Supporting Cooperative Caching in Ad Hoc Networks

Liangzhong Yin and Guohong Cao
Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802
E-mail: {yin, gcao}@cse.psu.edu

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 network 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. Simulation results show that the proposed schemes can significantly reduce the query delay and message complexity when compared to other caching schemes.**

I. 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 researches [1], [2], [3] in ad hoc networks focus 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 tourist information center may provide maps, pictures, history of attractive sites. Infostation 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*, which allows the sharing and coordination of cached data among multiple nodes, has been widely used to improve the Web performance in wired networks. These protocols can be classified as message-based, directory-based, or router-based. Wessels and Claffy introduced the Internet cache protocol (ICP) [4], which has been standardized and is widely used. As a message-based protocol, ICP supports communication between caching proxies using a query-response dialog. Directory-based protocols such as cache digests [5] and summary cache [6] enable caching proxies to exchange information about cached contents. The web cache coordination protocol [7], 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

This work was supported in part by the National Science Foundation (CAREER CCR-0092770 and ITR-0219711).

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.

In this paper, we design and evaluate cooperative caching techniques to efficiently support data access in ad 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 query delay and message complexity when compared to other caching schemes.

The rest of the paper is organized as follows. In Section II, we present the CacheData scheme and the CachePath scheme. Section III presents the HybridCache scheme. The performance of the proposed schemes is evaluated in Section IV. Section V concludes the paper.

II. Proposed Basic Cooperative Cache Schemes

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

A. System Model

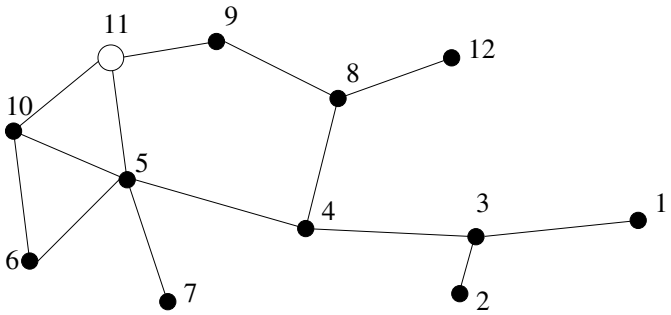


Fig. 1. An ad hoc network

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 $d_1, d_2, ..., d_n$. 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*.

B. 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.

C. Cache the Data Path (CachePath)

The idea of CachePath can be explained by 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 [8] and DSR [2]) provide the hop count information between the source and destination. By caching the data path for each data item, bandwidth and query delay can be reduced since the data can be obtained through less number of hops. However, recording the map between data items and caching nodes increases routing overhead. In the following, we propose some optimization techniques.

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 records the data path when it is closer (defined later) to the caching node than the data center.

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.

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 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.

Maintain cache consistency: There is a cache consistency issue in both CacheData and CachePath. We have done some work [9], [10] on maintaining strong cache consistency in 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.

D. Performance Analysis

In this section, we analyze the performance of the proposed schemes. We make some assumptions to simplify the analysis to get some conclusions. The simulation results in Section IV match the analytical results and verify that these assumptions are reasonable.

The performance is measured by the number of hops a request is expected to travel before it reaches the data. 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 easy 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.

Given the above 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

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

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 Equation (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 sub-cases 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. For Case 2(b), the request need 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 is $(1 + L_p + \overline{H} + \overline{H})/2 = \overline{H} + (1 + L_p)/2$. Thus

$$\begin{aligned} L_p \quad &= P_p^{'} \cdot 1 + (1 - P_p^{'}) \cdot P_{pp} \cdot \\ &\quad (P_i(\overline{H} + \tfrac{L_p+1}{2}) + (1 - P_i)(L_p + 1)) \\ &\quad + (1 - P_p^{'})(1 - P_{pp})(1 + L_p) \end{aligned} \tag{2}$$

So,

$$L_p \quad = \frac{{P_p}' + (1-{P_p}')P_{pp}(P_i\overline{H} - \frac{P_i}{2} + 1)) + (1-P_p^{'})(1-P_{pp})}{1-(1-{P_p}')P_{pp}(1-\frac{P_i}{2}) - (1-P_p^{'})(1-P_{pp})} \tag{3}$$

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

$$L_p = \frac{P_{dp}(1 - P_i)(\frac{P_i}{2} - P_i\overline{H}) + (P_i\overline{H} - \frac{P_i}{2} + 1)}{1 - (1 - P_{dp}(1 - P_i))(1 - \frac{P_i}{2}))} \tag{4}$$

$P_{pp} = 1$ 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

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

CachePath needs less cache space to store extra data[1]. Therefore $P_{dp} > P_{dd}$ when the cache size is not very big, which means $L_p < L_d$.

Suppose $P_i = 0.5$, and the cache size is big enough so that $P_{dp} = P_{dd}$. We obtain

$$L_d \quad = \quad \frac{2}{P_{dd}} \tag{6}$$

$$L_p \quad = \quad \frac{-2P_{dd}\overline{H} + P_{dd} + 4\overline{H} + 6}{2 + 3P_{dd}} \tag{7}$$

Thus,

$$L_d < L_p \Leftrightarrow \overline{H} > (4 - P_{dd}^2)/(4 - 2P_{dd}^2) \tag{8}$$

Note that $P_{dd} \in [0, 1]$ and

$$\frac{4 - P_{dd}^2}{4 - 2P_{dd}^2} \leq 1.5 \text{ if } P_{dd} \in [0, 1] \tag{9}$$

[1]Note that a cached path only contains the final destination node id, as explained in Section II-C. We assume that the size of any data item is larger than the size of a data id.

Combining Inequalities (8) and (9) yields,

$$L_d < L_p \quad \text{if} \quad \overline{H} > 1.5 \tag{10}$$

From the above equations, we can get the following conclusions:

- 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 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.

III. 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 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 re-sending 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.

Fig. 2 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.

(A) When a data item d_i arrives:
if (d_i is the requested data by the current node) **then**
cache data item d_i; return;
/* Data passing by */
if (an old version of d_i is in the cache) **then**
update the cached copy;
else if ($s_i < \mathcal{T}_s$ **or** there is an invalid copy in the cache **or** there is a cached path for d_i) **then**
cache data item d_i;
else if ($H_{save} > \mathcal{T}_H$ **and** $TTL_i > \mathcal{T}_{TTL}$) **then**
cache the path of d_i;

(B) When cache replacement is necessary:
while (not enough free space **and**
there are invalid data items in the cache) **do**
Remove an invalid data item;
while (not enough free space) **do** /*still need space*/
Remove a valid data item;

(C) When a request for data item d_i arrives:
if (there is a valid copy in cache) **then**
send d_i to the requester;
else if (there is a valid path for d_i in the cache) **then**
forward the request to the caching node;
else
forward the request to the data center;

Fig. 2. The hybrid caching scheme

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.

Comparing to Other Schemes

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 in [11]. 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.

Unlike data replication, caching does not rely on finding stable hosts and it has less overhead. In traditional caching schemes, referred to as **SimpleCache**, only the query node caches the received data. If another query request for the cached data comes before the cache expires, the node uses the cached data to serve the query. In case of a cache miss, it has to get the data from the data center. To utilize the caches of neighbor nodes, in the 7DS architecture [12], users can cache data and share with neighbors when experiencing intermittent connectivity to the Internet. However, the focus of 7DS is on single-hop environment instead of multi-hop. As a result, a user only broadcasts the request to its neighbors to see if the data can be served from their caches.

A cooperative caching scheme designed specifically for accessing multimedia objects in ad hoc networks has been proposed in [13]. When a query comes, this scheme relies on flooding to find the nearest node that has the requested object. We refer to this approach as the **FloodCache** scheme. 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.

The HybridCache scheme proposed in this paper does not use flooding. Its query delay may be higher than that of FloodCache in cases where the data are cached by some nearby nodes not along the route to the data center. However, in ad hoc networks FloodCache may not be a good choice due to its high message overhead.

When cooperative caching is used, mobile nodes need to cache 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.

IV. Performance Evaluation

The performance evaluation includes two parts. In the first part (Section IV-B), we verify the analytical results of CacheData and CachePath, and compare them to SimpleCache and HybridCache in terms of query delay. The second part (Section IV-C) compares HybridCache to SimpleCache and FloodCache in terms of query delay and message complexity.

A. The Simulation Model

The simulation is based on *ns-2* [14] with the CMU wireless extension. In our simulation, both AODV [8] and DSDV [3] were tested as the underlying routing algorithm. Because our schemes do not rely on specific routing protocols, the results from AODV and DSDV are similar. To save space, only the results based on AODV are shown here.

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.

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 [15]. The moving pattern follows the random way point movement model [16]. 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.

The client query model: The client query model is similar to what have been used in previous studies [9], [17]. Each node generates a single stream of read-only queries. The query generate time follows exponential distribution with mean value T_{query}. After a query is sent out, the node does not generate new query until the query is served. The access pattern is based on $Zipf-like$ distribution [18], which has been frequently used [19] to model non-uniform distribution. In the Zipf-like distribution, the access probability of the i^{th} ($1 \leq i \leq n$) 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. We choose θ to be 0.8 according to studies on real web traces [19].

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.

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 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.

TABLE I
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
$\mathcal{T}_H$	2	1 to 5
$\mathcal{T}_s$ (% of $(s_{min} + s_{max})$)	40	10 to 100
$\mathcal{T}_{TTL}$ (secs)	5000	500 to 10000

Most system parameters are listed in Table I. 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% confidence interval for the measured data is less than 10% of the sample mean.

B. Simulation Results: HybridCache

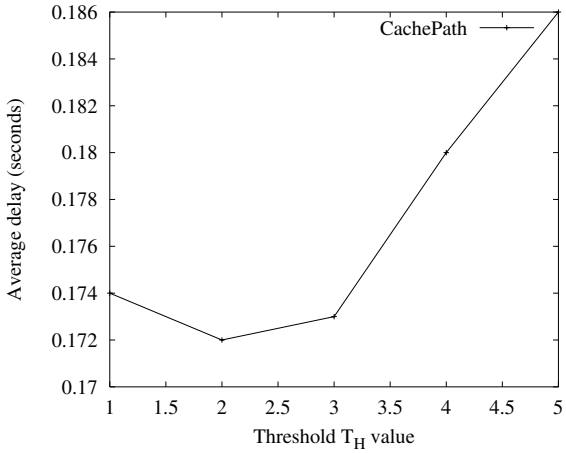
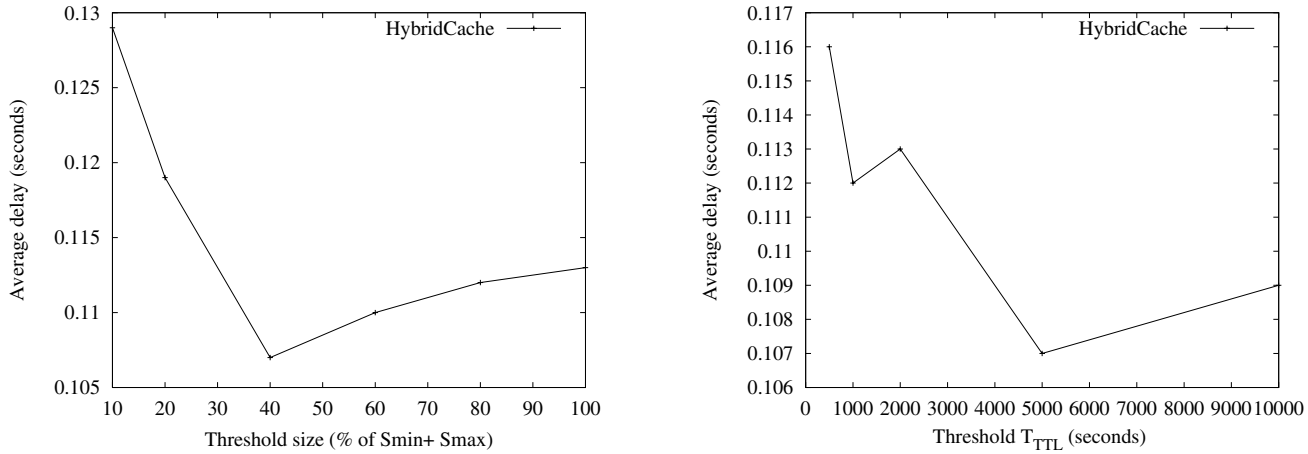


Fig. 3. Fine-tuning CachePath



(a) Threshold $\mathcal{T}_s$ (b) Threshold $\mathcal{T}_{TTL}$

Fig. 4. Fine-tune HybridCache

Experiments were run using different workloads and system settings. The performance analysis presented here is designed to compare the effects of different system parameters such as cache size and TTL on the performance of SimpleCache, CacheData, CachePath, and HybridCache. All schemes use the LRU algorithm for cache replacement. The effect of cache replacement is left as our future work.

1) Fine-tuning CachePath: As stated in Section II-C, 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. 3, $\mathcal{T}_H = 2$ achieves a balance, and we use it in the rest of our simulations.

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. 4 (a), 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.

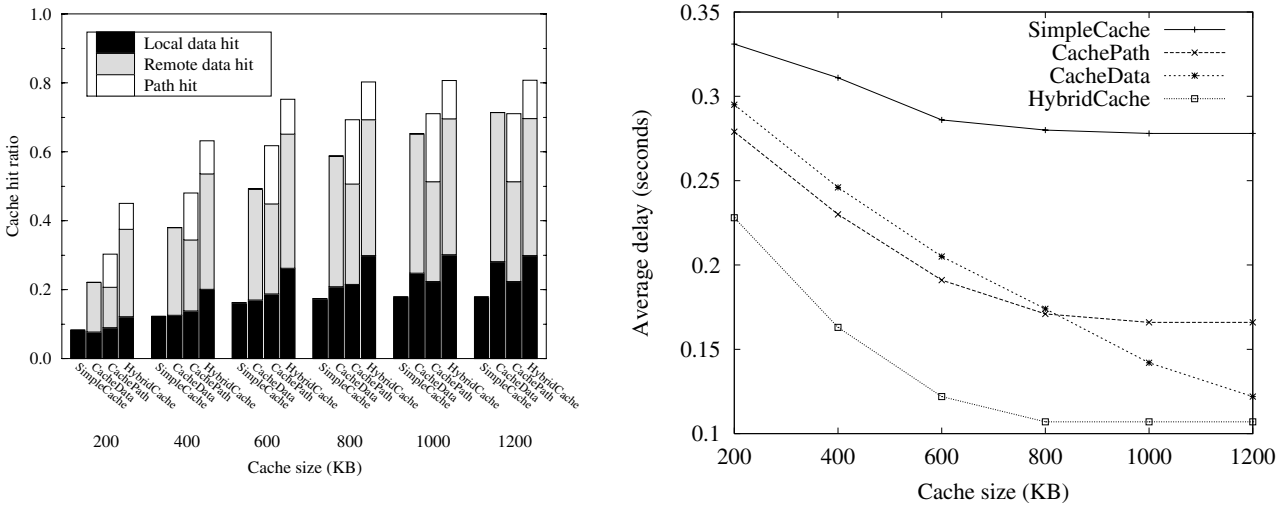
As shown in Fig. 4 (a), when the threshold value increases from 10% to 40%, the query delay drops sharply since more data are cached. If the threshold value keeps increasing beyond 40%, more passing-by data are cached, and the cache has less space to save the accessed data. As a result, some important data may be replaced, and the delay increases. We find that a threshold value of 40% gives the best performance.

Fig. 4 (b) shows the effect of $\mathcal{T}_{TTL}$ on the average query delay. The lowest query delay is achieved when $\mathcal{T}_{TTL} = 5000$ seconds. Compared to Fig. 4 (a), 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.

3) Effects of the Cache Size: Fig. 5 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 hit because the data are retrieved from remote nodes.

From Fig. 5 (a), 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 similar local data hit ratio in most cases, CacheData always has higher remote data hit ratio because it caches data for other nodes. Especially when the cache size is large, more data can be 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 reason 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



(a) Cache hit ratio (b) Query delay

Fig. 5. The system performances as a function of the cache size

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. 5). Comparing CachePath and 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 hop count of local data hit is 0 and the average hop count of remote data hit is lower than that of path hit, CacheData achieves low query delay. This figure also agrees with the performance comparisons of CachePath and CacheData in Section II-D.

Comparing these 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 advantages 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.

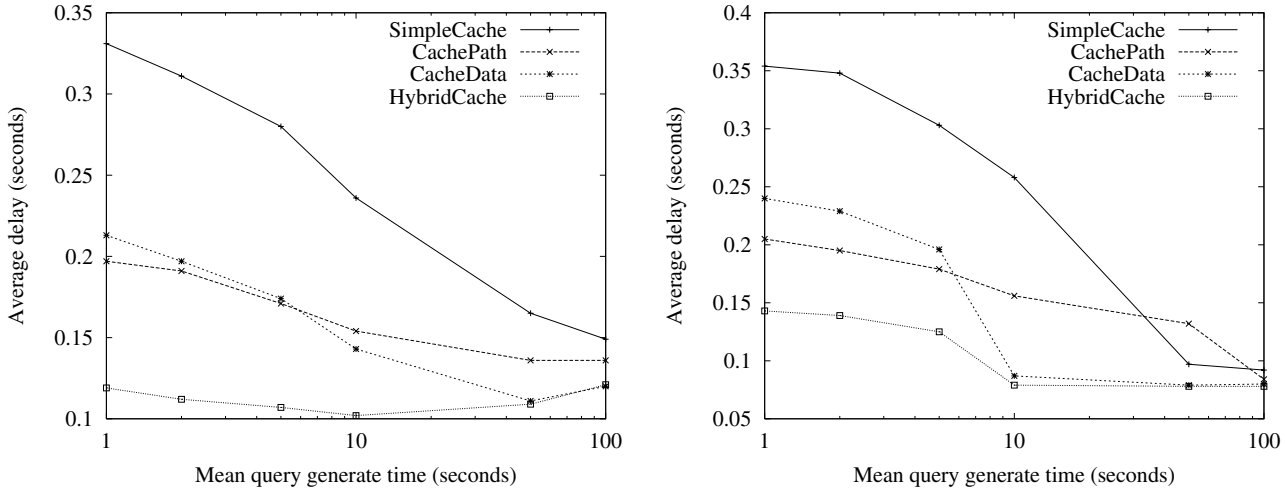
4) Effects of the Query Generate Time: Fig. 6 shows the average query delay as a function of the T_{query}. Both low mobility ($V_{max} = 2$ m/s) and high mobility ($V_{max} = 20$ m/s) settings are studied. We notice that all the trends are similar except CachePath. There are cases that CachePath even 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 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 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. 6 verifies this trend.

Under heavy system workload (T_{query} is small), HybridCache can reduce the query delay by as much as 40% 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 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.

5) Effects of TTL: Fig. 7 shows the average query delay when the TTL varies from 200 seconds to 10000 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



(a) $V_{max} = 2$ m/s (b) $V_{max} = 20$ m/s

Fig. 6. The average query delay as a function of the mean query generate time T_{query}

faster as TTL increases because nodes cooperate with each other to maximize the benefit of low update rate.

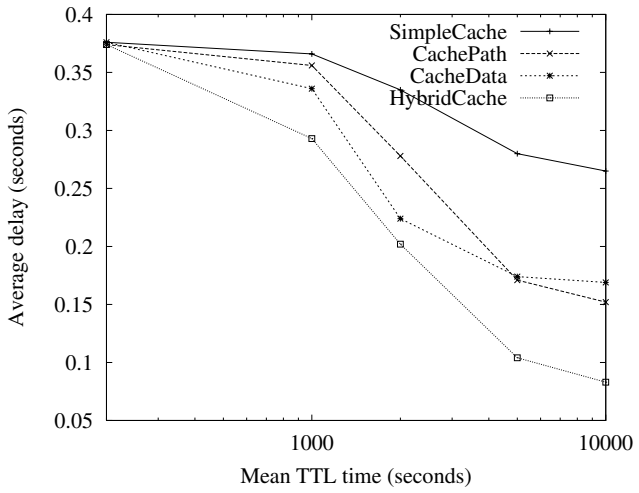


Fig. 7. The average query delay as a function of TTL

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%.

6) Effects of the Node Density: Fig. 8 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.

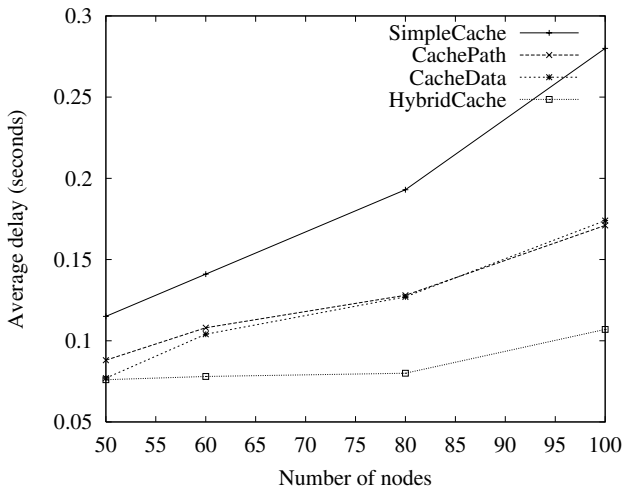
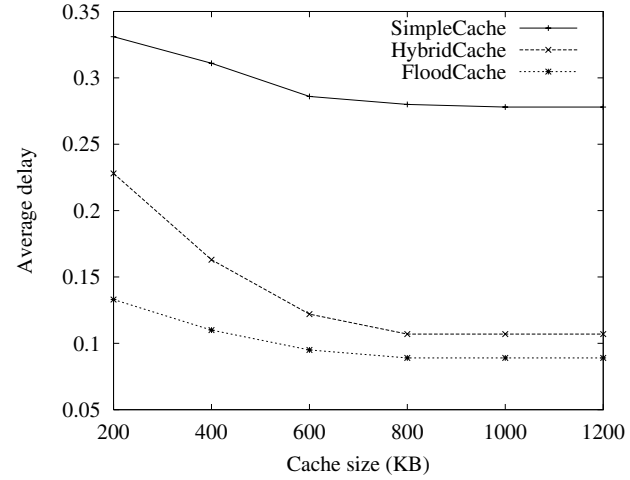


Fig. 8. The average query delay under different node density

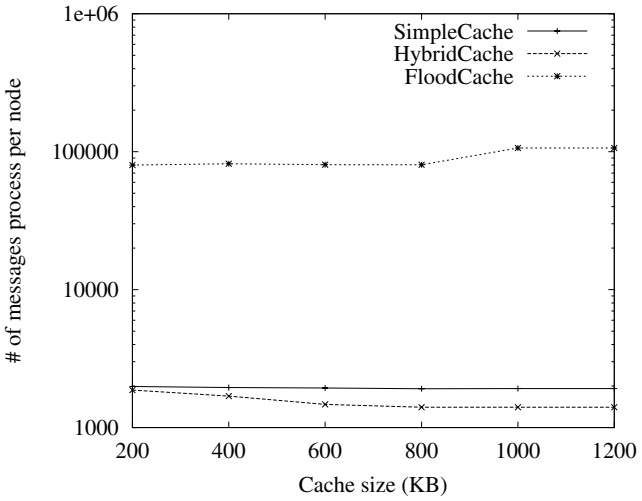
C. Simulation Results: Comparisons

In this subsection, we compare the performance of the HybridCache scheme to the SimpleCache scheme and the FloodCache scheme in terms of query delay and 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 message is processed (received and then re-broadcasted 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.

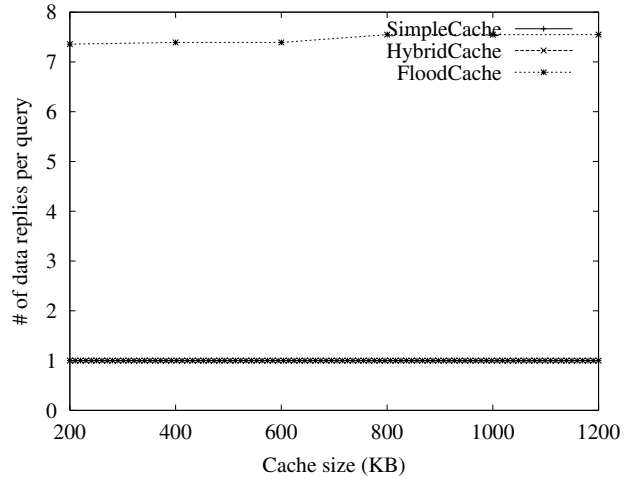
Fig. 9 shows the system performances when the cache size



(a) Query delay



(b) Message overhead



(c) # of reply messages per query

Fig. 9. The performance as a function of the cache size

varies. Fig. 9 (a) 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.

Comparing HybridCache and FloodCache, we can see that HybridCache does not perform as well as FloodCache in terms of query delay. However, Fig. 9 (b) shows that HybridCache incurs much less 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 less number of hops and mobile nodes need to process less 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 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 can not happen because only one request is sent out for each query in case of local cache miss. Fig. 9 (c) shows that more than 7 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 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.

V. 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. 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.

REFERENCES

[1] S. Das, C. Perkins, and E. Royer, "Performance comparison of two on-demand routing protocols for ad hoc networks," *IEEE INFOCOM*, pp. 3–12, 2000.

[2] D. Johnson and D. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Network," *Mobile Computing*, pp. 153–181, 1996.

[3] C. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers," *ACM SIGCOMM*, pp. 234–244, 1994.

[4] D. Wessels and K. Claffy, "ICP and the Squid Web Cache," *IEEE Journal on Selected Areas in Communication*, pp. 345–357, 1998.

[5] A. Rousskov and D. Wessels, "Cache digests," *Computer Networks and ISDN Systems*, vol. 30, no. 22-23, pp. 2155–2168, 1998.

[6] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide area web cache sharing protocol," *ACM SIGCOMM*, pp. 254–265, 1998.

[7] M. Cieslak, D. Foster, G. Tiwana, and R. Wilson, "Web cache coordination protocol v2.0," *IETF Internet draft, 2000. http://www.ietf.org/internet-drafts/ draft-wilson-wrec-wccp-v2-00.txt*.

[8] 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.

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

[10] G. Cao, "A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, September/October 2003 (A preliminary version appeared in ACM MobiCom'00).

[11] T. Hara, "Effective Replica Allocation in Ad Hoc Networks for Improving Data Accessibility," *IEEE INFOCOM*, 2001.

[12] M. Papadopouli and H. Schulzrinne, "Effects of power conservation, wireless coverage and cooperation on data dissemination among mobile devices," *ACM MobiHoc*, Oct. 2001.

[13] W. Lau, M. Kumar, and S. Venkatesh, "A Cooperative Cache Architecture in Supporting Caching Multimedia Objects in MANETs," *The Fifth International Workshop on Wireless Mobile Multimedia*, 2002.

[14] ns Notes and Documentation, "http://www.isi.edu/nsnam/ns/," 2002.

[15] Y. Xu, J. Heidemann, and D. Estrin, "Geography-informed Energy Conservation for Ad Hoc Routing," *ACM MobiCom*, pp. 70–84, July 2001.

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

[17] L. Yin, G. Cao, and Y. Cai, "A Generalized Target-driven Cache Replacement Policy for Mobile Environments," *The 2003 International Symposium on Applications and the Internet*, Jan. 2003.

[18] G. Zipf, "Human Behavior and the Principle of Least Effort," *Addison-Wesley*, 1949.

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