Optimal Storage Placement for Tree-Structured Networks with Heterogeneous Channel Costs

Ge-Ming Chiu, *Member, IEEE,* Li-Hsing Yen, *Member, IEEE,* and Tai-Lin Chin, *Member, IEEE*

Abstract—This work considers data query applications in tree-structured networks, where a given set of source nodes generate (or collect) data and forward the data to some halfway storage nodes for satisfying queries that call for data generated by all source nodes. The goal is to determine an optimal set of storage nodes that minimizes overall communication cost. Prior work toward this problem assumed homogeneous channel cost, which may not be the case in many network environments. We generalize the optimal storage problem for a tree-structured network by considering heterogeneous channel costs. The necessary and sufficient conditions for the optimal solution are identified, and an algorithm that incurs a linear time cost is proposed. We have also conducted extensive simulations to validate the algorithm and to evaluate its performance.

Index Terms—Algorithm/protocol design and analysis, Information Storage and Retrieval, Network problems.

1 Introduction

TREE topology is a network architecture commonly adopted by distributed networking applications due to its simple routing rules and ease of management. In addition, tree structure provides a baseline connection paradigm for a given set of communicating nodes. For instance, operations of wireless sensor networks are frequently based on tree topology.

Data query is one of the most important services for many networking applications. In such a service, a given set of nodes, called *source nodes*, are responsible for collecting or generating data periodically or in response to certain type of events. These data are required for answering queries issued by *requesters* in the network. In particular, a query often calls for data generated by *all* source nodes. For example, a query may ask for average of the collected temperature readings in certain timeframe in a wireless sensor network.

If each source node places its data in its own storage, a requesting node has to retrieve needed data from all source nodes, in order to satisfy its query demand. This is called *pull* mode of data query [8]. Alternatively, source nodes may forward their data to all possible requesters every time they generate new data, which is called *push* mode [8]. In general, source nodes may forward their data to some halfway nodes referred to as *storage nodes*, where data are stored for serving query demands from requesters. With this scheme, a requester only needs to issue a query message to the closest storage node to retrieve needed data. This operating mode is, in fact, a hybrid of push and pull services.

One major concern of the hybrid mode operation is to determine the optimal locations of storage nodes such that overall communication cost is minimized, a problem called *optimal storage problem*. Note that the optimality here is defined under the premise that the operation of data query is based on the existence of storage nodes. In this regard, the overall communication cost is the sum of communication costs introduced by transporting source data as well as query results. This problem is critical for networks where communication cost is a dominant metric for network performance. The aim of this work is to address the optimal storage problem for a tree-structured network.

The optimal storage problem has earlier been studied in the domain of database systems [5], [6], in which a database is optimally replicated under the assumption that each write to the database must be forwarded to all replicas, while a read from a node is satisfied by a replica that is nearest to the node. Prior approaches toward this problem were based on the notion of *tree median*. Basically, a median is a node in the tree that minimizes the sum of distances to the other nodes [27]. An optimal storage set can be found by starting with a median node and iteratively adding new neighboring node to the set as long as the addition results in cost reduction [13]. However, the optimality property associated with medians holds only if transmitting a unit of data over a given communication link costs the same for both directions. In other words, the validity of the previous work relies on the assumption of *homogeneous channel cost*.

While homogeneous channel cost may seem a reasonable assumption for some networks, *heterogeneous channel costs* are more common to tree-based networks. More specifically, data transmission from a node to a neighboring

- *G.-M. Chiu and T.-L. Chin are with the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan 106. E-mail: {chiu,tchin}@mail.ntust.edu.tw.*
- *L.-H. Yen is with the Department of Computer Science and Information Engineering, National University of Kaohsiung, Kaohsiung, Taiwan 811. E-mail: lhyen@nuk.edu.tw.*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

node along a communication link and that in the opposite direction may experience different degrees of contentions, induce different levels of interference, or progress with different priorities in MAC (medium access control) layer. For example, it is common that, in a tree network, upstream communication is treated differently from the downstream one. In addition, for wireless communication, two end nodes of a communication link may operate at different levels of transmission power when they communicate with each other, thus leading to different degrees of interference to the network. These differences should be reflected by distinction of the associated communication costs. However, it remains unknown how to find an optimal storage set for networks with heterogeneous channel costs.

In this paper, we generalize the optimal storage problem for a tree-structured network by considering heterogeneous channel costs. We have identified necessary and sufficient conditions for an optimal storage set for the network. In particular, we introduce the notion of fully-covered node for this purpose in the case of multiple sources. The proposed algorithm only incurs a linear time cost. Extensive simulations have been conducted to validate the algorithm and evaluate its performance.

Rest of this paper is organized as follows. Section 2 surveys related work and Section 3 presents network model and problem definition. In Section 4, we discuss how to deal with the problem with only one source node. Then in Section 5, we tackle the problem for the case of multiple sources. Correctness of the proposed algorithm is rigorously proved. Performance of the proposed algorithm is evaluated by extensive simulations in Section 6. Section 7 discusses related issues that are not specifically addressed in previous sections. The last section concludes this paper and discusses future research directions.

2 Related Work

The storage placement problem can be found in various contexts of research areas such as the warehouse location problem in operations research [1], [2], [29], the file assignment problem for database systems [4], [5], Web caching and peer-to-peer (P2P) applications in computer networks [7], [26], and data query in sensor networks [10], [11]. These studies resolve the storage location problem on different facilities or network platforms with a variety of purposes and restrictions.

The warehouse location problem has been studied extensively in the literature. The aim of the problem is to select a number of cites on a map as warehouse locations such that cost of transporting goods from factories to customers is minimized [1], [2], [14]. The problem has been proven NP-complete for general graphs and solvable in polynomial time for trees [14]. Our problem differs from the warehouse location problem in that, in our problem, each storage node must store data originated from all source nodes and one request node simply retrieves data from the storage node that is closest to itself. In contrast, in the warehouse location problem, goods produced by a particular factory may be transported and stored in different warehouses. A customer can get goods from different warehouses as long as total amount of the goods satisfies its demands.

The file assignment problem has also been investigated for distributed database systems [5], [12], [13], [30]. In this problem, data generated by a set of source nodes in a network are replicated to a number of storage nodes. A query node can retrieve replicas from the nearest storage node to increase retrieval speed. A number of cost evaluation models have been studied for this problem. For instance, Fisher and Hochbaum considered a simple write policy, where a separate copy of the data generated by a specific source node is sent to each storage node [5]. The total write cost is, therefore, sum of the distances from each source node to each of the storage nodes. In [13], the write policy is modeled similar to multicast in the sense that each piece of data is sent to multiple storage nodes via multicast communication. While most of the studies consider only data transmission cost, data storage cost is specifically included in the cost model in [12]. These studies are closely related to ours in that they also address the problem of storage placement in a network and have similar objective to ours. However, all of them assume that the communication cost is homogeneous in either direction between two neighbor nodes, while heterogeneous communication costs are considered in our network model to reflect realistic network environments.

Along with pervasive use of the Internet, Web and P2P applications are gaining popularity among today's Internet users who are desperate for increasing download speed. Web proxies are used to cooperatively cache and share Web contents [15], [17], [31], [32], [33]. For P2P overlay networks, Distributed Hash Table (DHT) has been exploited to place contents at specific locations and queries are directed to the associative locations [24], [25]. If contents are placed at random locations, queries are typically flooded to the peers in order to locate needed data [26], [34]. These studies, unlike ours, focus on searching locations for content placement in order to facilitate content dissemination.

Data replication and caching have also been widely studied for improving data accessibility in wireless environments [18], [20], [35]. Recently, considerable efforts have been made for reducing the cost of data dissemination and retrieval in wireless sensor networks [3], [10], [16], [21]. A typical usage of sensor networks is to collect data by tens of thousands of sensors which have limited computation and communication capability. The inherited limitations make the problem of efficient data retrieval a major challenge in sensor networks. In [10], a data-centric storage scheme based on DHT is proposed to locate certain type of data by the properties of the data. Queries for the same type of data can be served by one or a few nodes without resorting to flooding mechanism. GEM, which is another approach for data-centric storage, maps data to sensor nodes by embedding a logical graph of storage nodes to a physical network [21]. Hierarchical data storage and retrieval methods, which introduce an intermediate tier between sources and data-retrieval nodes, are also proposed for wireless or mobile sensor

networks [22], [23]. These studies focus on the design of data dissemination and retrieval schemes.

Tree-structured networks are usually exploited in wireless sensor networks [11], [19], [36]. An energy-conserving data placement scheme proposed for sensor networks is introduced in [3], [9]. The authors propose a greedy heuristic that places multiple copies of data in the network and transfers the data from sensors to observers using multicast. Essentially, storage locations are the medians concerning communication costs among the sender and observers. In [11], a data storage placement scheme is proposed for a tree-structured sensor network where data are eventually gathered at the sink. Storage nodes are placed between the sink and the sensors to reduce energy consumption for data transmission. In sensor networks, queries always originate from the sink. This work is similar to ours but queries in our setting can be issued by any network node.

3 Network Model and Problem Definition

We consider a tree network T where each source node s generates source data at a rate of g_s (times per unit time). We refer to g_s as the *source rate* of s. A non-source node simply has a source rate of zero. Each node i in the network issues queries for the collected data at a *query rate* r_i (times per unit time). We assume that each query calls for source data from *all* source nodes. Such requirement is very common for many practical applications. In light of this, a storage node is required to collect and store source data originated from all source nodes, and a node is made a *storage node* if all source nodes push source data to it. Data queries issued by a node are answered by a storage node that is the nearest to the querier. Without loss of generality, we assume that the sizes of source data and the data returned, as a result of a query, are both equal to one unit. In practice, these sizes may be different, in which case this difference can be captured by introducing a constant factor between g_s and r_i's.

As described earlier, different communication links may be situated in different transmission environments in a network. Moreover, communications across a communication link may operate differently in opposite directions in many cases, thus inducing different levels of impact on the network. To account for this characteristic of heterogeneity, we use $c_{i,j}$ to represent communication cost of transmitting one unit of data over a communication channel from node i to node j. Note that $c_{j,i}$ can be different from $c_{i,j}$.

Our goal is to find a set of storage nodes in T that minimizes overall communication cost. To facilitate our discussion, we refer to the communication cost for transmitting source data to storage nodes as *data push cost* and the communication cost for transmitting query results as *data query cost*. For a data query, the size of retrieved data is much larger than that of a query message in many network applications. In this case, data query cost is dominated by transmission of query result, rather than delivery of the query message. In fact, in some network applications, query requests are predetermined such that no explicit query messages need be transmitted to storage nodes. Rather, storage nodes may simply send query results to querying nodes in a proactive manner. This situation normally happens for applications with periodic query demands. Hence, we ignore the cost of transmitting query messages for now. However, we will consider including such cost in the cost model later in Section 5.3.

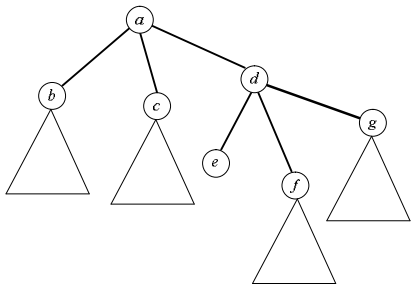


Fig. 1. A sample tree.

Suppose a query issued by node i travels through a path $i = n_0, n_1, n_2, \ldots, n_{l-1}, n_l = j$ to retrieve data from storage node j. The query result is returned from j to i in the reverse direction. Hence, the associated data query cost is the sum of communication cost induced by traversal of the query result from node j to node i.

The optimal storage problem can be illustrated by a sample tree shown in Fig. 1. Suppose that only nodes b and f are source nodes. If node a is the single storage node in the network, transmitting source data from b to a will incur cost $g_b \cdot c_{b,a}$. Similarly, the data push cost from f to a will be $g_f \cdot (c_{f,d} + c_{d,a})$, resulting in a total data push cost of amount $g_b \cdot c_{b,a} + g_f \cdot (c_{f,d} + c_{d,a})$. On the other hand, the data query cost for node e will be $r_e \cdot (c_{a,d} + c_{d,e})$, and data query costs for other nodes can be derived similarly. Now consider adding d as an extra storage node. It incurs additional data push cost amounted to $g_b \cdot c_{a,d}$. In return, data query costs for nodes d, e, f, and g are reduced as these nodes can now retrieve data from node d, instead of node a. Since every node could be a storage node, a naïve approach that examines every possible combination of storage nodes will suffer from combinatorial explosion problem.

We observed that the problem can be simplified if only one source node is present. In that case, the source node is a storage node by nature. Accordingly, there is only one possible direction in which source data could be passed to a particular storage node. In case of multiple sources, however, data are pushed from every source to every storage node. In the following sections, we present first a solution for the case of single source, which is inspired by the above-mentioned observation, followed by another approach which deals with cases with multiple sources.

TABLE 1
PARTIAL LIST OF SYMBOLS

Symbol	Semantics
g_s	Source rate of node s
r_i	Query rate of node i
$ch(i, j)$	The directional communication channel from nodes i to j
$c_{i,j}$	The communication cost of $ch(i, j)$
$T(i, j)$	The subtree containing i when link (i, j) is removed from T
$R(i, j)$	Sum of query rates of all nodes in $T(i, j)$
$S(i, j)$	Sum of source rates of all source nodes in $T(i, j)$
$N(x)$	Set of node x's neighboring nodes
$sn(x)$	The source-bound neighbor of node x
R_x	$R(x, sn(x))$
$d_T(i)$	Degree of node i in tree T
F	Set of fully-covered nodes
Φ	$\{j \mid S(j, i) < R(i, j)\}$; set of nodes that cover some neighbor
Σ	Optimal set of storage nodes

The key to the proposed solutions is to focus on *links* rather than *nodes* of the tree. Consider a link (i, j) between two neighboring nodes i and j. Observe that if (i, j) is deleted, T will be divided into two disjoint subtrees. We denote the one that contains node i by $T(i, j)$ and the other by $T(j, i)$. Link (i, j) is treated as two directional channels, $ch(i, j)$ from node i to node j and $ch(j, i)$ from j to i. For each $ch(i, j)$, two quantities are defined:

- $R(i, j)$: sum of query rates of all the nodes in $T(i, j)$.
- $S(i, j)$: sum of source rates of all the source nodes that are located in $T(i, j)$.

These two quantities are crucial to our solutions. The key idea is: if $R(i, j) > S(j, i)$, all source data in $T(j, i)$ must be pushed to node i. Detailed analysis will be presented in the following two sections. For the ensuing discussion, symbols that are mostly used are summarized in Table 1.

4 Single Source of Data Generation

In this section, we consider a tree network T in which there is only one source node s with source rate g_s. Let $N(x)$ denote the set of node x's neighboring nodes. For a node x ($\neq s$), we call the unique node $y \in N(x)$ that is located on the path from x to s the *source-bound neighbor* of x, denoted by $sn(x)$. We define R_x as the sum of query rates of all the nodes in $T(x, sn(x))$, that is, $R_x = R(x, sn(x))$. R_x possesses the property of *rate aggregation*, as stated by the following lemma.

Lemma 1. *Let $n_0, n_1, \ldots, n_l = s$, $l \geq 2$, be the path in T from node n_0 to the source node s. For all i, where $0 \leq i \leq l-2$, we have $R_i \leq R_{i+1}$.*

Proof. We have $sn(n_i) = n_{i+1}$ for all $0 \leq i \leq l-2$. Hence, the lemma can be directly derived from the fact that, for all $0 \leq i \leq l-2$, $T(n_i, sn(n_i)) \subset T(n_{i+1}, sn(n_{i+1}))$ and all query rates are non-negative. □

As stated earlier, a node is a storage node if source data is pushed to the node. A storage node may further forward the source data onto other storage nodes. In light of this, we have the following property.

Property 1. *If j is a storage node, then all nodes on the path from j to s are also storage nodes.*

Lemma 2. *Let H be a non-empty set of storage nodes and $i \in H$. If node j ($j \notin H$) is a neighbor of i and $R_j > g_s$, then the set $H \cup \{j\}$ offers a communication cost that is less than H.*

Proof. Since i and j are neighbors, either $i = sn(j)$ or $j = sn(i)$ is true. Property 1 implies that j cannot be $sn(i)$, so i must be $sn(j)$. Property 1 also implies that there is no storage node in $T(j, i)$, since otherwise j will be a storage node. Since $j \notin H$, $H \cup \{j\}$ is the result of adding j to H. It adds an extra data push cost of $c_{i,j} \cdot g_s$, but cuts data query cost by an amount of $c_{i,j} \cdot R_j$. Since $R_j > g_s$, overall communication cost is reduced as of adding node j to H. □

Lemma 3. *Node j ($\neq s$) must be included in any optimal set of storage nodes if $R_j > g_s$.*

Proof. Let $j = n_0, n_1, \ldots, n_l = s$, $l \geq 1$, be the path from j to s. Since $R_j > g_s$, we have $R_n > g_s$, for all $0 \leq i \leq l-1$, by Lemma 1. Suppose, by contradiction, that P is an optimal set of storage nodes and $j \notin P$. Obviously we have $s \in P$. By Property 1, there is a uniquely-defined i, $0 \leq i \leq l-1$, such that $n_0, n_1, \ldots, n_i$ are not in P while $n_{i+1}, \ldots, n_l$ are all in P. Applying Lemma 2 to $n_i, n_{i-1}, \ldots, n_0$ in sequence, we can easily see that P cannot be an optimal solution, a contradiction. □

Lemma 3 gives a necessary condition for an optimal solution. We now show that the same criterion also serves as a sufficient condition in the following lemma.

Lemma 4. *There always exists an optimal set of storage nodes that does not contain any node j ($\neq s$) with $R_j \leq g_s$.*

Proof. Let P be any optimal set of storage nodes that contains some node j ($\neq s$) with $R_j \leq g_s$. Let $P_j = \{i \mid i \in P$ and $i \in T(j, sn(j))\}$. Apparently, we have $j \in P_j$. Two cases are considered here.

Case 1: $|P_j| = 1$

In this case, j is the only node in P_j. Consider $P^* = P - \{j\}$. P^* adds an extra data query cost of $c_{sn(j),j} \cdot R_j$ but cuts data push cost by an amount of $c_{sn(j),j} \cdot g_s$. Since $R_j \leq g_s$, the overall communication cost of P^* is no greater than P.

Case 2: $|P_j| > 1$

Let $i \in P_j$ be a node such that there is no other node $x \in P_j$ for which $sn(x) = i$. The path from i to s has to pass through j. Lemma 1 leads to the fact of $R_i \leq g_s$. From the argument of case 1, we can remove i from P to obtain another solution whose communication cost is no greater than P. Repeatedly applying such pruning process to all nodes in P_j, we eventually obtain a set $P^* = P - P_j$ with overall communication cost no greater than P. Therefore, we can always remove j from P to obtain another optimal solution. □

Based on Lemmas 3 and 4, we can proceed to find an optimal set of storage nodes for T. The only issue is how to compute R_x efficiently for all node $x \neq s$. Let $d_T(x)$ represent the degree of node x in T. Since

$$R_x = \begin{cases} r_x & \text{if } d_T(x) = 1, \\ r_x + \sum_{y \in N(x) - \{sn(x)\}} R_y & \text{otherwise,} \end{cases}$$

computation of all R_x values must follow some particular order. For any two nodes x and y, we define $y \prec x$ if $y \in N(x) - \{sn(x)\}$. Clearly, the transitive closure of relation $\prec$ is a precedence ordering that captures computation dependency among nodes. If we represent $\prec$ as a directed acyclic graph $G = (V, E)$, where V is the set of nodes in T and $(y, x) \in E$ if $y \prec x$, computation of all R_x values should follow a topological ordering on G. It turns out that a topological sorting algorithm with slight modification can be utilized for the computation, which has a linear computation time.

5 Multiple Sources of Data Generation

We now focus on the case in which two or more source nodes in T can independently generate source data. Our single-source solution benefits from the property of unidirectional data flow: data originate from a single source and are pumped to all storage nodes. The lack of such property with multiple sources invalidates applicability of our single-source solution for multiple-source cases. In particular, unlike the single-source case in which the source node is by default a storage node, we do not know any pre-existing storage node to start with in multiple-source situations. In this section, we present a general yet efficient algorithm, which is mainly based on the notion of fully-covered node, to identify an optimal set of storage nodes for multiple-source scenarios.

Recall that in the single-source setting, any path from a non-source node to the source possesses the rate aggregation property (Lemma 1). The rate aggregation property founds a basis upon which locations of storage nodes can be determined. This property can be generalized to any path in the tree as stated in Lemma 5.

Lemma 5. *Let $n_0, n_1, \ldots, n_l$ be a path in T. For all i, $0 < i < l$, we have $R(n_{i-1}, n_i) \le R(n_i, n_{i+1})$, $R(n_i, n_{i-1}) \ge R(n_{i+1}, n_i)$, $S(n_{i-1}, n_i) \le S(n_i, n_{i+1})$, and $S(n_i, n_{i-1}) \ge S(n_{i+1}, n_i)$.*

Proof. It can be directly derived from the fact that $T(n_{i-1}, n_i) \subset T(n_i, n_{i+1})$, $T(n_i, n_{i-1}) \supset T(n_{i+1}, n_i)$, and all query and source rates are non-negative. □

Corollary 1. *Let $n_0, n_1, \ldots, n_l$ be a path in the tree. For all i, j, $0 \le i < j \le l$, we have $R(n_i, n_{i+1}) \le R(n_{j-1}, n_j)$, $R(n_{i+1}, n_i) \ge R(n_j, n_{j-1})$, $S(n_i, n_{i+1}) \le S(n_{j-1}, n_j)$, and $S(n_{i+1}, n_i) \ge S(n_j, n_{j-1})$.*

Source data must flood all storage nodes. If there is more than one storage node in T, all of these nodes must cluster together, forming a subtree. This property is similar to Property 1 but the correctness of the property is not trivial to see. Lemma 6 gives proof for the property.

Lemma 6. *Given any two storage nodes u and v, all nodes on the path connecting u and v are also storage nodes.*

Proof. Let $u = n_0, n_1, \ldots, n_l = v$ be the path from u to v. The lemma trivially holds when $l = 1$. Now consider any node n_i, where $0 < i < l$ and $l > 1$. If there is any source node in $T(n_i, n_{i+1})$, source data generated by the source node must pass through n_i in order to reach v. Similarly, the source data generated by any source node in $T(n_i, n_{i-1})$ must pass through n_i to reach u. Consequently, all source data will reach n_i, and therefore n_i will become a storage node as well. □

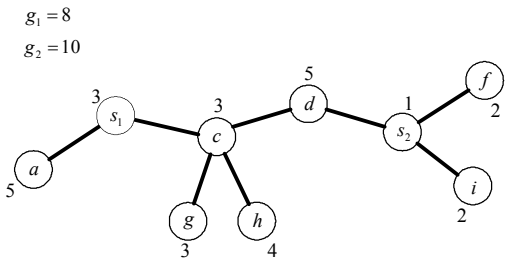


Fig. 2. An example having fully-covered nodes with s_1 and s_2 being source nodes. g_1 and g_2 are source rates of s_1 and s_2, respectively.

5.1 Notion of Fully-Covered Nodes

The design of our optimal algorithm is based on the following definition.

Definition 1. *Let node i be a neighbor of node j. Node i is said to* cover *j if $S(i, j) < R(j, i)$. Node j is said to be* fully-covered *if it is covered by all of its neighbors.*

Take Fig. 2 as an example, where two source nodes s_1 and s_2 with source rates g_1 and g_2, respectively, are present. The number beside each node represents query rate of the node. Source node s_1 covers node c since s_1 is a neighbor of c and $S(s_1, c) = 8 < R(c, s_1) = 20$. Node d also covers node c because d is a neighbor of c and $S(d, c) = 10 < R(c, d) = 18$. In addition, both nodes g and h cover c. Hence, c is fully-covered. Meanwhile, it is easy to see that node d is also fully-covered. However, no other nodes are fully-covered in the network.

In what follows, we first show that, if there exists any fully-covered node, the set of all fully-covered nodes constitutes an optimal solution for T. The situation in which there is no fully-covered node in the network will be investigated in Section 5.2.

Lemma 7. *Let i and j be two neighboring nodes. If node i covers node j, then there must be a storage node in $T(j, i)$ for any optimal solution.*

Proof. By definition, we have $S(i, j) < R(j, i)$. Suppose, by way of contradiction, that there is an optimal set of storage nodes that contains no node in $T(j, i)$. Consider the storage node x in the optimal set that is closest to j. Let $x = n_0, n_1, n_2, \ldots, n_{l-1} = i, n_l = j$ be the path from x to j, where $l \ge 1$. Note that there is no storage node in $T(n_1, x)$, since otherwise x cannot be the storage node in the optimal set that is closest to j by Lemma 6. Now consider including n_1 as an additional storage node in the optimal set. It adds a data push cost of amount $c_{x,n_1} \cdot S(x, n_1)$. On the other hand, the addition of n_1 as a storage node cuts data query cost of amount $c_{x,n_1} \cdot R(n_1, x)$. By Corollary 1, we know that $S(x, n_1) \le S(i, j)$ and $R(n_1, x) \ge R(j, i)$. Since $R(j, i) > S(i, j)$, we then have $R(n_1,$

$x) > S(x, n_1)$. Hence, adding n_1 to the optimal set of storage nodes will further decrease overall communicate cost, a contradiction. □

Corollary 2 is a derivation from Lemma 7, and Corollary 3 follows.

Corollary 2. *If j is covered by node i, all source nodes (if any) in T(i, j) must have their source data pushed to node j for any optimal solution.*

Corollary 3. *If node j is fully-covered, any optimal set of storage nodes must include j.*

Corollary 3 shows that the set of all fully-covered nodes must be included in any optimal storage set. We now prove that, if this set is not empty, it alone suffices to be an optimal solution.

Lemma 8. *Let i and j be two neighboring nodes. If node i covers node j and i is not fully-covered, then j does not cover i.*

Proof. If j is i's only neighbor, since i is not fully-covered, then j apparently does not cover i. Now consider the case in which i has more than one neighbor. Consider any neighbor k of i, $k \neq j$. Since i covers j, we have $S(i, j) < R(j, i)$ by definition. Considering the path k, i, to j, by Lemma 5, we have $S(k, i) \leq S(i, j)$ and $R(j, i) \leq R(i, k)$; thus $S(k, i) < R(i, k)$ follows. In other words, k covers i. Since i is not fully-covered, it must be true that j does not cover i. □

Theorem 1. *If there is at least one fully-covered node in the network, then the set of all fully-covered nodes is an optimal set of storage nodes.*

Proof. Let F represent the set of all fully-covered nodes. For any optimal solution F^*, we must have $F \subseteq F^*$ by Corollary 3. Let $Q = F^* - F$. If $Q = \phi$, then F is an optimal solution. Otherwise, one can readily see that Q consists of one or more disconnected tree fragments, each of which is a subtree. Consider any one such subtree, denoted as T_1. According to Lemma 6, there must be a node, say y, in T_1 such that y is a neighbor of some node $x \in F$. Two cases are considered here.

Case 1: T_1 contains only one node

This case happens when y is the only node in T_1. Consider the set $F^- = F^* - \{y\}$, i.e. deleting y from F^*. In comparison with F^*, F^- cuts a data push cost of amount $c_{x,y} \cdot S(x, y)$ and increases the data query cost by an amount of $c_{x,y} \cdot R(y, x)$. Since $x \in F$, y must cover x. The fact of $y \notin F$ implies that y is not fully-covered. According to Lemma 8, we have that x does not cover y, which means that $S(x, y) \geq R(y, x)$. We conclude that F^- offers a communication cost that is no greater than F^*.

Case 2: T_1 contains more than one node

In other words, T_1 includes at least one more node than y. Let z be a node in T_1 such that $z \neq y$, $z \in F^*$, and z has only one neighbor in T_1, that is, z is an edge node in T_1. Apparently, $z \notin F$. Consider the set $F^- = F^* - \{z\}$. F^- is still a connected subtree. Let $z = n_0, n_1, \ldots, n_{l-1} = y, n_l = x$, $l \geq 2$, represent the path from z to x. In comparison with F^*, deletion of z from F^* will cut a data push cost of amount $c_{n_1,z} \cdot S(n_1, z)$. On the other hand, it will increase data query cost by an amount of $c_{n_1,z} \cdot R(z, n_1)$. By Corollary 1, we have $S(n_1, z) \geq S(x, y)$ and $R(y, x) \geq R(z, n_1)$. From case 1, we have $S(x, y) \geq R(y, x)$, which then leads to $S(n_1, z) \geq R(z, n_1)$. In other words, F^- offers a communication cost that is no greater than F^*. By repeatedly applying the same pruning operation to nodes in T_1, one-by-one starting from edge nodes (nodes that has only one neighbor), and by the argument of case 1, we conclude that the set $F^* - \{j \mid j$ is a node in $T_1\}$ gives a communication cost that is no greater than F^*.

Applying the above argument to all of the subtrees of Q, we eventually obtain the set F, which offers a communication cost no greater than F^*. Therefore, F is also an optimal storage set. □

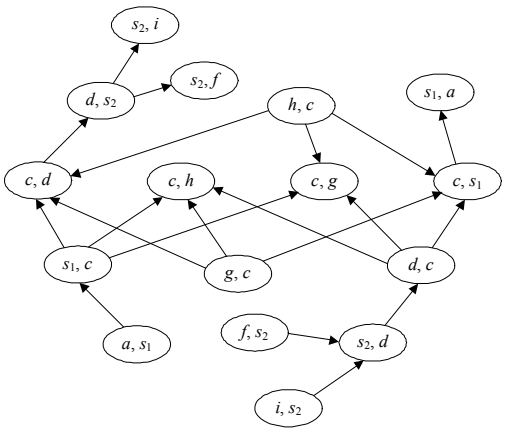


Fig. 3. Graph G_T for the tree in Fig. 2.

Theorem 1 founds the basis for locating an optimal set of storage nodes in a network. To turn this theorem into a practical algorithm, we need to compute $S(i, j)$ and $R(i, j)$ for every channel $ch(i, j)$. A channel $ch(i, j)$ is said to be *saturated* if and only if the values of $S(i, j)$ and $R(i, j)$ are both known. In general, $ch(i, j)$ is saturated only if all $ch(k, i)$ for which $k \in N(i) - \{j\}$ are saturated. For such a channel, we have

$$S(i, j) = g_i + \sum_{k \in N(i) - \{j\}} S(k, i) \quad (1)$$

and

$$R(i, j) = r_i + \sum_{k \in N(i) - \{j\}} R(k, i) . \quad (2)$$

For any channel $ch(i, j)$ where $d_T(i) = 1$, Eq. (1) reduces to $S(i, j) = g_i$ and Eq. (2) reduces to $R(i, j) = r_i$. We define $ch(k, i) \prec ch(i, j)$ if $k \in N(i) - \{j\}$. Given T, we can construct a directed graph $G_T = (V, E)$, where V is the set of all channels in T and $\forall x, y \in V$: $(x, y) \in E$ if $x \prec y$. Fig. 3 shows such a graph for the tree shown in Fig. 2. Clearly, computations of $R(i, j)$ and $S(i, j)$ should follow a topological ordering on G_T.

```
Algorithm Rate_Comp(T) {
1.  WQ ← φ;
2.  for (each node i ∈ T) {
        in_count[i] ← 0;
        if (d_T(i) = 1) { /* i is an edge node */
            R(i, j) ← r_i, where j is i's single neighbor;
            S(i, j) ← g_i, where j is i's single neighbor;
            WQ ← WQ ∪ {(i, j)};
        } else {
            R_sum[i] ← r_i;
            S_sum[i] ← g_i;
        }
    }
3.  P ← φ; /* records all channels that are already saturated */
4.  while (WQ ≠ φ) {
        remove an entry (x, y) from WQ;
        R_sum[y] ← R_sum[y] + R(x, y);
        S_sum[y] ← S_sum[y] + S(x, y);
        P ← P ∪ {(x, y)};
        in_count[y] ← in_count[y] + 1;
        if (in_count[y] = d_T(y) – 1) { /* one out-channel saturated */
            let z be the neighbor of y such that (z, y) ∉ P
            R(y, z) ← R_sum[y];
            S(y, z) ← S_sum[y];
            WQ ← WQ ∪ { (y, z) };
        } else if (in_count[y] = d_T(y))
            for (each z ∈ N(y) – {x}) { /* ch(y, x) already saturated */
                R(y, z) ← R_sum[y] – R(z, y);
                S(y, z) ← S_sum[y] – S(z, y);
                WQ ← WQ ∪{(y, z) };
            }
        }
    }
```

Fig. 4. Algorithm Rate_Comp.

A straightforward approach to the computation is therefore constructing G_T from T first and then performing a topology sorting on G_T. This approach, however, can be further optimized. The proposed algorithm, `Rate_Comp`, shown in Fig. 4, computes $R(i, j)$'s and $S(i, j)$'s from T directly. The key point is to count the number of incoming channels that are saturated for each node, and to accumulate the amount of source and query rates that have been pumped to each node during the computation. Array **in_count** serves for the former purpose while **S_sum** and **R_sum** together serve the latter. The algorithm also involves maintaining a working queue ***WQ***, where each entry contains the identification of a saturated channel. When **in_count**$[y]$ reaches $d_T(y) - 1$, we know that all but one channels pointing to y are saturated. Let $ch(z, y)$ be the only channel coming to y that is not yet saturated. At this moment we have $R(y, z)$ and $S(y, z)$ computed and stored in **R_sum**$[y]$ and **S_sum**$[y]$, respectively. In that case, $ch(y, z)$ becomes saturated and is added to ***WQ*** for later computation. When **in_count**$[y]$ equals $d_T(y)$, meaning that all channels coming into y are saturated, we are ready to compute $R(y, w)$ and $S(y, w)$ for all channel $ch(y, w)$ leaving y. The algorithm simply repeats the removal-and-addition process until ***WQ*** becomes empty.

```
Algorithm Multiple_Source(T) {
1.  F ← the set of all nodes in T;
    Φ ← φ;
    for (each ch(i, j) ∈ T) {
        if (S(j, i) < R(i, j))
            Φ ← Φ ∪ {j};
        else
            F ← F – {i}; /* i is not fully-covered */
    }
2.  if (F ≠ φ)
        Σ ← F;
    else /* no fully-covered node */
        Σ ← NoFullCov(Φ);
}
```

Fig. 5. Algorithm `Multiple_Source`.

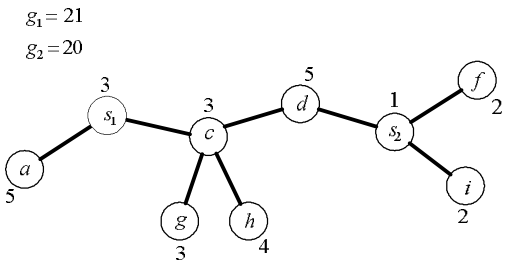


Fig. 6. An example having no fully-covered node. s_1 and s_2 are source nodes.

After $S(i, j)$ and $R(i, j)$ for all $ch(i, j)$ have been computed, we are able to determine whether u is fully-covered by Definition 1 for any node $u \in T$. This is achieved by algorithm `Multiple_Source` (Fig. 5). It uses Φ to indicate the set of nodes that cover some neighbor. The optimal set of storage nodes Σ is given by the set of fully-covered nodes (recorded in F) if there exists at least one such node. However, if F is empty, the algorithm will proceed to execute a special function called `NoFullCov` in order to find an optimal solution. Discussion about such a scenario, along with details of `NoFullCov`, is subsequently presented in Section 5.2. The set Φ will be used by function `NoFullCov`.

5.2 Scenarios of No Fully-Covered Node

Theorem 1 states that an optimal set of storage nodes is comprised of all fully-covered nodes in the network, if any fully-covered node can be identified. Unfortunately, there may be cases where no fully-covered node can be found. Take Fig. 6 as an example. The figure is the same as Fig. 2 except that now g_1 is 21 and g_2 is 20. In this example, none of the nodes in the network is fully-covered. In this case, we resort to function `NoFullCov` to locate an optimal set of storage nodes that offers minimum communication cost. This subsection describes the idea behind and details of `NoFullCov`.

Consider two neighboring nodes i and j. Suppose that i

covers j. Lemma 7 states that there must be a storage node in $T(j, i)$ for any optimal solution. Furthermore, by Lemma 8, we see that node i cannot be covered by node j in the case of no fully-covered node. This observation leads us to ignore all the nodes in the subtree $T(i, j)$ when finding an optimal solution for the network.

Recall that, in `Multiple_Source`, $j \in \Phi$ if node j covers any of its neighbors. If $j \notin \Phi$, j shall be considered by function `NoFullCov` as a possible storage location. In the example of Fig. 6, three nodes, namely s_1, c, and d, do not cover any of their neighbors. We now prove that all nodes in $T - \Phi$ form a connected subtree.

Lemma 9. *Let $U = T - \Phi$. The nodes in U form a subtree in T, i.e., the nodes in U are connected.*

Proof. If there is only one node in U, the lemma trivially follows. Let x and y be any two nodes in U and $x = n_0, n_1, n_2, \ldots, n_{i-1}, n_i, n_{i+1}, \ldots, n_l = y$ be the path from x to y with $l \geq 1$. If $l = 1$, the lemma also trivially follows. Now consider the case of $l > 1$. We prove the lemma by contradiction. Assume that n_i is the node that is the closest to x in Φ. Since $n_i \in \Phi$, n_i must cover some other neighbor node z and thus $S(n_i, z) < R(z, n_i)$. For the case of $z \neq n_{i-1}$, since $S(n_i, z) \geq S(n_{i-1}, n_i)$ and $R(z, n_i) \leq R(n_i, n_{i-1})$ by Lemma 5, it is obvious that $S(n_{i-1}, n_i) < R(n_i, n_{i-1})$. Therefore, n_{i-1} covers n_i and thus $n_{i-1} \in \Phi$, which contradicts the assumption. Consider the case of $z = n_{i-1}$. By Lemma 5, we have $S(n_i, z) \geq S(n_{i+1}, n_i)$ and $R(z, n_i) \leq R(n_i, n_{i+1})$. Since $S(n_i, z) < R(z, n_i)$, we then have $S(n_{i+1}, n_i) < R(n_i, n_{i+1})$. Hence, n_{i+1} covers n_i. By the same token, we have that n_{i+2} covers n_{i+1}, and so on. Eventually, one obtains that y covers node n_{l-1}. Therefore, y must be in Φ, which also contradicts the assumption. □

We call the subtree formed by the nodes in U a *residual tree* and denote it by T_U. T_U has the following properties.

Lemma 10. *Let x be a node in T_U and y is a neighbor of x that is not in T_U. Consider the path $n_0, n_1, \ldots, n_{l-1} = y, n_l = x$, $l \geq 1$, from node n_0 to x. We have n_i covers n_{i+1} for all $0 \leq i \leq l - 1$.*

Proof. By Lemma 9, we know that none of $n_0, n_1, \ldots, n_{l-2}$ is included in T_U if $l > 1$, since y is not in T_U. Note that y covers some of its neighbors as $y \in \Phi$. Assume that w is a neighbor covered by y and $w \neq x$. We have $S(y, w) < R(w, y)$. By Lemma 5, we have that $S(x, y) \leq S(y, w)$ and $R(w, y) \leq R(y, x)$. Thus $S(x, y) < R(y, x)$ follows. Therefore, x covers y, a contradiction with $x \notin \Phi$. Hence, we show that y has to cover x. This result also implies that $S(y, x) < R(x, y)$. Consider node n_{l-2}. Again, by Lemma 5, we have $S(n_{l-2}, y) \leq S(y, x)$ and $R(x, y) \leq R(y, n_{l-2})$, thus $S(n_{l-2}, y) < R(y, n_{l-2})$ follows, which implies that n_{l-2} covers y. By the same token, we can prove that n_i covers n_{i+1} for all $0 \leq i \leq l - 3$. □

Corollary 4 follows from Lemmas 10 and 8.

Corollary 4. *Let x be a node in T_U and y is a neighbor of x that is not in T_U. Consider the path $n_0, n_1, \ldots, n_{l-1} = y, n_l = x$, $l \geq 1$, from node n_0 to x. Then n_{i+1} does not cover n_i for all $0 \leq i \leq l - 1$.*

We now proceed to show that there exists an optimal solution that contains only one node in T_U.

Lemma 11. *In the case of no fully-covered node, there exists an optimal solution that contains only nodes in the residual tree T_U.*

Proof. We prove the lemma by contradiction. Assume that all of the optimal sets of storage nodes must contain at least one node that is not in T_U. Consider any optimal solution P. Let $Q = P - \{i \mid i \text{ is a node in } T_U\}$. Obviously, $Q \neq \phi$. In fact, one can readily see that Q consists of one or more disconnected tree fragments, each of which is a subtree. Consider any one such subtree, denoted by T_1. Let $b \in T_U$ and $c \in T_1$ be the closest nodes among all. Two cases are considered here.

Case 1: T_1 contains only one node

This case happens when c is the only node in T_1. Let $c = n_0, n_1, \ldots, n_l = b$, $l \geq 1$, be the path from node c to node b. Two subcases are further considered below.

Case 1.1: Nodes b and c are not neighbors in the network

This case occurs when $l > 1$. Note that node n_1 belongs to neither P nor T_U. Consider the set $P^+ = P \cup \{n_1\}$, i.e. adding n_1 to P. In comparison with P, P^+ increases a data push cost of amount $c_{c,n_1} \cdot S(c, n_1)$ and cuts data query cost by an amount of $c_{c,n_1} \cdot R(n_1, c)$. From Lemma 10, we know that c covers n_1. That is, $S(c, n_1) < R(n_1, c)$. Thus, P^+ offers a communication cost that is less than P, which is a contradiction.

Case 1.2: b and c are neighbors in the network

This case occurs when $l = 1$. If $b \notin P$, by the same argument given in case 1.1, we can show that the set $P \cup \{b\}$ offers a communication cost that is smaller than P, a contradiction. Now consider the case of $b \in P$. In this case, consider the set $P^* = P - \{c\}$. In comparison with P, P^* increases a data query cost of amount $c_{b,c} \cdot R(c, b)$ and cuts data push cost by an amount of $c_{b,c} \cdot S(b, c)$. From Corollary 4, we know that b does not cover c. That is, $S(b, c) \geq R(c, b)$. Therefore, P^* gives a communication cost that is no greater than P.

Case 2: T_1 contains more than one node

In other words, T_1 includes at least one more node than c. Let z be a node in T_1 such that $z \in P$ and z has only one neighbor in T_1, that is, z is an edge node in T_1. Consider the set $P^- = P - \{z\}$. P^- is still a connected subtree. Node z must traverse through node c to reach b. Let $z = n_0, n_1, \ldots, n_i = c, \ldots, n_l = b$, $l \geq 2$, represent the path from z to b. In comparison with P, P^- cuts a data push cost of amount $c_{n_1,z} \cdot S(n_1, z)$ and increases data query cost by an amount of $c_{n_1,z} \cdot R(z, n_1)$. From Corollary 4, we see that n_1 does not cover z, thus $S(n_1, z) \geq R(z, n_1)$. Hence, P^- gives a communication cost that is no greater than P.

By repeatedly applying the same pruning operation to nodes in T_1, one-by-one starting from edge nodes, and by the argument of case 1, we conclude that either P is not an optimal solution or the set $P - T_1$ gives a communication cost that is no greater than P. Applying the above arguments to all of the subtrees of Q, we eventually obtain that either P is not an optimal solution or the set $P - Q$ offers a communication cost less than or equal to P. Since $P - Q$ contains only nodes in T_U, thus a contradiction arises. □

The following theorem states that there exists an optimal solution that contains only one node in the residual tree for the case of no fully-covered node.

Theorem 2. *In the case of no fully-covered node, there exists an optimal solution that contains only one node in T_U.*

Proof. From Lemma 11, we see that there exists an optimal solution that contains only nodes in T_U. Let P be an optimal solution that contains only nodes in T_U and $|P| > 1$. Let x be any edge node in P and y be the sole neighbor of x in P. Consider the set $P^- = P - \{x\}$, i.e. deleting x from P. In comparison with P, P^- cuts a data push cost of amount $c_{y,x} \cdot S(y, x)$ and increases data query cost by an amount of $c_{y,x} \cdot R(x, y)$. Since y does not cover x, we have that $S(y, x) \geq R(x, y)$. Hence, P^- offers a communication cost that is no greater than P. Repeat the above pruning process to nodes in P, one-by-one starting from edge nodes, we will obtain an optimal solution with only one node left. □

Theorem 2 simply says that there exists an optimal solution in the residual tree and the solution contains only one node. A naïve method to find the optimal storage node is to take each node in the residual tree as a storage node and calculate overall communication cost for the network. However, this method can incur very high computation cost. The function `NoFullCov` listed in Fig. 7 provides a more efficient solution for finding the optimal storage node. The basic idea of `NoFullCov` is using the set of nodes in the residual tree T_U as a reference storage set. We then compute the amount of reduction in cost for each node x in T_U, with respect to the reference storage set, when x is the only storage node in the network. Redundant operations can be avoided using a technique similar to the `Rate_Comp` algorithm.

Assume that i is an edge node of T_U and T_U has more than one node. In comparison with T_U, deletion of i from T_U cuts a data push cost of amount $c_{j,i} \cdot S(j, i)$ and increases data query cost by an amount of $c_{j,i} \cdot R(i, j)$, where j is i's single neighbor node in T_U. Let $b(i, j) = c_{j,i} \cdot (S(j, i) - R(i, j))$. $b(i, j)$ is nothing but the amount of cost reduction when i is deleted from T_U. Since j does not cover its neighbors, we have $R(i, j) \leq S(j, i)$, which leads to $b(i, j) \geq 0$. Notice that $b(i, j)$ is also well-defined, even if i is not an edge node in T_U. Conceptually, the set that contains only one storage node x in T_U is obtained by deleting, from T_U, all the other nodes, one by one starting from edge nodes. Therefore, the amount of cost reduction for node x can be calculated by summing all $b(i, j)$, where $i \neq x$ and j is the next node from i to x. Such summations are done by accumulation, using a working queue. In fact, as shown in Fig. 7, function `NoFullCov` reuses ***WQ*** for this purpose. **`in_count`** and P are also reused in a similar manner, but $d_{T_U}(i)$ denotes the degree of node i in T_U. Let $T_U(i, j)$ represent the subtree of T_U that contains node i when link (i, j) is deleted from T_U. The entry **`b_sum`**$[i, j]$ maintains sum of all $b(x, y)$, where x is in $T_U(i, j)$ and y is the next node to j. We use **`cost_reduction`**$[i]$ to record the amount of cost reduction for node i. Eventually, the optimal storage node is the one with the highest cost reduction in the residual tree.

```
Function NoFullCov(Φ) {
1.  T_U ← the subtree of T that is induced by T − Φ ;
2.  for (each node i ∈ T_U) {
        cost_reduction[i] ← 0;
        in_count[i] ← 0;
    }
3.  WQ ← φ;
4.  for (each ch(i, j) ∈ T_U) {
        b(i, j) ← c_{j,i} · (S(j, i) − R(i, j));
        if (d_{T_U}(i) = 1) {  /* j is i's single neighbor in T_U */
            WQ ← WQ ∪ {(i, j)};
            b_sum[i, j] ← b(i, j);
        }
    }
5.  P ← φ;
    while (WQ ≠ φ) {
        remove an entry (x, y) from WQ;
        cost_reduction[y] ← cost_reduction[y] + b_sum[x, y];
        P ← P ∪ {(x, y)};
        in_count[y] ← in_count[y] + 1;
        if (in_count[y] = d_{T_U}(y) − 1) { /* one out-channel saturated */
            let z be the neighbor of y such that (z, y) ∉ P
            b_sum[y, z] ← cost_reduction[y] + b(y, z);
            WQ ← WQ ∪ {(y, z)};
        } else if (in_count[y] = d_{T_U}(y))
            for (each ch(y, z) ∈ T_U, z≠x) {  /* already saturated */
                b_sum[y, z] ← cost_reduction[y] − b_sum[z, y];
                WQ ← WQ ∪ {(y, z)};
            }
    }
6.  return {i | cost_reduction[i] = max_{j∈T_U} cost_reduction[j]};
}
```

Fig. 7. Function `NoFullCov`.

Take Fig. 6 as an example. The residual tree T_U contains three nodes, namely s_1, c, and d. Now assume that $c_{c,d} = 0.5$, $c_{d,c} = 1.2$, $c_{s_1,c} = 1.0$, and $c_{c,s_1} = 0.8$. We have $b(s_1, c) = 9.6$, $b(c, s_1) = 1.0$, $b(c, d) = 2.4$, and $b(d, c) = 5.5$. The set that contains only storage node d can be obtained by deleting edge node s_1 first, followed by deleting node c from T_U. Then the amounts of cost reduction are computed as **`cost_reduction`**$[s_1] = b(d, c) + b(c, s_1) = 6.5$, **`cost_reduction`**$[c] = b(s_1, c) + b(d, c) = 15.1$, and **`cost_reduction`**$[d] = b(s_1, c) + b(c, d) = 12.0$. Therefore, the optimal storage node is node c.

Theorem 3. *Function `NoFullCov` returns an optimal solution for the case of no fully-covered node.*

Proof. By Theorem 2, there exists an optimal solution in T_U that contains only one node. As described earlier, the set that contains only one storage node x in T_U can be obtained by deleting, from T_U, all the other nodes, one by one starting from edge nodes. Each such deletion induces an extra cost reduction of amount $b(i, j)$, where i is the deleted node and j is i's remaining neighbor. Hence, for node x, the amount of cost reduction is indeed the sum of all $b(i, j)$, where $i \neq x$ and j is the next node from i to x. By the accumulation prop-

erty of the function, we can see that **cost_reduction**[x] eventually records this sum. Obviously, the optimal solution would be the one that yields maximum cost reduction. □

Corollary 5 follows from Theorems 1 and 3.

Corollary 5. *Algorithm* `Multiple_Source` *produces an optimal solution.*

In algorithm `Rate_Comp`, each channel in the network is inserted into and removed from the working queue exactly once. Each removal involves O(1) additions and subtractions. A similar observation can also be made in function `NoFullCov`. Therefore, the time complexity of `Multiple_Source` on tree T is O(N), where N is the number of nodes in T.

5.3 Considering Cost for Transmitting Query Messages

Previously, we have ignored the cost for transmitting query messages. This treatment may introduce inaccuracy in some applications where the cost of transmitting query messages is too significant to be neglected. We now consider including such communication cost as part of the total cost.

It is easy to see that, if node i is a fully-covered node, then node i must be in the optimal solution. However, a non-fully covered node may need to be included in an optimal solution as adding more storage node may further cut the total cost due to reduction of the communication cost of transmitting query messages. To this end, for each channel $ch(i, j)$, we define one more quantity called *cost reduction due to query messages*, denoted by **cost_red_qm**[i, j], as the additional amount of cost reduction that can be achieved by making node i another storage node, provided that node j is already a storage node. More specifically, let **reduced_cost**[i, j] be given as

$$R(i,j)\cdot(c_{j,i}+q\cdot c_{i,j})-S(j,i)\cdot c_{j,i}+\sum_{\forall k\in N(i)-\{j\}}\textbf{cost\_red\_qm}[k,i]$$

where q represents the ratio of a query message to a data unit. Then **cost_red_qm**[i, j] is expressed as

$$\textbf{cost\_red\_qm}[i,j]=\begin{cases}\textbf{reduced\_cost}[i,j] & \text{if } \textbf{reduced\_cost}[i,j]>0\\ 0 & \text{otherwise}\end{cases}$$

Note that **reduced_cost**[i, j] is the accumulated maximum amount of cost saving that can be achieved if some storage nodes are placed in $T(i, j)$, provided that node j is already a storage node. If this amount is no greater than zero, it makes no sense to install storage node in $T(i, j)$. Hence, **cost_red_qm**[i, j] is set to zero in this case. Note that **cost_red_qm**[i, j] cannot be negative. The computation of **cost_red_qm**[i, j] can be performed alongside $S(i,j)$ and $R(i, j)$.

Consider the case in which some fully-covered node is found as a result of executing `Multiple_Source`. The storage set will contain all the fully-covered nodes in this case. For each node j in the storage set, we should add any non-fully-covered neighbor, say i, to the storage set if and only if **cost_red_qm**[i, j] is greater than zero. The same process must be recursively applied to all the nodes in the resulting storage set until no more new node can be added. We assert that, as a result of this operation, the final storage set will be the optimal one. The same process can be applied to the single storage node identified by the `NoFullCov` function in case no fully-covered node exists in the network. However, the result gives only near-optimal solution in this case.

6 Performance Evaluation

We have conducted extensive simulations to validate the proposed algorithm and evaluate its performance. The performance is compared with two other schemes.

6.1 Simulation Setup

In the simulation setting, the number of network nodes in a tree network varies from 100 to 300. A tree network containing a collection of nodes is randomly constructed with a maximum degree of six. Each source node is assigned a source rate that is randomly selected from some given range (per unit time), with the default range being [2.0, 4.0]. Non-source nodes have a source rate of zero. In the simulation, we vary the probability that a network node is a source node, so as to capture different network configurations. This probability is called *source probability* in the following treatment. Each network node (including a source node) is assigned a query rate that is randomly selected from the range of [2.0, 4.0] per unit time. In addition, each channel in the tree network is assigned a communication cost (i.e., cost for transmitting one unit of data over the channel) that is randomly selected from the range of [1.0, 3.0]. As described previously, we assume that sizes of source data and data returned as a result of a query are both equal to one data unit.

In the experiment, two other algorithms are also simulated for comparison with our algorithm. Recall that total communication cost composes of two components: data query cost and data push cost. In addition, any storage node set must always form a connected subtree of the network. The first algorithm, denoted as `QP_First`, first designates a node that has the largest sum of source rate and query rate as the first storage node. It then adds a neighbor of the storage node to the storage node set if the addition results in reduction of communication cost. The same process is repeated for any new storage node until no further cost reduction is possible. This algorithm is a greedy one and, basically, attempts to gain benefit by cutting total communication cost at the outset.

Note that the push-mode operation described earlier represents nothing but a special case in which all nodes in the tree are storage nodes, while the pull mode of operation does not resort to any storage node. Hence, we also choose to compare our scheme with the pull mode, called `Pull` in the following discussion. With `Pull`, a requesting node retrieves data from all source nodes to satisfy each of its queries, hence no data push cost is induced. Let T_F represent the subtree used to forward source data for a query issued by a node x with `Pull`. In the simulation, we assume that each source node s has to send to x newly collected source data that is still unknown to x. The

expected size of source data sent from s to x is then g_s / r_x. When measuring data query cost with `Pull`, we take account of *data aggregation*, which refers to the effect that data sent by various source nodes in response to the query may be aggregated to reduce the size of transmitted data. In the simulation, we have adopted three different aggregation models for `Pull`. Let node $i \neq x$ be a node on T_F. Node i may be a source node or an intermediary non-source node. Denote A the set of neighbors of i in T_F from which i receives (possibly aggregated) source data. Further, let $j \in T_F$ be the neighbor of i to which i forwards source data, and $Z(i, j)$ denote the average size of source data sent from i to j, in response to the query. It is apparent that $j \notin A$. For exposition purpose, we let $Z(i, i)$ represent the average size of source data generated by i that is sent to x, if i is a source node, and hence $Z(i, i) = g_i / r_x$. $Z(i, i)$ is zero if i is not a source node. The first aggregation model, called `max-aggr`, assumes that $Z(i, j)$ is given by

$$Z(i, j) = \max(\alpha \cdot \sum_{k \in A \cup \{i\}} Z(k, i), \max_{k \in A \cup \{i\}} Z(k, i))$$

where $\alpha \in [0,1]$ is the aggregation factor. That is, `max-aggr` assumes the size of source data forwarded from i to j is given by the maximum of different (aggregated) sources and the current aggregation result. The second aggregation model, denoted as `mnx-aggr`, is more aggressive in which $Z(i, j)$ is given by

$$Z(i, j) = \beta \cdot \max_{k \in A'} Z(k, i) + (1 - \beta) \min_{k \in A'} Z(k, i)$$

where $\beta \in [0,1]$ is the aggregation factor and $A' = A \cup \{i\}$ if i is a source node, $A' = A$ otherwise. Note that `max-aggr` with $\alpha = 1$ is identical to `mnx-aggr` with $\beta = 0$. The third model, called `unit-aggr`, assumes unit data size for all links on T_F (i.e. $Z(i, j) = 1$). This model represents one of the most aggressive data aggregation scenarios.

The main performance metric adopted in the simulation is average communication cost. We also examine the numbers of storage nodes produced by `QP_First` and our algorithm. For a given number of network nodes, communication cost is averaged over 1000 randomly constructed network topologies, with 100 sets of source nodes per source probability being randomly chosen from the network nodes for each of the topologies. In other words, each simulation result is averaged over 100,000 combinations of topologies and source nodes.

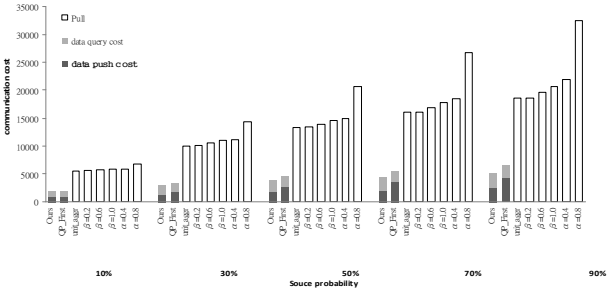


Fig. 8. Communication cost vs. source probability in a 100-node network.

6.2 Simulation Results

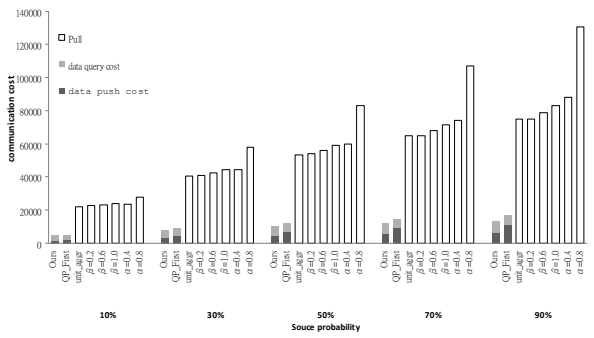
(a) Effects of Source Probability

Fig. 8 depicts total communication cost with source probability varying from 10% to 90% in a network with 100 nodes for the three algorithms. The method of `Pull` is evaluated using the three data aggregation models described above with α and β assuming various values for `max-aggr` and `mnx-aggr`, respectively. In the figure, communication cost consists of two components – data query cost and data push cost. Note that the cost for `Pull` includes no component of data push cost. Apparently, `Pull` performs much worse than `QP_First` and our algorithm under all data aggregation models, since source data may be transmitted redundantly for queries issued by different querying nodes. Our algorithm outperforms `QP_First` mostly due to a lower data push cost. As shown later, `QP_First` tends to yield more storage nodes than the optimal solution produced by ours. An optimal storage node set strikes a balance between data query cost and data push cost. Performance difference between our algorithm and the other two is more evident as source probability grows bigger. As we will show later, a higher source probability will lead to a smaller number of storage nodes in an optimal solution, causing data push cost to be sensitive to the number and locations of storage nodes. In fact, when no fully-covered node exists in the network, there is only one storage node in the optimal solution. In this case, selection of storage nodes also has more significant impact on data query cost as well. In contrast, the impact on network performance is most obvious for `Pull`.

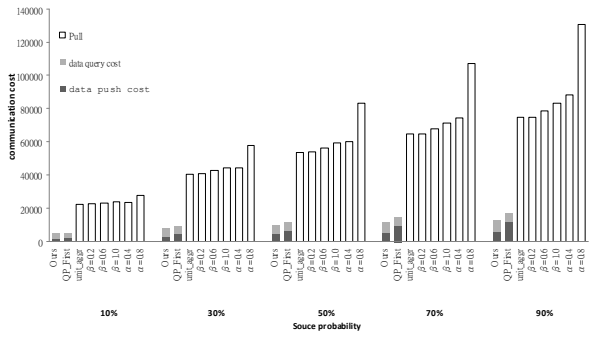
To examine performance of the algorithms with respect to network size, we show in Fig. 9 communication cost versus source probability for networks with 200 and 300 nodes. The other settings are identically set as the ones of Fig. 8. A trend similar to that of Fig. 8 is consistently observed in the figure. In addition, the simulation results reveal that, when network size grows, our algorithm proportionally gains more benefit when compared with the other two, which implies better scalability achieved by our algorithm.

(b) Source Rate vs. Query Rate

We have also evaluated performance of the algorithms when source rate is varied with respect to query rate. The simulated network has 200 nodes, source probability is set to 50%, and query rate is fixed in the range of [2.0, 4.0]. In Fig. 10, we compare the algorithms, in terms of total communication cost, with source rate falls in five different ranges: [0.0, 2.0], [1.0, 3.0], [2.0, 4.0], [3.0, 5.0], and [4.0, 6.0]. The figure shows that our algorithm outperforms all other algorithms in the simulation setting, while `Pull` has the worst performance with any aggregation model. For `Pull` with `unit-aggr` model, communication cost is constant, by nature, across the entire spectrum of source rate. For all the other algorithms, the cost demonstrates linear-like increase when source rate grows, with ours showing milder slope than the others. This can be explained as follows. When source rate increases relative to query rate in a network, data push load rises. As will be shown shortly, this will lead to more conservative instal-



(a) 200-node network



(b) 300-node network

Fig. 9. (a) Communication cost vs. source probability in a 200-node network. (b) Communication cost vs. source probability in a 300-node network.

lation of storage nodes for an optimal solution. Following an argument similar to the one given above for high source probability, it becomes evident that our algorithm gains relatively more advantage in this situation.

(c) Number of Storage Nodes

For a given network topology, the number of storage nodes produced by `QP_First` and our algorithm depends on values of source rate, query rate, and source probability. To gain more insight in this regard, we plot the number of storage nodes versus source probability for various settings of source rate for the algorithms in Fig. 11. In the simulation plotted, network size is 200 nodes. Three settings of source rate ranges are used: [0.0, 2.0], [2.0, 4.0], and [4.0, 6.0]. The number of storage nodes produced by both algorithms decreases when source rate or source probability increases. This is because the volume of source data grows at a higher source probability or source rate, which makes the data-push operation more costly. As a result, the algorithms refrain from data-push operations and hence yield less storage nodes.

In particular, our algorithm results in less number of storage nodes than `QP_First`. This is because, `QP_First` will grow to include at least all fully-covered nodes, when such nodes exist in the network. Furthermore, in the case of no fully-covered node, `QP_First` may likely add more storage nodes in order to cut down communication cost. Although more storage nodes lead to lower data query cost, they eventually result in higher data push cost as a whole. Difference in the number of produced storage nodes between our algorithm and

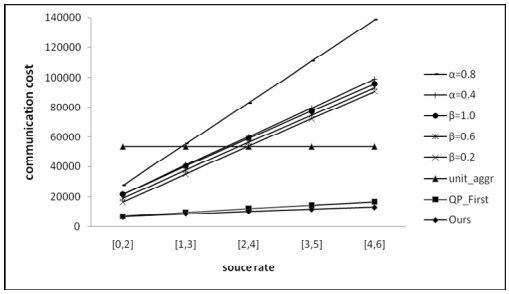


Fig. 10. Communication cost for the algorithms with respect to various ranges of source rate.

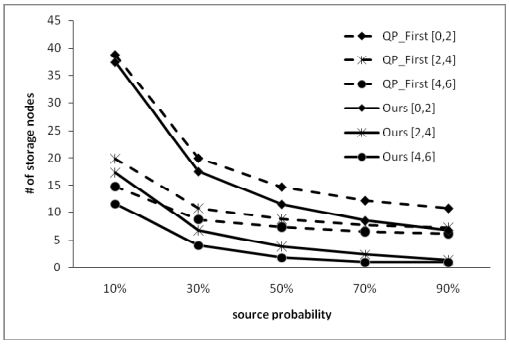


Fig. 11. Number of storage nodes vs. source probability.

TABLE 2
PERCENTAGE OF SCENARIOS WITH NO FULLY-COVERED NODES IN A 200-NODE NETWORK

Source rate	Source probability				
	10%	30%	50%	70%	90%
[2,4]	0%	0%	0%	0%	0%
[3,5]	0%	0%	0%	0.60%	44.34%
[4,6]	0%	0%	0%	39.84%	76.24%

`QP_First` is more obvious when source probability increases. From the discussion above, we see that the proposed optimal algorithm not only yields minimum amount of communication cost but also achieves it with a smallest number of storage nodes. This feature helps reduce storage space requirement for a network.

The optimal solution produced by our algorithm contains only one storage node when no fully-covered node exists in the network. Table 2 lists percentage of such network scenarios versus source probability for a network with 200 nodes. In the table, we show three sets of simulation results (employing our algorithm), with source rate falling in ranges of [2.0, 4.0], [3.0, 5.0], and [4.0, 6.0]. As source probability grows, the percentage of no fully-covered scenarios rises. Similar trend is also observed when source rate is increased. For example, nearly 40% of network scenarios yield no fully-covered node at source probability of 70%, when source rate falls in range [4.0, 6.0].

(d) Including Communication Cost of Transmitting Query Messages

As described in Section 5.3, if cost for transmitting

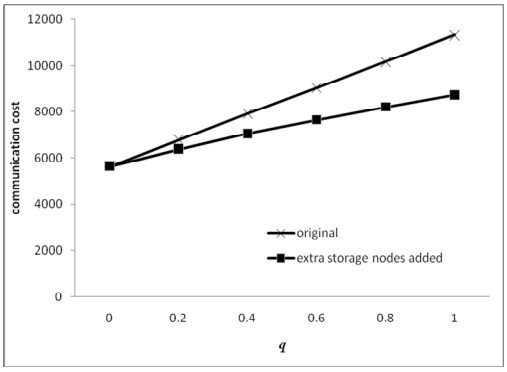


Fig. 12. Communication cost vs. q, ratio of the size of a query message to a data unit, with and without node-adding scheme implemented.

query messages is considered in the cost model, one may need to add more storage nodes after algorithm `Multiple_Source` is completed, in order to further reduce total cost. In what follows, we show effects of implementing such a scheme. Fig. 12 illustrates communication cost with respect to various q, ratio of the size of a query message to a data unit, for a 200-node network. In the figure, we compare communication cost with and without the above mentioned scheme implemented. In the simulation, both query rates and source rates are randomly chosen from [2.0, 4.0] and source probability is fixed at 50%.

From Fig. 12, we see that the proposed scheme constantly offers benefit, in terms of communication cost, for the network, and the cost reduction is more evident when q is increased. This is because traffic due to transmitting query messages becomes a more critical factor for communication cost when q grows larger, which renders data queries more costly. In this case, adding extra storage nodes are likely to cut down data query cost as a whole. For example, in the figure, overall communication cost can be reduced by as much as 15.5% and 22.8% with q equal to 0.6 and 1.0, respectively.

7 Related Issues

In this section, we discuss some issues that have not been specifically addressed in previous sections.

7.1 Storage Constraint

Previously, we have assumed that storage space would not be a problem for storage nodes. We now consider storage constraint problem. In many network applications, queries are issued in request of recently collected data, rather than old data. While new data are received by storage nodes, stale data can be discarded. Hence storage requirement can be less critical for the nodes. In addition, a storage node does not necessarily store all data in its raw form. For example, a storage node can convert collected data into an average value, in order to satisfy queries of the like. In this case, a storage node may only need to store a single average value. The same argument applies to many other queries of this nature, such as the maximum, the minimum, etc. Through such a mechanism the saving of memory space can be significant for storage nodes. Having a storage node receive all source data tends to make the data converting process more effective.

Consider the situation in which raw data is needed in order to satisfy some queries and the amount of data may exceed a storage node's capacity. Recall that storage nodes will cluster together to form a subtree. One can take advantage of this property and distribute the collected data among these nodes. In other words, a storage node no longer maintains an entire set of data. Rather, each storage node may only store a part of it, but all storage nodes collectively contain the entire set of data. Basically, we should have data generated by a source node be saved by the same set of storage nodes throughout the operation in order to simplify the coordination task. Further, we should avoid the partitioning of data collected by a source node. Each storage node is first assigned a number of branches of the network for which it is the closest storage node. Source data generated by source nodes in these branches have higher priority to be stored by the storage node. When the capacity of the node is exceeded, the storage node can then ask its neighboring storage nodes to offer space for storing data generated by some of the source nodes in its branches. The data distribution must be made known to all storage nodes. In this case, it may take more than one storage node to answer a query. When a query is issued, the first storage node contacted will take charge of it as usual, except that the node now has to parse the query and then request needed data from other storage nodes using an in-cluster multicast message. This operation is somewhat similar to a partial pulling scheme within the cluster of storage nodes. Communication cost due to coordination should be moderate since these storage nodes are close to each other. In case the total storage capacity of all storage nodes is not enough to accommodate the data, non-storage nodes in the vicinity of the storage set can be turned into storage nodes, if needed, in order to deal with storage constraint problem.

7.2 Distributed Version of the Proposed Algorithm

The proposed algorithm `Rate_Comp` can be easily extended to a distributed version as follows. All node x that is of degree one is allowed to send $S(x, y)$ and $R(x, y)$ to its only neighbor y. When y receives x's values, it adds $S(x, y)$ to $\mathbf{S\_sum}[y]$ and $R(x, y)$ to $\mathbf{R\_sum}[y]$, accompanied by increasing $\mathbf{in\_count}[y]$ by one. Node y is able to compute $S(y, z)$ and $R(y, z)$ for its neighbor z when $ch(x, y)$ is saturated for all of y's other neighbor $x \neq z$. In this way, when all channels starting from x are saturated, x can check by itself whether it is a fully-covered node.

However, we need to deal with the situation in which no fully-covered node is found. Lemma 9 states that all node $i \in T - \Phi$ are connected. Furthermore, if node i finds that it covers none of its neighbors, it can deduce from Corollary 4 that there is no other fully-covered node in the network, and hence it becomes a candidate storage node for an optimal solution. Function `NoFullCov` can be implemented in a distributed manner as described earlier. However, it is now $b(i, j)$ and $\mathbf{cost\_reduction}[i]$ that are computed. After this step is done, a comparison of the amounts of cost reduction between the nodes in the residual tree is performed to determine the optimal stor-

age node. Such a comparison can be implemented by a distributed election protocol [28].

8 Conclusions and Future Work

Although the assumption of heterogeneous channel costs seems to complicate the optimal storage problem, our treatment has tackled potential difficulties by identifying necessary and sufficient conditions that are irrelevant to specific channel costs. The proposed algorithm is of O(N) in terms of time complexity, where N is the number of nodes in the tree. This is comparable to that of our counterparts which assume homogeneous channel costs. The proposed optimal algorithm is also useful for determining storage node set for general topologies. One possibility is to first find a minimum spanning tree for the network. In this regard, we may assume that each (undirected) link is assigned a cost which is equal to the smaller of the two channel costs associated with the link. We can then apply our algorithm to the produced minimum spanning tree. Although this may not lead to an optimal solution, it however should offer a quality solution for the network.

There are several directions for future research. First, although our algorithm offers an optimal solution that requires the minimum number of storage nodes, we plan to specifically incorporate the cost taken by storage nodes in determining the optimal storage set in our future work. Also, more sophisticated cost model that takes other factors, such as queuing effect and link quality, into consideration shall be further studied. Finally, it is a long-term goal to design an efficient protocol that acts dynamically to any topology change in a wireless network for an optimal set of storage nodes.

Acknowledgments

This work was supported in part by the National Science Council, Taiwan, under grants NSC 97-2221-E-011-070-MY3, NSC 98-2221-E-390-024, and NSC 97-2221-E-011-032-MY3.

References

[1] B. M. Khumawala, "An efficient branch and bound algorithm for the warehouse location problem," *Management Science*, vol. 18, no. 12, Aug. 1972.

[2] D. B. Shmoys, E. Tardos, and K. Aardal, "Approximation algorithms for facility location problems," in *Proc. of 29th ACM Symposium on Theory and Computing*, pp. 265-274, 1997.

[3] S. Bhattacharya, H. Kim, S. Prabh, and T. Abdelzaher, "Energy-conserving data placement and asynchronous multicast in wireless sensor networks," in *Proc. of the 1st Int'l Conf. on Mobile Systems, Applications, and Services*, pp. 173-185, 2003.

[4] L. Dowdy and D. Foster, "Comparative models of the file assignment problem," *ACM Computing Surveys*, vol. 14, no. 2, 1982.

[5] M. Fisher and D. Hochbaum, "Database location in computer networks," *J. ACM*, vol. 27, no. 4, 1982.

[6] J. Douceur and R. Wattenhofer, "Optimizing file availability in a secure serverless distributed file system," in *Proc. of Reliable Distributed Systems*, 2001.

[7] C. Aggarwal, J. Wolf, and P. Yu, "Caching on the World Wide Web," *IEEE Trans. on Knowledge and Data Eng.*, vol. 11, no. 1, Jan./Feb. 1999.

[8] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy, "Adaptive push-pull: disseminating dynamic Web data," *IEEE Trans. on Computers*, vol. 51, no. 6, pp. 652-668, 2002.

[9] K. S. Prabh and T. F. Abdelzaher, "Energy-conserving data cache placement in sensor networks," *ACM Trans. on Sensor Networks*, vol. 1, no. 2, 2005.

[10] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu, "Data-center storage in sensornets with GHT, a geographic hash table," *Mobile Networks and Applications*, vol. 8, no. 4, pp. 427-442, 2003.

[11] B. Sheng, Q. Li, and W. Mao, "Data storage placement in sensor networks," in *Proc. of Mobihoc*, 2006.

[12] K. Kalpakis, K. Dasgupta, and O. Wolfson, "Optimal placement of replicas in trees with read, write, and storage costs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 12, no. 6, 2001.

[13] O. Wolfson and A. Milo, "The multicast policy and its relationship to replicated data placement," *ACM Trans. on Database Systems*, vol. 16, no. 1, pp. 181-205, Mar. 1991.

[14] G. Cornuejols, G. Nemhauser, and L. Wolsey, "Discrete location theory, chapter the uncapacitated facility location problem," *Lecture Note in Artificial Intelligence* (LNAI 1865), Wiley, pp. 119–171, 1990.

[15] D. Wessels and K. Claffy, "ICP and the Squid Web cache," *IEEE J. on Selected Areas in Comm.*, vol. 16, no. 3, 1998.

[16] B. Sheng, C. Tan, Q. Li, and W. Mao, "An approximation algorithm for data storage placement in sensor networks," in *Proc. of Wireless Algorithms, Systems and Applications*, 2007.

[17] K. Ross, "Hash routing for collections of shared Web caches," *IEEE Networks*, vol. 11, no. 6, pp. 37-44, 1997.

[18] T. Hara, "Effective replica allocation in ad hoc networks for improving data accessibility," in *Proc. of IEEE INFOCOM*, vol. 3, pp. 1568-1576, 2001.

[19] S. Madden, R. Szewczyk, M. J. Franklin, and D. Culler, "Supporting aggregate queries over ad-hoc wireless sensor networks," in *IEEE Workshop on Mobile Computing and Systems*, pp. 49-58, 2002.

[20] L. Yin and G. Cao, "Supporting cooperative caching in ad hoc networks," *IEEE Trans. on Mobile Computing*, vol. 5, no. 1, Jan. 2006.

[21] J. Newsome and D. Song, "GEM: graph embedding for routing and data-centric storage in sensor networks without geographic information," in *Proc. of the 1st Int'l Conf. on Embedded Networked Sensor Systems*, pp. 76-88, 2003.

[22] R. Shah, S. Roy, S. Jain, and W. Brunette, "Data MULEs: modeling a three-tier architecture for sparse sensor networks," in *Proc. of the 1st IEEE Int'l Workshop on Sensor Network Protocols and Applications*, pp. 30-41, May 2003.

[23] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-efficient communication protocols for wireless microsensor networks," in *Proc. of Int'l Conf. on System Sciences*, 2000.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. on Networking*, vol. 11, no. 1, 2003.

[25] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE J. on Selected Areas in Comm.*, vol. 22, no. 1, Jan. 2004.

[26] Gnutella Protocol Development, "Gnutella 0.6 RFC," June, 2002. [Online]. Available: http://rfc-gnutella.sourceforge.net/src/rfc-0.6-draft.html.

[27] B. Zelinka, "Medians and peripherians of trees," *Archivum Mathe-*

maticum, vol.4, no.2, pp. 87–95, 1968.
[28] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE Trans. on Computers*, vol. 31, no. 1, pp. 48-59, Jan. 1982.
[29] M. L. Brandeau and S. S. Chiu, "An overview of representative problems in location research," *Management Science*, vol. 35, no. 6, 1989.
[30] L. W. Dowdy and D. V. Foster, "Comparative models of the file assignment problem," *ACM Computing Survey*, vol. 14, no. 2, pp. 287-313, 1982.
[31] B. Li, M. J. Golin, G. F. Italiano, X. Deng, and K. Sohraby, "On the optimal placement of Web proxies in the Internet," in *Proc. of INFOCOM*, vol. 3, pp. 1282-1290, 1999.
[32] L. Qiu, V. N. Padmanabhan, and G. M. Voelker, "On the placement of Web replicas," in *Proc. of INFOCOM*, vol. 3, pp. 1587-1596, 2001.
[33] S. Sivasubramanian, M. Szymaniak, and G. Pierre, "Replication for Web hosting systems," *ACM Computing Survey*, vol. 36, no. 3, pp. 291-334, 2004.
[34] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making gnutella-like P2P systems scalable," in *Proc. of Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 407-418, 2003.
[35] J. Luo, J. P. Hubaux, and P. T. Eugster, "PAN: providing reliable storage in mobile ad hoc networks with probabilistic quorum systems," in *Proc. of Mobihoc*, pp. 1-12, 2003.
[36] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, "Next century challenges: scalable coordination in sensor networks," in *Proc. of Mobicom*, pp. 263-270, 1999.

Ge-Ming Chiu received the B.S. degree from National Cheng-Kung University, Taiwan, in 1976, the M.S. degree from the Texas Tech University in 1981, and the Ph.D. degree from the University of Southern California in 1991, all in electrical engineering. He is currently a professor in the Department of Computer Science and Information Engineering at National Taiwan University of Science and Technology, Taipei, Taiwan. His research interests include distributed computing, mobile computing, fault-tolerant computing, and parallel processing. Dr. Chiu is a member of the IEEE.

Li-Hsing Yen received the BS (1989), MS (1991), and PhD (1997) degrees in computer science, all from National Chiao Tung University, Taiwan. He was an assistant professor (1998-2003) and then an associate processor (2003-2006) with the Department of Computer Science and Information Engineering at Chung Hua University, Taiwan. He was an associate professor (2006 to 2010) and has been a full professor since 2010 with the Department of Computer Science and Information Engineering, National University of Kaohsiung, Taiwan. His current research interests include mobile computing, wireless networking, and distributed algorithms. Dr. Yen is a member of the IEEE.

Tai-Lin Chin obtained his BS degree in Computer Science and Information Engineering from National Chiao-Tung University, Taiwan, in 1995, and the MS and PhD degrees in Electrical and Computer Engineering from the University of Wisconsin-Madison in 2004 and 2006, respectively. He is currently with the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, as an Assistant Professor. His primary research interests include wireless ad hoc, vehicular, and sensor networks, mobile and pervasive computing, and distributed algorithms. He is a member of the IEEE.