loss rate, delay, or transmission power. For the cache placement problem addressed in this article, there are multiple data items and each data item is served by its server (a network node may act as a server for more than one data items). Each network node has limited memory and can cache multiple data items subject to its memory

capacity limitation. The objective of our cache placement problem is to minimize the overall access cost. Below, we give a formal definition of the cache placement problem addressed in this article.

Problem Formulation. Given a general ad hoc network graph $G(V, E)$ with p data items $D_1, D_2, \ldots, D_p$, where a data item D_j is served by a server S_j. A network node may act as a server for multiple data items. *For clarity of presentation*, we assume uniform-size (occupying unit memory) data items for now. Our techniques easily generalize to non-uniform size data items, as discussed later. Each node i has a memory capacity of m_i units. We use a_{ij} to denote the access frequency with which a node i requests the data item D_j, and d_{il} to denote the weighted distance between two network nodes i and l. The *cache placement problem* is to select a *set of sets* $M = \{M_1, M_2, \ldots, M_p\}$, where M_j is a set of network nodes that store a copy of D_j, to minimize the total access cost

$$\tau(G, M) = \sum_{i \in V} \sum_{j=1}^{p} a_{ij} \times min_{l \in (\{S_j\} \cup M_j)} d_{il},$$

under the memory capacity constraint that

$$|\{M_j | i \in M_j\}| \leq m_i \qquad \text{for all } i \in V,$$

which means each network node i appears in at most m_i sets of M. The cache placement problem is known to be NP-hard [6].

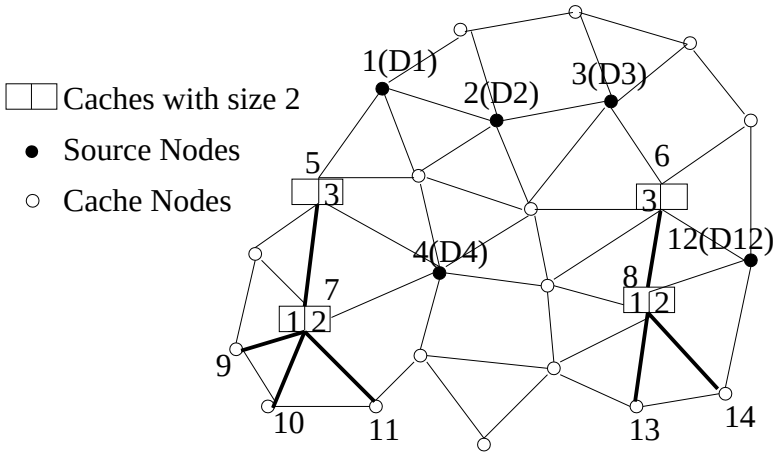


Figure 2.1: Illustrating cache placement problem under memory constraint.

EXAMPLE 1 Figure 2.1 illustrates the above described cache placement problem in a small ad hoc network. In Figure 2.1, each graph edge has a

unit weight. All the nodes have the same memory capacity of 2 pages, and the size of each data item is 1 memory page. Each of the nodes 1, 2, 3, 4, and 12 have one distinct data item to be served (as shown in the parenthesis with their node numbers). Each of the client nodes (9, 10, 11, 13, and 14) accesses each of the data items D_1, D_2, and D_3 with unit access frequency. Figure 2.1 shows that the nodes 5, 6, 7, 8 have cached one or more data items, and also shows the cache contents in those nodes. As indicated by the bold edges, the clients use the nearest cache node instead of the server to access a data item. The set of cache nodes of each data item are: $M_1 = \{7, 8\}, M_2 = \{7, 8\}, M_3 = \{5, 6\}$. One can observe that total access cost is 20 units for the given cache placement. □

2.2.1 Related Work

Below, we categorize the prior work by number of data items and network topology.

Single Data Item in General Graphs. The general problem of determining optimal cache placements in an arbitrary network topology has similarity to two problems in graph theory viz. facility location problem and the k-median problem. Both the problems consider only a single facility type (data item) in the network. In the facility-location problem, setting up a cache at a node incurs a certain fixed cost, and the goal is to minimize the sum of total access cost and the setting-up costs of all caches, without any constraint. On the other hand, the k-median problem minimizes the total access cost under the number constraint, i.e., that at most k nodes can be selected as caches. Both problems are NP-hard, and a number of constant-factor approximation algorithms have been developed for each of the problems [16, 18, 30], under the assumption of triangular inequality of edge costs. Without the triangular inequality assumption, either problem is as hard as approximating the set cover [30, 37] and thus, cannot be approximated better than $O(\log |V|)$ unless $\mathbf{P} = \mathbf{NP}$. Here, $|V|$ is the size of the network. In other related work, Nuggehalli et al. [38] formulate the caching problem in ad hoc networks as a special case of the connected facility location [47].

Single Data Item in Tree Topology. Several papers in the literature circumvent the hardness of the facility-location and k-median problems by assuming that the network has a tree topology [11, 33, 35, 48, 52]. In particular, Tamir [48] and Vigneron et al. [52] design optimal dynamic programming polynomial algorithms for the k-median problem in undirected and directed trees respectively. In other works, Krishnan et al. [35] consider placement of k "transparent" caches, Kalpakis et al. [33] consider a cost model involving reads, writes, and storage, and Bhattacharya et al. [11] present a distributed

algorithm for sensor networks to reduce the total power expended. All of the above works consider only a single data time in a tree network topology.[1]

Multiple Data Items. Hara [25] proposes three algorithms for cache placement of multiple data items in ad hoc networks. In the first approach, each node caches the items most frequently accessed by itself; the second approach eliminates replications among *neighboring* nodes introduced by the first approach; the third approach requires creation of "stable" groups to gather neighborhood information and determine caching placements. The first two approaches are largely localized, and hence, would fare very badly when the percentage of client nodes in the network is low, or the access frequencies are uniform. For the third approach, it is hard to find stable groups in ad hoc networks because of frequent failures and movements. All the above approaches assume the knowledge of access frequencies. In extensions of the above work, [26] and [27] generalize the above approaches for push-based systems and updates respectively. In other related works, Xu et al. [55] discuss placement of "transparent" caches in tree networks.

Our work on cache placement problem is most closely related to the works by Yin and Cao [56] and Baev and Rajaraman [6]. Yin and Cao [56] design and evaluate three simple distributed caching techniques, viz., *CacheData* which caches the passing-by data item, *CachePath* which caches the path to the nearest cache of the passing-by data item, and *HybridCache* which caches the data item if its size is small enough, else caches the path to the data. They use LRU policy for cache replacement. To the best of our knowledge, [56] is the only work that presents a distributed cache placement algorithm in a multihop ad hoc network with memory constraint at each node. Thus, we use the algorithms in [56] as a comparison point for our study.

Baev and Rajaraman [6] design a 20.5-approximation algorithm for the cache placement problem with uniform-size data items. For the non-uniform size data items, they show that there is no polynomial-time approximation unless $\mathbf{P} = \mathbf{NP}$. They circumvent the non-approximability by increasing the given node memory capacities by the size of the largest data item, and generalize their 20.5-approximation algorithm. However, their approach (as noted by themselves) is not amenable to an efficient distributed implementation.

Our Work. In this article, we circumvent the non-approximability of the cache placement problem by choosing to maximize the benefit (*reduction* in total access cost) instead of minimizing the total access cost. In particular, we design a simple centralized algorithm that delivers a solution whose benefit is at least one-fourth (one-half for uniform-size data items) of the optimal benefit

[1] [35] formulates the problem in general graphs, but designs algorithms for tree topologies with single server.

without using any more than the given memory capacities. To the best of our knowledge, ours and [6] are the only[2] works that present approximation algorithms for the general placement of cache placement for *multiple* data items in networks with *memory constraint*. However, as noted before, [6]'s approach is not amenable to an efficient distributed implementation, while our approximation algorithm yields a natural distributed implementation which is localized and shown (using ns2 simulations) to be efficient even in mobile and dynamic traffic conditions. Moreover, as stated in Theorem 2, our approximation result is an improvement over that of [6] when optimal access cost is at least $(1/40)^{th}$ of the total access cost without the caches. Finally, unlike [6], we do not make the assumption of the cost function satisfying the triangular inequality.

2.3 Cache Placement Algorithms

In this section, we first present our centralized approximation algorithm. Then, we design its localized distributed implementation that performs very close to the approximation algorithm in our simulations.

2.3.1 Centralized Greedy Algorithm (CGA)

The designed centralized algorithm is essentially a greedy approach, and we refer to it as CGA (Centralized Greedy Algorithm). CGA starts with all network nodes having all empty memory pages, and then, iteratively caches data items into memory pages maximizing the benefit in a greedy manner at each step. Thus, at each step, the algorithm picks a data item D_j to cache into an empty memory page r of a network node such that the benefit of caching D_j at r is the maximum among all possible choices of D_j and r at that step. The algorithm terminates when all memory pages have been cached with data items.

For formal analysis of CGA, we first define a set of variables A_{ijk}, where selection of a variable A_{ijk} indicates that the k^{th} memory page of node i has been selected for storage of data item D_j, and reformulate the cache placement problem in terms of selection of A_{ijk} variables. Recall that for simplicity we have assumed that each data item is of unit size, and occupies one memory page of a node.

Problem Formulation using A_{ijk}. Given a network graph $G(V, E)$, where each node $i \in V$ has a memory capacity of m_i pages, and p data items $D_1, \ldots, D_p$ in the network with the respective servers $S_1, \ldots, S_p$. Select a set

[2] [4] presents a competitive online algorithm, but uses polylog-factor bigger memory capacity at nodes compared to the optimal.

Γ of *variables* A_{ijk}, where $i \in V$, $1 \leq j \leq p$, $1 \leq k \leq m_i$, and if $A_{ijk} \in \Gamma$ and $A_{ij'k} \in \Gamma$ then $j = j'$, such the total access cost $\tau(G, \Gamma)$ (as defined below) is minimized. Note that the memory constraint is subsumed in the restriction on Γ that if $A_{ijk} \in \Gamma$, then $A_{ij'k} \notin \Gamma$ for any $j' \neq j$. The total access cost $\tau(G, \Gamma)$ for a selected set of variables can be easily defined as:

$$\tau(G, \Gamma) = \sum_{j=1}^{p} \sum_{i \in V} a_{ij} \times min_{l \in (\{S_j\} \cup \{i' | A_{i'jk} \in \Gamma\})} d_{il}.$$

Note that the set of cache nodes M_j that store a particular data item D_j can be easily derived from the selected set of variables Γ.

Centralized Greedy Algorithm (CGA). CGA works by iteratively selecting a variable A_{ijk} that gives the highest "benefit" at that stage. The benefit of adding a variable A_{ijk} into an already selected set of variables Γ is the reduction in the total access cost if the data item D_j is cached into the empty k^{th} memory page of the network node i. The benefit of selecting a variable is formally defined below.

Definition 1 (Benefit of selecting A_{ijk}.) Let Γ denote the set of variables that have been already selected by the centralized greedy algorithm at some stage. The *benefit of a variable* A_{ijk} ($i \in V$, $j \leq p$, $k \leq m_i$) with respect to Γ is denoted as $\beta(A_{ijk}, \Gamma)$ and is defined as follows:

$$\beta(A_{ijk}, \Gamma) = \begin{cases} \text{Undefined} & \text{if } A_{ij'k} \in \Gamma,\ j' \neq j \\ 0 & \text{if } A_{ijk'} \in \Gamma \\ \tau(G, \Gamma) - \tau(G, \Gamma \cup \{A_{ijk}\}) & \text{otherwise} \end{cases}$$

where $\tau(G, \Gamma)$ is as defined before. The first condition of the above definition stipulates that if the k^{th} memory page of the node i is not empty (i.e., has already been selected to store another data item j' due to $A_{ij'k} \in \Gamma$), then the benefit $\beta(A_{ijk}, \Gamma)$ is undefined. The second condition specifies that the benefit of a variable A_{ijk} with respect to Γ is zero if the data item D_j has already been stored at some other memory page k' of the node i. □

Algorithm 1 Centralized Greedy Algorithm (CGA)

BEGIN
  $\Gamma = \emptyset$;
  while (there is a variable A_{ijk} with defined benefit)
    Let A_{ijk} be the variable with maximum $\beta(A_{ijk}, \Gamma)$.
    $\Gamma = \Gamma \cup \{A_{ijk}\}$;
  end while;
  RETURN Γ;
END. ◊

The total running time of CGA is $O(p^2|V|^3\overline{m})$, where $|V|$ is the size in the network, $\overline{m}$ is the average number of memory pages in a node, and p is the total number of data items. Note that the number of iterations in the above algorithm is bounded by $|V|\overline{m}$, and at each stage we need to compute at most pV benefit values where each benefit value computation may take $O(pV)$ time.

Theorem 1 *CGA (Algorithm 1) delivers a solution whose total benefit is at least half of the optimal benefit.*

Proof: Let L be the total number of iterations of CGA. Note that L is equal to the total number of memory pages in the network. Let Γ_l be the set of variables selected at the end of l^{th} iteration, and let ζ_l be the variable added to the set Γ_{l-1} in the l^{th} iteration. Let ζ_l be a variable A_{ijk} signifying that in the l^{th} iteration CGA decided to store j^{th} data item in the k^{th} memory page of the i node. Without loss of generality, we can assume that the optimal solution also stores data items in all memory pages. Now, let λ_l be the variable $A_{ij'k}$ where j' is the data item stored by the optimal solution in the k^{th} memory page of node i. By the greedy choice of ζ_l, we have

$$\beta(\zeta_l, \Gamma_{l-1}) \geq \beta(\lambda_l, \Gamma_{l-1}), \qquad \forall l \leq L. \tag{2.1}$$

Let O be the optimal benefit,[3] and C be the benefit of the CGA solution. Note that[4]

$$C = \sum_{l=1}^{L} \beta(\zeta_l, \Gamma_{l-1}). \tag{2.2}$$

Now, consider a modified network G' wherein each node i has a memory capacity of $2m_i$. We construct a cache placement solution for G' by taking a union of data items selected by CGA and data items selected in an optimal solution for each node. More formally, for each variable $\lambda_l = A_{ij'k}$ as defined above, create a variable $\lambda'_l = A_{ij'k'}$ where $k' = m_i + k$. Obviously, the benefit O' of the set of variables $\{\zeta_1, \zeta_2, \ldots, \zeta_L, \lambda'_1, \lambda'_2, \ldots, \lambda'_L\}$ in G' is greater than or equal to the optimal benefit O in G. Now, to compute O', we add the variables in the order of $\zeta_1, \zeta_2, \ldots, \zeta_L, \lambda'_1, \lambda'_2, \ldots, \lambda'_L$ and add up the benefits of each newly added variable. Let $\Gamma'_l = \{\zeta_1, \zeta_2, \ldots, \zeta_L\} \cup \{\lambda_1, \lambda_2, \ldots, \lambda_l\}$, and

[3]Note that a solution with optimal benefit also has optimal access cost.

[4]Note that $O \neq \sum_{l=1}^{L} \beta(\lambda_l, \Gamma_{l-1})$. Also, in spite of (2.2), the benefit value C is actually independent of the order in which ζ_l are selected.

recall that $\Gamma_l = \{\zeta_1, \zeta_2, \ldots, \zeta_l\}$. Now, we have

$$\begin{aligned} O &\leq O' = \sum_{l=1}^{L} \beta(\zeta_l, \Gamma_{l-1}) + \sum_{l=1}^{L} \beta(\lambda'_l, \Gamma'_{l-1}) \\ &= C + \sum_{l=1}^{L} \beta(\lambda'_l, \Gamma'_{l-1}) \qquad \text{From (2)} \\ &\leq C + \sum_{l=1}^{L} \beta(\lambda_l, \Gamma_{l-1}) \quad \text{Since } \lambda_l = \lambda'_l,\ \Gamma_{l-1} \subseteq \Gamma'_{l-1} \\ &\leq 2C \qquad \text{From (1) and (2)} \end{aligned}$$

■

The following theorem follows from the above theorem and the definition of benefit, and shows that our above result is an improvement of the 20.5-approximation result of [6] when the optimal access cost is at least $(1/40)^{th}$ of the total access cost without the caches.

Theorem 2 *If the access cost without the caches is less than 40 times the optimal access cost using optimal cache placement, then the total access cost of the CGA solution is less than 20.5 times the optimal access cost.*

Proof: Let the total access cost without the caches be W, and the optimal access cost (using optimal cache placement) be O. Thus, the optimal benefit is $W - O$. Since the benefit of the CGA solution is at least half of the optimal benefit, the total access time of the CGA solution is at most $W - (W - O)/2$ which is at most $20.5O$. ■

Non-uniform Size Data Items. To handle non-uniform size data items, at each stage, CGA selects a data item to cache at a node such that the (data item, node) pair has the maximum benefit per page at that stage. CGA continues to cache data items at nodes in the above manner until each node's memory is *exceeded* by the last data item cached. Let S be the solution obtained at the end of the above process. Now, CGA picks the better of the following two feasible solutions: (S_1) Each node caches only its last data item, (S_2) Each node caches all the selected data items except the last. For the above solutions to be feasible, we assume that size of the largest data item in the system is less than the memory capacity of any node. Below, we show that the better of the above two solutions has a benefit of at least 1/4 of the optimal benefit.

Theorem 3 *For non-uniform size data items, the above described modified CGA algorithm delivers a solution whose benefit is at least one fourth of the optimal benefit. We assume that the size of the largest data time is at most the size of any node's memory capacity.*

Proof: First, note that since the solution S is a union of solutions S_1 and S_2, the benefit of either S_1 or S_2 is at least half of the benefit of S. We prove the theorem by showing below that the benefit of S is at least half of the optimal benefit. The below proof is similar to that of Theorem 1.

As before, we define variables A_{ijk} signifying that (part of) the data item D_j was stored in the k^{th} memory page of node i. However, note that A_{ijk} only corresponds to a *unit* memory page, while a data item may occupy more than one memory page. Thus, a cache placement solution may select one or more variables A_{ijk} with the same i and j. Moreover, in this context, the benefit $\beta(A_{ijk}, \Gamma)$ is defined as the benefit *per unit space* of caching D_j at node i, when the certain data items have already been cached at nodes (as determined by the variables in $\Gamma - \{A_{ij*}\}$).

Now, let L' be the total amount of memory used by the solution S. Note that L' may be more than the total number of memory pages in the network. As in Theorem 1, let Γ_l be the set of variables selected at the end of l^{th} iteration, and let ζ_l be the variable added to the set Γ_{l-1} in the l^{th} iteration. Thus, the solution S is the set of variables $\{\zeta_1, \zeta_2, \ldots, \zeta_{L'}\}$. Similarly, let the optimal solution be $\{\lambda_1, \lambda_2, \ldots, \lambda_L\}$, where λ_l corresponds to the same node and memory page as ζ_l. Without loss of generality, we assume the optimal solution is "completely" different than S. That is, there is no data item that is cached by S as well as the optimal solution at the same node.[5] As in Theorem 1, by the greedy choice of ζ_l and the above assumption of completely different solutions, we have

$$\beta(\zeta_l, \Gamma_{l-1}) \geq \beta(\lambda_l, \Gamma_{l-1}), \qquad \forall l \leq L. \tag{2.3}$$

Now, consider a modified network G' wherein each node i has a memory capacity of $2m_i$. We construct a cache placement solution for G' by taking a union of the solution S and the optimal solution at each node. More formally, for each variable $\lambda_l = A_{ij'k}$ as defined above, create a variable $\lambda'_l = A_{ij'k'}$ where $k' = m'_i + k$, where m'_i is the memory used by the solution S at node i. Obviously, the benefit O' of the set of variables $\{\zeta_1, \zeta_2, \ldots, \zeta_L, \lambda'_1, \lambda'_2, \ldots, \lambda'_L\}$ in G' is greater than or equal to the optimal benefit. Now, to compute O', we add the variables in the order of $\zeta_1, \zeta_2, \ldots, \zeta_L, \lambda'_1, \lambda'_2, \ldots, \lambda'_L$ and add up the benefits of each newly added variable. Let $\Gamma'_l = \{\zeta_1, \zeta_2, \ldots, \zeta_L\} \cup \{\lambda_1, \lambda_2, \ldots, \lambda_l\}$, and recall that $\Gamma_l = \{\zeta_1, \zeta_2, \ldots, \zeta_l\}$. Let C be the benefit of solution S. Now, we

[5]Else, we could remove the common data items from both solutions and prove the below claim about the remaining solutions.

have

$$\begin{aligned} O &\leq O' = \sum_{l=1}^{L'} \beta(\zeta_l, \Gamma_{l-1}) + \sum_{l=1}^{L} \beta(\lambda'_l, \Gamma'_{l-1}) \\ &= C + \sum_{l=1}^{L} \beta(\lambda'_l, \Gamma'_{l-1}) \\ &\leq C + \sum_{l=1}^{L} \beta(\lambda_l, \Gamma_{l-1}) \quad \text{Since } \lambda_l = \lambda'_l,\ \Gamma_{l-1} \subseteq \Gamma'_{l-1} \\ &\leq 2C \qquad\qquad \text{From (1) and (2)} \end{aligned}$$

■

2.3.2 Distributed Greedy Algorithm (DGA)

In this subsection, we describe a localized distributed implementation of CGA. We refer to the designed distributed implementation as DGA (Distributed Greedy Algorithm). The advantage of DGA is that it adapts to dynamic traffic conditions, and can be easily implemented in an operational (possibly, mobile) network with low communication overheads. While we cannot prove any bound on the quality of the solution produced by DGA, we show through extensive simulations that the performance (in terms of the quality of the solution delivered) of the DGA is very close to that of the CGA. The DGA is formed of two important components – nearest-cache tables and localized caching policy – as described below.

Nearest-cache Tables. For each network node, we maintain the *nearest* node (including itself) that has a copy of the data item D_j for each data item D_j in the network. More specifically, each node i in the network maintains a *nearest-cache* table, where an entry in the nearest-cache table is of the form (D_j, N_j) where N_j is the closest node that has a copy of D_j. Note that if i is the server of D_j or has cached D_j, then N_j is i. In addition, if a node i has cached D_j, then it also maintains an entry (D_j, N_j^2), where N_j^2 is the *second-nearest cache*, i.e., the closest node (other than i itself) that has a copy of D_j. The second-nearest cache information is helpful when node i decides to remove the cached item D_j. Note that if i is the server of D_j, then N_j is i. The above information is in addition to any information (such as routing tables) maintained by the underlying routing protocol. The nearest-cache tables at network nodes in the network are maintained as follows in response to cache placement changes.

Addition of a Cache. When a node i caches a data item D_j, N_j^2 (the second-nearest cache) is set to its current N_j (nearest-cache node) and N_j is updated

to i itself. In addition, the node i broadcasts an `AddCache` message to all of its neighbors. The `AddCache` message contains the tuple (i, D_j) signifying the ID of the originating node and the ID of the newly cached data item. Consider a node l that receives the `AddCache` message (i, D_j). Let (D_j, N_j) be the nearest-cache table entry at node l signifying that N_j is the cache node currently closest to l that has the data item D_j. If $d_{li} < d_{lN_j}$,[6] then the nearest-cache table entry (D_j, N_j) is updated to (D_j, i), and the `AddCache` message is forwarded by l to all of its neighbors. Otherwise, the node l sends the `AddCache` message to the single node N_j (which could be itself) so that N_j can possibly update information about its second-nearest cache. The above process maintains correctness of nearest-cache entries and second-nearest cache entries in a static network with bounded communication delays, because of the following observations.

O1: Consider a node k whose nearest-cache table entry *needs* to change (to i) in response to addition of a cache at node i. Then, every intermediate node on the shortest path connecting k to i also needs to change its nearest-cache table entry (and hence, forward the `AddCache` message).

O2: Consider a cache node k such that its second-nearest cache node *should* be changed to i in response to addition of a cache at node i. Then, there exists two distinct *neighboring* nodes i_1 and i_2 (not necessarily different from k or i) on the shortest path from k to i such that the nearest-cache node of i_1 is k and the nearest-cache node of i_2 is i.

The first observation ensures correctness of nearest-cache entries since any node k that needs to receive the `AddCache` message receives it through the intermediate nodes on the shortest path connecting k to i. The second observation ensures correctness of the second-nearest cache entries, since for any cache node k whose second-nearest cache entry much change to the newly added cache i, there exists a node i_1 that sends the `AddCache` message (received from the forwarding neighboring node i_2) to k (i_1's nearest cache node). We now prove the above two observations.

We prove the first observation O1 by contradiction. Consider a node k whose nearest-cache table entry *needs* to change (to i) in response to addition of a cache at node i. Assume that there is an intermediate node j on the shortest path $\mathcal{P}$ connecting k to i such that j does not need to change its nearest-cache entry. Thus, there is another cache node l that is nearer to j than i. Then, the cache node l is also closer to k than i, and thus, k does not need to change its nearest-cache entry due to addition of cache at node

[6]The distance values are assumed to be available from the underlying routing protocol.

i – which is a contradiction. The second observation O2 is true because the nearest-cache node of each intermediate node on the shortest path connecting the *cache nodes* k and i (such that i is the second-nearest cache node of k) is either k or i.

Deletion of a Cache. To efficiently maintain the nearest-cache tables in response to deletion of a cache, we maintain a *cache list* C_j for each data item D_j at its server S_j. The cache list C_j contains the set of nodes (including S_j) that have cached D_j. To keep the cache list C_j up to date, the server S_j is informed whenever the data time D_j is cached at or removed from a node. Note that the cache list C_j is almost essential for the server S_j to efficiently update D_j at the cache nodes. Now, when a node i removes a data item D_j from its local cache, it updates its nearest-cache node (N_j) to its second-nearest cache (N_j^2) and deletes the second-nearest cache entry. In addition, the node i requests C_j from the server S_j, and then, broadcasts a `DeleteCache` message with the information (i, D_j, C_j) to all of its neighbors. Consider a node l that receives the `DeleteCache` message and let (D_j, N_j) be its nearest-cache table entry. If $N_j = i$, then the node l updates its nearest-cache entry using C_j, and forwards the `DeleteCache` message to all its neighbors. Otherwise, the node l sends the `DeleteCache` message to the node N_j. The above process ensures correctness of nearest-cache and second-nearest cache entries due to the above two observations (O1 and O2). If maintenance of a complete cache list at the server is not feasible, then we can either broadcast the `DeleteCache` message with $\{S_j\}$ as the cache list or not use any `DeleteCache` messages at all. In the latter case, when a data request for the deleted cache is received, the data request can be redirected to the server.

Integrated Cache-Routing Tables. Nearest-caching tables can be used in conjunction with any underlying routing protocol to reach the nearest cache node, as long as the distances to other nodes are maintained by the routing protocol (or available otherwise). If the underlying routing protocol maintains routing tables [40], then the nearest-cache tables can be integrated with the routing tables as follows. For a data item D_j, let H_j be the next node on the shortest path to N_j, the closest node storing D_j. Now, if we maintain a *cache-routing* table having entries of the form (D_j, H_j, δ_j) where δ_j is the distance to N_j, then there is no need for routing tables. However, note that maintaining cache-routing tables instead of nearest-cache tables *and* routing tables doesn't offer any clear advantage in terms of number of messages transmissions.

Mobile Networks. To handle mobility of nodes, we could maintain the integrated cache-routing tables in the similar vein as routing tables [40] are maintained in mobile ad hoc networks. Alternatively, we could have the servers periodically broadcast the latest cache lists. In our simulations, we adopted the latter strategy, since it precludes the need to broadcast `AddCache` and

`DeleteCache` messages to some extent.

Localized Caching Policy. The caching policy of DGA is as follows. Each node computes benefit of data items based on its "local traffic" observed for a sufficiently long time. The *local traffic* of a node i includes its own local data requests, non-local data requests to data items cached at i, and the traffic that the node i is forwarding to other nodes in the network.

Local Benefit. We refer to the benefit computed based on node's local traffic as the *local benefit*. For each data item D_j *not* cached at node i, the node i calculates the local benefit gained by caching the item D_j, while for each data item D_j cached at node i, the node i computes the local benefit lost by removing the item. In particular, the local benefit B_{ij} of caching (or removing) D_j at node i is the reduction (or increase) in access cost given by

$$B_{ij} = t_{ij}\delta_j,$$

where t_{ij} is the access frequency observed by node i for the item D_j in its local traffic, and δ_j is the distance from i to N_j or N_j^2 – the nearest-node other than i that has the copy of the data item D_j. Using the nearest-cache tables, each node can compute the local benefits of data items in a localized manner using only local information. Since the traffic changes dynamically (due to new cache placements), each node needs to continually recompute local benefits based on most recently observed local traffic.

Caching Policy. A node decides to cache the most beneficial (in terms of local benefit per unit size of data item) data items that can fit in its local memory. When the local cache memory of a node is full, the following cache replacement policy is used. Let $|D|$ denote the size of a data item (or a set of data items) D. If the local benefit of a newly available data item D_j is higher than the total local benefit of some set D of cached data items where $|D| > |D_j|$, then the set D is replaced by D_j. Since, adding or replacing a cache entails communication overhead (due to `AddCache` or `DeleteCache` messages), we employ a concept of *benefit threshold*. In particular, a data item is newly cached only if its local benefit is higher than the benefit threshold, and a data item replaces a set of cached data items only if the difference in their local benefits is greater than the benefit threshold.

Distributed Greedy Algorithm (DGA). The above components of nearest-cache table and cache replacement policy are combined to yield our Distributed Greedy Algorithm (DGA) for cache placement problem. In addition, the server uses the cache list to periodically update the caches in response to changes to the data at the server. The departure of DGA from CGA is primarily in its inability to gather information about all traffic (access frequencies). In addition, the inaccuracies and staleness of the nearest-cache table entries (due

to message losses or arbitrary communication delays) may result in approximate local benefit values. Finally, in DGA, the placement of caches happens simultaneously at all nodes in a distributed manner, which is in contrast to the sequential manner in which the caches are selected by the CGA. However, DGA is able to cope with dynamically changing access frequencies and cache placements. As noted before, any changes in cache placements trigger updates in the nearest-cache table, which in turn affect the local benefit values. Below is a summarized description of the DGA.

Algorithm 2 Distributed Greedy Algorithm (DGA)
Setting
A network graph $G(V, E)$ with p data items. Each node i has a memory capacity of m_i pages. Let Θ be the benefit threshold.
Program of Node i
BEGIN
When a data item D_j passes by:
if local memory has available space and $(B_{ij} > \Theta)$
then cache D_j
else if there is a set D of cached data items such that (local benefit of $D < B_{ij} - \Theta$) and $(|D| \geq |D_j|)$, then replace D with D_j.
When a data item D_j is added to local cache
Notify the server of D_j.
Broadcast an `AddCache` message containing (i, D_j)
When a data item D_j is deleted from local cache
Get the cache list C_j from the server of D_j
Broadcast a `DeleteCache` message with (i, D_j, C_j)
On receiving an `AddCache` message (i', D_j)
if i' is nearer than current nearest-cache for D_j
then update nearest-cache table entry and broadcast `AddCache` message to neighbors
else send the message to the nearest-cache of i
On receiving a `DeleteCache` message (i', D_j, C_j)
if i' is the current nearest-cache for D_j
then update the nearest-case of D_j using C_j, and broadcast the `DeleteCache` message
else send the message to the nearest-cache of i
For **mobile networks**, instead of `AddCache` and `DeleteCache` messages, for each data item, its server periodically broadcasts (to the entire network) the latest cache list.

END. ◊

Performance Analysis. Note that the performance guarantee of CGA (i.e., proof of Theorem 1) holds even if the CGA were to consider the memory pages

in some arbitrary order and select the most beneficial caches for each one of them. Now, based on the above observation, if we assume that local benefit is reflective of the accurate benefit (i.e., if the local traffic seen by a node i is the only traffic that is affected by caching a data item at node i), then DGA also yields a solution whose benefit is one-fourth of the optimal benefit. Our simulation results in Section 2.4.1 show that DGA and CGA indeed perform very close.

Data Expiry and Cache Updates. We incorporate the concepts of data expiry and cache updates in our overall framework as follows. For data expiry, we use the concept of *Time-to-Live (TTL)* [56], which is the time till which the given copy of the data item is considered valid/fresh. The data item or its copy is considered *expired* at the end of the TTL time value. We consider two data expiry models, viz. *TTL-per-request* and *TTL-per-item*. In the *TTL-per-request* data expiry model [56], the server responds to any data item request with the requested data item and an appropriately generated TTL value. Thus, *each* copy of the data item in the network is associated with an independent TTL value. In the *TTL-per-item* data expiry model, the server associates a TTL value with each *data item* (rather than each request), and all requests for the same data item are associated with the same TTL (until the data item expires). Thus, in the *TTL-per-item* data expiry model, all the fresh copies of a data item in the network are associated with the same TTL value. On expiry of the data item, the server generates a new TTL value for the data item.

For updating the cached data items, we consider two mechanisms. For the case of *TTL-per-request* data expiry model, we use the *cache deletion* update model, where each cache node independently deletes its copy of the expired data item. Such deletions are handled in the similar way as describe before, i.e., by broadcasting a `DeleteCache` request. In the case of *TTL-per-item* data expiry model, all the copies of a particular data item expire simultaneously. Thus, we use the *server multicast* cache update model, wherein the server multicasts the fresh copy of the data item to all the cache nodes, on expiration of the data item (at the server). If the cache list is not maintained at the server, then the above update is implemented using a network wide broadcast.

2.4 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 solutions returned by CGA and DGA. Then, we turn our attention to application level performance in complex network settings, and evaluate our designed DGA with respect to a naive distributed

algorithm and the HybridCache algorithm [56] using the ns-2 simulator [21].

2.4.1 CGA vs. DGA

In this subsection, we evaluate the relative performance of CGA and DGA, by comparing the benefits of the solutions delivered.

For the purposes of implementing a centralized strategy, we use our own simulator for implementation and comparison of our designed algorithms. In our simulator, DGA is implemented as a dynamically evolving process wherein initially all the memory pages are free and the nearest-cache table entries point to the corresponding servers. This initialization of nearest-cache table entries results in traffic being directed to servers, which triggers caching of data items at nodes, which in turn causes changes in the nearest-cache tables and further changes in cache placements. The process continues until convergence. To provide a semblance of an asynchronous distributed protocol, our simulation model updates routing and nearest-cache table entries in an arbitrary order across nodes.

Simulation Parameters. In our cache placement problem, the relevant parameters are: (i) number of nodes in the network, (ii) transmission radius T_r (two nodes can directly transmit with each other iff they are within T_r distance from each other), (iii) number of data items, (iv) number of clients accessing each data item, (v) memory capacity on each node. The first two parameters are related to network topology, the next two parameters are application-dependent, and the last parameter is the problem constraint (property of the nodes). Here, we assume each data item to be of unit size (one memory page). Below, we present a set of plots wherein we vary some of the above parameters, while keeping the others constant.

Varying Number of Data Items and Memory Capacity. Figure 2.2(a) plots the access costs for CGA and DGA against the number of data items in the network for different local memory capacities. Here, the network size is 500 nodes in a 30×30 area.[7] We use a transmission radius (T_r) of 5 units. The memory capacity in each node is expressed as the percentage of the number of data items in the network. We vary the number of data items from 500 to 1000, and the memory capacity of each node from 1% to 5% of the number of data items. The number of clients accessing each data items is fixed at 50% of the number of nodes in the network.

We observe that the access cost increases with the number of data items as expected. Also, as expected, we see that CGA performs slightly better

[7]Since the complexity of CGA is a high-order polynomial, the running time is quite slow. Thus, we have not been able to evaluate the performance on very large networks.

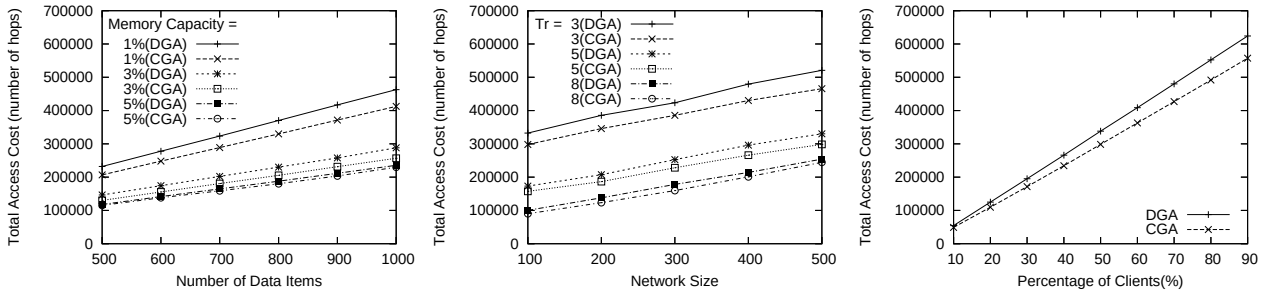


Figure 2.2: Performance comparison of CGA and DGA. (a) Varying number of data items and memory capacity, (b) Varying number of nodes and transmission radius (T_r), (c) Varying client percentage. Unless being varied - the number of nodes is 500, transmission radius is 5, number of data items is 1000, number of clients (for each data item) is 250, and each node can store 20 data items in its memory.

since it exploits global information, but DGA performs quite close to CGA. The performance difference between the algorithms decreases with increasing memory capacity, since with increasing memory capacity both the algorithms must converge to the same solution (access cost zero) as all client nodes will eventually be able to cache all the data items they wish to access. While this degenerate situation is not reached, the trend is indeed observed.

Varying Network Size and Transmission Radius. In the next plot (Figure 2.2(b)), we fix the number of data items in the network to 1000 and the memory capacity of each node to 2% of data items. As before, 50% of the network nodes act as clients for each of the data item. In this plot, we vary the network size from 100 nodes to 500 nodes and transmission radius (T_r) from 3 to 8. Essentially, Figure 2.2(b) shows the access cost as a function of network size and transmission radius for the two algorithms. Once again, as expected CGA slightly outperforms DGA, but DGA performs very close to CGA.

Varying Client Percentage. We also investigated the effect of the number of clients on the access cost. See Figure 2.2(c). We note a similar behavior. The performances are more similar as the independent parameter is varied towards the degenerate case. Here, the degenerate case represents a single client, where both algorithms must perform similarly.

2.4.2 DGA vs. HybridCache

In this subsection, we compare DGA with the HybridCache approach proposed in [56] by simulating both approaches in *ns2* [21] (version 2.27). The *ns2* simulator contains models for common ad hoc network routing protocols, IEEE Standard 802.11 MAC layer protocol, and two-ray ground reflection propaga-

tion models [13]. The DSDV routing protocol [40] is used to provide routing services. For comparison, we also implemented a *Naive* approach, wherein each node caches any passing-by data item if there is free memory space and uses LRU (least recently used) policy for replacement of caches. We start with presenting the simulation setup, and then present the simulation results in the next subsection. In all plots, each data point represents an average of five to ten runs. *In some plots, we show error bars indicating the 95% confidence interval; for sake to clarity, we show confidence intervals in only those graphs that are relevant to our claims.*

B.1 Simulation Setup

In this subsection, we briefly discuss the network set up, client query model, data access pattern model, and performance metrics used for our simulations.

Network Setup. We simulated our algorithms on a network of randomly placed 100 nodes in an area of 2000×500 m^2. Note that the nominal radio range for two directly communicating nodes in the *ns2* simulator is about 250 meters. In our simulations, we assume 1000 data items of varying sizes, two randomly placed servers 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. We choose the size of a data item randomly between 100 and 1500 bytes.[8]

Client Query Model. In our simulations, each network node is a client node. 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. In our DGA scheme, the query is forwarded to the nearest cache (based on the nearest-cache table entry). In the Naive 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. The time interval between two consecutive queries is known as the *query generate time* and follows exponential distribution with mean value T_{query} which we vary from 3 to 40 seconds. We do not consider values of T_{query} less than 3 seconds, since they result in a query success ratio of much less than 80 % for Naive and HybridCache approaches. Here, the *query success ratio* is defined as the percentage of the queries that receive the requested data item within the *query success timeout* period. In our simulations, we use a query success timeout of 40 seconds.

The above client query model is similar to the model used in previous studies [15, 56]. However, query generation process differs slightly from the one used in [56] in how the queries are generated. In [56], if the query response is not received within the query success timeout period, then the same query

[8]The maximum data size used in [56] is 10 KBytes, which is not a practical choice due to lack of MAC layer fragmentation/reassembly mechanism in the 2.27 version of *ns2* we used.

is sent repeatedly until it succeeds, while on success of a query, a new query is generated (as in our model) after some random interval.[9] Our querying model is better suited (due to exact periodicity of querying) for comparative performance evaluation of various caching strategies, while the querying model of [56] depicts a more realistic model of a typical application (due to repeated querying until success).

Data Access Models. For our simulations, we use the following two patterns for modeling data access frequencies at nodes.

1. Spatial pattern. In this pattern of data access, the data access frequencies at a node depends on its geographic location in the network area such that nodes that are closely located have similar data access frequencies. More specifically, we start with laying the given 1000 data items uniformly over the network area in a grid-like manner resulting in a virtual coordinate for each data item. Then, each network node accesses the 1000 data items in a *Zipf-like* distribution [12, 57], with the access frequencies of the data items ordered by the distance of the data item's virtual coordinates from the network node. More specifically, the probability of accessing (which can be mapped to access frequency) the $j^{th}(1 \leq j \leq 1000)$ closest data item is represented by $P_j = \frac{1}{j^\theta \sum_{h=1}^{1000} 1/h^\theta}$, where $0 \leq \theta \leq 1$. Here, we have assumed the number of data items to be 1000. When $\theta = 1$, the above distribution follows the strict Zipf distribution, while for $\theta = 0$, it follows the uniform distribution. As in [56], we choose θ to be 0.8 based on real web trace studies [12].

2. Random pattern. In this pattern of data access, each node uniformly accesses a predefined set of 200 data items chosen randomly from the given 1000 data items.

Performance Metrics. We measure three performance metrics for comparison of various caching strategies, viz., average query delay, total number of messages, and query success ratio. Query delay is defined as the time elapsed between query request and query response, and average query delay is the average of query delays over all queries. Total number of messages includes *all* message transmissions between neighboring nodes, including messages due to queries, maintenance of nearest-cache tables and cache-lists, and periodic broadcast of cache-lists in mobile networks. Messages to implement routing protocol are not counted, as they are the same in all three approaches compared. Query success

[9] In the original simulation code of HybridCache ([56]), the time interval between two queries is actually 4 seconds plus the query generate time (which follows exponential distribution with mean value T_{query}).

ratio has been defined before. Each data point in our simulation results is an average over five different random network topologies, and to achieve stability in performance metrics, each of our experiments is run for sufficiently long time (20000 seconds for our experiments).

DGA Parameter Values. We now present a brief discussion on choice of values of benefit threshold and local traffic window size for DGA. For static networks, we compute local benefits based on the most recent 1000 queries. Since, the data access frequencies remains static in our experiment setting, computing local benefits based on as large a number of queries as possible is a good idea. However, we observed that most recent 1000 queries are sufficient to derive complete knowledge of local traffic. For mobile networks with spatial data access pattern, the access frequencies at a client node change with the node's location. Thus, we compute local benefits using only 50 recent queries.

Also, we chose a benefit threshold value of 0.008 when the cache size is default 75 KBytes (capable of storing 100 average sized data items), based on the typical benefit value of the 100^{th} most beneficial data item at a node. We use similar methodology for choosing benefit threshold values for other values of cache sizes. In general, the chosen benefit threshold value should be higher than the communication overhead incurred (in terms of maintenance of the nearest-cache tables and the cache list) due to caching of a data item.

B.2 Simulation Results

We now present simulation results comparing the three caching strategies, viz., Naive Approach, HybridCache approach of [56], and our DGA, under the random and spatial data access patterns (as defined above) and study the effect of various parameter values on the performance metrics.

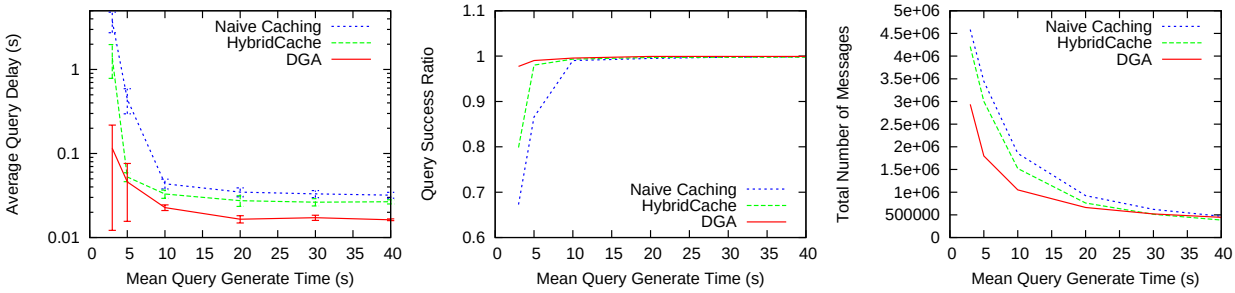


Figure 2.3: Varying mean query generate time on spatial data access pattern. (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

Varying Mean Query Generate Time. In Figure 2.3, we vary the mean query generate time T_{query} in the spatial data access pattern while keeping the cache

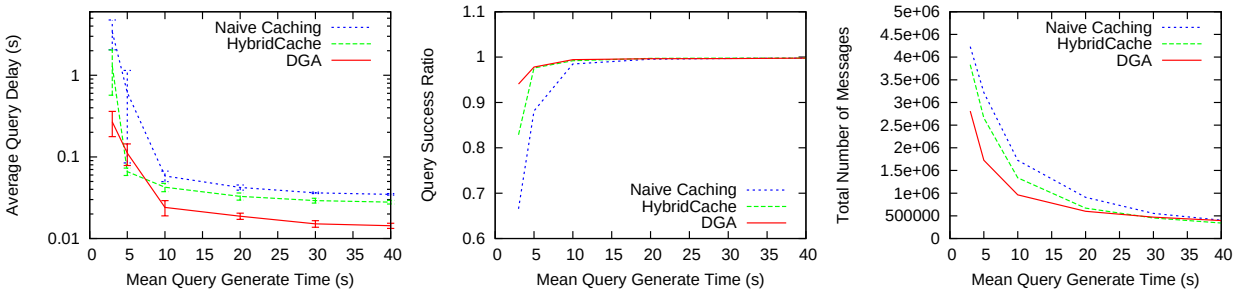


Figure 2.4: Varying mean of query generate time on random data access pattern. (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

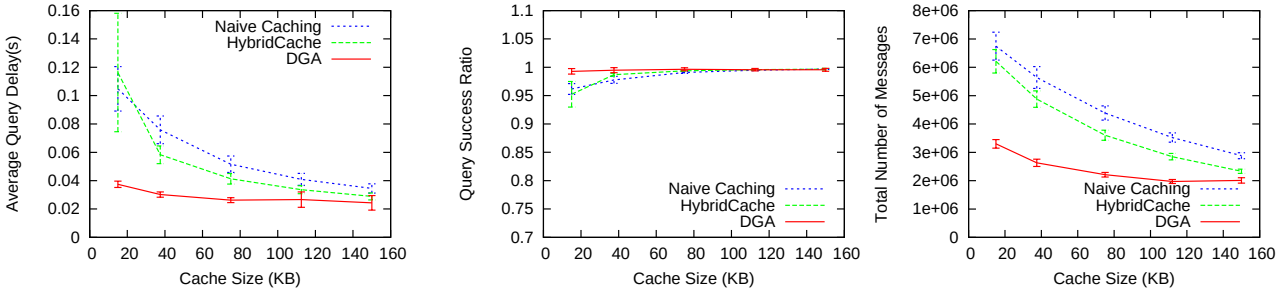


Figure 2.5: Varying cache size on spatial data access pattern. Here, $T_{query} = 10$ secs. (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

size as constant and all network nodes as client nodes. We choose the cache size to be big enough to fit about 100 average sized data items (i.e., 75 KBytes). We observe that our DGA outperforms the other two approaches in terms of all three performance metrics of query average delay, query success ratio, and total number of messages. In comparison with HybridCache strategy, our DGA has an average query delay of less than half for all parameter values (corroborated by confidence intervals of 95%), always has better query success ratio and lower message overhead. For the mean query generate time of 3 seconds, average query delay in all approaches is high, but our DGA outperforms HybridCache by a more than a factor of 10. Also, for very low mean query generate times, our DGA has a significantly better query success ratio. Figure 2.4 depicts similar observations for the random access data patterns, except that for mean query generate time of 5 second we have a slightly worse average query delay than that of HybridCache (but a significantly better query success ratio).

Varying Cache Memory Size. In Figure 2.5, we vary the local cache size of each node in the spatial data access pattern while keeping the mean query generate

time T_{query} constant at 10 seconds. We vary the local cache size from 15 KBytes (capable of storing 20 data items of average size) to 150 KBytes. We observe in Figure 2.5 that our DGA outperforms the HybridCache approach consistently for all cache sizes and in terms of all three performance metrics. The difference in the average query delay is much more significant for lower cache size – which suggests that our DGA is very judicious in choice of data items to cache. Note that HybridCache performs even worse than the Naive Approach when each node's memory is 15 KB.

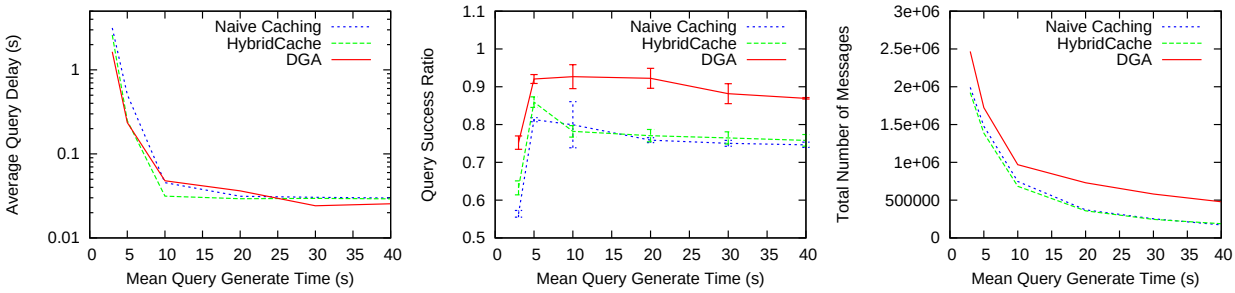


Figure 2.6: Varying mean query generate time in spatial data access pattern with v_{max}= 10 m/s. (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

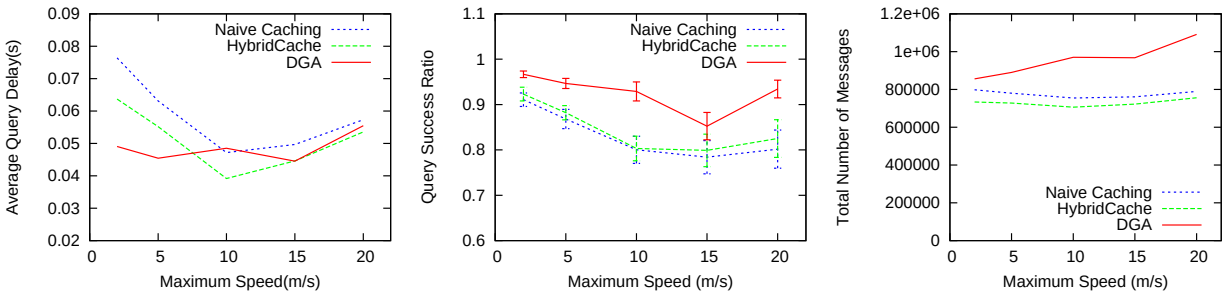


Figure 2.7: Varying v_{max} in spatial data access pattern. Here, T_{query} = 10 seconds. (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

Mobile Networks. Till now, we have restricted our discussion and simulations to ad hoc networks with static nodes. Now, we present performance comparison of various caching strategies for mobile ad hoc networks, wherein the mobile nodes move based on the "random waypoint" movement model [13]. In the random waypoint movement model, initially nodes are placed randomly in the area. Each node selects a random destination and moves towards the destination with a speed selected randomly from (0 m/s, v_{max} m/s). After the node reaches its destination, it pauses for a period of time (chosen to be 300

seconds in our simulations as in [56]) and repeats the movement pattern. In our simulations, the server broadcasting the cache lists every 100 seconds; this time interval is sufficient for notifying the nodes in a timely manner without incurring too much overhead.

In Figure 2.6, we compare various cache placement algorithms under the spatial data access pattern for varying mean query generate time, while keeping other parameters constant ($v_{\max} = 10$ m/s and local cache size = 75 KBytes). We observe that our all schemes perform similarly in terms of query delay, but DGA outperforms the other schemes by a significant margin in terms of query success ratio (again, corroborated by confidence intervals). Note that a significantly better query success ratio is much more desirable than a slightly better average query delay. In Figure 2.7, we compare various schemes under the spatial data access pattern for varying $v_{\max}$ value, while keeping other parameters constant ($T_{\text{query}} = 10$ seconds and local cache size = 75 KBytes). In terms of query delay, DGA outperforms other schemes for low mobilities, but has a slightly worse query delay for higher mobilities. But, more importantly, DGA has a *significantly* better query success ratio than all schemes for all mobilities. As noted before, a much better query success ratio is more desirable than slightly better query delay. We have the following explanation for the unusual (nonmonotonic) pattern of the graphs in Figure 2.7, which is the only figure in this article where we have varied mobilities. Firstly, Naive and Hybrid schemes display similar patterns – the query success ratio initially decreases with increase in mobility (as expected), and then, stabilizes. The initial decrease in query delay is largely due to the effect of decrease in query success ratio (due to loss of longer delay queries). The later increase in query delay with increase in mobility is as expected, when the query success ratio remains largely unchanged. In contrast, we notice that the DGA scheme has a relatively unchanged query delay and query success ratio, suggesting that higher mobility does not deteriorate much the performance of DGA due to the presence of nearest-cache table structure.

Varying Client Percentage. In all previous experiments in this section, we have assumed that each network node is a client node. In Figure 2.8, we vary the percentage of client nodes in the static network for the spatial data access pattern while keeping $T_{\text{query}} = 10$ seconds and cache size as 75 KBytes. We can see that DGA outperforms HybridCache for all client percentage values. The performance difference is seen to be very less at very low percentage of client nodes because of minimal traffic. Figure 2.9 shows similar trend and results for mobile networks with mobility ($v_{\max} = 10$ m/s).

Incorporating Data Expiry and Cache Updates. In all of our previous experiments, we have not considered data expiration or cache updates. We now incorporate data expiry and cache updates into our simulations. We run our

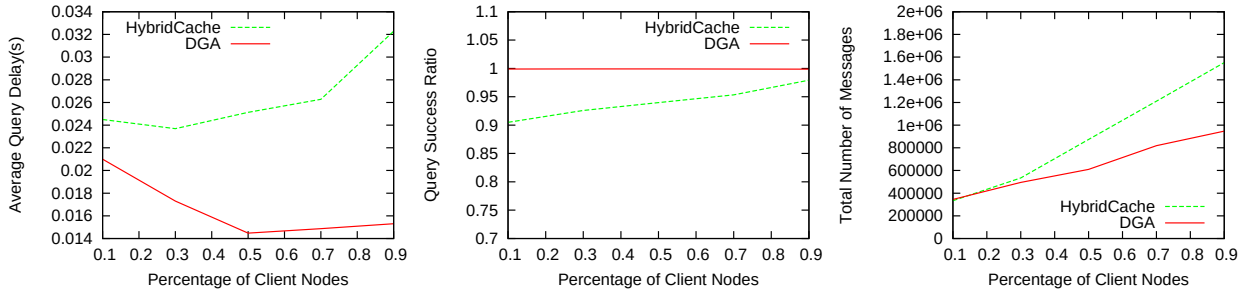


Figure 2.8: Varying client percentage on spatial data access pattern in static networks. Here, $T_{query} = 10$ seconds. (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

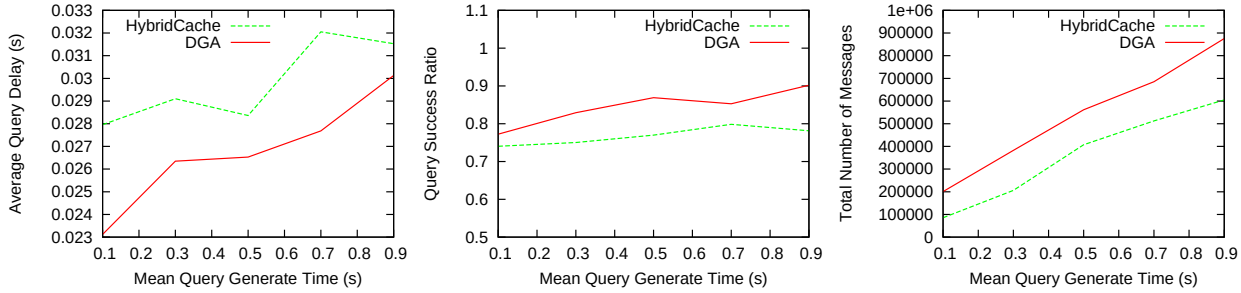


Figure 2.9: Varying client percentage on spatial data access pattern for mobile networks with $v_{max} = 10$ m/s. Here, $T_{query} = 10$ seconds. (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

simulations for a total run time of 200,000 seconds, and generate *TTL* values as *current time* plus a random number in [10000, 20000]. We use both data expiry models, viz., TTL-per-request and TTL-per-item. As mentioned before, for the TTL-per-request data expiry model, we use the cache deletion update mechanism, while for the TTL-per-time model we use the server multicast update mechanism. For all the three caching algorithms (Naive, Hybrid, and DGA), a data item request destined to a node with expired data item is redirected to the server, and the TTL value of a cached expired data item is updated using the TTL values of a passing by fresh copy of the data item. Figure 2.10 and Figure 2.11 show the comparison of the three caching techniques for TTL-per-item and TTL-per-request data expiry models respectively. In Figure 2.10, we see that our DGA technique outperforms HybridCache and Naive Caching in all three performance metrics; the relative performance is similar to that in Figure 2.3. However, for the case of TTL-per-request data expiry model (Figure 2.11), our DGA has a lower query success ratio (95%) due to increase in the number of `DeleteCache` messages; our DGA still outperforms the other

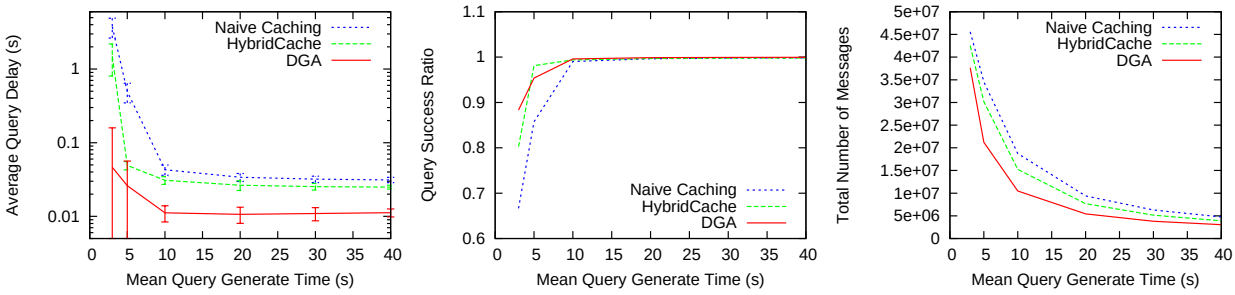


Figure 2.10: Varying mean query generate time on spatial data access pattern with cache update in static networks. Here, the data expiry model is TTL-per-item and the cache update model is server multicast. (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

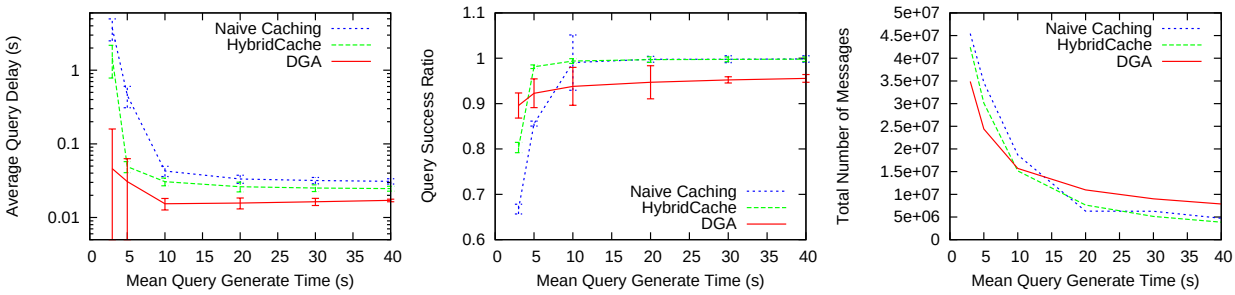


Figure 2.11: Varying mean query generate time on spatial data access pattern with cache update in static networks. Here, the data expiry model is TTL-per-request and the cache update model is cache deletion. (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

two techniques in terms of average query delay by a significant margin. Figure 2.12 and Figure 2.13 show the comparison of the DGA and HybridCache in mobile networks with $v_{max} = 2$ m/s, for TTL-per-item and TTL-per-request data expiry models respectively. The total run time for these experiments is 100,000 seconds, at which the average query delay and the query success ratio values had stabilized. In this very general setting of mobility, data expiration and cache updates, we continue to see that our DGA technique outperforms HybridCache in terms of average query delay and query success ratio.

Compare with Random Caching/Nearest Cache Table. To demonstrate that the better performance of DGA is not just due to the presence of nearest-cache table, but also due to the way the caches are placed, we compare our DGA scheme with a *Random Caching* scheme aided with the nearest-cache table. In the Random Caching scheme, we cache/place data items randomly in each node's cache memory and appropriately initialize the nearest-cache table. The placement

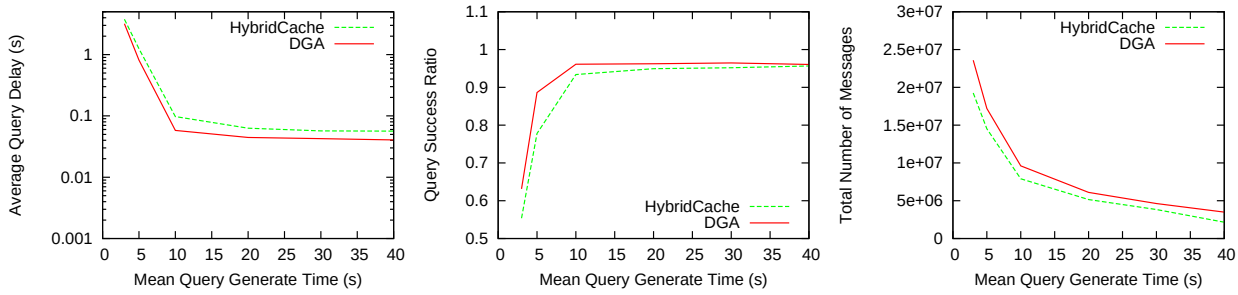


Figure 2.12: Varying mean query generate time on spatial data access pattern with cache update in mobile networks with $v_{max} = 10$ m/s. Here, the data expiry model is TTL-per-item and the cache update model is server multicast. (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

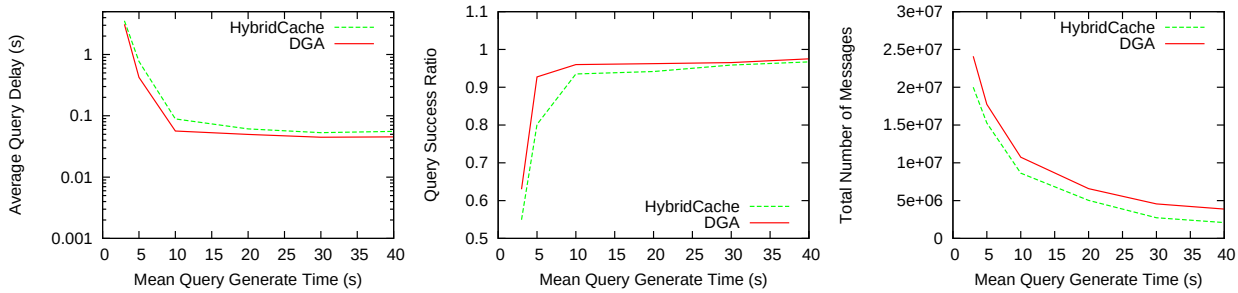


Figure 2.13: Varying mean query generate time on spatial data access pattern with cache update in mobile networks with $v_{max} = 10$ m/s. Here, the data expiry model is TTL-per-request and the cache update model is cache deletion. (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

of caches and initialization of nearest-cache tables is done in a centralized way without any communication overhead, which only favors the Random Caching scheme. In Figure 2.14, we compare the DGA, HybridCache, and Random Caching schemes in highly mobile networks (i.e., with $v_{max} = 10$ m/s). We observe that due to high-mobility all three different schemes have similar average query delays. However, DGA has significantly better query success ratio than the other schemes. These results demonstrate that the superior performance of our DGA scheme is not just due to the nearest-cache table structure.

Summary of Simulation Results. Our simulation results can be summarized as follows. Both the HybridCache and DGA approaches outperform the Naive approach in terms of all three performance metrics, viz., average query delay, query success ratio, and total number of messages. Our designed DGA almost

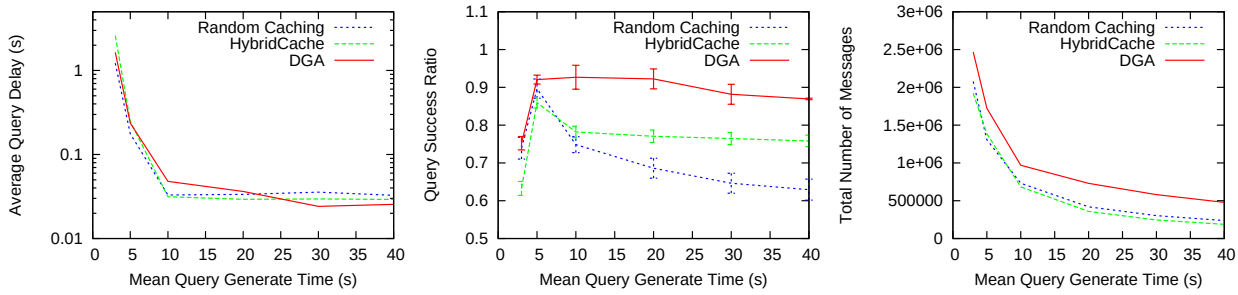


Figure 2.14: Varying mean query generate time on spatial data access pattern by comparing Random Caching, HybridCache and DGA, with $v_{max} = 10$ m/s (a) Average Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

always outperforms the Hybrid approach in terms of *all* performance metrics for a wide range of parameters of mean query generate time, local cache size, and mobility speed. In particular, for frequent queries or smaller cache size, the DGA approach has a significantly better average query delay and query success ratio. For very high mobility speeds, sometimes, the DGA approach has a slight worse average query delay than Hybrid, but with significantly better query success ratio, which is certainly the more desirable performance metric. We show that the success of DGA comes not only from maintenance of the nearest-cache tables, but also from the near-optimal placement of caches. The optimized placement of caches not only reduces query delay, but also message transmissions, which in turn leads to less congestion and hence fewer lost messages due to collisions or buffer overflows at the network interfaces. This, in turn provides a better success ratio. This "snowballing" effect is very apparent in challenging cases such as frequent queries and small cache sizes.

2.5 Conclusions

We have developed a paradigm of data caching techniques to support effective data access in ad hoc networks. In particular, we have considered memory capacity constraint of the network nodes, and developed efficient algorithms to determine near-optimal cache placements to maximize reduction in overall access cost. Reduction in access cost leads to communication cost savings and hence, better bandwidth usage and energy savings. Our later simulation experience with *ns2* also shows that better bandwidth usage also in turn leads to less message losses and thus, better query success ratio.

The novel contribution in our work is the development of a 4-approximation centralized algorithm, which is naturally amenable to a localized distributed

implementation. The distributed implementation uses only local knowledge of traffic. However, our simulations over a wide range of network and application parameters show that the performance of the two algorithms is quite close. We note that ours is the first work that presents a distributed implementation based on an approximation algorithm for the problem of cache placement of multiple data items under memory constraint.

We further compare our distributed algorithm with a competitive algorithm (HybridCache) presented in literature that has a similar goal. This comparison uses the *ns2* simulator with a complete wireless networking protocol stack including dynamic routing. We consider a broad range of application parameters and both stationary and mobile networks. These evaluations show that our algorithm significantly outperforms HybridCache, particularly in more challenging scenarios, such as higher query frequency and smaller memory.

Chapter 3

Cache Placement Under Update Cost Constraint

3.1 Introduction

Advances in embedded processing and wireless networking have made possible creation of sensor networks [5, 20]. A sensor network consists of sensor nodes with short-range radios and limited on-board processing capability, forming a multi-hop network of irregular topology. Sensor nodes must be powered by small batteries, making energy efficiency a critical design goal. There has been a significant interest in designing algorithms, applications, and network protocols to reduce energy usage of sensors. Examples include energy-aware routing [29], energy-efficient information processing [17, 20], and energy-optimal topology construction [53]. In this article, we focus on designing techniques to conserve energy in the network by caching data items at selected sensor nodes in a sensor network. The techniques developed in this paper are orthogonal to some of the other mentioned approaches, and can be used in combination with them to conserve energy.

Existing sensor networks assume that the sensors are preprogrammed and send data to a sink node where the data is aggregated and stored for offline querying and analysis. Thus, sensor networks provide a simple sample-and-gather service, possibly with some in-network processing to minimize communication cost and energy consumption. However, this view of sensor network architecture is quite limited. With the rise in embedded processing technology, sensor networks are set to become a more general-purpose, heterogeneous, distributed databases that generate and process time-varying data. As energy and storage limitations will always remain an issue – as much of it comes from pure physical limitations – new techniques for efficient data handling, storage, and dissemination must be developed. In this article, we take a general view of the

sensor network where a subset of sensor nodes (called *servers*) generate data and another subset of nodes (called *clients*) consume this data. The data generation and consumption may not be synchronous with each other, and hence, the overall communication cost can be optimized by caching generated data at appropriately selected intermediate nodes. In particular, the data-centric sensor network applications which require efficient data dissemination [11, 29] will benefit from effective data caching strategies.

In our model of the sensor network, there is a single data item at a given server node, and many client nodes. (See Section 3.6 for a discussion on multiple data items and servers.) The server is essentially the data item producer and maintains the original copy of the item. All the nodes in the network cooperate to reduce the overall communication cost of accessing the data via a caching mechanism, wherein any node in the network can serve as a cache. A natural objective in the above context could be to select cache nodes such that the sum of the overall access and update cost is minimized. However, such an objective does not guarantee anything about the general distribution of energy usage across the sensor network. In particular, the updates always originate from the server node, and hence, the server node and the surrounding nodes bear most of the communication cost incurred in updating. Hence, there is a need to constrain the total update cost incurred in the network, to prolong the lifetime of the server node and the nodes around it – and hence, possibly of the sensor network. Thus, in this article, we address the cache placement problem to minimize the total access cost under an update cost constraint. More formally, we address the problem of selecting nodes in the network to serve as caches in order to minimize the total access cost (communication cost incurred in accessing the data item by all the clients), under the constraint that the total update cost (communication cost incurred in updating the cache nodes using an optimal Steiner tree over the cache nodes and the server) is less than a given constant. Note that since we are considering only a single data item, we do not need to consider memory constraints of a node.

Paper Outline. We start with formulating the problem addressed in this article and a discussion on related work in Section 3.2. For the cache placement problem under an update cost constraint, we consider a tree topology and a general graph topology of the sensor network. For the tree topology, we design an optimal dynamic programming algorithm in Section 3.3. The optimal algorithm for the tree topology can be applied to the general graph topology by extracting an appropriate tree from the given network graph. For the general graph topology, we consider a simplified multiple-unicast update cost model, and design a constant-factor approximation algorithm in Section 3.4.1. In Section 3.4.2, we present an efficient heuristic for the general cache placement problem under an update cost constraint, i.e., for a general update cost model

in general graph topology. In Section 3.4.3, we present an efficient distributed implementation. Finally, we present simulation results in Section 4.5, and give concluding remarks in Section 3.6.

3.2 Problem Formulation and Related Work

In this section, we formulate the problem addressed in this article. We start with describing the sensor network model.

Sensor Network Model. A sensor network consists of a large number of sensor nodes distributed randomly in a geographical region. Each sensor node has a unique identifier (ID). Each sensor node has a radio interface and can communicate directly with some of the sensor nodes around it. For brevity, we sometimes just use *node* to refer to a sensor node. The sensor network can be modeled as an undirected weighted graph $G = (V, E)$, where V is the set of nodes, 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. The edge weight may represent a link metric such as loss rate, delay, or transmission power. We use d_{ij} to denoted the weighted distance between any two nodes i and j in G. The network has a data item, which is stored at a unique node called a *server*, and is updated at a certain update frequency. Each sensor node could be a client node. A client node i requests the data item with an *access frequency* a_i. The cost of accessing a data item (*access cost*) by a node i from a node j (the server or a cache) is $a_i d_{ij}$, where d_{ij} is the weighted distance between nodes i and j.

Problem. Informally, our article addresses the following *cache placement problem* in sensor networks. Select a set of nodes to store copies of the data item such that the total access cost is minimized under a given update cost constraint. The total access cost is the sum of all individual access costs over all clients for accessing the data item from the nearest node (either a cache or the server) having a copy of the data item. The *update cost* incurred in updating a set of caches M is modeled as the cost of the optimal Steiner tree [22] spanning the server and the set of caches. This problem is obviously NP-hard, as even the Steiner tree problem is known to be NP-hard [9]. In this article, we look at the above problem in various stages – a tree topology, a graph topology with a simplified update cost model, a graph topology with the general update cost model – and present optimal, approximation, and heuristic-based algorithms respectively.

More formally, given a sensor network graph $G = (V, E)$, a server r with the data item, and an update cost Δ, select a set of cache nodes $M \subseteq V$ $(r \in M)$

to store the data item such that the total access cost

$$\tau(G, M) = \sum_{i \in V} a_i \times min_{j \in M} d_{ij}$$

is minimum, under the constraint that the total update cost $\mu(M)$ is less than a given constant Δ, where $\mu(M)$ is the cost (i.e., weight) of the minimum weighted Steiner tree over the set of nodes M. We use the words cost and weight interchangeably, in this article. Note that in the above definition *all* network nodes are considered as potential clients. If some node i is not a client, the corresponding a_i would be zero.

Related Work. The general problem of determining optimal cache placements in an arbitrary network topology has similarity to two problems widely studied in graph theory viz., facility location problem and the k-median problem. Both problems consider only a single facility type (data item) in the network. In the facility-location problem, setting up a cache at a node incurs a certain fixed cost, and the goal is to minimize the sum of total access cost and the setting-up costs for all the caches, without any constraint. On the other hand, the k-median problem minimizes the total access cost under the number constraint, i.e., that at most k nodes can be selected as caches. Both problems are NP-hard, and a number of constant-factor approximation algorithms have been developed for each of the problems [16, 18, 30], under the assumption that the edge costs in the graph satisfy the triangular inequality. Without the triangular inequality assumption, either problem is as hard as approximating the set cover [30, 37], and therefore cannot be approximated better than $O(\log |V|)$ unless $\mathbf{NP} \subseteq \tilde{\mathbf{P}}$. Here, $|V|$ is the size of the network.

Several papers in the literature circumvent the hardness of the facility-location and k-median problems by assuming that the network has a tree topology [36, 55]. In particular, Li et al. [36] address the optimal placement of web proxies in a tree topology, essentially designing an $O(n^3k^2)$ time dynamic programming algorithm to solve the k-median problem optimally in a tree of n nodes. In other related works on cache placement in trees, Xu et al. [55] discuss placement of "transparent" caches to minimize the sum of reads and writes, Krishnan et al. [35] consider a cost model based on cache misses, and Kalpakis et al. [33] consider a cost model involving reads, writes, and storage. In sensor networks, which consist of a large number of energy-constrained nodes, the constraint on the number of cache nodes is of little relevance.

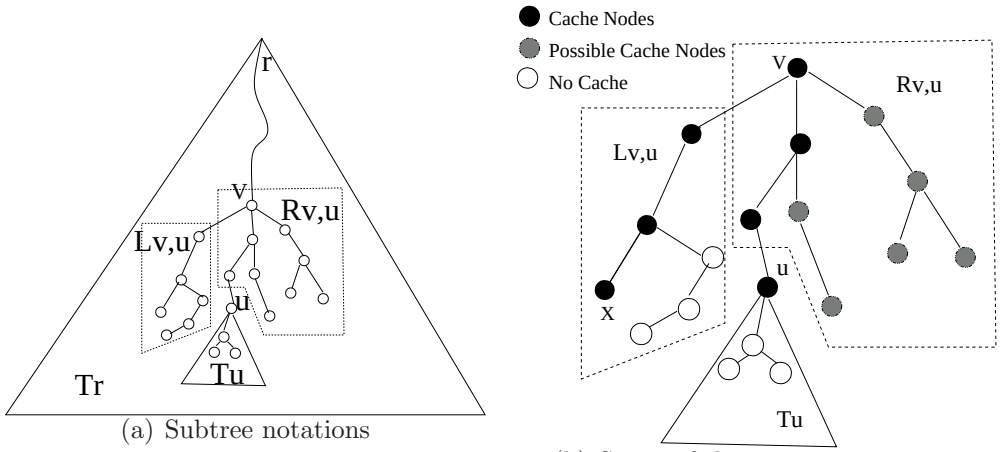
Cache placement has also been widely used in the web environment [8, 10, 43] and peer-to-peer networks [19, 25, 38] to alleviate problems such as server overloading, delayed respond time, and inadequate bandwidth. In particular, Qiu et al. [43] have addressed effective placement of web server replicas over the Internet and evaluated several placement algorithms. Cohen and Shenkar [19]

discuss the data replica placement problem in peer-to-peer networks and formulate the data replication strategies as a mapping from the query cost to the number of replicas. Relatively less work has been done on the cache placement problem in the specific context of ad hoc networks. Hara [25] addresses replica allocation methods for mobile ad hoc networks that can experience frequent disconnection. Yin and Cao [56] design and evaluate three simple cooperative caching techniques to efficiently support data access in ad hoc networks. In particular, they propose that intermediate nodes either cache data and/or nearest-cache path information to serve future requests. None of the above described works offer any performance guarantee on the solutions.

Caching in sensor networks is equally important, since caching sensed information at intermediate nodes can greatly reduce overall communication cost which is the main source of energy consumption. Shenker et al. [46] propose data centric storage (DCS) as a data dissemination paradigm for sensor networks. In DCS, data is stored, according to event type, at corresponding sensornet nodes. Data is also replicated to avoid overloading. Recently, Sheng et al. [45] study the storage node placement problem to minimize the total energy for data collection and data query. Intanagonwiwat et al. [29] propose directed diffusion, a data dissemination paradigm for sensor networks, which adopts a data centric approach and enables diffusion to achieve energy savings by selecting empirically good paths and by caching/processing data in-network. Bhattacharya et al. [11] develop a distributed framework that improves energy consumption by application layer data caching and asynchronous update multicast. Prabh et al. [42] improve upon [11] by presenting and analyzing the optimality properties of Steiner data caching tree over all the cache nodes. None of the above works take into consideration the update cost incurred for the selected caches. In this article, we consider cache placement in sensor networks wherein the objective is to minimize the access cost under the constraint of maximum allowable update cost. As mentioned before, the update cost is typically mostly borne by the server and the surrounding nodes, and hence, is a critical constraint. To the best of our knowledge, we are not aware of any prior work that considers the cache placement problem under an update cost constraint.

3.3 Tree Topology

In this subsection, we address the cache placement problem under the update cost constraint in a tree network. The motivation of considering a tree topology (as opposed to a general graph model which we consider in the next section) is two fold. Firstly, data dissemination or gathering in sensor networks is typically done over an appropriately constructed network tree. Secondly, for



(a) Subtree notations

(b) Setup of the recursive equation

Figure 3.1: Dynamic Programming algorithm for the tree topology.

the tree topology, we can actually design polynomial time *optimal* algorithms. Thus, we can apply such optimal algorithms for the tree topology to the general graph topology by extracting an appropriate tree (e.g., shortest-path tree or near-optimal Steiner tree connecting the clients) from the general graph. In Section 4.5, we show through extensive simulations that such a strategy of applying an optimal tree algorithm to a general graph topology yields heuristics that deliver near-optimal cache placement solutions.

Consider an ad hoc network tree T rooted at the node r. Since the communication edges are bidirectional, any node in the network could be designated as the root; thus, we assume that the root node r is also the server for the data item. The cache placement problem under update cost constraint in a tree topology can be formally defined as follows.

Given the tree network T rooted at r, a data item whose server is r, and an update cost constraint Δ, find a set of cache nodes $M \subseteq T$ ($r \in M$) for storing copies of the data item, such that the total access cost

$$\tau(T, M) = \sum_{i \in T} a_i \times min_{j \in M} d_{ij}$$

is minimized under the constraint that the total update cost $\mu(M)$ is less than Δ, where $\mu(M)$ is the weight of minimum-weighted Steiner tree over M. Note that the minimum weighted Steiner tree spanning over a set of nodes M is simply the smallest subtree connecting the root r to all the nodes in M.

3.3.1 Dynamic Programming Algorithm

In this subsection, we present an optimal dynamic programming algorithm for the above described cache placement problem under the update cost constraint in a tree topology. We first start with some subtree notations [36] that are needed to describe our dynamic programming algorithm.

Subtree Notations. Consider the network tree T rooted at r. We use T_u to denote the subtree rooted at u in the tree T with respect to the root r; the tree T_r represents the entire tree T. For ease of presentation, we use T_u to also represent the set of nodes in the subtree T_u. We use $p(i)$ to denote the parent node of a node i in the tree T_r. Let $\pi(i,j)$ denote the unique path from node i to node j in T_r, and $d_{k,\pi(i,j)}$ denote the distance of a node k to the closest node on $\pi(i,j)$.

Let us assume a left to right ordering of the children at each node. Consider two nodes v and u in the network tree, where v in an ancestor of u in T_r. See Figure 3.1(a). Let $L_{v,u}$ be the subgraph induced by the set of nodes on the left of and excluding the path $\pi(v,u)$ in the subtree T_v, and $R_{v,u}$ be the subgraph induced by the set of nodes on the right of and including the path $\pi(v,u)$, as shown in Figure 3.1(a). It is easy to see T_v can be divided into three distinct subgraphs, viz., $L_{v,u}$, T_u, and $R_{v,u}$. Note that T_u and $R_{v,u}$ are trees, while $L_{v,u}$ may not be a tree.

DP Algorithm. Consider a subtree T_v and a node x on the leftmost branch of T_v. Let us assume that all the nodes on the path $\pi(v,x)$ (including v and x) have already been selected as caches. Let $\tau(T_v, x, \delta)$ denote the optimal access cost for all the nodes in the subtree T_v under the *additional* update cost δ, where we do *not* include the cost of updating the already selected caches on the path $\pi(v,x)$. Below, we derive a recursive equation to compute $\tau(T_v, x, \delta)$, which would essentially yield a dynamic programming algorithm to compute $\tau(T_r, r, \Delta)$ – the minimum value of the total access cost for the entire network tree T_r under the update cost constraint Δ.

Let O_v be an optimal set (not including and in addition to $\pi(v,x)$) of cache nodes in T_v that minimizes the total access time under the constraint of additional update cost δ. Let u be the leftmost deepest node (i.e., deepest among the leftmost) of O_v in T_v, i.e., the node u is such that $L_{v,u} \cap O_v = \emptyset$ and $T_u \cap O_v = \{u\}$. It is easy to see that adding the nodes along the path $\pi(v,u)$ to the optimal solution O_v does not increase the additional update cost incurred by O_v, but may reduce the total access cost. Thus, without loss of generality, we assume that the optimal solution O_v includes all the nodes along the path $\pi(v,u)$ as cache nodes, if u is the leftmost deepest node of O_v in T_v.

Recursive Equation. As described above, consider an optimal solution O_v that minimizes $\tau(T_v, x, \delta)$, and let u be the leftmost deepest node of O_v in T_v. Note

that O_v does not include the nodes on $\pi(v,x)$. Based on the definition of u and possible cache placements, a node in $L_{v,u}$ will access the data item from either the nearest node on $\pi(v,u)$ or the nearest node on $\pi(v,x)$. In addition, any node in T_u will access the data item from the cache node u, while all other nodes (i.e., the nodes in $R_{v,u}$) will choose one of the cache nodes in $R_{v,u}$ to access the data item. See Figure 3.1(b). Thus, the optimal access cost $\tau(T_v, x, \delta)$ can be recursively defined in terms of $\tau(R_{v,u}, p(u), \delta - d_{u,\pi(v,x)})$ as shown below. Below, the quantity $d_{u,\pi(v,x)}$ denotes the shortest distance in T_v from u to a node on the path $\pi(v,x)$ and hence, is the additional update cost incurred in updating the caches on the path $\pi(v,u)$. We first define $S(T_v, x, \delta)$ as the set of nodes u such that the cost of updating u is less than δ, the additional update cost. That is,

$$S(T_v, x, \delta) = \{u | u \in T_v \wedge (\delta > d_{u,\pi(v,x)})\}.$$

Now, the recursive equation can be defined as follows.

$$\tau(T_v, x, \delta) = \begin{cases} \sum_{i \in T_v} a_i \times d_{i,\pi(v,x)}, & \text{if } S(T_v, x, \delta) = \emptyset \\ min_{u \in S(T_v,x,\delta)} \left(\begin{array}{l} \sum_{i \in L_{v,u}} a_i \times min(d_{i,\pi(v,u)}, d_{i,\pi(v,x)}) \\ + \sum_{i \in T_u} a_i d_{iu} \\ + \tau(R_{v,u}, p(u), \delta - d_{u,\pi(v,x)}) \end{array} \right), & \text{otherwise.} \end{cases}$$

In the above recursive equation, the first case corresponds to the situation when the additional update cost δ is not sufficient to cache the data item at any more nodes (other than already selected cache nodes on $\pi(v,x)$). For the second case, we compute the total (and minimum possible) access cost for each possible value of u, the leftmost deepest additional cache node, and pick the value of u that yields the minimum total access cost. In particular, for a fixed u, the first term corresponds to the total access cost of the nodes in $L_{v,u}$. Note that for a node in $L_{v,u}$ the closest cache node is either on the path $\pi_{v,x}$ or $\pi_{v,u}$. The second and third terms correspond to the total access time of nodes in T_u and $R_{v,u}$ respectively. Since the tree T_u is devoid of any cache nodes, the cache node closest to any node in T_u is u. The minimum total access cost of all the nodes in $R_{v,u}$ can be represented as $\tau(R_{v,u}, p(u), \delta - d_{u,\pi(v,x)})$, since the remaining available update cost is $\delta - d_{u,\pi(v,x)}$ where $d_{u,\pi(v,x)}$ is the update cost used up by the cache node u.

Time Complexity. Note that the above recursive equation can also be used to compute the optimal *placement* of cache nodes required needed within T_v to attain the optimal cost $\tau(T_v, x, \delta)$. Also, our original problem of finding an optimal set of cache nodes in T_r under the given update constraint Δ can be solved by evaluating $\tau(T_r, r, \Delta)$.

For time efficiency, we first precompute the terms $\sum_{i \in L_{v,u}} a_i \times min(d_{i,\pi(v,u)}, d_{i,\pi(v,x)})$ and $\sum_{i \in T_u} a_i d_{iu}$ for all combinations of values of v, u, and x. It is easy to see that the precomputation can be done in $O(n^4)$ time. Next, we compute $\tau(T_v, x, \delta)$ for all values of v, x and δ. Using the above precomputed values, each such $\tau(T_v, x, \delta)$ value takes $O(n)$ time for computation. Since, there are a total of $n^2\Delta$ combinations of v, x and δ, the overall time complexity of our dynamic programming algorithm is $O(n^4 + n^3\Delta))$. In unweighted graphs, the time complexity can be simplified to $O(n^4)$, since $\Delta = O(n)$.

3.4 General Graph Topology

The tree topology assumption makes it possible to design a polynomial-time optimal algorithm for the cache placement problem under update cost constraint. In this subsection, we address the cache placement problem in a general graph topology. In the general graph topology, the cache placement problem becomes NP-hard. Thus, our focus here is on designing polynomial-time algorithms with some performance guarantee on the quality of the solution.

As defined before, the total update cost incurred by a set of caches nodes is the minimum-weight of an optimal Steiner tree over the set of cache nodes and the server; we refer to this update cost model as the Steiner tree update cost model. Since the minimum-weighted Steiner tree problem is NP-hard in general graphs, we solve the cache placement problem in two steps. First, we consider a simplified multiple-unicast update cost model and present a greedy algorithm with a provable performance guarantee for the simplified model. Then, we improve our greedy algorithm based upon the more efficient Steiner tree update cost model.

3.4.1 Multiple-Unicast Update Cost Model

In this section, we consider the cache placement problem for general network graph under a simplified update cost model. In particular, we consider the multiple-unicast update cost model, wherein we model the total update cost incurred in updating a set of caches as the sum of the individual shortest path lengths from the server to each cache node. More formally, the total update cost of a set M of cache nodes is $\mu(M) = \sum_{i \in M} d_{si}$, where s is the server. Using this simplified update cost model, the cache placement problem in general graphs for update cost constraint can be formulated as follows.

Problem Under Multiple-Unicast Model. Given an ad hoc network graph $G = (V, E)$, a server s with the data item, and an update cost Δ, select a set of cache nodes $M \subseteq V$ $(s \in M)$ to store the data item such that the total access

cost $\tau(G, M) = \sum_{i \in V} a_i \times min_{j \in M} d_{ij}$ is minimum, under the constraint that the total update cost $\mu(M) = \sum_{i \in M} d_{si} < \Delta$.

The cache placement problem with the above simplified update cost model is still NP-hard, as can be easily shown by a reduction from the k-median problem. A number of constant-factor approximation algorithms have been proposed [16, 30] for the k-median problem which can also be used to solve the above cache placement problem. However, all the constant-factor approximation algorithms are based on the assumption that the edge weights in the network graph satisfy the triangular inequality. Moreover, the proposed approximation algorithms for k-median problem cannot be easily extended to the more efficient Steiner tree update cost model. Below, we present a greedy algorithm that returns a solution whose "access benefit" is at least 63% of the optimal benefit, where access benefit is defined as the reduction in total access cost due to cache placements.

Greedy Algorithm. In this section, we present a greedy approximation algorithm for the cache placement problem under the multiple-unicast update cost constraint in general graphs, and show that it returns a solution with near-optimal reduction in access cost. We start with defining the concept of a benefit of a set of nodes which is important for the description of the algorithm.

Definition 2 (Benefit of Nodes) Let A be an arbitrary set of nodes in the sensor network. The *benefit* of A with respect to an already selected set of cache nodes M, denoted as $\beta(A, M)$, is the decrease in total access cost resulting due to the selection of A as cache nodes. More formally, $\beta(A, M) = \tau(G, M) - \tau(G, M \cup A)$, where $\tau(G, M)$, as defined before, is the total access cost of the network graph G when the set of nodes M have been selected as caches. The *absolute benefit* of A denoted by $\beta(A)$ is the benefit of A with respect to an empty set, i.e., $\beta(A) = \beta(A, \emptyset)$.

The *benefit per unit update cost* of A with respect to M is $\beta(A, M)/\mu(A)$, where $\mu(A)$ is the total update cost of the set A under the multiple-unicast update cost model. □

Our proposed Greedy Algorithm works as follows. Let M be the set of caches selected at any stage. Initially, M is empty. At each stage of the Greedy Algorithm, we add to M the node A that has the highest benefit per unit update cost with respect to M at that stage. This process continues until the update cost of M reaches the allowed update cost constraint. The algorithm is formally presented below.

Algorithm 3 Greedy Algorithm
Input: A sensor network graph $V = (G, E)$.
Update cost constraint Δ.
Output: A set of cache nodes M.
BEGIN
$M = \emptyset$;
while $(\mu(M) < \Delta)$
Let A be the node with maximum $\beta(A, M)/\mu(A)$.
$M = M \cup \{A\}$;
end while;
RETURN $M - \{A\}$ or $\{A\}$, whichever has the higher benefit.
END. $\Diamond$

The running time of the above greedy algorithm is $O(kn^2)$, where n is the number of nodes in the network and $k(\leq n)$ is the number of iterations.

Performance Guarantee of the Greedy Algorithm. We now show that the Greedy Algorithm returns a solution that has a benefit at least 31% of that of the optimal solution. We start with presenting a lemma about the benefit function that leads to the final approximation result. The following lemma shows that the total benefit of a set of sets of nodes is at most the sum of the benefit of individual sets.

Lemma 1 *Let $O_1, O_2, ..., O_m$ and M be arbitrary sets of nodes. Then, $\beta(O_1 \cup O_2 \ldots \cup O_m, M) \leq \sum_{i=1}^{m} \beta(O_i, M)$.*

Proof: Without loss of generality, we prove the lemma for $m = 2$. By definition of the benefit function, we have

$$\beta(O_1 \cup O_2, M) = \beta(O_1, M) + \beta(O_2, M \cup O_1).$$

In the next paragraph, we show that

$$\beta(O_2, M \cup O_1) \leq \beta(O_2, M).$$

Thus, we get $\beta(O_1 \cup\ O_2, M) \leq \beta(O_1, M) + \beta(O_2, M)$.

To complete the proof, we now show that $\beta(O_2, M) \geq \beta(O_2, M \cup O_1)$ for arbitrary sets of nodes M, O_1, and O_2. Let V be the set of all nodes in the given network graph, and let $d(i, M)$ denote the distance (number of hops) from a node i to the closest node in the set M. Note that for an arbitrary node i and arbitrary sets of nodes M, O_1, and O_2, we have

$$d(i, M) - d(i, M \cup O_2) \geq d(i, M \cup O_1) - d(i, M \cup O_1 \cup O_2).$$

To see the above, consider the following three cases viz. the closest node to i in the set $M \cup O_1 \cup O_2$ is in M, or O_1 or O_2. In the first case, both sides of the above equation are zero. For the second case, the right hand side is zero while the left hand side is positive. For the third case, $d(i, M \cup O_1 \cup O_2) = d(i, M \cup O_2) = d(i, O_2)$ and $d(i, M) \geq d(i, M \cup O_1)$.

Now, by the definition of the benefit function, we have

$$\begin{aligned} \beta(O_2, M) &= \sum_{i \in V} a_i \times (d(i, M) - d(i, M \cup O_2)) \\ &\geq \sum_{i \in V} a_i \times (d(i, M \cup O_1) - d(i, M \cup O_1 \cup O_2)) \\ &= \beta(O_2, M \cup O_1) \end{aligned}$$

■

Now, we show that the Greedy Algorithm returns a solution with near-optimal benefit. The proof technique used here is similar to that used in [23] for the closely related problem of selection of views in a data warehouse.

Theorem 4 *Greedy Algorithm (Algorithm 3) returns a solution whose absolute benefit is of at least* $(1-1/e)/2$ *times the absolute benefit of an optimal solution.*

Proof: Let M be the set of cache nodes selected by Greedy Algorithm at the end of the while loop, i.e., before the very last step. Below, we show that the benefit of M is at least $(1 - 1/e)$ times the absolute benefit of an optimal solution (we actually allow the optimal solution to use $\mu(M)$ update cost). Since, Greedy Algorithm partitions M into two feasible solutions and return the better of them, we get the desired approximation result.

Let $\mu(M)$, the total multiple-unicast update cost of M, be equal to k. Let the optimal solution using at most k units of multiple-unicast update cost be O.

Consider a stage when the greedy algorithm has already chosen a set $M = G_l$ occupying l units of update cost with "incremental" benefits $b_1, b_2, \ldots, b_l$. Incremental benefit b_i is defined as the increase in benefit when the node with the i^{th} unit of update cost is added into the set of cache nodes. So, the absolute benefit of G_l, $\beta(G_l) = \sum_{i=1}^{l} b_i$. Since, the absolute benefit of $O \cup G_l$ is at least that of O, we have $\beta(O, G_l) \geq \beta(O) - \sum_{i=1}^{l} b_i$.

Let $O = \{o_1, o_2, \ldots, o_m\}$. By Lemma 1 for the sets $\{o_i\}$'s, we have $\beta(O, G_l) \leq \sum_{i=1}^{m} \beta(\{o_i\}, G_l)$. Now, we show by contradiction that there exists a node o_h in O such that $\beta(\{o_h\}, G_l)/\mu(o_h) \geq \beta(O, G_l)/k$. Note that O and G_l may not be disjoint. Let us assume that there is no such node o_h in O. Then, $\beta(\{o_i\}, G_l) <$

$(\beta(O, G_l)/k)\mu(o_i)$ for every node $o_i \in O$. Thus, $\sum_{i=1}^{m} \beta(\{o_i\}, G_l) < (\beta(O, G_l)/k) \sum_{i=1}^{m} \mu(o_i) = \beta(O, G_l)$, which violates Lemma 1. Therefore, there exists a node o_h in O such that

$$\beta(\{o_h\}, G_l)/\mu(o_h) \geq \beta(O, G_l)/k \geq (\beta(O) - \sum_{i=1}^{l} b_i)/k.$$

Now, the benefit per unit update cost with respect to G_l of the node C selected by the algorithm is at least that of o_h, which is at least $(\beta(O) - \sum_{i=1}^{l} b_i)/k$, as shown above. Distributing the benefit of C over each of its unit update costs equally (for the purpose of analysis), we get

$$b_{l+j} \geq (\beta(O) - \sum_{i=1}^{l} b_i)/k \qquad \text{for } \ 0 < j \leq \mu(C),$$

where $\mu(C)$ is the update cost for C.

As the above analysis is true for each node C selected at any stage, we have for $0 < j \leq k$:

$$b_j \geq (\beta(O) - \sum_{i=1}^{j-1} b_i)/k.$$

Rearranging a few terms, we get:

$$(\beta(O) - \sum_{i=1}^{j} b_i) \leq (\beta(O) - \sum_{i=1}^{j-1} b_i)(k-1)/k.$$

Application of the above equation j times, we get $(\beta(O) - \sum_{i=1}^{j} b_i) \leq ((k-1)/k)^j \beta(O)$, which yields $(\beta(O) - \sum_{i=1}^{k} b_i) \leq ((k-1)/k)^k \beta(O)$ when $j = k$. Thus, we get $(\sum_{i=1}^{k} b_i)/\beta(O) \geq 1 - (\frac{k-1}{k})^k \geq 1 - 1/e$. Since, the absolute benefit of M is $\beta(M) = \sum_{i=1}^{k} b_i$, we have $\beta(M)/\beta(O) \geq 1 - 1/e$. ∎

3.4.2 Steiner Tree Update Cost Model

Recall that the constant factor performance guarantee of the Greedy Algorithm described in the previous section is based on the multiple-unicast update cost model, wherein whenever the data item in a cache node needs to be updated, the updated information is transmitted along the individual shortest path between the server and the cache node. However, the more efficient method of updating a set of caches from the server is by using the optimal (minimum-weighted) Steiner tree over the selected cache nodes and the server. In this section, we improve the performance of our Greedy Algorithm by using the more efficient Steiner tree update cost model, wherein the total update cost

incurred for a set of cache nodes is the cost of the optimal Steiner tree over the set of nodes M and the server of the data item.

Since the minimum-weighted Steiner tree problem is NP-hard, we adopt the simple 2-approximation algorithm [22] for the Steiner tree construction, which constructs a Steiner tree over a set of nodes L by first computing a minimum spanning tree in the "distance graph" of the set of nodes L. We use the term 2-approximate Steiner tree to refer to the solution returned by the 2-approximation Steiner tree approximation algorithm. Based on the notion of 2-approximate Steiner tree, we define the following update cost terms.

Definition 3 (Steiner Update Cost) The *Steiner update cost* for a set M of cache nodes, denoted by $\mu'(M)$, is defined as the cost of a 2-approximate Steiner tree over the set of nodes M and the server s.

The *incremental Steiner update cost* for a set A of nodes with respect to a set of nodes M is denoted by $\mu'(A, M)$ and is defined as the increase in the cost of the 2-approximate Steiner tree due to addition of A to M, i.e., $\mu'(A, M) = \mu'(A \cup M) - \mu'(M)$. □

Based on the above definitions, we describe the Greedy-Steiner Algorithm which uses the more efficient Steiner tree update cost model as follows.

Algorithm 4 Greedy-Steiner Algorithm
Input: A network graph $V = (G, E)$.
Update cost constraint Δ.
Output: The set of cache nodes M.
BEGIN
$M = \emptyset$;
while $(\mu'(M) < \Delta)$
Let A be the node with maximum $\beta(A, M)/\mu'(A, M)$.
$M = M \cup \{A\}$;
end while;
RETURN $M - \{A\}$ or $\{A\}$, whichever has the higher benefit.
END. ◊

Unfortunately, there is no performance guarantee of the solution delivered by the Greedy-Steiner Algorithm. However, as we show in Section 4.5, the Greedy-Steiner Algorithm performs the best among all our designed algorithms for the cache placement problem under an update cost constraint.

3.4.3 Distributed Implementation

In this subsection, we design a distributed version of the centralized Greedy-Steiner Algorithm (Algorithm 4). Using similar ideas as presented in this section, we can also design a distributed version of the centralized Greedy Algorithm (Algorithm 3). However, since the centralized Greedy-Steiner Algorithm outperformed the centralized Greedy Algorithm for all ranges of parameter values in our simulations, we present only the distributed version of Greedy-Steiner Algorithm. As in the case of centralized Greedy-Steiner Algorithm, we cannot prove any performance guarantee for the presented distributed version. However, we observe in our simulations that solution delivered by the distributed version is very close to that delivered by the centralized Greedy-Steiner Algorithm. Here, we assume the presence of an underlying routing protocol in the sensor network. Due to limited memory resources at each sensor node, a proactive routing protocol [41] that builds routing tables at each node is unlikely to be feasible. In such a case, a location-aided routing protocol such as GPSR [34] is sufficient for our purposes, if each node is aware of its location (either through GPS [28] or other localization techniques [7, 14]).

Distributed Greedy-Steiner Algorithm. The distributed version of the centralized Greedy-Steiner Algorithm consists of rounds. During a round, each non-cache node A estimates its benefit per unit update cost, i.e., $\beta(A, M)/\mu'(A, M)$, as described in the next paragraph. If the estimate at a node A is maximum among all its communication neighbors, then A decides to cache itself, and sends the estimated incurred update cost $\mu'(A, M)$ to the server. During each round, a number of sensor nodes may decide to cache the data item according to the above criteria. At the end of each round, the server node sums the update cost incurred by newly added cache nodes, and calculates the remaining update cost by deducting it from the given update cost constraint. Then the remaining update cost is broadcast by the server to the entire network and a new round is *initiated*. To avoid many cache nodes being selected in the first round, we can have a node selecting itself as a cache node only if its estimate of benefit per unit cost estimate is the maximum among all its k-hop neighbors, where $k > 1$. The constant k may be chosen iteratively, until the number of nodes selecting themselves are small enough. Note that we do not need to assume a synchronized mode, since each round is initiated by the server using a message. If there is no remaining update cost, then the server decides to discard some of the recently added caches (to keep the total update cost under the given update cost constraint), and the algorithm terminates. In this case, the server can deal with it by order. The algorithm is formally presented below.

Algorithm 5 Distributed Greedy-Steiner Algorithm

Input: A network graph $V = (G, E)$.
Update cost constraint Δ.
Output: The set of cache nodes M.
BEGIN
$M = \emptyset$;
while $(\mu'(M) < \Delta)$
Let $\mathcal{A}$ be the set of nodes each of which (denoted as A)
has the maximum $\beta(A, M)/\mu'(A, M)$ among its
non-cache neighbors.
$M = M \cup \mathcal{A}$;
end while;
RETURN M;
END.
$\Diamond$

Estimation of $\mu'(A, M)$. Let A be a non-cache node, and T_A^S be the shortest path tree from the server to the set of communication neighbors of A. Let $C \in M$ be the cache node in T_A^S that is closest to A, and let d be the distance from A to C. In the above Distributed Greedy-Steiner Algorithm, we estimate the incremental Steiner update cost $\mu'(A, M)$ to be $d \times u$, where u is the update frequency of the server. The value d can be computed in a distributed manner at the start of each round as follows. As mentioned before, the server initiates a new round by broadcasting a packet containing the remaining update cost to the entire network. If we append to this packet all the cache nodes encountered on the way, then each node should get the set of cache nodes on the shortest path from the server to itself. Now, to compute d, each node only needs to exchange the above information with all its immediate neighbors.

Estimation of $\beta(A, M)$. A non-cache node A considers only its "local" traffic to estimate $\beta(A, M)$, the benefit with respect to an already selected set of cache nodes M. The local traffic of A is defined as the data access requests that use A as an intermediate/origin node. Thus, the local traffic of a node includes its own data requests. We estimate the benefit of caching the data item at A as $\beta(A, M) = d \times t$, where t is the frequency of the local traffic observed at A and d is the distance to the nearest cache from A (which is computed as shown in the previous paragraph). The local traffic t can be computed if we let the normal network traffic (using only the already selected caches in previous rounds) run for some time between successive rounds. The data access requests of a node A during normal network traffic between rounds can be directed to the nearest cache in the tree T_A^S as defined in the previous paragraph.

Dynamic Topologies. The sensor network topology may be very dynamic due to node/link failures, mobility of sensor nodes, new sensor nodes entering

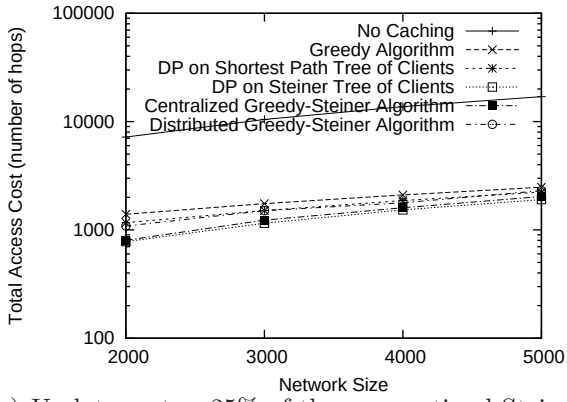
the network, etc. The Distributed Greedy-Steiner Algorithm can be adapted to handle node failures if the active cache nodes periodically send a probe to the server node, and the server initiates a new round if the current update cost is sufficiently less than the update cost constraint. If the server node is static, then mobility of cache nodes can be handled in a similar way. However, in this case, the server node may need to discard cache nodes that have moved too far away. The situation is more challenging if the server node itself is mobile. In the most general scenario of mobile server and client nodes, the server node may need to gather latest location of active cache nodes' by periodically flooding the network (in absence of a proactive routing scheme that adapts to mobility of nodes). New nodes entering the network automatically become part of the network and play a useful role in later rounds of the algorithm.

3.5 Performance Evaluation

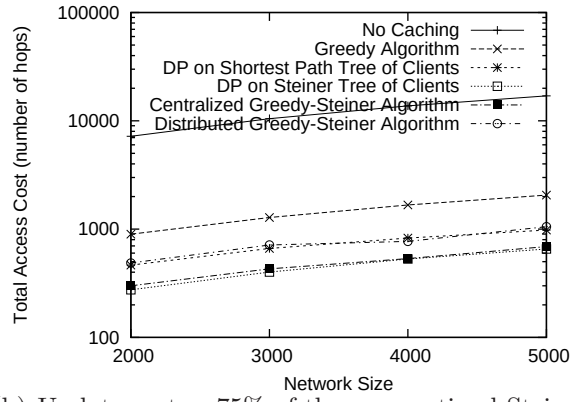
We empirically evaluate the relative performances of the cache placement algorithms for randomly generated sensor networks of various densities. As the focus of our work is to optimize access cost, this metric is evaluated for a wide range of parameters – (i) network-related – such as the number of nodes and network density, (ii) application-related – such as the number of clients accessing each data item.

We study various caching schemes (listed below) on a randomly generated sensor network of 2,000 to 5,000 nodes in a square region of 30×30. The distances are in terms of arbitrary units. We assume all the nodes have the same transmission radius (T_r), and all edges in the network graph have unit weight. We have varied the number of clients over a wide range. For clarity, we first present the data for the case where number of clients is 50% of the number of nodes, and then present a specific case with varying number of clients. All the data presented here are representative of a very large number of experiments we have run. Each point in a plot represents an average of five runs, in each of which the server is randomly chosen. The access costs are plotted against number of nodes and transmission radius and several caching schemes are evaluated:

- *No Caching* – serves as a baseline case.
- *Greedy Algorithm* — greedy algorithm using the multiple-unicast update cost model (Algorithm 3).
- *Centralized Greedy-Steiner Algorithm* — greedy algorithm using the Steiner tree-based update cost model (Algorithm 4).



(a) Update cost = 25% of the near-optimal Steiner tree cost.

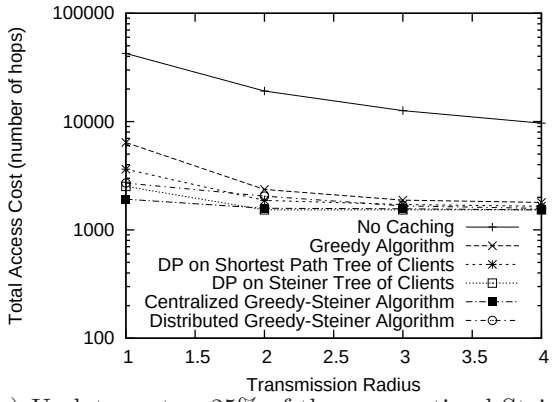


(b) Update cost = 75% of the near-optimal Steiner tree cost.

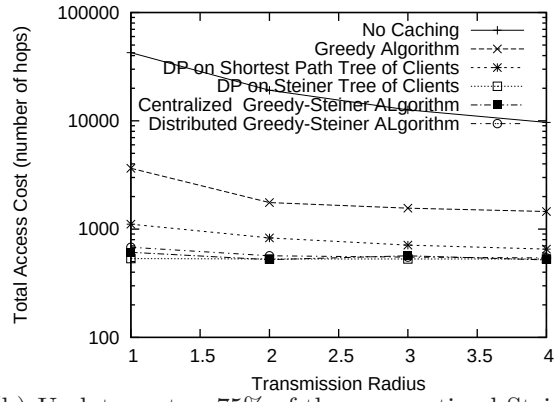
Figure 3.2: Access cost with varying number of nodes in the network for different update cost constraints. Transmission radius (T_r) = 2. Number of clients = 50% of the number of nodes, and hence increases with the network size.

- *Distributed Greedy-Steiner Algorithm* – distributed implementation of the Greedy-Steiner Algorithm (Algorithm 5).
- *DP on Shortest Path Tree of Clients* – Dynamic Programming algorithm (Section 3.3.1) on the tree formed by the shortest paths between the clients and the server.
- *DP on Steiner Tree of Clients* – Dynamic Programming algorithm (Section 3.3.1) on the 2-approximate Steiner tree over the clients and the server.

Varying Network Size for Multiple Update Constraints. We first compare the performance of the six algorithms under different update cost con-



(a) Update cost = 25% of the near-optimal Steiner tree cost.



(b) Update cost = 75% of the near-optimal Steiner tree cost.

Figure 3.3: Access cost with varying transmission radius (T_r) for different update cost constraints. Number of nodes = 4000, and number of clients = 2000 (50% of number of nodes).

straints with varying number of nodes (See Figure 3.2). The transmission radius (T_r) is fixed at 2 (we will vary this in a later evaluation). Instead of using absolute cost values to describe the update cost constraint, we represent it in terms of a fraction of the cost of the near-optimal (2-approximate [9]) Steiner tree over all clients and the server node. Clearly, this cost represents a measure of the *maximum* possible update cost. The update cost constraint is set to 25% and 75% of the cost of the near-optimal Steiner tree. Figure 3.2 shows that the proposed algorithms perform significantly better (up to an order of magnitude) than the no-caching case (note the logarithm scale for the vertical axis). Figure 3.2(a) shows that when the update cost constraint is small, all our proposed algorithms perform very similarly, especially for large

network size. However, a closer look shows that Greedy Algorithm using the multiple-unicast update cost model performs the worst among all our five designed algorithms. The performance differences can be seen more clearly in Figure 3.2(b), where the update cost constraint is larger. In particular, the best performing algorithms are the Steiner tree based centralized algorithms viz. DP on Steiner tree of clients and Centralized Greedy-Steiner Algorithm. Finally, we observe that the Distributed Greedy-Steiner Algorithm performs quite closely to its centralized version.

Varying Transmission Radius. Figure 3.3 shows the effect of the transmission radius (T_r) on access cost. A network of 4,000 nodes is chosen for these experiments. The transmission radius T_r is varied from 1 to 4. This range is sufficient for evaluation. T_r smaller than 1 disconnects the network with high probability. On the other end, a convergence of behavior of our caching algorithms is seen near $T_r = 4$, as the network is already dense enough. So, T_r is not increased any further. The total access cost of all the algorithms decreases with the increase in T_r, since clients come closer to the server in terms of number of hops as the network density increases. However, when the update cost is large (75% of the near-optimal Steiner tree) as shown in Figure 3.3(b), the performances of the two Steiner-tree based centralized algorithms is almost same for all values of T_r. Moreover, we again observe that the Distributed Greedy-Steiner Algorithm performs very close to its centralized version.

Summary. The general trend in these two sets of plots (Figures 3.2 and 3.3) is similar. Aside from the fact that our algorithms offer much less total access cost than the no-caching case, the plots show that (i) the two Steiner tree-based algorithms (DP on Steiner Tree of Clients and Centralized Greedy-Steiner Algorithm) perform equally well and the best among all algorithms except for very sparse graphs; (ii) the Greedy-Steiner Algorithm provides the best overall behavior; (iii) the Distributed Greedy-Steiner Algorithm performs very closely to its centralized version. Figure 3.4 shows the total access cost as a function of number of clients for a network with 3,000 nodes. The general behavior is no different from before.

3.6 Conclusions

We have developed a suite of data caching techniques to support effective data dissemination in sensor networks. In particular, we have considered update cost constraint and developed efficient algorithms to determine optimal or near-optimal cache placements to minimize overall access cost. Minimization of access cost leads to communication cost savings and hence, energy efficiency. The choice of update constraint also indirectly contributes to resource effi-

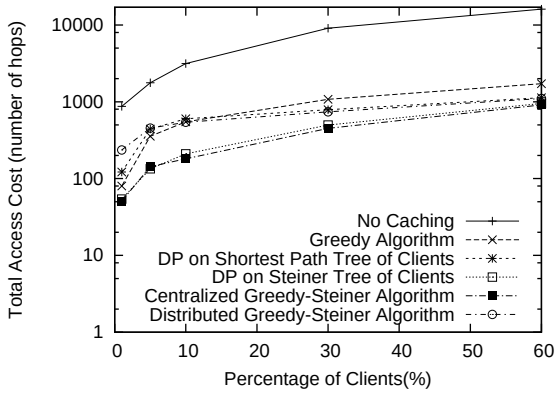


Figure 3.4: Effect of the number of clients on the access cost. $T_r = 2$. Update cost = 50% of the minimum Steiner tree cost. Number of nodes = 3,000.

ciency. Two models have been considered – one for a tree topology, where an optimal algorithm based on dynamic programming has been developed, and the other for the general graph topology, which presents a NP-hard problem where a polynomial-time approximation algorithm has been developed. We also designed efficient distributed implementations of our centralized algorithms, and empirically showed that they perform well for random sensor networks.

Cache placement of multiple data items at different servers can be solved as independent single data item cache placement problems, since the update cost constraint at different servers would presumably be independent. The cache placement problem of multiple data items at a single server is challenging, but we can use a heuristic of allocating update costs for each item in proportion to the sum of access frequencies. Each of the above scenarios assumes no memory constraints at network nodes. Since, sensor nodes are characterized by limited memory capacity and limited battery energy, we are currently addressing the more general cache placement problem in sensor networks under memory and update constraints for multiple data items.

Chapter 4

Data Caching Under Number Constraint

4.1 Introduction

In recent years, with the advent of wireless technology and file sharing applications, the traditional client-server model has begun to lose its prominence. Instead, information sharing by spontaneously connected nodes has emerged as a new framework. In such networks, all network nodes are equal in terms of capacity and functionality. Moreover, the ownership of the files is not critical, and a file (data item) does not belong to a specific node and hence, is read and written by multiple nodes in the network. Caching an object at various network nodes can play an important role in improving overall system performance by drastically reducing the time to read an object.

In this article, we address the data caching problem in above described multi-hop networks wherein the given data item may be read and written by multiple other network nodes, and the objective is to minimize the total reading, writing, and storage cost by placing a limited number of caches. Here, the cost of reading the data item by a node is defined as the distance to the closest cache node times the read frequency, the cost of writing is defined as the cost of the minimum Steiner tree over the writing node and all the cache nodes times the write frequency, and the storage cost is the given cost of caching the data item at the node.

The rest of the paper is organized as follows. In Section 4.2, we present our network model, formulate the data caching problem addressed in this article, and present an overview of the related work. Section 4.3 presents the optimal dynamic programming algorithm for tree topology networks. In Section 4.4, we design centralized and distributed heuristics for general graph networks. Simulation results are presented in Section 4.5, and concluding remarks in

Section 4.6.

4.2 Data Caching Problem Formulation

In this section, we present our model of the network, give a formal definition of the problem, and present a discussion on related work. We use the term *cache node* to refer to a network node that caches the data item.

Network Model and Notations. We model the network as a connected general graph, $G(V, E)$, where V is the set of nodes/vertices and E is the set of edges. We use n to denote the total number of nodes in the given network, i.e., $n = |V|$. Each edge has a nonnegative weight associated with it. There is a single data item in the network, which is to be cached at selected network nodes. For each node $i \in V$, the frequency of reading the data item is r_i, the frequency of writing the data item is w_i, and the cost of caching (i.e., storing) the data item at node i is s_i. Let d_{ij} denote the shortest distance (minimum total weight) between any two nodes i, j, and let $d(i, M) = \min_{j \in M} d_{ij}$ be the shortest distance from i to some node in a set of nodes M. Also, let $S(X)$ be the optimal cost of a Steiner tree over the set of nodes X. Given a set of cache nodes M where the data item is cached, the total cost of reading the data item by a node i is $r_i d(i, M)$, while the cost of writing by node i is $w_i S(M \cup \{i\})$. The tree used by a writer i to write onto the set of caches is refered to as the *write-tree* for the writer i. Note that we do not assume a server for the data item in the network, since in our model, a server can be looked upon as a predetermined cache node.

Data Caching Problem. The *data caching problem* in the above network model can be defined as follows. Given a network graph $G(V, E)$ and a number P ($1 \leq P \leq n$), select at most P cache nodes such that the total (reading, writing, and storage) cost is minimized. For a given network graph G and a set of cache nodes M, the total cost is denoted by $\tau(G, M)$ and is defined as:

$$\tau(G, M) = \sum_{i \in V} r_i d(i, M) + \sum_{i \in V} w_i S(\{i\} \cup M) + \sum_{i \in M} s_i \tag{4.1}$$

In the above equation, the terms on the right hand side represent total read cost, total write cost, and total storage cost respectively. Essentially, the data caching problem is to select a set of cache nodes M ($|M| \leq P$) such that the total cost $\tau(G, M)$ is minimized.

Related Work. When there are no writers and $P = n$, the data caching problem is exactly the same as well-known facility-location problem. When the number of cache nodes are constrained to be at most P and the cost

is comprised only of reading and storage costs, the data caching problem is the well-known P-median problem. Both the problems (facility-location and P-median) are NP-hard, and a number of constant-factor approximation algorithms have been developed for each of the problems [16, 18, 30], under the assumption that the edge costs in the graph satisfy the triangular inequality. Without the triangular inequality assumption, either problem is as hard as approximating the set cover [30, 37], and therefore cannot be approximated better than $O(\log n)$ unless $\mathbf{NP} \subseteq \tilde{\mathbf{P}}$. Several papers in the literature circumvent the hardness of the facility-location and P-median problems by assuming that the network has a tree topology [35, 36, 48]. In particular, the best known algorithm for solving P-median in trees is by Tamir [48], who gives an $O(Pn^2)$ time dynamic programming algorithm. In this article, we essentially generalize Tamir's algorithm for our data caching problem in trees, and also present centralized and distributed heuristics for general graphs.

In a recent work, Wolfson and Milo [54] consider a simpler version of our data caching problem, wherein there are no storage costs, and the write cost is equal to the minimum spanning tree over the distance graph of the cache nodes. They design optimal algorithms for trees, rings, and complete graphs. In the most related work, Kalpakis et al. [33] consider the problem of finding a Steiner-optimal P-replica set in a tree topology in order to minimize the sum of reading, writing, and storing costs. They developed a very complicated (more than 20 pages of case analysis) optimal dynamic programming algorithm that runs in $O(n^6P^2)$ time and finds a Steiner-optimal replica set of size *exactly* P in tree topologies. In our understanding, their work gives a $O(n^6P^3)$-time algorithm for finding a Steiner-optimal replica set of size *at most* P in trees. In this article, we essentially address the same problem and design a much simpler dynamic programming optimal algorithm that runs in $O(n^2P^2)$ time and finds an optimal set of caches of size at most P. In addition, we design centralized and distributed heuristics to solve the problem in general graph topologies, and show through extensive simulations that our proposed heuristics perform well in practice. In the preliminary version [24] of this work, we proposed an $O(n^2P^3)$ dynamic programming algorithm for the data caching problem in trees with an assumption that read requests are satisfied by an "ancestor" cache node rather than the nearest cache node.[1]

[1]The assumption is not stated in the preliminary version [24], since we failed to realize it at the time of publication. This work presents a correct (i.e., without the assumption) and more efficient dynamic programming algorithm based on an entirely different technique.

4.3 Data Caching in Tree Topology

In this section, we study the data caching problem in special case of a tree topology, and present an optimal dynamic programming algorithm. Before we present our algorithm, we first review Tamir's dynamic programming (DP) algorithm for the P-median problem in a tree topology [48], since it forms the basis of our own proposed algorithm.

Tamir's DP Algorithm for P-Median in Trees. As mentioned before, the P-median problem is to select a set S of at most P nodes that minimizes the sum of the storage costs of nodes in S and the access costs. As in our data caching problem, the access cost is defined as the sum of the distances of each node v in the tree to the node nearest to v in S. Tamir [48] presents an $O(n^2P)$ time DP algorithm for the above. The brief description of the DP algorithm in [48] is as follows. First, [48] presents a linear algorithm to transform an arbitrary tree (rooted at some distinguished node v_1) into a full binary tree, wherein each node either has two children or is a leaf. The transformation guarantees that solving the problem on the original tree is equivalent to solving it on the transformed full binary tree. Let $T = (V, E)$ be the resulting binary tree, where $V = \{v_1, ..., v_n\}$. For each node $v_j \in V$, the subtree rooted at v_j is denoted as T_j, and the set of nodes in T_j is denoted as V_j. Then, for each node v_j in V, [48] computes and sorts the distances from v_j to all nodes in V, and denotes the sequence as $L = \{l_j^1, ..., l_j^n\}$,[2] where $l_j^i \leq l_j^{i+1}$ and $l_j^1 = 0$. The node corresponding to l_j^i is denoted as v_j^i. Based on the above notations, [48] defines the following terms G and F, which can be computed recursively from "leaves to root" using a dynamic programming approach.

- $G(v_j, q, l_j^i)$. It is defined as the optimal value of the subproblem defined on the subtree T_j, given that a total of at least 1 and *at most* q cache nodes can be selected in T_j, and that at least one of them has to be in $\{v_j^1, v_j^2, ..., v_j^i\} \cap V_j$. In the above definition, it is implicitly assumed that there is no interaction between the nodes in T_j and the rest of nodes in T.

- $F(v_j, q, l)$. It is defined as the optimal value of the subproblem defined on the subtree T_j under the following constraints: (i) A total of *at most* q cache nodes can be selected in T_j, (ii) There are already some selected cache nodes in $T - T_j$, and the closest amongst them to v_j is at a distance of l from v_j.

Tamir's dynamic programming (DP) algorithm starts from leaves of T, and recursively computes G and F values at each node in T in terms of the G

[2] [48] uses the notation $\{r_j^1, \ldots, r_j^n\}$ instead.

and F values of its children. The optimal *value* of the problem is given by $\min(G(v_1, P, l_1^n), G(v_1, 0, l_1^n))$, where v_1 is the root of the tree and n is the total number of nodes in the network. The algorithm can be easily modified to select the actual set of cache nodes that yields the optimal value.

4.3.1 Generalizing Tamir's DP to Our Data Caching Problem

Our data caching problem essentially generalizes the P-median problem by including the concept of writers and writing costs in the overall cost. Below, we present our generalized DP algorithm for the data caching problem in trees. First, we start with an overview of our simplified notations. Then, we generalize the definitions of G and F from [48] for our data caching problem, and present the recursive equations for computing G and F values at each node in the tree. Finally, we will use the values of G and F to define another function $\mathcal{G}$ for each node in the network, which essentially solves our data caching problem.

Simplified Notations. Let $T(V, E)$ be a given binary tree with nonnegative edge weights. For clarity, we drop the subscript j from the notations used in [48]. In particular, we use v to represent a node in T (instead of v_j in [48]), and T_v to denote the subtree (or the set of nodes in the subtree) rooted at v. Without loss of generality, we pick some node $\mathcal{R}$ as the root of the given tree. For each non-leaf node $v \in T$, we use v_1 and v_2 to denote v's left and right children. Finally, for each node $v \in T$, we compute and sort the distances from v to all the nodes in T and denote the corresponding node sequence as $\{v^1, ..., v^n\}$, where $d_{vv^i} \leq d_{vv^{i+1}}$ for $i = 1, ..., n-1$ and $v^1 = v$.

Defining G and F using Γ. For the purposes of defining our generalized versions of G and F functions, we first define the total cost $\Gamma(T_v, M, M_o)$ in a subtree T_v due to M, a set of cache nodes inside T_v, where M_o is the set of cache nodes outside T_v. The cost $\Gamma(T_v, M, M_o)$ is defined as:

$$\Gamma(T_v, M, M_o) = \sum_{k \in T_v} r_k d(k, M \cup M_o) + \sum_{k \in M} s_k + \sum_{k \notin T_v} w_k S(\{v\} \cup M) + \sum_{k \in T_v} w_k S(\{k\} \cup \{v\} \cup M)$$

The above expression includes the storage costs of the set M of cache nodes inside T_v, the total reading costs of all the nodes in T_v using the cache nodes M as well as M_o, and total writing cost over the edges in T_v due to all the writers in T. For the writing cost, we assume (even if M_o is empty) that there *are* some cache nodes outside T_v, i.e, v is part of each write-tree.[3] Note that

[3]Eventually, we will define another function $\mathcal{G}$ that computes the writing costs assuming no outside caches nodes. For clarity of presentation, we defer definition of $\mathcal{G}$.

M_o can also be represented by the node in M_o that is closest to v, but we use the above notation for sake of clarity in defining G and F.

Defining $G(v, q, v^i)(1 \leq q \leq |T_v|)$. We define $G(v, q, v^i)$ as the optimal cost Γ in the subtree T_v given that there are *exactly* q cache nodes in T_v and the closest to v among them is at most d_{vv^i} distance away from v. Also, the access costs are computed using only the caches inside T_v (i.e., $M_o = \{\}$). More formally,

$$G(v, q, v^i) = \min_{|M|=q, d(v,M) \leq d_{vv^i}} \Gamma(T_v, M, \{\}).$$

Defining $F(v, q, v^i)(0 \leq q \leq |T_v|$. We define $F(v, q, v^i)$ as the optimal cost Γ in the subtree T_v given that there are *exactly* q cache nodes in T_v and the closest outside cache is v^i. More formally,

$$F(v, q, v^i) = \min_{|M|=q} \Gamma(T_v, M, \{v^i\}).$$

Note that $F(v, q, v^i)$ is not defined when $v^i \in T_v$, and

$$F(v, 0, v^i) = \sum_{k \in T_v} (r_k d_{kv^i} + w_k d_{kv}).$$

Recursive Equations for Computing G and F. We now define recursive equations for computing G and F at a node v in terms of the G and F at the children of v. The G and F values will be eventually used to compute the solution of our data caching problem.

G and F Values at a Leaf Node. When v is a leaf node, the value G is defined only for $q = 1$ and F is defined for $q = 0$ or 1. Also, F is not defined for $v^i = v$, i.e., $i = 1$. Now, it is easy to see that:

$$\begin{aligned} G(v, 1, v^i) &= s_v, & i = 1, ..., n \\ F(v, 0, v^i) &= r_v d_{vv^i}, & i = 2, ..., n \\ F(v, 1, v^i) &= s_v, & i = 2, ..., n \end{aligned}$$

Intuition for the Below Recursive Equations. Recall that v_1 and v_2 are used to denote the two children of v. Now, for a non-leaf node v, the cost $\Gamma(T_v, M, M_o)$ can be expressed in terms of the function Γ over T_{v_1} and T_{v_2}, the access and storage cost for node v, and the write cost over the edges (v, v_1) and (v, v_2). The exact expression for the above depends on the composition of M, i.e., whether M includes v, a node in T_{v_1}, and/or a node in T_{v_2}. Based on the above

observation, the values G and F at a node v can be appropriately defined in terms of G and F values at its children v_1 and v_2, as shown in the following paragraphs.

Computing $G(v, q, v^1)$ (i.e., for $i = 1$). Here, since $i = 1$, the node v is also a cache node. First, when $q = 1$, we have

$$G(v, 1, v^1) = F(v_1, 0, v) + F(v_2, 0, v) + d_{vv_1} \sum_{k \in T_{v_1}} w_k + d_{vv_2} \sum_{k \in T_{v_2}} w_k.$$

Note that v is an outside cache node for the subtrees T_{v_1} and T_{v_2}. For $q > 1$, the total cost on T_v includes the storage cost on node v, the cost on the subtrees T_{v_1} and T_{v_2}, and the write cost on the edges of (v, v_1) and (v, v_2). In particular, there are three cases:

(a) There are no cache nodes in T_{v_1}, but there is at least one cache node in T_{v_2}. In this case, the edge (v, v_1) is included in the write-trees of only the writer nodes in T_{v_1}. However, the edge (v, v_2) is included in the write-trees of all writers in the network.

(b) There are no caches nodes in T_{v_2}, but there is at least one cache node in T_{v_1}. This case is similar to the above case (a).

(c) There is at least one cache node in T_{v_1} as well as T_{v_2}; this case is only possible if $q > 2$. In this case, the path (v_1, v, v_1) is included in the write-trees of each writer node in the network.

Based on the above three cases, the value $G(v, q, v^1)$ for $q > 1$ can be defined as below.

$$G(v, q, v^1) = s_v + \min \left(\begin{array}{l} F(v_1, 0, v) + F(v_2, q-1, v) + d_{vv_1} \sum_{k \in T_{v_1}} w_k + d_{vv_2} \sum_{k \in T} w_k, \\ F(v_1, q-1, v) + F(v_2, 0, v) + d_{vv_2} \sum_{k \in T_{v_2}} w_k + d_{vv_1} \sum_{k \in T} w_k, \\ \min_{1 \le q_1 < q-1} \left(F(v_1, q_1, v) + F(v_2, q-1-q_1, v) + d_{v_1 v_2} \sum_{k \in T} w_k \right) \end{array} \right)$$

Computing $G(v, q, v^i)$ for $1 < i \le n$. Here, there are two cases:

1. In the first case, at least one of the nodes in $\{v^1, v^2, \ldots, v^{i-1}\}$ is selected as a cache node. In this case, $G(v, q, v^i)$ is equal to $G(v, q, v^{i-1})$. Note that this case includes the scenario when $v^i \notin T_v$.

2. In the second case, v^i *must* be selected as a cache node. Here, there are two subcases, viz., (2-a): $v^i \in T_{v_1}$, (2-b): $v^i \in T_{v_2}$.

Let us analyze the subcase (2-a); the subcase (2-b) is similar. We denote the total cost for the subcase (2-a) as Q_1, and compute it as a minimum of two values: (i) When there are no cache nodes in T_{v_2}, (ii) when there is at least one cache node in T_{v_2}. The above case analysis yields the following expression for $G(v, q, v^i)$.

$$\begin{aligned} G(v,q,v^i) &= \min\big(G(v,q,v^{i-1}), Q_1\big) && \text{if } v^i \in T_{v_1} \\ G(v,q,v^i) &= \min\big(G(v,q,v^{i-1}), Q_2\big) && \text{if } v^i \in T_{v_2} \end{aligned}$$

where

$$Q_1 = r_v d_{vv^i} + \min\left(\begin{array}{l} G(v_1,q,v^i) + F(v_2,0,v^i) + d_{vv_1}\sum_{k\in T} w_k + d_{vv_2}\sum_{k\in T_{v_2}} w_k, \\ \min_{1\le q_1<q}\Big(G(v_1,q_1,v^i) + F(v_2,q-q_1,v^i) + d_{v_1v_2}\sum_{k\in T} w_k\Big) \end{array}\right)$$

$$Q_2 = r_v d_{vv^i} + \min\left(\begin{array}{l} G(v_2,q,v^i) + F(v_1,0,v^i) + d_{vv_2}\sum_{k\in T} w_k + d_{vv_1}\sum_{k\in T_{v_1}} w_k, \\ \min_{1\le q_1<q}\Big(G(v_2,q_1,v^i) + F(v_1,q-q_1,v^i) + d_{v_1v_2}\sum_{k\in T} w_k\Big) \end{array}\right)$$

As shown in the above equation for Q_1, when there are no caches nodes in T_{v_2}, the edge (v, v_1) is part of the write-tree for all the writers in the network, and the edge (v, v_2) is part of the write-tree for all the writers in T_{v_2}. On the other hand, when there is at least one cache node in T_{v_2}, the path (v_1, v, v_2) is part of the write-tree of all writer nodes in the network. On the other hand, The cost Q_2 is similarly defined.

Computing $F(v, q, v^i)$. Recall that $F(v, q, v^i)$ is the optimal value of $\Gamma(T_v, M, \{v^i\})$ where M is a set of q cache nodes in T_v, and v^i is not in T_v. If M includes a cache node $u \in T_v$ such that $d_{uv} < d_{vv^i}$, then the optimal value of $\Gamma(T_v, M, \{v^i\})$ is $G(v, q, v^{i-1})$. Else, v^i is the closest cache to v (in particular, v is not a cache node), and there are the following three cases. (i) There are no caches nodes in T_{v_2}, (ii) There are no cache nodes in T_{v_1}, and (iii) There is at least one cache node in T_{v_1} as well as T_{v_2}. For the last case, note that the cache node closest to v_1 (v_2) outside of T_{v_1} (T_{v_2}) is still v^i, since M does not include any node u such that $d_{uv} < d_{vv^i}$. Also, since $q > 1$, there *must* be a cache node in either T_{v_1} or T_{v_2}. The above case analysis and observations yield the following equation for computing F.

$$F(v, q, v^i) = \min\{G(v, q, v^{i-1}), Q_3\}$$

where

$$Q_3 = r_v d_{vv^i} + \min\left(\begin{array}{l} F(v_1,q,v^i) + F(v_2,0,v^i) + d_{vv_1}\sum_{k\in T} w_k + d_{vv_2}\sum_{k\in T_{v_2}} w_k, \\ F(v_2,q,v^i) + F(v_1,0,v^i) + d_{vv_2}\sum_{k\in T} w_k + d_{vv_1}\sum_{k\in T_{v_1}} w_k, \\ \min_{1\le q1<q}\Big(F(v_1,q_1,v^i) + F(v_2,q-q1,v^i) + d_{v_1v_2}\sum_{k\in T} w_k\Big) \end{array}\right)$$

Solving the Data Caching Problem. The computation of the above defined G and F values does not solve the data caching problem, since definition of Γ (and hence, G) assumes (for the purposes of write cost) that there is an ouside cache node. Thus, we now define another function $\mathcal{G}$, which is similar to the definition of G but assumes (even for writing costs) that there are no outside cache nodes. More formally, for a given node v and $1 \leq i \leq n$ and $1 \leq q \leq |T_v|$, we define $\mathcal{G}(v, q, v^i)$ as

$$\mathcal{G}(v,q,v^i) = \min_{|M|=q, d(v,M) \leq d_{vv^i}} \Big(\sum_{k \in T_v} r_k d(k,M) + \sum_{k \in M} s_k + \sum_{k \notin T_v} w_k S(\{v\} \cup M) + \sum_{k \in T_v} w_k S(\{k\} \cup M \Big).$$

The function $\mathcal{G}$ at a node v can be computed in terms of G, F, and $\mathcal{G}$ values at v_1 and v_2, as shown below. The below equations are similar to the recursive equations for G, except that when all the cache nodes are in T_{v_1} (T_{v_2}), the edge (v, v_1) ((v, v_2)) is only used by the write-trees for writers *outside* T_{v_1} (T_{v_2}).

$$\begin{array}{rcll} \mathcal{G}(v,q,v^1) & = & G(v,q,v^1) & \\ \mathcal{G}(v,q,v^i) & = & \min\big(\mathcal{G}(v,q,v^{i-1}), Q_4\big) & \text{if } \ v^i \in T_{v_1} \\ \mathcal{G}(v,q,v^i) & = & \min\big(\mathcal{G}(v,q,v^{i-1}), Q_5\big) & \text{if } \ v^i \in T_{v_2} \end{array}$$

where

$$Q_4 = r_v d_{vv^i} + \min \left(\begin{array}{l} \mathcal{G}(v_1,q,v^i) + F(v_2,0,v^i) + d_{vv_1} \sum_{k \notin T_{v_1}} w_k + d_{vv_2} \sum_{k \in T_{v_2}} w_k, \\ \min_{1 \leq q_1 < q} \Big(G(v_1,q_1,v^i) + F(v_2,q-q_1,v^i) + d_{v_1 v_2} \sum_{k \in T} w_k \Big) \end{array} \right)$$

$$Q_5 = r_v d_{vv^i} + \min \left(\begin{array}{l} \mathcal{G}(v_2,q,v^i) + F(v_1,0,v^i) + d_{vv_2} \sum_{k \notin T_{v_2}} w_k + d_{vv_1} \sum_{k \in T_{v_1}} w_k, \\ \min_{1 \leq q_1 < q} \Big(G(v_2,q_1,v^i) + F(v_1,q-q_1,v^i) + d_{v_1 v_2} \sum_{k \in T} w_k \Big) \end{array} \right)$$

Data Caching Problem Solution. The solution of the data caching problem can now be computed as $\min_{1 \leq q \leq P} \mathcal{G}(\mathcal{R}, q, \mathcal{R}^n)$, where $\mathcal{R}$ is the root and $\mathcal{R}^n$ is the farthest node in the network from $\mathcal{R}$. Starting from the leaves towards the root, for each node v, we compute G, F, and $\mathcal{G}$ values for each q and i. Thus, there are total $3n^2P$ values to be computed. If we *precompute* ($\sum_{k \in T_v} w_k$) and ($\sum_{k \notin T_v} w_k$) terms for all v in total $O(n^2)$ time, then computation of each G or F or $\mathcal{G}$ value can be done in $O(P)$ time. Thus, the overall time complexity of our proposed dynamic programming algorithm is $O(n^2P^2)$.

4.4 General Graph Topology

In this section, we address the data caching problem in a general graph topology. In a general graph, the data caching problem is NP-hard, since it reduces

to the facility-location problem when the write frequencies are zero. Here, we first design a centralized greedy algorithm, and then present a distributed implementation of the centralized algorithm. We have used similar techniques in our recent work [49] on a related problem of data caching under update cost constraint. We will show through simulations that the centralized heuristic developed in this section perform close to the optimal solution in small general graph networks.

4.4.1 Centralized Greedy Algorithm

We now present a polynomial-time Centralized Greedy Algorithm for the data caching problem. We start with defining the concept of benefit of a set of nodes.

Definition 4 (Benefit of Node) Let M be the set of nodes that have been already selected as cache nodes by the Centralized Greedy Algorithm at some stage. The *benefit* of an arbitrary node A, denoted as $\beta(A, M)$, is the reduction in total cost due to selection of A as a cache node. More formally, $\beta(A, M) = \tau(G, M) - \tau(G, M \cup \{A\})$, where $\tau(G, M)$ is the total cost of selecting a set of cache nodes M in graph G, as defined in Equation 4.1. □

Note that since the minimum-cost Steiner tree problem is NP-hard, we adopt the 2-approximation Steiner tree algorithm [22] to compute writing costs.

Based on the above definition of benefit, our proposed Greedy Algorithm can be described as follows. Let M be the set of cache nodes selected at any given stage. Initially, M is empty. At each stage of the Greedy Algorithm, we add to M the node A that has the highest benefit with respect to M at that stage. The process continues until P caches nodes have been selected or there is no node with positive benefit. The running time of the above described algorithm is $O(Pn^5)$, since the time to compute a 2-approximation Steiner tree over a set of s nodes is $O(sn^2)$.

4.4.2 Distributed Greedy Algorithm

In this subsection, we present a distributed localized implementation of the Centralized Greedy Algorithm. To facilitate communication between nodes, we assume presence of a *coordinator* in the network. Our Distributed Greedy Algorithm consists of rounds. During each round, each non-cache node A estimates the benefit (as described in the next paragraph) of caching the data item at A. If the benefit estimate at a node A is positive and is the maximum among all its non-cache neighbors, then A decides to cache the data item. At the end of a round, the coordinator node gathers information about the cache

nodes newly added. The number of cache nodes that can be further added is then broadcast by the coordinator to the entire network. The algorithm terminates, when either more than P cache nodes have already been added or no more cache nodes were added in a round.

Estimation of $\beta(A, M)$. A non-cache node A considers only its "local" traffic and estimation of distance to the nearest cache node, to estimate $\beta(A, M)$, the benefit with respect to an already selected set of cache nodes M. In particular, a node A observes its local traffic, i.e., the data access requests that A forwards to other cache nodes. Of course, the local traffic of a node includes its own data requests. We estimate the benefit of caching the data item at A as

$$\beta(A, M) = fd - s_a - d \sum_{i \in V} w_i,$$

where f is the frequency of the local data access traffic observed at A, d is the distance to the nearest cache from A (which is computed as shown in the next paragraph), s_a is the storage cost at A, and w_i is the write frequency at a node i in the network. In the above equation, we have estimated the increase in total writing cost due to caching at A as $d \sum_{i \in V} w_i$. The local traffic f can be computed if we let the normal network traffic (using only the already selected cache nodes in previous rounds) run for some time between successive rounds.

Estimation of d – the distance to the nearest cache from A. Let A be a non-cache node, and T_A be the shortest path tree from the coordinator to the set of communication neighbors of A. Let $C \in M$ be the cache node in T_A that is closest to A. In the above Distributed Greedy Algorithm, we estimate d to be $d(A, C)$, the distance from A to C. The value $d(A, C)$ can be computed in a distributed manner at the start of each round as follows. As mentioned before, the coordinator initiates a new round by broadcasting a packet containing the remaining number constraint to the entire network. If we append to this packet all the cache nodes encountered on the way, then each node should get the set of cache nodes on the shortest path from the server to itself. Now, to compute $d(A, C)$, each node only needs to exchange the above information with all its immediate neighbors.

4.5 Performance Results

In this section, we evaluate the relative performances of the various cache placement algorithms proposed in our article.

Experiment Setup. We use a network of 50 to 400 nodes placed randomly in a square region of size 30×30. We consider unit-disk graphs wherein two nodes can communicate with each other if the distance between them is less than a

given number (called the *transmission radius*). For our simulations, we use a transmission radius of 9, which is the minimum to keep even small networks of size 50 connected. We vary various parameters such as network size, the maximum number of cache nodes P, percentage of readers and writers in the network, and the *ratio* R of average write frequency to average read frequency. Note that in practical settings we expect R to be low. The read frequency of a reader node is chosen to be a random number between 0 and 100, the write frequency of a writer node is chosen to be a random number between 0 and $100R$, and the storage cost at a node is chosen to be a random number between 0 and 100. Each data point in the graph plots is an average over five different random graph topologies. In our simulations, we compare the performance of various data caching placement algorithms, viz., Centralized Greedy Algorithm, Distributed Greedy Algorithm, and Dynamic Programming Algorithm (DP) on the spanning tree with near-minimum stretch factor (as described below).

Computing a Spanning Tree with Near-Minimum Stretch Factor. Before presenting the algorithm for constructing a spanning tree with near-optimal stretch factor, let us first define *stretch factor*. Consider a graph $G = (V, E)$; the *stretch factor* of an edge $(u, v) \in E$ in a subgraph $G'(V, E' \subset E)$ is defined as the shortest distance between u and v in G'. The *stretch factor* of the subgraph G' is defined as the maximum stretch factor over all edges in G. The minimum stretch-factor spanning tree problem is to find a spanning tree with minimum stretch factor in the given graph. The problem is known to be NP-hard [**?**].

We now describe an approximation algorithm for the above problem [**?**] in unit-disk graphs. We will use this algorithm to construct a near-minimum stretch-factor spanning tree, which will be input to our dynamic programmg algorithm (since it runs only on tree topologies). The approximation algorithm consists of the following steps.

1. Construct a dominating set of the given unit-disk graph.
2. Connect the nodes in the dominating set that are at most three hops away. This results in a connected dominating graph.
3. Extract the Gabriel Graph (which is planar) from the above connected dominating graph.
4. Compute the dual graph of the Gabriel Graph. The *dual graph* contains a vertex for every face of the Gabriel Graph, and an edge between any two adjacent faces. The weight of the edge in the dual graph is the number of common edges of the corresponding faces in the Gabriel Graph.

5. We now associate an appropriate defined weight with each vertex in the above dual graph, and then, construct a "shortest path tree" in the above dual graph.

6. Finally, in the Gabriel Graph, we delete a common edge between any two adjacent faces that are connected in the above constructed shortest-path tree in the dual graph.

The resulting graph can be shown [**?**] to be a spanning tree with a stretch factor of $(OPT)^4$, where OPT is the optimal (minimum) stretch factor.[4]

Comparison with Optimal Algorithm in Small Networks. An optimal solution for the data caching problem can be computed by looking at all $O(n^P)$ subsets of nodes of size at most P, and picking the subset of nodes that gives the minimum total cost as the solution. Due to the high time complexity of the above algorithm, we choose the network size $n = 50$ and vary P from 1 to upto 6. We pick R (the ratio of average write frequency to the average read frequency) as 0.1, since it was just small enough to result in maximum number of cache nodes being selected. We observe in Figure 4.1 that the Centralized Greedy Algorithm performs very close to the optimal cost. Thus, in the following experiments, we use the Centralized Greedy Algorithm as a benchmark of comparison. We also observe that the DP algorithm performs only about 15% worse than the optimal algorithm.

Varying R. In this experiment, we vary R (the ratio of average read frequency to the average write frequency) from 0.001 to 0.1 in a network of size 200 with P (the maximum number of cache nodes allowed) as 25. We keep the percentage of readers and writers in the network at 50%. Figure 4.2 plots the total cost $\tau(G, M)$ corresponding to the set M of cache nodes delivered by various algorithms for given parameters. We see that the Centralized Greedy outperforms the Distributed Greedy Algorithm only by about 15%. However, when R is small, the centralized and distributed greedy algorithms perform very closely, but their relative performance becomes almost constant after $R = 0.02$. This implies that the estimation of writing costs done by the Distributed Greedy Algorithm is not as accurate as the estimation of reading costs. In contrast, we see that the DP algorithm actually performs close to the Centralized Greedy for very low values of R. For higher values of R, the DP algorithm performs worse than the Distributed Greedy. Thus, the strategy of extracting the shortest path tree rooted at an appropriate node seems effective when the writing cost is relatively very low. For $R = 0.1$, we observed that the number of cache

[4]We note that the best known approximation for the minimum stretch-factor spanning tree problem is $\log n$ [**?**, **?**]; however, we choose the technique from [**?**] for the sake of its relative simplicity.

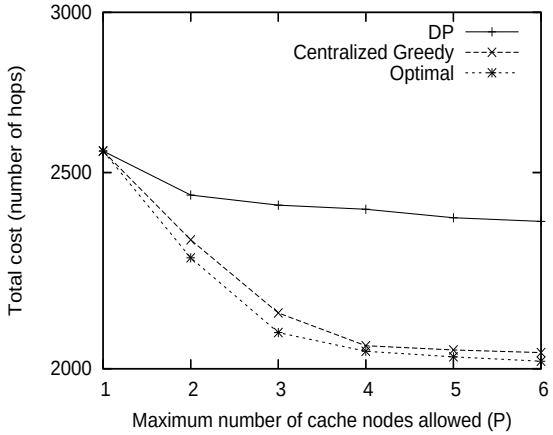


Figure 4.1: Comparison of Centralized Greedy Algorithms with the optimal algorithm. Here, the network size is 50, R (the ratio of average write to average read frequency) as 0.1, and percentage of readers and writers is 50%.

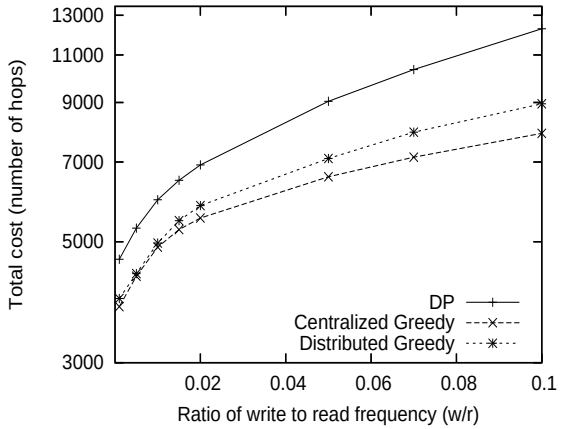


Figure 4.2: Varying R, the ratio of average write to average read frequency. Here, the network size is 200, $P = 25$, percentage of readers and writers is 50.

nodes selected by any algorithm was very low (1 or 2). Thus, we did not increase the value of R beyond 0.1. Based on Figure 4.2, we fix R as 0.02 for all the remaining experiments, since for $R = 0.02$ the number of cache nodes is large enough (around 10) and the relative performance observed at $R = 0.02$ is representative of the general trend.

Varying Network Size. In Figure 4.3, we vary the network size from 100 to 400 and plot $\tau(G, M)$ corresponding to the solution M delivered by various algorithms. As suggested before, we fix $P = 25$ and $R = 0.02$. Also, the percentage of readers and writers in the network is kept as 50%. In Figure 4.3, we can see that the Centralized Greedy Algorithm and the Distributed Greedy Algorithm perform quite closely; both perform better than the DP algorithm. More importantly, we observe that the relative performance of the various algorithms remains relatively stable, and hence, in all other simulations, we fix the network size to be 200.

Varying Percentage of Readers and Writers. In Figure 4.4 and Figure 4.5, we vary the percentage of reader and writer nodes respectively in the network and plot the values of $\tau(G, M)$ for the solution delivered by various algorithms. As suggested in previous paragraphs, we fix R as 0.02 and the network size as 200. In addition, we use P as 25. In Figure 4.4, we vary the percentage of reader nodes from 10 to 100%, while keeping the percentage of

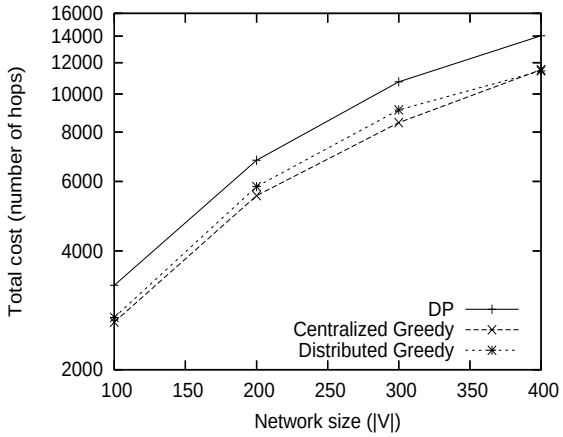


Figure 4.3: Varying network size. Here, $P = 25, R$ (the ratio of average write to average read frequency) is 0.02, and percentage of readers and writers is 50.

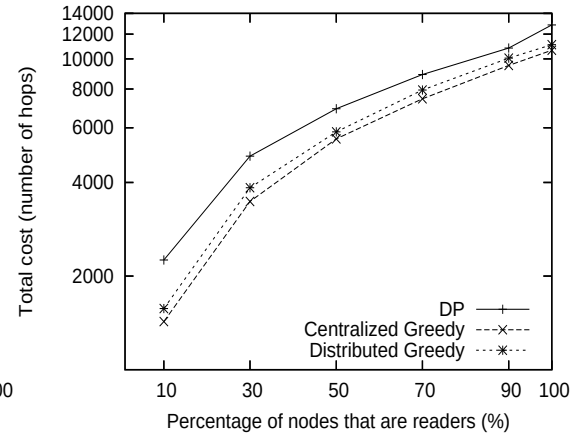


Figure 4.4: Varying percentage of reader nodes in the network. Here, the network size is 200, $P = 25$, $R = 0.02$, and the percentage of writer nodes is 50%.

writer nodes fixed at 50%. Similarly, in Figure 4.5, we vary the percentage of writer nodes from 0 to 100%, while keeping the percentage of reader nodes fixed at 50%. We observe that the relative performance of the various algorithms remains largely unchanged with the change in percentages of readers or writers. In generally, we see the performance gap between various algorithm to be limited by 10-15%.

Varying P. In Figure 4.6, we vary P, the maximum number of cache nodes allowed, and plot $\tau(G, M)$ for various algorithms. We see that with the increase in P, the relative performance gap between the Centralized and Distributed Greedy Algorithms reduces. After $P = 10$, the performance of the various algorithms remains unchanged since for the given parameter values all algorithms place at most 10 caches. Again, we see the performance gap between various algorithm to be limited by 10-15%.

4.6 Conclusions

In this paper, we addressed the problem of selection on nodes to cache a data item in a network, wherein multiple nodes can read or update the data items, individual nodes have storage limitations, and there is a limit on the number of nodes that can be selected to cache the data item. The objective of our problem was to minimize the sum of appropriately defined total reading cost, writing cost, and storage cost. For the above data caching problem, we designed

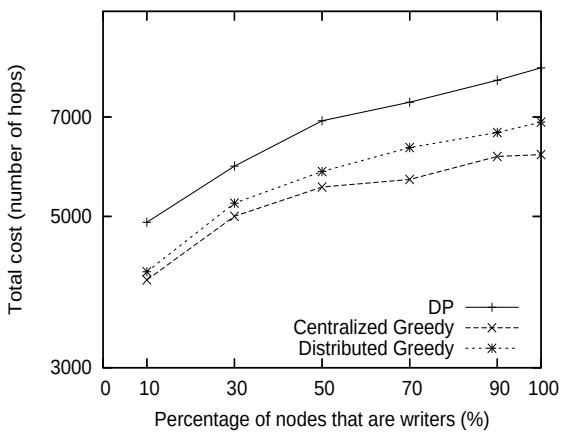


Figure 4.5: Varying percentage of writer nodes in the network. Here, the network size is 200, $P = 25$, $R = 0.02$, and percentage of reader nodes is 50%.

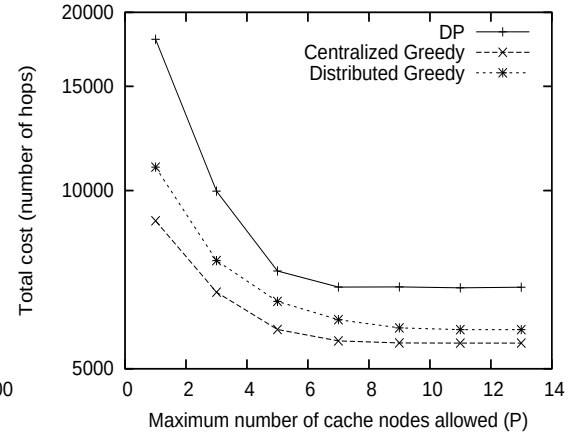


Figure 4.6: Varying P. Here, the network size is 200, $R = 0.02$, and percentage of readers and writers is 50%.

an optimal dynamic programming algorithm for tree networks. In addition, for general network graphs, we proposed Centralized Greedy and Distributed Greedy heuristics, and evaluated the performance of our proposed algorithms through extensive simulations. We observe that the Centralized Greedy performs very close to the optimal algorithm for small networks, and for larger networks, the Distributed Greedy and the dynamic programming algorithm on an appropriately extracted tree perform very close to the Centralized Greedy.

Chapter 5

Conclusion and Future Work

We have studied the data placement problems in ad hoc and sensor networks. In particular, we have proposed to solve the data placement problem under different constraints: memory constraint, update cost constraint, and number of allowable caches constraint. We have developed optimal, near optimal centralized algorithms in either tree networks or general networks. We also present their distributed implementations.

There are still un-answered questions: How to design efficient caching algorithms when nodes fail or join and leave the networks dynamically? Can caching technique take advantage of the wireless broadcast medium for better system performance? To name a few.

Moreover, ad hoc networks rely on the cooperation of participating nodes to route messages from source to destination that are outside each other's communication range. Data forwarding costs nodes both bandwidth and battery energy, making the user who operates the node unwilling to cooperate. This also happens to the caching strategies – with limited local memory and battery, each node prefers to cache the most beneficial data items from its own perspective, not the ones for the network as a whole. Our distributed caching techniques have to be modified to address this important observation. There is no any prior work addressing such *selfish caching* in a purely distributed manner in ad hoc networks. How to analyze all these behavior of nodes in ad hoc networks involves game theoretical approach, in which each node is a rational player who wants to maximize its own utility. The central concept *benefit* in this work captures this very idea. However, instead of maximizing each node's own benefit, how to design mechanism to guide nodes' behaviors to achieve benefit maximization of the whole network remains a challenge.

Bibliography

[1] A. Aazami, S. Ghandeharizadeh, and T. Helmi. Near optimal number of replicas for continuous media in ad-hoc networks of wireless devices. In *Intl. Workshop on Multimedia Information Systems*, 2004.

[2] A. Adya, P. Bahl, J. Padhye, A. Wolman, and L. Zhou. A multi-radio unification protocol for IEEE 802.11 wireless networks. In *Proceddings of the International Conference on Broadband Communications, networks, and systems(BROADNETS)*, 2004.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 2002.

[4] B. Awerbuch, Y. Bartal, and A. Fiat. Heat & dump: Competitive distributed paging. In *International Conference on Foundations of Computer Science (FOCS)*, 1993.

[5] B. Badrinath, M. Srivastava, K. Mills, J. Scholtz, and K. Sollins, editors. *Special Issue on Smart Spaces and Environments,* IEEE Personal Communications, 2000.

[6] I. Baev and R. Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2001.

[7] P. Bahl and V. N. Padmanabhan. Radar: An in-building RF-based user-location and tracking system. In *Proceedings of the IEEE Conference on. Computer Communications (INFOCOM)*, 2000.

[8] G. Barish and K. Obraczka. World wide web caching: Trends and technologies. *IEEE Communications Magazine, Internet Technology Series*, 2000.

[9] P. Berman and V. Ramaiyer. Improved approximation algorithms for the steiner tree problem. *Journal of Algorithms*, 17:381–408, 1994.

[10] T. Berners-Lee and H. F. Mielsen. Propagation, caching and replication on the web. http://www.w3.org/Propagation.

[11] S. Bhattacharya, H. Kim, S. Prabh, and T. Abdelzaher. Energy-conserving data placement and asynchronous multicast in wireless sensor networks. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003.

[12] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the IEEE Conference on. Computer Communications (INFOCOM)*, 1999.

[13] J. Broch, D. A. Maltz, D. B. Johnson, Y-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of the International Conference on Mobile Computing and Networking (MobiCom)*, 1998.

[14] N. Bulusu, J. Heidemann, and D. Estrin. GPS-less low cost outdoor localization for very small devices. *IEEE Personal Communications Magazine*, 7(5), 2000.

[15] G. Cao. Proactive power-aware cache management for mobile computing systems. *IEEE Transactions on Computer*, 51(6), 2002.

[16] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *International Conference on Foundations of Computer Science (FOCS)*, 1999.

[17] M. Chu, H. Haussecker, and F. Zhao. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. *IEEE Journal of High Performance Computing Applications*, 2002.

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

[19] E. Cohen and S. Shenkar. Replication strategies in unstructured peer-to-peer networks. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2002.

[20] D. Estrin, R. Govindan, and J. Heidemann, editors. *Special Issue on Embedding the Internet,* Communications of the ACM, volume 43, 2000.

[21] K. Fall and K. Varadhan (Eds.). The *ns* manual. At http://www-mash.cs.berkeley.edu/ns/.

[22] E. N. Gilbert and H. O. Pollak. Steiner minimal trees. *SIAM Journal Applied Math*, 16, 1968.

[23] H. Gupta. *Selection and Maintenance of Views in a Data Warehouse*. PhD thesis, Computer Science Department, Stanford University, 1999.

[24] H. Gupta and B. Tang. Data caching under number constraint. In *Proceedings of the International Conference on Communications (ICC)*, 2006.

[25] T. Hara. Effective replica allocation in ad hoc networks for improving data accessibility. In *Proceedings of the IEEE Conference on. Computer Communications (INFOCOM)*, 2001.

[26] T. Hara. Cooperative caching by mobile clients in push-based information systems. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 2002.

[27] T. Hara. Replica allocation in ad hoc networks with periodic data update. In *Intl. Conf. on Mobile Data Management (MDM)*, 2002.

[28] B. Hofmann-Wellenhof, H. Lichtenegger, and J. Collins. *Global Positioning System: Theory and Practice*. Springer-Verlag Telos, 1997.

[29] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the International Conference on Mobile Computing and Networking (MobiCom)*, 2000.

[30] 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), 2001.

[31] S. Jin. Replication of partitioned media streams in wireless ad hoc networks. In *ACM MULTIMEDIA*, 2004.

[32] S. Jin and L. Wang. Content and service replication strategies in multi-hop wireless mesh networks. 2005.

[33] K. Kalpakis, K. Dasgupta, and O. Wolfson. Steiner-optimal data replication in tree networks with storage costs. In *International Database Engineering and Applications Symposium (IDEAS)*, 2001.

[34] Brad Karp and H.T Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proceedings of the International Conference on Mobile Computing and Networking (MobiCom)*, 2000.

[35] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE Transactions on Networkings (TON)*, 8, 2000.

[36] B. Li, M. J. Golin, G. F. Italiano, and X. Deng. On the optimal placement of web proxies in the internet. In *Proceedings of the IEEE Conference on. Computer Communications (INFOCOM)*, 1999.

[37] J.-H. Lin and J. Vitter. Approximation algorithms for geometric median problems. *Information Processing Letters*, 44(5), 1992.

[38] P. Nuggehalli, V. Srinivasan, and C. Chiasserini. Energy-efficient caching strategies in ad hoc wireless networks. In *Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2003.

[39] C. Perkins, editor. *Ad Hoc Networking*. Addison Wesley, 2001.

[40] C. Perkins and P. Bhagwat. Highly dynamic dsdv routing for mobile computers. In *SIGCOMM*, 1994.

[41] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 1994.

[42] S. Prabh and T. Abdelzaher. Energy-conserving data cache placement in sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 1(2):178–203, November 2005.

[43] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *Proceedings of the IEEE Conference on. Computer Communications (INFOCOM)*, 2001.

[44] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu. Data-centric storage in sensornets with ght, a geographic hash table. *Mobile Networks and Applications*, 8(4), 2003.

[45] B. Sheng, Q. Li, and W. Mao. Data storage placement in sensor networks. In *Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2006.

[46] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. Data-centric storage in sensornets. 2002.

[47] C. Swamy and A. Kumar. Primal-dual algorithms for connected facility location problems. In *Intl. Workshop on APPROX*, 2002.

[48] A. Tamir. An $o(pn^2)$ algorithm for p-median and related problems on tree graphs. *Operations Research Letters*, 19, 1996.

[49] B. Tang, S. Das, and H. Gupta. Cache placement in sensor networks under update cost constraint. In *Proceedings of the International Conference on AD-HOC Networks and Wireless(ADHOCNOW)*.

[50] B. Tang, Samir Das, and Himanshu Gupta. Benefit-based data caching in ad hoc networks. 2006.

[51] B. Tang and H. Gupta. Cache placement in sensor networks under update cost constraint. *Journal of Discrete Algorithms*, 5(3):422–435, September 2007.

[52] A. Vigneron, L. Gao, M. J. Golin, G. F. Italiano, and B. Li. An algorithm for finding a k-median in a directed tree. *Info. Proc. Letters*, 74, 2000.

[53] R. Wattenhofer, L. Li, P. Bahl, and Y.-M. Wang. Distributed topology control for wireless multihop ad-hoc networks. In *Proceedings of the IEEE Conference on. Computer Communications (INFOCOM)*, 2001.

[54] O. Wolfson and A. Milo. The multicast policy and its relationship to replictated data placement. *ACM Transactions on Data Base Systems*, 16(1):181–205, 1991.

[55] J. Xu, B. Li, and D. L. Lee. Placement problems for transparent data replication proxy services. *IEEE Journal on Selected Areas in Communications*, 20(7), 2002.

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

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