Supporting Cooperative Caching in Ad Hoc Networks

Liangzhong Yin, *Student Member*, *IEEE*, and Guohong Cao, *Member*, *IEEE*

Abstract—Most researches in ad hoc networks focus on routing and not much work has been done on data access. A common technique used to improve the performance of data access is caching. Cooperative caching, which allows the sharing and coordination of cached data among multiple nodes, can further explore the potential of the caching techniques. Due to mobility and resource constraints of ad hoc networks, cooperative caching techniques designed for wired networks may not be applicable to ad hoc networks. In this paper, we design and evaluate cooperative caching techniques to efficiently support data access in ad hoc networks. We first propose two schemes: CacheData, which caches the data, and CachePath, which caches the data path. After analyzing the performance of those two schemes, we propose a hybrid approach (HybridCache), which can further improve the performance by taking advantage of CacheData and CachePath while avoiding their weaknesses. Cache replacement policies are also studied to further improve the performance. Simulation results show that the proposed schemes can significantly reduce the query delay and message complexity when compared to other caching schemes.

Index Terms—Cooperative cache, cache management, cache replacement policy, ad hoc networks, data dissemination, simulations.

1 Introduction

Wireless ad hoc networks have received considerable attention due to the potential applications in battlefield, disaster recovery, and outdoor assemblies. Ad hoc networks are ideal in situations where installing an infrastructure is not possible because the infrastructure is too expensive or too vulnerable. Due to lack of infrastructure support, each node in the network acts as a router, forwarding data packets for other nodes. Most of the previous research [7], [11], [12], [25] in ad hoc networks focuses on the development of dynamic routing protocols that can efficiently find routes between two communicating nodes. Although routing is an important issue in ad hoc networks, other issues such as information (data) access are also very important since the ultimate goal of using ad hoc networks is to provide information access to mobile nodes. We use the following two examples to motivate our research on data access in ad hoc networks.

Example 1. In a battlefield, an ad hoc network may consist of several commanding officers and a group of soldiers around the officers. Each officer has a relatively powerful data center, and the soldiers need to access the data centers to get various data such as the detailed geographic information, enemy information, and new commands. The neighboring soldiers tend to have similar missions and thus share common interests. If one soldier accessed a data item from the data center, it is quite possible that nearby soldiers access the same data some time later. It saves a large amount of battery power, bandwidth, and time if later accesses to the same data are served by the nearby soldier who has the data instead of the faraway data center.

Example 2. Recently, many mobile infostation systems have been deployed to provide information for mobile users. For example, infostations deployed by a tourist information center may provide maps, pictures, and the history of attractive sites. Infostations deployed by a restaurant may provide menus. Due to limited radio range, an infostation can only cover a limited geographical area. If a mobile user, say Jane, moves out of the infostation range, she will not be able to access the data provided by the infostation. However, if mobile users are able to form an ad hoc network, they can still access the information. In such an environment, when Jane's request is forwarded to the infostation by other mobile users, it is very likely that one of the nodes along the path has already cached the requested data. Then, this node can send the data back to Jane to save time and bandwidth.

From these examples, we can see that if mobile nodes are able to work as request-forwarding routers, bandwidth and power can be saved, and delay can be reduced. Actually, *cooperative caching* [22], [8], [6], which allows the sharing and coordination of cached data among multiple nodes, has been widely used to improve the Web performance. Although cooperative caching and proxy techniques have been extensively studied in wired networks, little has been done to apply this technique to ad hoc networks. Due to mobility and resource constraints, techniques designed for wired networks may not be applicable to ad hoc networks. For example, most research on cooperative caching in the Web environment assumes a fixed topology, but this may not be the case in ad hoc networks due to mobility. Since the cost of the wireless link is different from the wired link, the decision regarding where to cache the data and how to get the cached data may be different.

In this paper, we design and evaluate cooperative caching techniques to efficiently support data access in ad

• *The authors are with the Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802. E-mail: {yin, gcao}@cse.psu.edu.*

Manuscript received 20 Feb. 2004; revised 11 June 2004; accepted 30 Aug. 2004; published online 16 Nov. 2005.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-0034-0204.

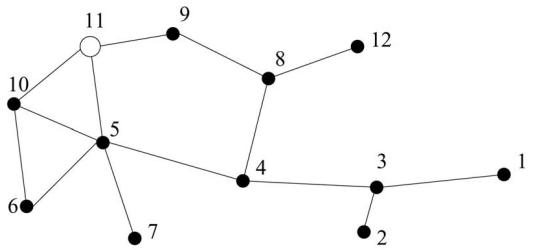

Fig. 1. An ad hoc network.

hoc networks. Specifically, we propose three schemes: CachePath, CacheData, and HybridCache. In CacheData, intermediate nodes cache the data to serve future requests instead of fetching data from the data center. In CachePath, mobile nodes cache the data path and use it to redirect future requests to the nearby node which has the data instead of the faraway data center. To further improve the performance, we design a hybrid approach (HybridCache), which can further improve the performance by taking advantage of CacheData and CachePath while avoiding their weaknesses. Simulation results show that the proposed schemes can significantly improve the performance in terms of the query delay and the message complexity when compared to other caching schemes.

The rest of the paper is organized as follows: In Section 2, we present the CacheData scheme and the CachePath scheme. Section 3 presents the HybridCache scheme. The performance of the proposed schemes is evaluated in Section 4. Section 5 discusses the related work. Section 6 concludes the paper.

2 Proposed Basic Cooperative Cache Schemes

In this section, we propose two basic cooperative cache schemes and analyze their performance.

2.1 System Model

Fig. 1 shows part of an ad hoc network. Some nodes in the ad hoc network may have wireless interfaces to connect to the wireless infrastructure such as wireless LAN or cellular networks. Suppose node N_{11} is a data source (center), which contains a database of n items $d_1, d_2, \ldots, d_n$. Note that N_{11} may be a node connecting to the wired network which has the database.

In ad hoc networks, a data request is forwarded hop-by-hop until it reaches the data center and then the data center sends the requested data back. Various routing algorithms have been designed to route messages in ad hoc networks. To reduce the bandwidth consumption and the query delay, the number of hops between the data center and the requester should be as small as possible. Although routing protocols can be used to achieve this goal, there is a limitation on how much they can achieve. In the following, we propose two basic cooperative caching schemes: *CacheData* and *CachePath*.

2.2 Cache the Data (*CacheData*)

In CacheData, the node caches a passing-by data item d_i locally when it finds that d_i is popular, i.e., there were many requests for d_i, or it has enough free cache space. For example, in Fig. 1, both N_6 and N_7 request d_i through N_5, N_5 knows that d_i is popular and caches it locally. Future requests by N_3, N_4, or N_5 can be served by N_5 directly. Since CacheData needs extra space to save the data, it should be used prudently. Suppose the data center receives several requests for d_i forwarded by N_3. Nodes along the path $N_3 - N_4 - N_5$ may all think that d_i is a popular item and should be cached. However, it wastes a large amount of cache space if three of them all cache d_i. To avoid this, a conservative rule should be followed: *A node does not cache the data if all requests for the data are from the same node*. As in the previous example, all requests received by N_5 are from N_4, which in turn are from N_3. With the new rule, N_4 and N_5 do not cache d_i. If the requests received by N_3 are from different nodes such as N_1 and N_2, N_3 will cache the data. If the requests all come from N_1, N_3 will not cache the data, but N_1 will cache it. Certainly, if N_5 receives requests for d_i from N_6 and N_7 later, it may also cache d_i. Note that d_i is at least cached at the requesting node, which can use it to serve the next query.

This conservative rule is designed to reduce the cache space requirement. In some situations, e.g., when the cache size is very large or for some particular data that are interested by most nodes, the conservative rule may decrease the cache performance because data are not cached at every intermediate node. However, in mobile networks, nodes usually have limited cache spaces and we do not assume that some data are interested by all nodes. Therefore, the conservative rule is adopted in this paper.

2.3 Cache the Data Path (*CachePath*)

The idea of CachePath can be explained by using Fig. 1. Suppose node N_1 has requested a data item d_i from N_{11}. When N_3 forwards the data d_i back to N_1, N_3 knows that N_1 has a copy of d_i. Later, if N_2 requests d_i, N_3 knows that the data center N_{11} is three hops away whereas N_1 is only one hop away. Thus, N_3 forwards the request to N_1 instead of N_4. Note that many routing algorithms (such as AODV [17] and DSR [11]) provide the hop count information between the source and destination. By caching the data path for each data item, bandwidth and the query delay can be reduced since the data can be obtained through fewer number of hops. However, recording the map between data items and caching nodes increases routing overhead. In the following, we propose some optimization techniques.

When saving the path information, a node need not save all the node information along the path. Instead, it can save only the destination node information, as the path from current router to the destination can be found by the underlying routing algorithm.

In CachePath, a node does not need to record the path information of all passing-by data. For example, when d_i flows from N_{11} to destination node N_1 along the path $N_5 - N_4 - N_3$, N_4 and N_5 need not cache the path information of d_i since N_4 and N_5 are closer to the data center than the caching node N_1. Thus, a node only needs to record the data path when it is closer to the caching node than the data center.

Due to mobility, the node which caches the data may move. The cached data may be replaced due to the cache

size limitation. As a result, the node which modified the route should reroute the request to the original data center after it finds out the problem. Thus, the cached path may not be reliable and using it may adversely increase the overhead. To deal with this issue, a node N_i caches the data path only when the caching node, say N_j, is very close. The closeness can be defined as a function of its distance to the data center, its distance to the caching node, the route stability, and the data update rate. Intuitively, if the network topology is relatively stable, the data update rate is low, and its distance to the caching node (denoted as $H(i,j)$) is much lower than its distance to the data center (denoted as $H(i,\mathcal{C})$), the routing node should cache the data path. Note that $H(i,j)$ is a very important factor. If $H(i,j)$ is small, even if the cached path is broken or the data are unavailable at the caching node, the problem can be quickly detected to reduce the overhead. Certainly, $H(i,j)$ should be smaller than $H(i,\mathcal{C})$. The number of hops that a cached path can save is denoted as

$$H_{save} = H(i,\mathcal{C}) - H(i,j),$$

where H_{save} should be greater than a system tuning threshold, called $\mathcal{T}_H$, when CachePath is used.

2.3.1 Maintain Cache Consistency

There is a cache consistency issue in both CacheData and CachePath. We have done some work [4], [5] on maintaining strong cache consistency in the single-hop-based wireless environment. However, due to bandwidth and power constraints in ad hoc networks, it is too expensive to maintain strong cache consistency, and the weak consistency model is more attractive. A simple weak consistency model can be based on the Time-To-Live (TTL) mechanism, in which a node considers a cached copy up-to-date if its TTL has not expired, and removes the map from its routing table (or removes the cached data) if the TTL expires. As a result, future requests for this data will be forwarded to the data center.

Due to TTL expiration, some cached data may be invalidated. Usually, invalid data are removed from the cache. Sometimes, invalid data may be useful. As these data have been cached by the node, it indicates that the node is interested in these data. When a node is forwarding a data item and it finds there is an invalid copy of that data in the cache, it caches the data for future use. To save space, when a cached data item expires, it is removed from the cache while its id is kept in "invalid" state as an indication of the node's interest. Certainly, the interest of the node may change, and the expired data should not be kept in the cache forever. In our design, if an expired data item has not been refreshed for the duration of its original TTL time (set by the data center), it is removed from the cache.

When cooperative caching is used, mobile nodes need to check passing-by data besides routing. This may involve cross-layer optimization, and it may increase the processing overhead. However, the processing delay is still very low compared to the communication delay. Since most ad hoc networks are specific to some applications, cross-layer optimization can also reduce some of the processing overhead. Considering the performance improvement, the use of cooperative cache is well justified.

2.4 Performance Analysis

In this section, we analyze the performance of the proposed schemes. The performance metric is the distance (hop count) between a requester and a node that has the requested data. This node can be a caching node, or the data center when no caching node is found. Reducing the hop count can reduce the query delay, the bandwidth, and the power consumption since fewer nodes are involved in the query process. Further, reducing the hop count can also reduce the workload of the data center since requests served by caches will not be handled by the data center.

The notations used in the analysis are as follows:

- $\overline{H}$: The average number of hops between a mobile node and the data center.
- P_{dd}: The probability that a data item is in the cache in the CacheData scheme.
- P_{dp}: The probability that a data item is in the cache in the CachePath scheme.
- P_{pp}: The probability that a path is in the cache in the CachePath scheme.
- P_i: The probability that a cached item is not usable. This may be caused by TTL expiration or broken paths because of node movement.
- L_d: In CacheData, the average length of the path for a request to reach the node (or the original server) which has a valid copy of the data. If the requester has a valid copy of the data, $L_d = 1$ for ease of presentation.
- L_p: In CachePath, the average length of the path for a request to reach the node (or the original server) which has a valid copy of data. $L_p = 1$ if the requester has a valid copy of the data.

We make some assumptions to simplify the analysis. For example, we assume that parameters such as P_{dd}, P_{dp}, P_{dp}, and P_i in all nodes are the same. These assumptions allow us to study the ad hoc network where these parameters are affected by various factors such as cache size, topology changes, node/link failures, etc. Most of these factors are not easy to model, especially when they can affect each other. The simulation results in Section 4 match the analytical results and verify that these assumptions are reasonable.

Given these notations, we can obtain the expected number of hops that a request takes from node N_i to the node which has the data. Let $P'_d = P_{dd}(1 - P_i)$, then

$$\begin{aligned} L_d &= P'_d \cdot 1 + (1 - P'_d) \cdot P'_d \cdot 2 + \ldots \\ &\quad + (1 - P'_d)^{H(i,\mathcal{C})-1} \cdot P'_d \cdot H(i,\mathcal{C}) \\ &= \sum_{k=1}^{H(i,\mathcal{C})} (1 - P'_d)^{k-1} \cdot P'_d \cdot k \\ &\approx \frac{1}{P_{d'}} = \frac{1}{P_{dd}(1 - P_i)}. \end{aligned} \tag{1}$$

This equation is an approximation of L_d since, in practice, P_{dd} may be different at different nodes. Equation (1) helps us understand the effects of many important factors, and we believe the approximation is reasonable. Note that L_d is bounded by $\overline{H}$. When P'_d is not too small, i.e., not less than $1/\overline{H}$, line 4 of (1) provides an adequate approximation.

To calculate L_p, three cases need to be considered:

1. The requested data item is in the local cache.
2. A path is found in the local cache which indicates N_i caches the requested data. Two subcases are possible:
   a. A valid data item is found in N_i.
   b. The data item in N_i is not usable because of broken path or TTL expiration.
3. No data or path is found in the local cache.

Let $P'_p = P_{dp}(1 - P_i)$. The probabilities of Cases 1, 2(a), 2(b), and 3 are P'_p, $(1 - P'_p)P_{pp}(1 - P_i)$, $(1 - P'_p)P_{pp}P_i$, and $(1 - P'_p)(1 - P_{pp})$, respectively. The number of hops needed for a request to get the data is 1 for Case 1 and $1 + L_p$ for Case 2(a) and Case 3. Note that, for Case 3, the distance is not $\overline{H}$ because intermediate nodes also check their local cache for the requested data or path. Thus, it is different from forwarding the request directly to the data center. For Case 2(b), the request needs to travel $1 + L_p$ to reach N_i. Then, it is redirected to the data center which is $\overline{H}$ away. At last, the data item is sent back to the requester in $\overline{H}$ hops. Therefore, the average number of hops needed for the request[1] $(1 + L_p) + \overline{H} + \overline{H}/2 = \overline{H} + (1 + L_p)/2$.

Thus,

$$L_p = P'_p \cdot 1 + (1 - P'_p) \cdot P_{pp} \cdot \left(P_i \left(\overline{H} + \frac{L_p + 1}{2} \right) + (1 - P_i)(L_p + 1) \right) + (1 - P'_p)(1 - P_{pp})(1 + L_p). \quad (2)$$

So,

$$L_p = \frac{P_{p\prime} + (1 - P_{p\prime})P_{pp}\left(P_i\overline{H} - \frac{P_i}{2} + 1\right) + (1 - P'_p)(1 - P_{pp})}{1 - (1 - P_{p\prime})P_{pp}\left(1 - \frac{P_i}{2}\right) - (1 - P'_p)(1 - P_{pp})}. \quad (3)$$

In (3), P_{pp} is specific to CachePath. Therefore, it needs to be fixed when comparing L_p to L_d. If $P_{pp} = 0$, $L_p = 1/(P_{dp}(1 - P_i))$, and, if $P_{pp} = 1$, then

$$L_p = \frac{P_{dp}(1 - P_i)\left(\frac{P_i}{2} - P_i\overline{H}\right) + \left(P_i\overline{H} - \frac{P_i}{2} + 1\right)}{1 - (1 - P_{dp}(1 - P_i))\left(1 - \frac{P_i}{2}\right)}. \quad (4)$$

$P_{pp} = 1$ gives the performance upper bound of CachePath. Equations (1) and (4) are still complex as they contain several parameters. We can fix some parameters to get a better understanding of the relation between L_d and L_p.

Suppose $P_i = 0$ (i.e., all the data items in the cache are valid), we have

$$L_d = \frac{1}{P_{dd}} \text{ and } L_p = \frac{1}{P_{dp}}. \quad (5)$$

CachePath needs less cache space to store extra data.[2] Therefore, $P_{dd} < P_{pd}$ when the cache size is not very big, which means $L_p < L_d$.

1. The average number of hops is the round-trip distance divided by two.
2. Note that a cached path only contains the final destination node id, as explained in Section 2.3. We assume that the size of any data item is larger than the size of a data id.

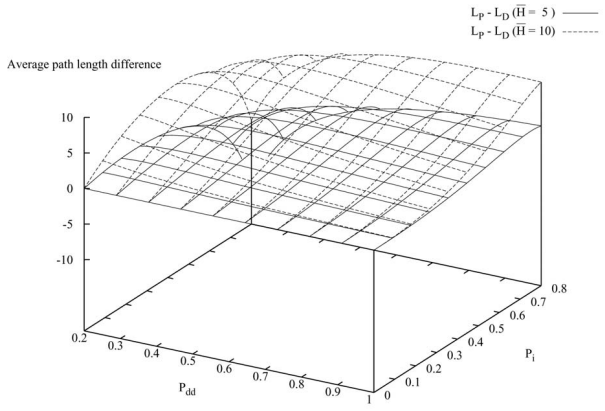

Fig. 2. Performance comparison.

Suppose $P_{dd} = P_{dp}$. This assumption favors CacheData since, in practice, the cache size is limited and $P_{dd} < P_{dp}$. Fig. 2 shows some numerical results of CachePath and CacheData by comparing their average path lengths L_p and L_d under two different $\overline{H}$ values: five and 10. As can be seen, in both cases, L_p is similar to L_d when P_i is small, which shows the advantage of CachePath considering that the assumption $P_{dd} = P_{dp}$ favors CacheData when the cache size is small. If the cache size is large enough so that P_{dp} is similar to P_{dd}, CacheData performs better as L_d is similar to or less than L_p. When P_i is high, the difference between L_p and L_d is also high because many requests follow paths that lead to data not useful in CachePath when P_i is high, which is essentially *chasing the wrong path*. In such a situation, it is better to adopt CacheData because it does not redirect requests.

Comparing these two cases: $\overline{H} = 5$ and $\overline{H} = 10$ in Fig. 2, we can see that a large $\overline{H}$ value decreases the performance of CachePath when compared to CacheData. This is because the penalty of chasing a wrong path is high when the network size is large (large $\overline{H}$). We can summarize the analytical results as follows:

- Both schemes can reduce the average number of hops between the requester and the node which has the requested data. For example, when $P_i = 0$, the number of hops can be reduced if the cache hit ratio is greater than $1/\overline{H}$. If there is no cached data or path available, our schemes fall back to traditional caching scheme, where requests are sent directly to the data center.
- When the cache size is small, CachePath is better than CacheData; when the cache size is large, CacheData is better.
- When the network size is small (small $\overline{H}$), CachePath is a good approach; when the network size is large, CacheData performs better.
- When the data items are updated slowly or mobile nodes move slowly, i.e., P_i is small, CachePath is a good approach; in other cases, CacheData performs better.

3 A Hybrid Caching Scheme (HybridCache)

The performance analysis showed that CachePath and CacheData can significantly improve the system performance. We also found that CachePath performs better in some situations such as small cache size or low data update

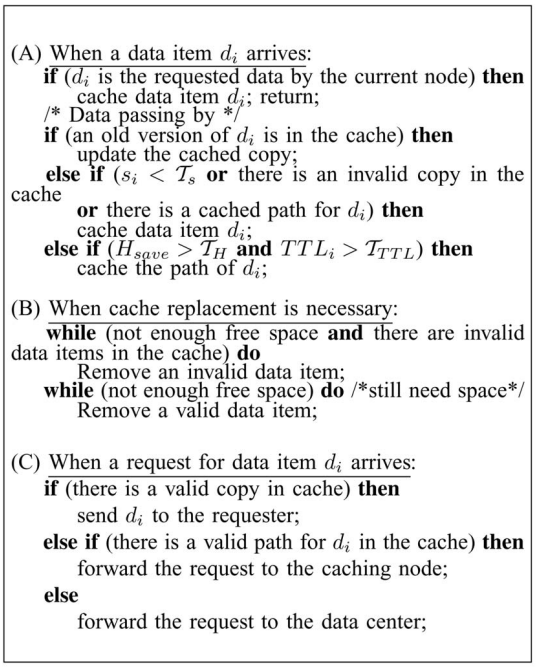

Fig. 3. The hybrid caching scheme.

rate, while CacheData performs better in other situations. To further improve the performance, we propose a hybrid scheme *HybridCache* to take advantage of CacheData and CachePath while avoiding their weaknesses. Specifically, when a node forwards a data item, it caches the data or path based on some criteria. These criteria include the data item size s_i, the TTL time TTL_i, and the H_{save}. For a data item d_i, the following heuristics are used to decide whether to cache data or path:

- If s_i is small, CacheData should be adopted because the data item only needs a very small part of the cache; otherwise, CachePath should be adopted to save cache space. The threshold value for data size is denoted as $\mathcal{T}_s$.
- If TTL_i is small, CachePath is not a good choice because the data item may be invalid soon. Using CachePath may result in chasing the wrong path and end up with resending the query to the data center. Thus, CacheData should be used in this situation. If TTL_i is large, CachePath should be adopted. The threshold value for TTL is a system tuning parameter and denoted as $\mathcal{T}_{TTL}$.
- If H_{save} is large, CachePath is a good choice because it can save a large number of hops; otherwise, CacheData should be adopted to improve the performance if there is enough empty space in the cache. We adopt the threshold value $\mathcal{T}_H$ used in CachePath as the threshold value.

These threshold values should be set carefully as they may affect the system performance. Their effects and how to set them are studied through simulations in Sections 4.2.1 and 4.2.2.

Fig. 3 shows the algorithm that applies these heuristics in HybridCache. In our design, caching a data path only needs to save a node id in the cache. This overhead is very small. Therefore, in HybridCache, when a data item d_i needs to be cached using CacheData, the path for d_i is also cached. Later, if the cache replacement algorithm decides to remove d_i, it removes the cached data while keeping the path for d_i. From some point of view, CacheData degrades to CachePath for d_i. Similarly, CachePath can be upgraded to CacheData again when d_i passes by.

3.1 Cache Replacement Policy

Because of limited cache size, a cache replacement policy must be adopted to evict data from the cache when new data arrive. One widely used cache replacement policy is LRU, which removes the least-recently-used data from the cache. However, some researches ([21], [27]) show that LRU can be outperformed by policies that consider various system parameters such as the data size, transfer time, data invalidation rate, etc. The problem with policies proposed in [21], [27] is that they require the input of many system parameters that are constantly changing and not easy to estimate.

In this paper, we focus on two parameters that are easier to get. The first parameter is the data size s_i. Data with larger size are better candidates for replacement because they occupy a large amount of cache space. Replacing them can make room for more incoming data items. The second parameter is $Order(d_i)$, the order of d_i according to the access interest. Let $k = Order(d_i)$, then d_i is the kth most frequently accessed data. Intuitively, data that are less likely to be accessed should be replaced first. Our policy, called the *Size*Order* cache replacement policy (SXO), combines these two parameters in the following value function:

$$value(d_i) = s_i * Order(d_i). \tag{6}$$

The data item with the largest $value(d_i)$ is replaced from the cache first.

In (6), s_i is known to mobile nodes because d_i is in the cache. Although $Order(d_i)$ is not available, it can be derived from the mobile node's access rate to d_i, denoted as a_i. In order to get a_i, we can apply similar techniques used by Shim et al. [21] as follows:

$$a_i = \frac{K}{T - T_{a_i}(K)}, \tag{7}$$

where T is the current time and $T_{a_i}(K)$ is the time of the Kth most recent access. If less than K samples are available, all available samples are used. It is shown in [21] that K can be as small as two or three to achieve the best performance. Thus, the spatial overhead to store recent access time is relatively small. Once the access frequency to each data item is available, it is easy to get the $Order(d_i)$ value.

Because $Order(d_i)$ is given through estimation, we cannot guarantee that $value(d_i)$ used in the cache replacement is absolutely accurate. Therefore, the SXO policy should remain effective even if $Order(d_i)$ is not very accurate. In our simulations, the sensitivity of SXO to data inaccuracy is studied by introducing noise to $Order(d_i)$. The result shows that SXO is able to perform well even when $Order(d_i)$ is not very accurate.

4 Performance Evaluation

The performance evaluation includes four sections. The simulation model is given in Section 4.1. In Section 4.2, we verify the analytical results of CacheData and CachePath and compare them to HybridCache and **SimpleCache**, which is the traditional cache scheme that only caches the received data at the query node. Section 4.3 compares HybridCache to SimpleCache and the cooperative caching scheme proposed by Lau et al. [13], referred to as **FloodCache**. FloodCache is designed for accessing multimedia data in ad hoc networks. When a query comes, it relies on flooding to find the nearest node that has the requested data. In both Sections 4.2 and 4.3, all schemes use LRU as the cache replacement policy. In Section 4.4, we study the effect of cache replacement policies on the query delay and compare the performance of HybridCache-SXO, which is the HybridCache scheme using the SXO cache replacement policy, to SimpleCache and HybridCache which use LRU.

4.1 The Simulation Model

The simulation is based on *ns-2* [14] with the CMU wireless extension. In our simulation, both AODV [17] and DSDV [16] were tested as the underlying routing algorithm. To save space, only the results based on DSDV are shown here because of the similarity between these two results.

The node density is changed by choosing the number of nodes between 50 and 100 in a fixed area. We assume that the wireless bandwidth is 2 Mb/s, and the radio range is 250m.

4.1.1 The Node Movement Model

We model a group of nodes moving in a 1500m $\times$ 320m rectangle area, which is similar to the model used in [25]. The moving pattern follows the random way point movement model [3]. Initially, nodes are placed randomly in the area. Each node selects a random destination and moves toward 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 and repeats this movement pattern. Two v_{max} values, 2 m/s and 20 m/s, are studied in the simulation.

4.1.2 The Client Query Model

The client query model is similar to what has been used in previous studies [4], [27]. Each node generates a single stream of read-only queries. The query generating time follows exponential distribution with mean value T_{query}. After a query is sent out, the node does not generate a new query until the query is served. The access pattern is based on $Zipf\text{-}like$ distribution [28], which has been frequently used [2] to model nonuniform distribution. In the Zipf-like distribution, the access probability of the ith ($1 \leq i \leq n$) data item is represented as follows:

$$P_{a_i} = \frac{1}{i^{\theta} \sum_{k=1}^{n} \frac{1}{k^{\theta}}},$$

where $0 \leq \theta \leq 1$. When $\theta = 1$, it follows the strict Zipf distribution. When $\theta = 0$, it follows the uniform distribution. Larger θ results in more "skewed" access distribution.

The access pattern of mobile nodes can be location-dependent; that is, nodes that are around the same location tend to access similar data, such as local points of interests. To simulate this kind of access pattern, a "biased" Zipf-like access pattern is used in our simulation. In this pattern, the whole simulation area is divided into 10 (X axis) by 2 (Y axis) grids. These grids are named grid 0, 1, 2,... 19 in a column-wise fashion. Clients in the same grid follow the same Zipf pattern, while nodes in different grids have different offset values. For example, if the generated query should access data id according to the original Zipf-like access pattern, then, in grid i, the new id would be $(id + n \bmod i) \bmod n$, where n is the database size. This access pattern can make sure that nodes in neighboring grids have similar, although not the same, access pattern.

TABLE 1
Simulation Parameters

Parameter	Default value	Range
Database size n	1000 items	
s_{min} (KB)	1	
s_{max} (KB)	10	
Number of nodes	100	50 to 100
v_{max} (m/s)	2	2, 20
Bandwidth (Mb/s)	2	
TTL (secs)	5000	200 to 10000
Pause time (secs)	300	
Client cache size (KB)	800	200 to 1200
Mean query generate time T_{query} (secs)	5	1 to 100
Zipf parameter θ	0.8	0 to 1
$\mathcal{T}_H$	2	1 to 5
$\mathcal{T}_s$ (% of $(s_{min} + s_{max})$)	40	10 to 100
$\mathcal{T}_{TTL}$ (secs)	5000	500 to 10000

4.1.3 The Server Model

Two data servers: server0 and server1 are placed at the opposite corners of the rectangle area. There are n data items at the server side and each server maintains half of the data. Data items with even ids are saved at server0 and the rests are at server1. The data size is uniformly distributed between s_{min} and s_{max}. The data are updated only by the server. The servers serve the requests on a FCFS (first-come-first-service) basis. When the server sends a data item to a mobile node, it sends the TTL tag along with the data. The TTL value is set exponentially with a mean value. After the TTL expires, the node has to get the new version of the data either from the server or from other nodes before serving the query.

Most system parameters are listed in Table 1. The second column lists the default values of these parameters. In the simulation, we may change the parameters to study their impacts. The ranges of the parameters are listed in the third column. For each workload parameter (e.g., the mean TTL time or the mean query generate time), the mean value of the measured data is obtained by collecting a large number of samples such that the confidence interval is reasonably small. In most cases, the 95 percent confidence interval for the measured data is less than 10 percent of the sample mean.

4.2 Simulation Results: *HybridCache*

Experiments were run using different workloads and system settings. The performance analysis presented here is designed to compare the effects of different workload parameters such as cache size, mean query generate time, node density, node mobility, and system parameters such

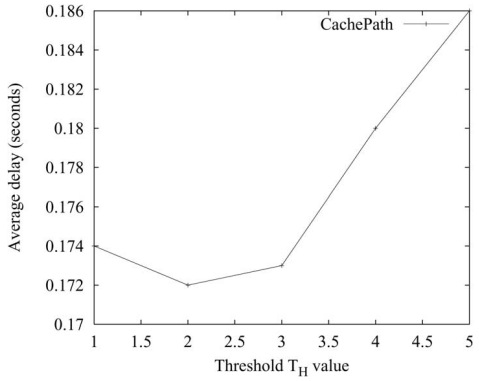

Fig. 4. Fine-tuning CachePath.

as TTL and θ on the performance of SimpleCache, CacheData, CachePath, and HybridCache.

4.2.1 Fine-Tuning CachePath

As stated in Section 2.3, the performance of CachePath is affected by the threshold value $\mathcal{T}_H$ as a path is only cached when its H_{save} value is greater than $\mathcal{T}_H$. A small $\mathcal{T}_H$ means more paths are cached, but caching too many less-valuable paths may increase the delay because the cached paths are not very reliable. A large $\mathcal{T}_H$ means only some valuable paths are cached. However, if $\mathcal{T}_H$ is too large, many paths are not cached because of the high threshold. As shown in Fig. 4, $\mathcal{T}_H = 2$ achieves a balance, and we use it in the rest of our simulations.

4.2.2 Fine-Tuning HybridCache

In HybridCache, if a data item size is smaller than $\mathcal{T}_s$, it is cached using CacheData. If $\mathcal{T}_s$ is too small, HybridCache fails to identify some small but important data items; if it is too large, HybridCache caches all the data using CacheData. To find an optimal value for $\mathcal{T}_s$, we measure the query delay as a function of $\mathcal{T}_s$. As $\mathcal{T}_s$ is related to data size, in Fig. 5a, we use a relative value: $\mathcal{T}_s/(S_{min} + S_{max})$, which can give us a clearer idea of what the threshold value should be.

As shown in Fig. 5a, when the threshold value increases from 10 percent to 40 percent, the query delay drops sharply since more data are cached. If the threshold value keeps increasing beyond 40 percent, more passing-by data are cached, and the cache has less space to save the accessed data. As a result, some important data may be replaced, and the delay increases. We find that a threshold value of 40 percent gives the best performance.

Fig. 5b shows the effect of $\mathcal{T}_{TTL}$ on the average query delay. The lowest query delay is achieved when $\mathcal{T}_{TTL} = 5,000$ seconds. Compared to Fig. 5a, the performance difference between different $\mathcal{T}_{TTL}$ is not significant. This is because the database we studied has heterogeneous data size. Data size varies from 1 KB to 10 KB. As data size is a very important factor for caching, it makes the effect of $\mathcal{T}_{TTL}$ less obvious.

4.2.3 Effects of the Cache Size

Fig. 6 shows the impacts of the cache size on the cache hit ratio and the average query delay. Cache hits can be divided into three categories: *local data hit*, which means that the requested data item is found in the local cache, *remote data hit*, which means that the requested data item is found in one of the intermediate node when the request is forwarded in the network, and *path hit*, which means that a path is found for the request and a valid data item is found in the destination node of that path. Both remote data hit and path hit are considered as remote cache hits because the data are retrieved from remote nodes.

From Fig. 6a, we can see that the local hit ratio of SimpleCache is always the lowest. When the cache size is small, CacheData performs similar to SimpleCache because small cache size limits the aggressive caching of CacheData. When the cache size is large, CacheData can cache more data for other nodes. These data can be used locally and, hence, the local data hit ratio increases. CachePath does not cache data for other nodes, but its cached data can be refreshed by the data passing by. Therefore, its local data hit ratio is still slightly higher than that of SimpleCache. HybridCache prefers small data items when caching data for other nodes. Therefore, it can accommodate more data and achieve a high local data hit ratio.

Although CacheData and CachePath have a similar local data hit ratio in most cases, CacheData always has a higher remote data hit ratio because it caches data for other nodes. Especially when the cache size is large, more data can be

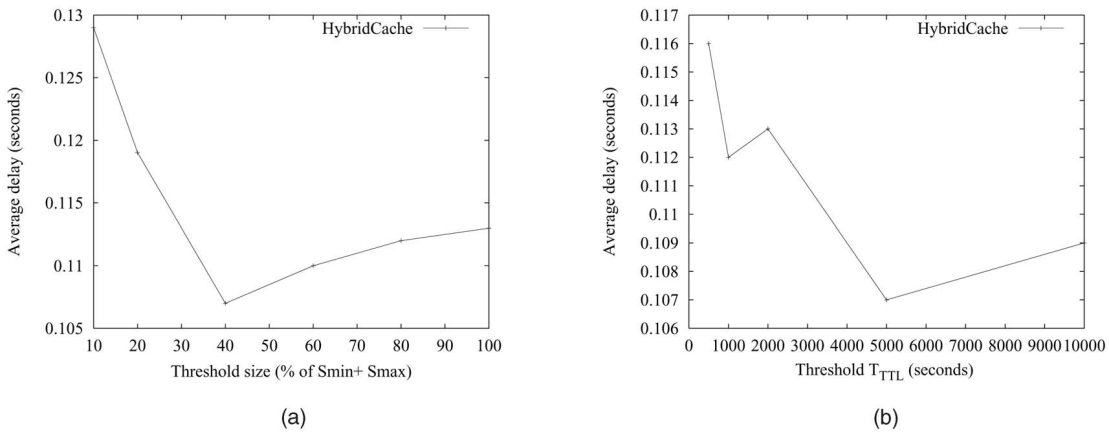

Fig. 5. Fine-tune HybridCache. (a) Threshold $\mathcal{T}_s$. (b) Threshold $\mathcal{T}_{TTL}$.

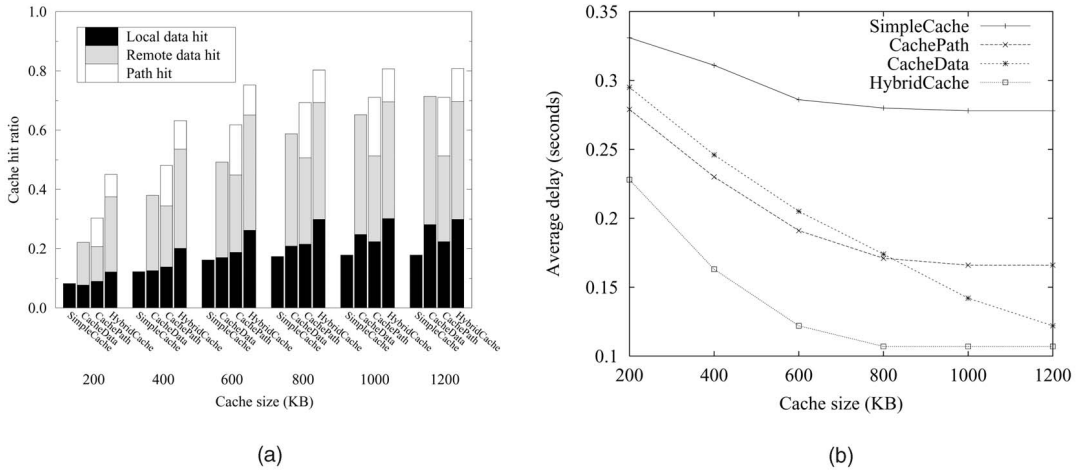

Fig. 6. The system performances as a function of the cache size. (a) Cache hit ratio. (b) Query delay.

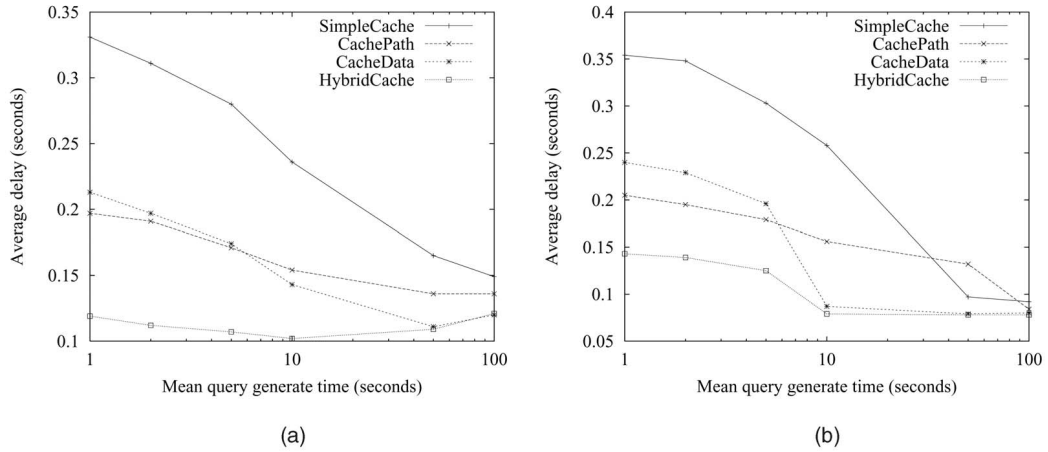

Fig. 7. The average query delay as a function of the mean query generate. (a) $V_{max} = 2$ m/s. (b) $V_{max} = 20$ m/s.

cached in CacheData and its remote data hit ratio is significantly higher than that of CachePath. HybridCache has a high remote data hit ratio due to similar reasons as for its high local data hit ratio. Even if the path hit is not considered, HybridCache still has highest cache hit ratio in most cases. It is worth noticing that CachePath and HybridCache almost reach their best performance when the cache size is 800 KB. This demonstrates their low cache space requirement. This particularly shows the strength of HybridCache as it also provides the best performance at the same time.

Because of the high cache hit ratio, the proposed schemes perform much better than SimpleCache (see Fig. 6). Comparing CachePath with CacheData, when the cache size is small, CachePath has lower query delay because its path hit helps reduce the average hop count. When the cache size is greater than 800 KB, these two schemes have similar total cache hit ratio, but CacheData has higher local data hit ratio and remote data hit ratio. Because the average hop count of local and remote data hit is lower than that of path hit, CacheData achieves low query delay. This figure also agrees with the performance analysis of CachePath and CacheData in Section 2.4.

Comparing all three proposed schemes, we can see that HybridCache performs much better than CacheData or CachePath, because HybridCache applies different schemes (CacheData or CachePath) to different data items, taking advantage of both CacheData and CachePath. As the result of the high local data hit ratio, remote data hit ratio, and overall cache hit ratio, HybridCache achieves the best performance compared to other schemes.

4.2.4 Effects of the Query Generate Time

Fig. 7 shows the average query delay as a function of the T_{query}. Both low mobility ($V_{max} = 2$ m/s) and high mobility ($V_{max} = 20$ m/s) settings are studied. We notice that all the trends are similar except CachePath. There are cases that CachePath even performs worse than SimpleCache. This is due to the fact that high node mobility causes more broken paths, which affects the performance of CachePath. In a high mobility setting, CacheData performs better and HybridCache still performs the best in most cases.

When T_{query} is small, more queries are generated and the system workload is high. As a result, the average query delay is high. As T_{query} increases, less queries are generated and the average query delay drops. If T_{query} keeps increasing, the average query delay only drops slowly or

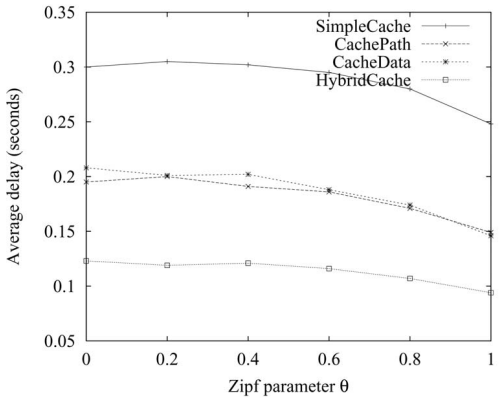

Fig. 8. The average query delay as a function of θ.

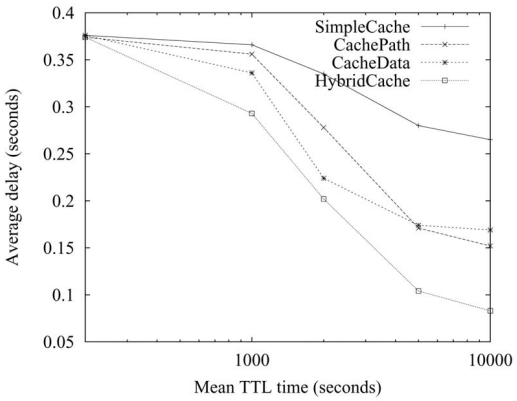

Fig. 9. The average query delay as a function of TTL.

even increases slightly. The reason is that the query generating speed is so low that the number of cached data is small and many cached data are not usable because their TTL have already expired before queries are generated for them. Fig. 7 verifies this trend.

Under heavy system workload (T_{query} is small), HybridCache can reduce the query delay by as much as 40 percent compared to CacheData or CachePath. When the system workload is extremely light, the difference between different schemes is not very large. This is because under an extreme light workload, the cache hit ratio is low. Therefore, most of the queries are served by the remote data center and different schemes perform similarly.

We can also find that, when the query generating speed increases (T_{query} decreases), the delay of HybridCache does not increase as fast as other schemes. This demonstrates that HybridCache is less sensitive to workload increases and it can handle much heavier workload.

4.2.5 Effects of the Zipf Parameter θ

The Zipf parameter θ defines the access pattern of mobile nodes. When θ is small, the access distribution is more like a uniform distribution. The average query delay is high since the cache is not large enough to save all the data. When θ is large, the access is focused on the hot (frequently accessed) data, and the average query delay is lower since most of these hot data can be cached. By changing θ, we can see how different access patterns affect the performance. As shown in Fig. 8, our schemes perform much better than the SimpleCache scheme because the cooperation between nodes can significantly reduce the query delay.

4.2.6 Effects of TTL

Fig. 9 shows the average query delay when the TTL varies from 200 seconds to 10,000 seconds. TTL determines the data update rate. Higher update rate (smaller TTL) makes the cached data more likely to be invalidated and, hence, the average query delay is higher. When the TTL is very small (200 sec.), all four schemes perform similarly because most data in the cache are invalid and then the cache hit ratio is very low. Since SimpleCache does not allow nodes to cooperate with other nodes, its average query delay does not drop as fast as our schemes when TTL increases. The delay of our schemes drops much faster as TTL increases because nodes cooperate with each other to maximize the benefit of low update rate.

Comparing CachePath to CacheData, CacheData performs better when TTL is small, whereas CachePath performs better when TTL is big. This result again agrees with the performance analysis. HybridCache further reduces the query delay by up to 45 percent.

4.2.7 Effects of the Node Density

Fig. 10 shows the average query delay as a function of the number of nodes in the system. As node density increases, the delay of all four schemes increases because more nodes compete for limited bandwidth. However, the delay of our schemes increases much slower than SimpleCache. This can be explained by the fact that more data can be shared as the number of nodes increases in our schemes, which helps reduce the query delay. When the total number of nodes is small, HybridCache performs similar as CacheData and CachePath. When the number of nodes increases, HybridCache performs much better than other schemes. This indicates that HybridCache scales well with the number of nodes.

4.3 Simulation Results: Comparisons

In this section, we compare the performance of the HybridCache scheme to the SimpleCache scheme and the FloodCache scheme in terms of the query delay and the message complexity. A commonly used message complexity metric is the total number of messages injected into the network by the query process [13]. Since each broadcast

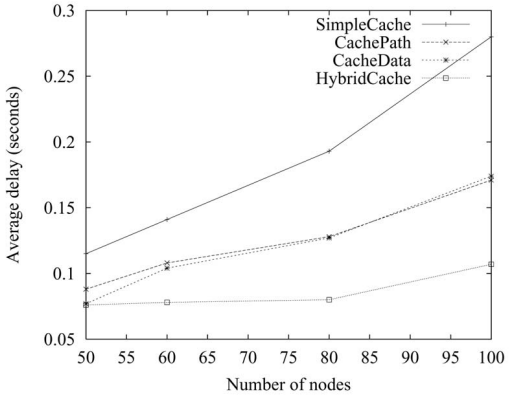

Fig. 10. The average query delay under different node density.

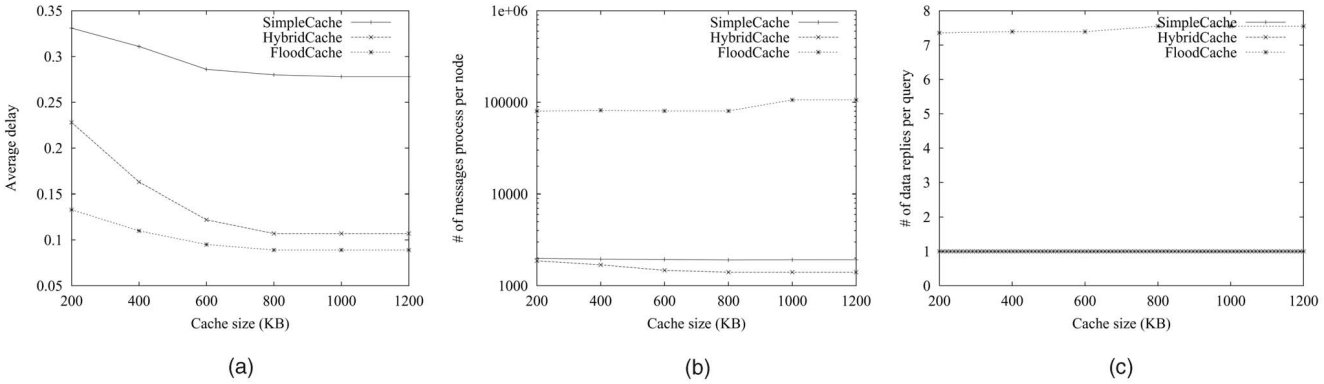

Fig. 11. The performance as a function of the cache size. (a) Query delay. (b) Message overhead. (c) Duplicated reply messages.

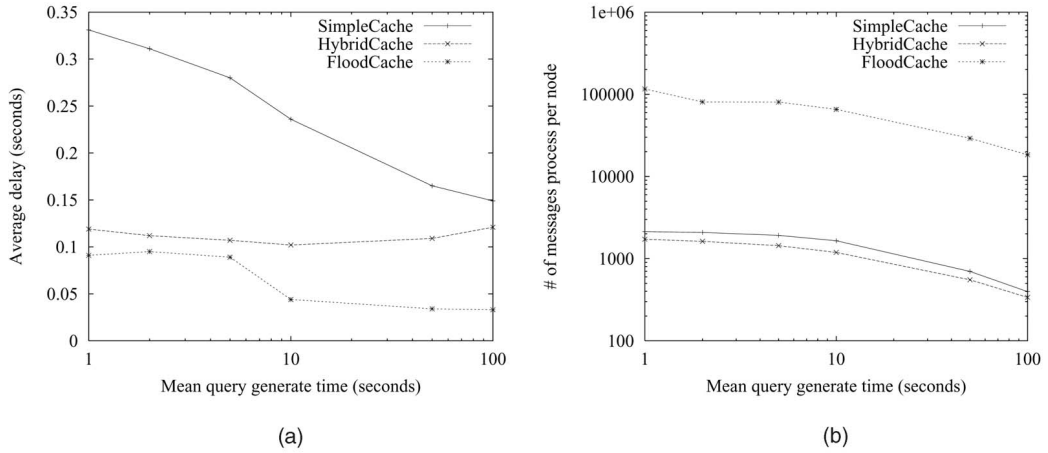

Fig. 12. The performance as a function of the mean query generate time T_{query}. (a) Query delay. (b) Message overhead.

message is processed (received and then rebroadcasted or dropped) by every node that received it, "the number of messages processed per node" is used as the message complexity metric to reflect the efforts (battery power, CPU time, etc.) of the mobile node to deal with the messages.

4.3.1 Effects of the Cache Size

Fig. 11 shows the impacts of the cache size on the system performance. Fig. 11a shows that the query delay decreases as the cache size increases. After the cache size increases beyond 800 KB, mobile nodes have enough cache size and the query delay does not drop significantly. The SimpleCache scheme is outperformed by cooperative caching schemes under different cache size settings. This demonstrates that mobile nodes can benefit from sharing data with each other.

FloodCache performs better compared to HybridCache in terms of the query delay. This is because, by flooding, FloodCache can find the nearest node that caches the requested data, which reduces the query delay. However, Fig. 11b shows that HybridCache incurs a much lower message overhead than FloodCache. The message overhead of HybridCache is even less than that of SimpleCache. The reason is that HybridCache gets data from nearby nodes instead of the faraway data center if possible. Therefore, the data requests and replies need to travel fewer number of hops and mobile nodes need to process fewer number of messages. As the cache size increases, the cache hit ratio of HybridCache increases and its message overhead decreases. Because FloodCache uses flooding to find the requested data, it incurs a much higher message overhead compared to SimpleCache and HybridCache.

In FloodCache, the request is sent out through flooding, and multiple copies of data replies may be returned to the requester by different nodes that have the requested data. In SimpleCache and HybridCache, this cannot happen because only one request is sent out for each query in case of local cache miss. Fig. 11c shows that more than seven copies of data replies are returned per query in FloodCache. The number of duplicated data replies increases slightly as the cache size increases because data can be cached in more nodes. In our simulation, the data size is relatively small (from 1 KB to 10 KB) and, hence, the duplicated messages do not affect the performance significantly. For some other environments such as multimedia accessing, transmitting duplicated data messages may waste much more power and bandwidth. As one solution, instead of sending the data to the requester upon receiving a request, mobile nodes which have the data may send back an acknowledgment. The requester can then send another unicast request to the nearest node among them to get the data. The drawback of this approach is that the query delay will be significantly increased.

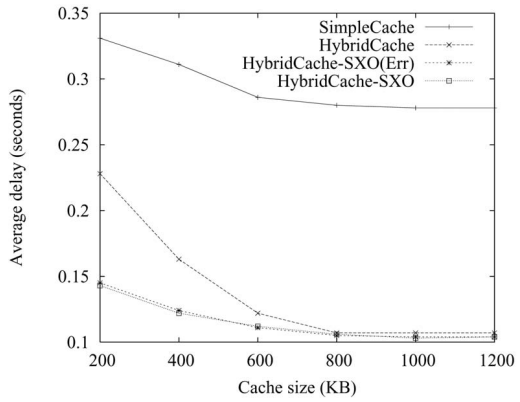

Fig. 13. The performance as a function of the cache size.

4.3.2 Effects of the Mean Query Generate Time T_{query}

Fig. 12 shows the effects of T_{query} on system performance. HybridCache performs similar to FloodCache when T_{query} is small. When the system workload is low (T_{query} is large), the difference between HybridCache and FloodCache increases. As explained in Section 4.2.4, when T_{query} is large, many cached data are not usable because of the TTL expiration. Therefore, the cache hit ratio is very low. Because FloodCache can find the nearest valid data, its query delay is small. HybridCache may not find the valid data item before a request reaches the data center. Therefore, its query delay is a little bit longer. Fig. 12a shows that the query delay of FloodCache is the lowest. However, as can be seen from Fig. 12b, the message overhead of FloodCache is significantly higher than that of HybridCache.

When considering the results from both Fig. 11 and Fig. 12, we can see that FloodCache uses significantly higher message overhead to get a very small query delay improvement over HybridCache. Thus, FloodCache may not be suitable for ad hoc networks where bandwidth and power are scarce. HybridCache performs well because it reduces the query delay compared to SimpleCache and incurs much less overhead compared to FloodCache.

4.4 Simulation Results: Cache Replacement Policy

The effect of the cache replacement policy is shown in Fig. 13. HybridCache-SXO denotes the HybridCache scheme that applies the SXO cache replacement policy. In order to study the effect of inaccurate input on SXO, noise is introduced by changing the value function to:

$$value(d_i) = s_i * (Order(d_i) + uniform(0, 20)).$$

Note that the $Order(d_i)$ used in SXO is already an estimated value. Here, more noise (uniform(0, 20)) is added to the estimated value to test the robustness of our scheme. The result is shown in Fig. 13 as HybridCache-SXO(Err).

Fig. 13 shows that HybridCache-SXO clearly outperforms HybridCache when the cache size is small since data size is a very important factor for cache replacement in SXO. Because mobile nodes usually have limited cache size, HybridCache-SXO is suitable for them. Even when noise is deliberately introduced to $value(d_i)$, the performance is almost the same as before, which shows the robustness of the proposed cache replacement policy.

When cache size is large enough (more than 800KB), the difference between HybridCache-SXO and HybridCache becomes small. When mobile nodes have enough cache space, there is always enough space for new data, and then cache replacement policies does not affect the cache performance too much. In such cases, we prefer HybridCache due to its low complexity.

5 Related Work

5.1 Caching Schemes in Wired Networks

Cooperative caching has been widely studied in the Web environment. These protocols can be classified as message-based, directory-based, hash-based, or router-based. Wessels and Claffy introduced the Internet cache protocol (ICP) [22], which has been standardized and is widely used. As a message-based protocol, ICP supports communication between caching proxies using a query-response dialog. It has been reported that ICP scales poorly with an increasing number of caches. Directory-based protocols for cooperative caching enables caching proxies to exchange information about cached content. The information is compressed using arrays of bits. Notable protocols of this class include Cache Digests [20] and summary cache [8]. The most notable hash-based cooperative caching protocol constitutes the cache array routing protocol (CARP) [19]. The rationale behind CARP constitutes load distribution by hash routing among Web proxy cache arrays. Wu and Yu introduced several improvements to hash-based routing, considering network latency and allowing local replication [24]. Web cache coordination protocol (WCCP) [6], as a router-based protocol, transparently distributes requests among a cache array. These protocols usually assume fixed network topology and often require high computation and communication overhead. However, in an ad hoc network, the network topology changes frequently. Also, mobile nodes have resource (battery, CPU, and wireless channel) constraints and cannot afford high computation or communication overhead. Therefore, existing techniques designed for wired networks may not be applied directly to ad hoc networks.

5.2 Caching Schemes in Wireless Networks

Most of the previous research [7], [11], [12], [25] in ad hoc networks focuses on routing, and not much work has been done on data access. The directed diffusion proposed by Intanagonwiwat et al. [10] addressed the cooperation among sensor nodes during data collection. Ye et al. [26] applied the query-forwarding concept to sensor networks. They proposed a two-tier data dissemination (TTDD) model for wireless sensor networks. TTDD requires the construction of a grid structure in fixed sensor networks. The nodes at the grid crossing points work as routers, which forward queries to the source and forward data to the sink. Although both approaches use cache, their focus is on data aggregation and compression for sensor networks, not on cooperative caching and data access.

To effectively disseminate data in ad hoc networks, data replication and caching can be used. Data replication schemes in ad hoc networks have been studied by Hara [9]. However, these schemes may not be very effective due

to the following reasons: First, because of frequent node movement, powering off or failure, it is hard to find stable nodes to host the replicated data. Second, the cost of initial distribution of the replicated data and the cost of redistributing the data to deal with node movement or failure is very high.

Similar to the idea of cooperative caching, Papadopouli and Schulzrinne [15] proposed a 7DS architecture, in which a couple of protocols are defined to share and disseminate data among users that experience intermittent connectivity to the Internet. It operates either on a prefetch mode to gather data for serving the users' future needs or on an on-demand mode to search for data on a single-hop multicast basis. Unlike our work, they focus on data dissemination instead of cache management. Further, their focus is the single-hop environment instead of the multihop environment as in our work.

A cooperative caching scheme designed specifically for accessing multimedia objects in ad hoc networks has been proposed by Lau et al. [13]. When a query comes, this scheme relies on flooding to find the nearest node that has the requested object. Using flooding can reduce the query delay since the request may be served by a nearby node instead of the data center faraway. Thus, it is good for multimedia applications which have strict delay requirements. Another benefit of using flooding is that multiple nodes that contain the requested data can be found. If the data size is very large, when the link to one node fails, the requester can switch to other nodes to get the rest of the requested data. Using flooding incurs significant message overhead. To reduce the overhead, in [13], flooding is limited to nodes within k hops from the requester, where k is the number of hops from the requester to the data center, but the overhead is still high. In a wireless network where nodes are uniformly distributed, on average, there are πk^2 nodes within k-hops range of a mobile node. Therefore, πk^2 messages are needed to find a data item using this method. Moreover, when a message is broadcast in the network, many neighbors will receive it. Even if the mobile node is able to identify and drop duplicated messages, each node still needs to broadcast the messages at least once to ensure full coverage. If a node has c neighbors, on average, the total number of messages needs to be processed is $c\pi k^2$. Although the message complexity is still $O(k^2)$, the constant factor may be very high, especially when the network density is high.

5.3 Cache Replacement Policies

Aggarwal et al. [1] classifies the existing cache replacement policies into three categories: *direct-extension*, *key-based*, and *function-based*. In the direct-extension category [18], traditional policies such as LRU or FIFO are extended to handle data items of nonhomogeneous size. The difficulty with such policies in general is that they fail to pay sufficient attention to the data size. In the key-based policies [23], keys are used to prioritize some replacement factors over others; however, such prioritization may not always be ideal. Recently, function-based replacement policy has received considerable attention [1], [21], [27]. The idea in function-based replacement policies is to employ a function of different factors such as time since last access, entry time of the data item in the cache, transfer time cost, data item expiration time, and so on. For example, the LNC-R-W3-U algorithm, proposed by Shim et al. [21], aims to minimize the response time in Web caching. Their cost function incorporates many system parameters such as the transfer time, the document size, and the invalidation rate. In [27], we proposed a generalized cost function for wireless environments. However, the solution was proposed for cellular networks and we did not consider mobility, which is different from this work.

6 Conclusions

In this paper, we designed and evaluated cooperative caching techniques to efficiently support data access in ad hoc networks. Specifically, we proposed three schemes: CachePath, CacheData, and HybridCache. In CacheData, intermediate nodes cache the data to serve future requests instead of fetching data from the data center. In CachePath, mobile nodes cache the data path and use it to redirect future requests to the nearby node which has the data instead of the faraway data center. HybridCache takes advantage of CacheData and CachePath while avoiding their weaknesses. Cache Replacement policies are also studied to further improve the cache performance. Simulation results showed that the proposed schemes can significantly reduce the query delay when compared to SimpleCache and significantly reduce the message complexity when compared to FloodCache.

Acknowledgments

The authors would like to thank the anonymous referees whose insightful comments helped us to improve the presentation of the paper. A preliminary version of the paper appeared in IEEE INFOCOM, 2004. This work was supported in part by the US National Science Foundation (CAREER CNS-0092770 and ITR-0219711).

References

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

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," *Proc. IEEE INFOCOM,* 1999.

[3] J. Broch, D. Maltz, D. Johnson, Y. Hu, and J. Jetcheva, "A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols," *Proc. ACM MobiCom,* pp. 85-97, Oct. 1998.

[4] G. Cao, "Proactive Power-Aware Cache Management for Mobile Computing Systems," *IEEE Trans. Computer,* vol. 51, no. 6, pp. 608-621, June 2002.

[5] G. Cao, "A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 5, Sept./Oct. 2003, preliminary version appeared in *Proc. ACM MobiCom'00.*

[6] M. Cieslak, D. Foster, G. Tiwana, and R. Wilson, "Web Cache Coordination Protocol v2.0," IETF Internet draft, http://www.ietf.org/internet-drafts/org/internet-drafts/draft-wilson-wrec-wccp-v2-00.txt, 2000.

[7] S. Das, C. Perkins, and E. Royer, "Performance Comparison of Two On-Demand Routing Protocols for Ad Hoc Networks," *Proc. IEEE INFOCOM,* pp. 3-12, 2000.

[8] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide Area Web Cache Sharing Protocol," *Proc. ACM SIGCOMM*, pp. 254-265, 1998.
[9] T. Hara, "Effective Replica Allocation in Ad Hoc Networks for Improving Data Accessibility," *Proc. IEEE INFOCOM*, 2001.
[10] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks," *Proc. ACM MobiCom*, Aug. 2000.
[11] D.B. Johnson and D.A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," *Mobile Computing*, pp. 153-181, Kluwer, 1996.
[12] Y. Ko and N. Vaidya, "Location-Aided Routing in Mobile Ad Hoc Networks," *Proc. ACM MobiCom*, pp. 66-75, 1998.
[13] W. Lau, M. Kumar, and S. Venkatesh, "A Cooperative Cache Architecture in Supporting Caching Multimedia Objects in MANETs," *Proc. Fifth Int'l Workshop Wireless Mobile Multimedia*, 2002.
[14] ns Notes and Documentation, http://www.isi.edu/nsnam/ns/, 2002.
[15] M. Papadopouli and H. Schulzrinne, "Effects of Power Conservation, Wireless Coverage, and Cooperation on Data Dissemination among Mobile Devices," *Proc. ACM MobiHoc*, Oct. 2001.
[16] C. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers," *Proc. ACM SIGCOMM*, pp. 234-244, 1994.
[17] C. Perkins, E. Belding-Royer, and I. Chakeres, "Ad Hoc On Demand Distance Vector (AODV) Routing," IETF Internet draft, draft-perkins-manet-aodvbis-00.txt, Oct. 2003.
[18] J. Pitkow and M. Recker, "A Simple Yet Robust Caching Algorithm Based on Dynamic Access Patterns," *Proc. Second Int'l World Wide Web Conf.*, 1994.
[19] K. Ross, "Hash Routing for Collections of Shared Web Caches," *IEEE Networks*, pp. 37-44, 1997.
[20] A. Rousskov and D. Wessels, "Cache Digests," *Computer Networks and ISDN Systems*, vol. 30, no. 22-23, pp. 2155-2168, 1998.
[21] J. Shim, P. Scheuermann, and R. Vingralek, "Proxy Cache Algorithms: Design, Implementation, and Performance," *IEEE Trans. Knowledge and Data Eng.*, vol. 11, no. 4, July/Aug. 1999.
[22] D. Wessels and K. Claffy, "ICP and the Squid Web Cache," *IEEE J. Selected Areas in Comm.*, pp. 345-357, 1998.
[23] S. Williams, M. Abrams, C. Standridge, G. Abdulla, and E. Fox, "Removal Policies in Network Caches for World-Wide Web Documents," *Proc. ACM Sigcomm96*, 1996.
[24] K. Wu and P. Yu, "Latency-Sensitive Hashing for Collaborative Web Caching," *Proc. World Wide Web Conf.*, pp. 633-644, 2000.
[25] Y. Xu, J. Heidemann, and D. Estrin, "Geography-Informed Energy Conservation for Ad Hoc Routing," *Proc. ACM MobiCom*, pp. 70-84, July 2001.
[26] F. Ye, H. Luo, J. Cheng, S. Lu, and L. Zhang, "A Two-Tier Data Dissemination Model for Large-Scale Wireless Sensor Networks," *Proc. ACM MobiCom*, 2002.
[27] L. Yin, G. Cao, and Y. Cai, "A Generalized Target-Driven Cache Replacement Policy for Mobile Environments," *Proc. 2003 Int'l Symp. Applications and the Internet (SAINT)*, Jan. 2003.
[28] G. Zipf, *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.

Liangzhong Yin received the BE and the ME degrees in computer science and engineering from the Southeast University, Nanjing, China, in 1996 and 1999, respectively. He is currently working toward the PhD degree in the Department of Computer Science and Engineering at the Pennsylvania State University. From 1999 to 2000, he was an engineer in the Global Software Group at Motorola. His research interests include wireless/ad hoc networks and mobile computing. He is a student member of the IEEE and the IEEE Computer Society.

Guohong Cao received the BS degree from Xian Jiaotong University, Xian, China. He received the MS and PhD degrees in computer science from the Ohio State University in 1997 and 1999, respectively. Since then, he has been with the Department of Computer Science and Engineering at the Pennsylvania State University, where he is currently an associate professor. His research interests are wireless networks and mobile computing. He has published about 100 papers in the areas of sensor networks, cache management, data dissemination, resource management, wireless network security, and distributed fault-tolerant computing. He is an editor of the *IEEE Transactions on Mobile Computing* and *IEEE Transactions on Wireless Communications*, a coguest editor of the special issue on heterogeneous wireless networks in *ACM/Kluwer Mobile Networking and Applications*, and has served on the program committee of many conferences. He was a recipient of the Presidential Fellowship at the Ohio State University in 1999, and a recipient of the US National Science Foundation CAREER award in 2001. He is a member of the IEEE and the IEEE Computer Society.