J. Parallel Distrib. Comput. 73 (2013) 595–607

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Improved group-based cooperative caching scheme for mobile ad hoc networks

I-Wei Ting, Yeim-Kuan Chang *

Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan

ARTICLE INFO

Article history:
Received 4 December 2010
Received in revised form
1 December 2012
Accepted 20 December 2012
Available online 3 January 2013

Keywords:
Mobile ad hoc networks
Cooperative caching
Multi-point relays

ABSTRACT

Data caching is a popular technique that improves data accessibility in wired or wireless networks. However, in mobile ad hoc networks, improvement in access latency and cache hit ratio may diminish because of the mobility and limited cache space of mobile hosts (MHs). In this paper, an improved cooperative caching scheme called group-based cooperative caching (GCC) is proposed to generalize and enhance the performance of most group-based caching schemes. GCC allows MHs and their neighbors to form a group, and exchange a bitmap data directory periodically used for proposed algorithms, such as the process of data discovery, and cache placement and replacement. The goal is to reduce the access latency of data requests and efficiently use available caching space among MH groups. Two optimization techniques are also developed for GCC to reduce computation and communication overheads. The first technique compresses the directories using an aggregate bitmap. The second employs multi-point relays to develop a forwarding node selection scheme to reduce the number of broadcast messages inside the group. Our simulation results show that the optimized GCC yields better results than existing cooperative caching schemes in terms of cache hit ratio, access latency, and average hop count.

1. Introduction

Mobile ad hoc networks (MANETs) [15] comprise various mobile devices called mobile hosts (MHs), such as notebooks, PDAs, and cell phones. These MHs form a wireless data communication network without the aid of any network infrastructure, such as access points or base stations. Each MH can communicate with its one-hop neighbors directly by broadcast messages, in which the one-hop neighbors of an MH are within the transmission range of its broadcast channel. Each MH can also freely move to any location [5] and communicate with another MH using multi-hop wireless links at any given time.

In the last decade, studies in the area of wireless and mobile networks have mainly focused on designing routing protocols [4] for packet forwarding. These protocols create routing paths usually composed of multiple intermediate nodes between two communicating nodes. In addition, caching techniques are developed to improve data accessibility. Caching is used to store recently accessed data based on the properties of temporal and spatial data locality, or commonly interesting data that may be accessed by numerous hosts. Caching has been successfully applied in the design of CPUs, multi-processors, and routers. It has also been employed in Internet-based technologies, such as the cache design of the World Wide Web (WWW) [1], proxy caches [24], Internet Cache Protocol [26] for proxy servers, cache digests [22] and summary caches [12], and various distributed systems, with the purpose of reducing data access delays. Furthermore, several caching nodes can also work together to form a cooperative caching environment, further improving the overall caching performance.

Designing an efficient cooperative caching scheme that considers essential factors such as mobility, battery power, and limited wireless bandwidth is a challenge for MANETs, for the following reasons. First, the caches of some MHs may become unavailable when they are shut down because of power shortage, or switched to sleep mode to save power. The caches of MHs that are turned off must be cleared because they are unaware of possible data updates, even once they rejoin the network. Second, the cache space of an MH is usually much smaller than that of the total data set in the network. A better cooperative caching scheme is required to efficiently use the caches of all MHs. Third, if the cache of an MH is shared by a large number of other MHs, the cache space may become full and the battery power of this MH may be quickly drained. Existing proposed cooperative caching protocols consider different aspects including the deployment of query directory nodes [2] and application managers [16], cache placement and replacement [6,7,9,17,23,27,28], replication of data objects [10,13,14], redirection of

* Corresponding author.
E-mail addresses: p7893113@mail.ncku.edu.tw (I.-W. Ting), ykchang@mail.ncku.edu.tw (Y.-K. Chang).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.12.013

data requests [8,7,19,27,28], and utilization of the caching space of neighboring MHs [9].

In this paper, the proposed group-based cooperative caching scheme (GCC) is based on the concept of group caching as in the many existing caching algorithms stated earlier. GCC allows each MH and its *k*-hop neighbors to form a group, instead of only 1-hop neighbors forming a group, as described in previous work [25]. A directory of cached data items is maintained in each node. Each MH obtains the directories of its *k*-hop group members through broadcasts in the group. As a result, each MH knows whether the requested data object is cached in its group. The requesting MH can avoid the search overhead of global *flooding* when one of the group members caches the requested data.

Cache placement and replacement are also important issues that affect the cache hit ratio in cooperative caches. Numerous studies on wired or wireless networks consider different parameters such as data object size, access frequency, and latency. However, most of these do not consider caching data status and node mobility when performing cache replacement. In ad hoc networks, if MHs connect to the networks, their caches cannot be used immediately, and the overall caching utilization decreases. Therefore, GCC employs efficient cache placement and replacement policies to consider the available caching space and node mobility for improving data accessibility in a group. The cache hit ratio in the group can be increased and the average search latency can be reduced significantly. GCC is also optimized by a hierarchical bitmap scheme and an efficient forwarding node selection scheme based on multi-point relay (MPR) [21] to reduce the computation and communication overheads. The aggregate bitmap can reduce the memory requirement for the stored directories, and the forwarding node selection scheme can reduce the number of broadcasts. In summary, the contributions of this paper are as follows.

1. The development of a group-based cooperative caching scheme generalizes the existing zone or group-based caching schemes that fully exploit broadcasts in the group to effectively enhance data accessibility in MANETs.
2. Data placement and replacement algorithms, including mobility and timestamp factors, are proposed to efficiently utilize the caches and improve data accessibility in the group.
3. Two optimizations are proposed to reduce the computation and communication overheads.
4. Prevalent existing cooperative caching schemes are evaluated and compared.

The rest of the paper is organized as follows. Section 2 presents a review of related work in MANETs. The proposed GCC scheme is explained in Section 3. Section 4 presents the optimized GCC scheme using the proposed hierarchical bitmap and forwarding node selection schemes. Section 5 shows the simulation results of GCC, compared with those of existing cooperative caching schemes. The conclusion is presented in Section 6.

2. Related work

The original data request-and-response model in MANETs is similar to the traditional client–server model in wired networks. Because nodes in MANETs freely move, the routing protocol usually broadcasts the route discovery messages (i.e., message flooding) to find the routing path from the source node to the destination node (or data source). The routing path is easily broken when the intermediate nodes move out of the transmission range of upstream or downstream nodes on the routing path. Thus, the source node or an intermediate node needs to rebroadcast the route discovery messages to reconstruct the routing path. The route discovery messages are received and processed by all nodes; therefore, the communication overhead is significantly high [18]. The average hop count and latency of data accesses are affected mainly by the node transmission range, node movement speed, and node density of the networks. If the transmission range is high, the hop count to the data source is smaller but the energy consumption is higher. In an environment containing no cache, the routing protocol plays a major efficient data communication role in MANETs. Several caching schemes that are closely related to the proposed GCC are reviewed as follows.

The SimpleCache scheme is similar to the client cache of the traditional client–server model in wired networks. When a data request is generated, the source node first searches its local cache to check if the requested data is cached (cache hit) or not (cache miss). If there is a cache hit, the requested data can be served locally. Otherwise, the data request is sent to the data source and waits for a response. SimpleCache does not consider node mobility, battery power, and limited wireless bandwidth that differentiate the MANET from the wired network. The nodes in SimpleCache do not use the cache space available in their neighbors or any other nodes.

The CacheData [27,28] scheme caches the requested data at intermediate nodes on the routing path between the source node and the data source. Any intermediate node caches popular passing data and is responsible for serving the request. When an intermediate node receives the data request, the local cache is looked up. If there is a cache hit, the cached data in the intermediate node is replied to the source node. CacheData is similar to the proxy servers sitting between the client and original server of the traditional WWW client–server model on the Internet. Because the intermediate nodes are closer to the source node than the data source is, access delay is reduced. In the CachePath scheme, the intermediate nodes are allowed to record only the distance (hop count) between the data source and source node provided by the routing protocol. Thus, when the data request is received in the intermediate nodes, the nodes can redirect the data request to the caching node if they find that the distance to the caching node is less than the distance of the data source from the source node.

In the NeighborCaching (NC) scheme [9], nodes use the caches of their 1-hop neighbors to store the data evicted by the cache replacement algorithm. The *ranking threshold* of a node is defined as the time value of the *k*th recently used data in the node, where *k* is a predefined number; all the nodes possess identical *k* numbers. When a source node evicts data to accommodate newly requested data, it attempts to select one of its 1-hop neighbors to store the evicted data as follows. The source node discards the evicted data if the ranking thresholds of all its 1-hop neighbors are not earlier than its ranking threshold. Otherwise, the neighbor that has the earliest ranking threshold is selected to store the evicted data. Then, the source node has to negotiate with the selected node to make sure its ranking threshold is not earlier than the selected node because the ranking information maintained in the source node may be old. If the ranking threshold of the selected node is not earlier than that of the source node, the 1-hop neighbor with the second earliest ranking threshold is selected and the same negotiation process is performed. The source node records the selected neighbor that stores the data evicted from its cache. Therefore, if the source node later requires the data that is cached in its neighbors, it can directly obtain the data from the neighbor caching the data.

The ZoneCooperative (ZC) cache scheme [7] is similar to CacheData in that the caches of the intermediate nodes on the routing path are also searched to find the requested data. However, the difference between ZC and CacheData lies in the manner by which the caches of the 1-hop neighbors of the intermediate nodes are searched. A cost function based on data popularity, distance between the requesting node and the caching node, time-to-live (TTL) values, and data size is developed for cache replacement. The evicted data by the replacement algorithm in one node cannot be stored in its 1-hop neighbors for reuse.

The Cooperative and Adaptive Caching System (COACS) [2] is similar to CachePath in that the path information of data in the caching nodes is cached in a predetermined number of nodes called *query directory* (QD) servers. QDs are the nodes that have the highest *scores* that summarize their resource capabilities, such as expected lifetime, battery life, available bandwidth, and available memory. The complete list of QDs is broadcast to all the nodes in the network. When a node requires a specific data, the nearest QD is first queried to check if it caches the path of the caching node that caches the requested data. If the nearest QD does not have a matching path, then the second nearest QD is queried until a matching path is found, or until all the QDs are exhausted. If a QD has the matching path of the caching node, it forwards the request to the caching node which, in turn, returns the requested data to the requester. However, if no QD contains the matching path, the requester is acknowledged and then the data is obtained from the data source. When receiving the data, the requester will notify the nearest query directory to update the path of the caching node for future queries.

The Data Pull/Index Push (DPIP) scheme [8] is similar to the ZC scheme, but it only allows the caching nodes to broadcast the index of the cached data in the zone (1-hop neighbors). When a node on the routing path receives the index packet, it knows which data is cached, as well as which caching nodes the data originated from. If a node finds that the requested data is in the zone (a zone hit) by examining the recorded index, it broadcasts the data request in the zone. Otherwise, data pull is performed by sending the data pull packets in the zone. The nodes receiving the data pull packets also check whether their 1-hop neighbors have the requested data. If the requested data is found in a node, the data request is redirected to that node. If no response is replied, the data request is sent to the next intermediate node toward the data source. In general, DPIP performs data recovery in the zone containing the 1-hop and 2-hop neighbors of the requesting node.

Table 1 summarizes the differences of all schemes. The third column shows the nodes that are looked up for requested data, excluding the source and destination nodes. The fourth column shows the actions taken by the nodes receiving the requests. In SimpleCache and NC, intermediate nodes on the routing path do nothing but forward the requests to the destination. In CacheData, CachePath, ZC, and the proposed GCC, only the intermediate nodes on the routing path receive and process the requests. In COACS, query directories redirect the requests to the caching nodes if a matching path is found and the caching nodes return the requested data to the requester. In DPIP, the request can be processed by 1-hop and 2-hop neighbors of the requesting node by sending the data pull packets. The last column shows the nodes that perform cache placement and replacement algorithms.

3. The proposed group-based cooperative caching scheme

3.1. Assumption and definition

Each MH has a unique ID which may be its unique host ID or IP address. "*Hello*" messages sent periodically as *Keep-Alive* signals are used to maintain the connectivity of the network. With "*Hello*" messages, each MH knows which nodes are its neighbors or k-hop neighbors. The one-hop neighbors of an MH are the nodes within its transmission range. MHs can obtain the k-hop neighbor information by "piggybacking" onto the $(k-1)$-hop neighbor information in "*Hello*" messages.

A *k-group* of a node x and k-group(x) are the set of nodes that are at most k hops away from node x. In the GCC scheme, each MH and its k-hop neighbors form a group in which k can be set empirically.

```
// The request for data $d_v$ is initiated from the source node (i.e., $s$).
// $MH_1,\ldots, MH_n$ are the nodes on the routing path from $s$ to destination node $MH_n$ (home of $d_v$).
// $MH_1,\ldots, MH_n$ may receive request($d_v$).
// $MH_j$ is the current node which receives the request from its upstream neighbor $MH_{j-1}$.
Request(Data $d_v$)
{
01 if ($MH_j$ caches $d_v$ or $MH_j$ is the home of $d_v$)
02     send response($d_v$) message to $MH_{j-1}$ and exit;
03     else if (the $v^{th}$ bit of group_bitmap is set) {
04     broadcast request_in_group messages to $MH_j$'s group members based on MPR protocol;
05     wait for response_in_group message from one of $MH_j$'s group members before time-out;
06     if ($d_v$ is received before time-out)
07         send response($d_v$) message to $MH_{j-1}$ and exit;
08 }
09 if ($MH_j$ is the source node) Run the routing protocol to create a routing path;
10 forward request($d_v$) to $MH_{j+1}$;
}
```

Fig. 1. Data discovery process.

In general, when the group size k is large, the communication overhead is high. However, a high remote cache hit ratio can be obtained with a large k because MHs can recognize the caching information of a large number of neighbors in the group.

3.2. Maintaining a directory using a bitmap

Two bitmaps, *self_bitmap* and *group_bitmap,* are maintained for each MH. The length of the bitmap is equal to the number of data objects in the network. Thus, bit i of the bitmap is mapped to the*i*th data object. When an MH caches a data object, the corresponding bit of its *self_bitmap* is set. Each MH periodically broadcasts its *self_bitmap* along with the information about its available caching space and the oldest timestamp of its cached data objects to its group members. The available caching space and the oldest timestamp are needed in cache placement and replacement algorithms, described later. When an MH receives *self_bitmaps* from all its group members, it will merge them into *group_bitmap*. By examining the *group_bitmap* corresponding to the requested data, a requesting node immediately knows if the requested data is cached in the group. Then, data discovery can be accomplished using broadcasting inside the group.

Although GCC adopts a "*stateful*" mechanism, only bitmap information is recorded and transmitted. Exchanging bitmaps in a group only consumes a small amount of communication bandwidth. Generally, the size of bitmap information is significantly smaller than the size of the data object. Thus, the control overhead is insignificant. The main benefit is being able to save memory and communication bandwidth.

3.3. Data discovery process

When receiving a request, the node first searches its cache for the requested data. If the requested data is not found locally, a special request called *request_in_group* is broadcast inside the group of the node. The MPR [21] protocol is used to broadcast *request_in_group* messages in the group. If the requested data is still not found, the request is forwarded to the next node on the routing path to continue the data discovery process. Fig. 1 shows the detailed algorithm for processing a request for data d_v in the intermediate node MH_j. Lines 1 and 2 deal with the simple case in which MH_j is either the home of the requested data or has a copy of the data in its cache. Lines 3–8 perform the data discovery process inside the group of MH_j if the vth bit of *group_bitmap* is set, that is, the requested data is cached in one of the $\neq$group members of MH_j. If MH_j receives the response containing d_v before time-out, it sends the data back to MH_{j-1}. The response is eventually sent back along the reversed routing path to the requester. If no response containing d_v is returned before time-out, the routing path must be created first if the current node is the source node and the request is then forwarded to the next node on the routing path to continue the data discovery process.

Fig. 2 shows the algorithm for a node MH_i in the group to process the *request_in_group* message sent from MH_{i-1}. The group

Table 1
Comparisons of various cooperative caching schemes.

Scheme	Feature				
	Local cache hit	Nodes that are searched in data discovery process excluding source and destination nodes	Actions taken by the nodes receiving a data request	Remote cache hit	Nodes performing cache placement and replacement
NoCache	No	None	None	No	No
Simple Cache	Yes	None	None	No	Source node
CacheData	Yes	Intermediate nodes	Reply if cache hit in the intermediate nodes	Yes	Source node and intermediate nodes
CachePath	Yes	Intermediate nodes and other nearby caching nodes indicated by the path information	Reply if cache hit in the intermediate node or redirect the request to the caching node	Yes	Source node and intermediate nodes, but only the path information is cached
NC	Yes	The 1-hop neighbors of source node	None	Yes	Source node and its 1-hop neighbors
ZC	Yes	Zone (1-hop) neighbors of source node and intermediate nodes	Reply if zone cache hit in the 1-hop neighbors of source node and intermediate nodes	Yes	Source node
COACS	Yes	Caching nodes indexed by the path information stored in the query directory nodes	Forward the request to caching node if hit in current or next query directory node	Yes	Source node
DPIP	Yes	Zone (1-hop and 2-hop) neighbors of source and intermediate nodes	Reply if zone cache hit in the (1 or 2-hop) neighbors of source and intermediate nodes	Yes	Source node and intermediate nodes
The proposed GCC	Yes	k-group members of the source node and intermediate nodes	Reply if k-group cache hit or forward the request to next intermediate node	Yes	k-group members of the source node and intermediate nodes on the routing path

```
//broadcast request_in_group messages inside the group of $MH_i$ based on MPR protocol
// $MH_i$ is the current node which receives request_in_group from $MH_{i-1}$.
Request_in_group(Data $d_v$)
{
01 if (the same request_in_group($d_v$) is not received before) {
02     if ($MH_i$ caches $d_v$ or $MH_i$ is the home of $d_v$)
03         send response_in_group($d_v$) to $MH_{i-1}$ and exit;
04     decrement ttl-hop-count by 1
05     if (ttl-hop-count $\neq$ 0 and $MH_i$ is the forwarding node based on MPR) {
06         broadcast request_in_group messages to $MH_i$'s 1-hop neighbors
07         wait for response_in_group($d_v$) message from one of $MH_i$'s 1-hop neighbors;
08         if (response_in_group($d_v$) is received before time-out)
09             send response_in_group($d_i$) to $MH_{i-1}$ and exit;
10     }
11 }
}
```

Fig. 2. Data discovery process in the group.

of a node is assumed to be a k-group that consists all i-hop neighbors for $i \leq k$. Thus, to confine the *request_in_group* messages to the group, the *time-to-live hop count* (*ttl-hop-count*) is set to k. Line 1 checks if the same message has not been received before. If MH_i has cached the requested data d_v, or MH_i is the home of d_v, it immediately sends d_v to MH_{i-1}. Otherwise, *ttl-hop-count* is decremented by one. If *ttl-hop-count* is not zero and MH_i is a forwarding node based on MPR, MH_i forwards the *request_in_group* message to the 1-hop neighbors of MH_i and waits for response. If the response containing data d_v comes back before time-out, it is returned back to MH_{i-1}.

Fig. 3 shows an example of node S requesting the data originally owned by node MH_n based on the 2-group. Node s first searches 2-group(S). If the requested data is not found, the routing path between the source node S and destination node MH_n is created. Assuming that the intermediate nodes on the routing path are $MH_1, \ldots, MH_{n--1}$, the request is then passed to MH_1 which searches its 2-group(MH_1) for the data. This data recovery process is repeated on the intermediate nodes along the routing path until the requested data is found or the request reaches the destination node MH_n.

3.4. Cache placement and replacement algorithm

The detailed algorithm in Fig. 4 shows how and where to place the data contained in the response received by the intermediate node (say MH_j). In line 1, MH_j starts to search its*self_bitmap* and *group_bitmap* to check if the received data object d_v has been

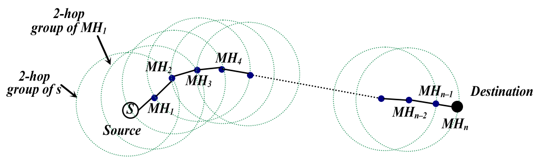


Fig. 3. Data discovery process initiated from source node s for the data owned by destination node MH_n using the proposed GCC with 2-hop groups.

```
// The response($d_v$) is executed by $MH_j$ after obtaining data $d_v$ from its k-group members,
// where $MH_j$ is an intermediate node on the routing path from source node to data source.
Response(Data $d_v$)
{
01 if (the $v^{th}$ bit of self_bitmap and group_bitmap of $MH_j$ is not set) {
02     if ($MH_j$ has enough space for $d_v$)
03         $MH_j$ performs cache_placement($d_v$) to cache $d_v$ and updates self_bitmap;
04     else {
05         $MH_j$ performs cache_replacement_LRU($d_v$) to cache $d_v$ and update self_bitmap;
         //Assuming $d_r$ is replaced data and $t_r$ is its timestamp;
06         if (any of $MH_j$'s k-group members has enough space for $d_r$) {
07             $MH_j$ selects the stable group member, say $MH_p$;
08             $MH_j$ sends caching_data($d_r$) message to $MH_p$;
09         } else
10             if ($t_r$ is the oldest timestamp in the group) remove $d_r$;
11             else {
12                 $MH_j$ stores $d_r$ to the nearby and stable group member which owns the
                       cached data with the oldest timestamp in the group, say $MH_s$;
13                 $MH_j$ sends caching_data($d_r$) message to $MH_s$; }
14     }
15 }
16 Send response($d_v$) to $MH_{j-1}$;
}
Receive_caching_data($d_r$) // MH receives the cached data, $d_r$
{
01 if (the $v^{th}$ bit of self _bitmap is not set)
02     if (MH has enough space for $d_r$)
03         cache_placement($d_r$) and update self_bitmap.
04     else
05         cache_replacement_LRU($d_r$) and update self_bitmap.
}
```

Fig. 4. Placement and replacement after receiving the response.

cached in the group. If yes, d_v will not be cached in MH_j to reduce the degree of cached data redundancy in the group. Otherwise, if the free cache space is large enough to hold d_v, MH_j caches d_v. If the free cache space is not large enough to hold d_v as in line 5, MH_j performs the well-known least recently used (LRU) replacement to cache d_v and removes the least recently used data first, say d_r. Then, in lines 7–9, d_r is stored in the most "*stable*" group member (defined later) if the available cache space of the group member is sufficient to store d_r. The objective is to allow MH_j to quickly retrieve the replaced data later.

For node MH_j, we say that a node MH_m is more stable than another node MH_n when MH_j receives more "*Hello*" messages from MH_m than from MH_n in a fixed period of time. For example, MH_j receives three, seven, and six "*Hello*" messages from MH_k, MH_m, and MH_n in a period, respectively. Therefore, MH_m is more stable for MH_j than MH_k and MH_n. Higher stability between two nodes indicates that the connection between them is not expected to break down. Selecting stable nodes for storing replaced data can minimize the negative impacts of node mobility and disconnection and thus improve data accessibility. If two or more nodes become the most stable group members, we break the tie by selecting the closer nodes. Also, if two or more nodes are the most stable and closest group members, the node that has the largest available caching space is selected.

A simple mechanism to determine the degree of node stability is explained as follows. Each MH uses counters for each group member. When the MH receives and recognizes a "*Hello*" message from one of its group members, the corresponding counter is increased by one. MHs may move in or out of the transmission range area of the group members. Therefore, MHs receive different numbers of "*Hello*" messages from their group members in a period. Thus, when an MH needs the stable group member to cache the replaced data, it selects the member with the largest counter.

If MH_j and all its group members do not have sufficient space to cache the received data as in lines 11–14, LRU is performed by MH_j. The timestamp t_r retrieved from the replaced data is compared with the recorded timestamp piggybacked in the "*Hello*" messages from the stable group members. If t_r is the oldest, d_r is removed. Otherwise, d_r is stored in the stable group member that owns the oldest timestamp. When the node receives d_r from MH_j, it repeatedly removes the oldest cached data to increase the available cache space until the received data object can be cached. If no group member is considered to be stable, d_r is removed directly.

Generally, the local hit ratio increases as the cache size increases. However, in GCC, the replaced data is first moved to the group members that have the available caching space. Thus, when a node joins the network, its caching space is used by other nodes. This method improves the local hit ratio of the newly joining nodes.

3.5. Mobility effect

GCC enables the requesting node to answer "*the requested data is cached or not cached in the group members*", in which the mobility is considered in a seamless manner. Such information of group members minimizes the negative impacts of node mobility. If the group member that has the requested data moves away from the group, the information may still be valid, because it is possible that any one of the group members may have another copy of the requested data. In addition, the newly proposed cache replacement algorithm also considers the mobility of group members.

A group search in a node is considered a hit if one of the group members of the node caches and replies to the data requested. If the source node or any intermediate node encounters a local cache miss and determines that the bit in *group_bitmap* corresponding to the requested data is set, it starts a search in its group. However, because nodes may move away from a group, or become disconnected from the network after shutdown, the node that initiates the group search may not be able to obtain the requested data from its group members. Three factors namely node speed, group size, and the *self_bitmap* update interval affect the group search hit ratio. When the node mobility is high, some group members may move out of the group before the *group_bitmap* is updated. When the group size is large, the node has a high probability of acquiring the response from its group members. When the bitmap update interval is short, the *group_bitmap* maintained in each node will almost always be valid. However, the communication overhead will be increased due to frequent broadcasts of *self_bitmap*. The effect of node mobility on the group hit ratio is evaluated in Section 5.

3.6. Cache consistency issue

When the data objects are updated in the data sources, the cached copies in the caching nodes may become inconsistent. Therefore, *weak consistency* and *strong consistency* are two models that are usually applied to maintain data consistency. Under the weak consistency model, a cached copy of a data object is associated with the time-to-live (TTL) attribute. The cached data object is declared invalid by the caching nodes if its TTL expires. Weak consistency is simple and does not cause much communication overhead for maintaining data consistency. Under strong consistency, the caching node must check the consistency of the requested data with the data source before a cached copy can be returned to the requesting nodes. The main drawback of strong consistency is that the query message needs to be broadcast to find the data source, and, as a result, a heavy communication overhead occurs. In GCC, the weak consistency model is used because it consumes low energy and wireless bandwidth.

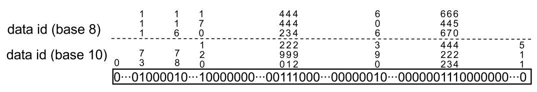
4. Optimized GCC

In this section, an aggregate bitmap scheme and the forwarding node selection are proposed to speed up the search in the directory and reduce the memory usage and communication overhead needed in GCC.

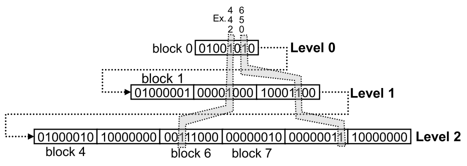
4.1. Aggregate bitmap

In the network containing N_D data objects, each node maintains an N_D-bit *self_bitmap* to record which data objects are cached. The ith bit of *self_bitmap* is set to 1 if the node caches the ith data object for $i = 0 \ldots N_D - 1$. Because the number of data objects that can be cached in a node is much smaller than N_D, *self_bitmap* will be a sparse bitmap. In other words, only a small number of bits in *self_bitmap* are set to 1 and the other bits are set to 0. To reduce the memory usage of *self_bitmap* and thus reduce the bandwidth usage for transmitting it over the network, a scheme called *aggregate bitmap* that aggregates a block of bits into one bit is proposed. In the aggregate bitmap with *aggregate size k*, k bits are collapsed into a single bit if at least one of these k bits is originally set to 1.

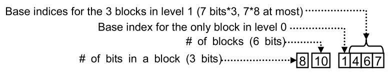
Fig. 5(a) shows an example of a 512-bit bitmap and the corresponding aggregate bitmap with the aggregate size of 8 bits. Assume that the ten set bits in the original bitmap are bits 73, 78, 120, 290, 291, 292, 390, 422, 423, and 424. First, all data with IDs 0–63 of the original bitmap are not set. Thus, the first bit of level 0 in aggregate bitmap is not set because 64 bits are aggregated in one bit. Second, several of data with IDs 64–127 are set (73, 78, and 120). Thus, the second bit of level 0 is set. As a result, each bit in level 0 could be set based on the original bitmap. Three blocks in level 1 are also induced from the number of set bits in level 0. The procedure of each bit setting in each block from the level 1 is performed as level 0 but only 8 bits are aggregated in one bit. For example in Fig. 5(b), the second bit of the first block in level 1 is set because any one of bits 64–71 are set in original bitmap. The blocks of level 2 are also induced from the bit set in each block from level 1. Finally, based on the number of blocks in each level, we are able to know which data is cached and reconstruct the original bitmap. The resulting aggregate bitmap consists of ten 8-bit blocks as shown in Fig. 5(b). Three additional values are required to complete the aggregate bitmap. The first value is the aggregate size. The second is the total number of blocks in the aggregate bitmap, which is 10 in the example of Fig. 5. The third value is the base index of the non-bottom-level blocks. Notice that the base index for the only block in level 0 is always 1. In Fig. 5(c), the base indices of the three blocks in level 1 are 4, 6, and 7. These additional values are stored into a data



(a) Original 512-bit bitmap.



(b) Array of 8-bit blocks in aggregate bitmap denoted as $B[0\ldots 9]$.


Base indices for the 3 blocks in level 1 (7 bits*3, 7*8 at most)
Base index for the only block in level 0
# of blocks (6 bits)
# of bits in a block (3 bits)
8 10 1 4 6 7

(c) Aggregate bitmap header including the block size, number of blocks, and the array of base indices pointing to a block in level $i + 1$ from a block in level i.

Fig. 5. Bitmap with ten set bits and corresponding aggregate 3-3-3 bitmap.

```
// $S = \{b_0^0 b_1^0 \cdots b_{d-1}^0, b_0^1 b_1^1 \cdots b_{d-1}^1, \cdots, b_0^{n-1} b_1^{n-1} \cdots b_{d-1}^{n-1}\}$, where the data IDs are in the increasing
   order and each data ID is represented as a $d$-digit value where each digit is a number of
   base $k$ and the IDs are assumed to be in the range of 0 to $k^d - 1$.
// Set $y_0$=1 and $y_i$=the number of distinct values in $\{b_0^0 \cdots b_{i-1}^0, b_0^1 \cdots b_{i-1}^1, \cdots, b_0^{n-1} \cdots b_{i-1}^{n-1}\}$.
// So, $\{p_0^0 \cdots p_{i-1}^0, \cdots, p_0^{y_i-1} \cdots p_{i-1}^{y_i-1}\}$ is $\{b_0^0 \cdots b_{i-1}^0, \cdots, b_0^{n-1} \cdots b_{i-1}^{n-1}\}$ after redundancy removal;
Construct_aggregate_bitmap(S)
01 $t = \sum_{i=0}^{d-2} y_i$ and Create an array of $t$ blocks, $B[0\ldots t-1]$;
02 cur_blk=0; num_of_blks=1;
03 B[cur_blk].ptr=num_of_blks; //set the index pointer of block 0 to 1;
04 enable bits $p_0^0$ to $p_0^{y_1-1}$ in B[cur_blk];
05 if ($d > 1$) num_of_blks += $y_1$;
06 for ($i$=1 to $d-1$) {
07     for ($f = p_0^0 \cdots p_{i-1}^0$ to $p_0^{y_i-1} \cdots p_{i-1}^{y_i-1}$) {
08         cur_blk++;
09         $\{fq_i^0, \cdots, fq_i^{z_i-1}\}$ is the subset of $\{p_0^0 \cdots p_i^0, \cdots, p_0^{y_{i+1}-1} \cdots p_i^{y_{i+1}-1}\}$ with the same prefix $f$;
10         $z_i$ is the size of $\{fq_i^0, \cdots, fq_i^{z_i-1}\}$;
11         enable bits $q_i^0$ to $q_i^{z_i-1}$ in B[cur_blk];
12         if ($i < d-1$) { B[cur_blk].ptr=num_of_blks; num_of_blks += $z_i$; }
13     }
14 }
```

Fig. 6. Constructing the aggregate bitmap.

structure called *header*. The data objects recorded in each node are different; thus, each node can select the best aggregate size to construct a smaller aggregate bitmap. The detailed algorithm for constructing an aggregate bitmap is shown in Fig. 6. The algorithm denotes the set $\left\{p_0^0 \cdots p_{i-1}^0, \ldots, p_0^{y_i-1} \cdots p_{i-1}^{y_i-1}\right\}$ of size y_i as the one after the redundant elements in $\left\{b_0^0 \cdots b_{i-1}^0, \ldots, b_0^{n-1} \cdots b_{i-1}^{n-1}\right\}$ are removed. For example, an MH computes $y_1 = 3$ and $y_1 = 6$, and the total number of 8-bit blocks is 10 in the example of aggregate size 8 in Fig. 5. Line 1 computes the total number of blocks needed (say, t) and allocates t blocks as $B[0\ldots t-1]$. In lines 2–3, the pointer of the current block 0 (i.e., $B[cur\_blk]$) is set to block 1 (*num_of_blks*). In line 4, bits p_0^0 to $p_0^{y_1-1}$ in B [1] are set to 1. In lines 6–14, the bit patterns of blocks in the non-bottom levels are set in a top-down fashion. There are two *for* loops. The outer *for* loop *repeats for* non-bottom levels. The inner *for* loop repeats for all the blocks in each level. There are y_i blocks in level i and thus, the inner loop repeats y_i times for each distinct element f in $\left\{p_0^0 \cdots p_{i-1}^0, \ldots, p_0^{y_i-1} \cdots p_{i-1}^{y_i-1}\right\}$. In lines 9–11, the subset $\left\{p_0^0 \cdots p_i^0, \ldots, p_0^{y_{i+1}-1} \cdots p_i^{y_{i+1}-1}\right\}$ with the same prefix f is denoted as $\left\{fq_i^0, \ldots, fq_i^{z_i-1}\right\}$ and the bits q_i^0 to $q_i^{z_i-1}$ in the current block ($B[cur\_blk]$) are enabled. Finally, if the current block is not in the bottom level, its pointer is set according to the size of $\left\{fq_i^0, \ldots, fq_i^{z_i-1}\right\}$.

Fig. 7 shows the search process for determining if data object *id* is recorded in the aggregate bitmap of an MH. To simplify the presentation, the algorithm assumes that the size of the original bitmap is $L = k^d$ and the aggregate size is k. The function *Aggregate_bitmap_search*(B, id) is given in Fig. 7, in which the data *id* is represented as a string of d base-k numbers, that is, $b_0 b_1 \ldots b_{d-1}$.

```
// $k^d$ is the size of the original bitmap in bits.
// $id = b_0 b_1 \ldots b_{d-1}$ is represented as a number of base $k$.
// $B[0\ldots s-1]$ is a linear array of $s$ $k$-bit blocks.
Aggregate_bitmap_search(B, id)
{
01    m=0;
02    for ($i$=0 to $d-1$) {
03        if (bit $b_i$ of block B[m]=0) return False;
04        if ($i$=$d-1$) return True;
05        B0=B[m]; //save B[m] to temporary block B0
06        Set the bits $b_i \ldots 0$ in block B0 to zeros;
07        Compute $f$=the number of set bits in B0;
08        m=B[m].ptr + $f$;
09    }
```

Fig. 7. Searching for a data object in the aggregate bitmap.

```
// Merge_aggregate_bitmaps(Aggregate_Bitmap $AB_1$, Aggregate_Bitmap $AB_2$)
01 Extract data object IDs $S1 = \{i_0^0 i_1^0 \cdots i_{L-1}^0, i_0^1 i_1^1 \cdots i_{L-1}^1, \cdots, i_0^{n_1-1} i_1^{n_1-1} \cdots i_{L-1}^{n_1-1}\}$ from $AB_1$;
02 Extract data object IDs $S2 = \{j_0^0 j_1^0 \cdots j_{L-1}^0, j_0^1 j_1^1 \cdots j_{L-1}^1, \cdots, j_0^{n_2-1} j_1^{n_2-1} \cdots j_{L-1}^{n_2-1}\}$ from $AB_2$;
03 Merge sets S1 and S2 into $S = \{b_0^0 b_1^0 \cdots b_{L-1}^0, b_0^1 b_1^1 \cdots b_{L-1}^1, \cdots, b_0^{n-1} b_1^{n-1} \cdots b_{L-1}^{n-1}\}$;
04 Call Construct_aggregate_bitmap(S)
```

Fig. 8. Procedure for merging two aggregate bitmaps.

```
// This algorithm computes $MPR_j(z)$ that is the set of forwarding nodes $j$ hops from $z$ which is the
// source node $S$ or an intermediate node $MH_i$ on the routing path between S and data source.
Algorithm_Forwarding_Node_Selection(z, G(z))
{
01 for ($j = k-1$ to 1) {
02    $MPR_j(z)$ is initialized to empty;
03    if ($j = k-1$) TSet is initialized to $G(z)$;
04    else TSet is initialized to $MPR_{j+1}(MH_j)$;
05    CSet is initialized to j-hop(S);
06    while (TSet is not empty) {
07        $x_{max} = y_1$; $T(x_{max}) = 0$;
08        for ($i$ = 1 to $n$) { // assume CSet consists of nodes $y_1$ to $y_n$, where $n$ = |CSet|
09            Compute the covered set of $y$, $T(y_i)$, consisting of the nodes in TSet covered by $y_i$;
10            if ($|T(y_i)| > |T(x_{max})|$) $x_{max} = y_i$;
11        }
12        Added $x_{max}$ in $MPR_j(z)$;
13        Removed $x_{max}$ from CSet;
14        Removed $T(x_{max})$ from TSet;
15    }
16 }
}
```

Fig. 9. Forwarding node selection scheme.

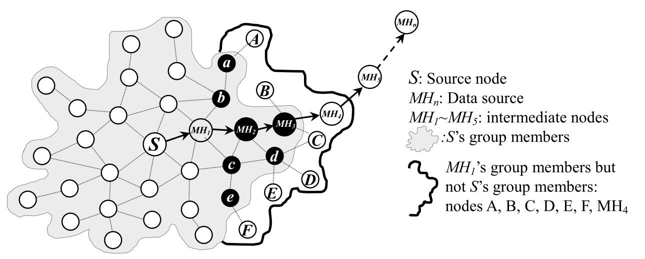

S
MH1
MH2
MH3
MH4
MHn
a
b
c
d
e
A
B
C
D
E
F
S: Source node
MHn: Data source
MH1~MH5: intermediate nodes
:S's group members
MH1's group members but not S's group members: nodes A, B, C, D, E, F, MH4

Fig. 10. Concept of forwarding node selection scheme in the group ($k = 3$).

The base-k number b_i is used to search the blocks in level i of the aggregate bitmap. Because each node receives all the *self_bitmaps*

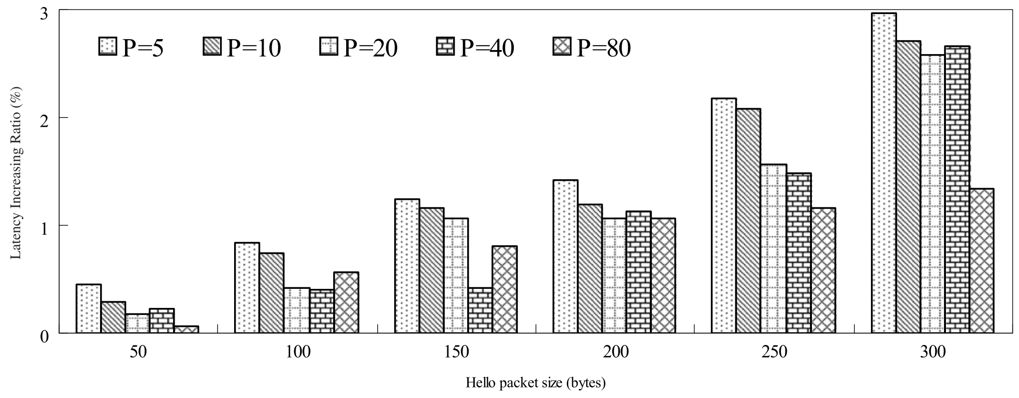


Fig. 11. Communication overhead in terms of latency increasing ratio.

from its neighboring nodes, the nodes merge the *self_bitmaps* into one to make the lookup operations efficient. The merge procedure is shown in Fig. 8. The merge operation of two aggregate bitmaps is done as follows. First, each aggregate bitmap is converted into the set of original data IDs in the format of a string of base-k numbers. The bases of the numbers in these two aggregate bitmaps can be different. Next, these two sets of original data IDs are merged. Finally, the procedure *Construct_aggregate_bitmap(S)* is called to create the merged aggregate bitmap.

Note that the aggregate bitmap is a deterministic scheme. Non-deterministic schemes, such as hashing or a bloom filter can also achieve the same goal. If the non-deterministic scheme is developed, the issues of selecting the hash function, collisions, and false positives need to be addressed.

4.2. Forwarding node selection scheme

In addition to the aggregate bitmap, the forwarding node selection scheme based on MPR [21] can also be used to reduce the overhead of the broadcast operations in the group. MPR avoids using all the nodes in the network to perform the broadcast operations. In other words, only a subset of nodes is selected as the *forwarding nodes* that are responsible for transmitting the broadcast messages to their neighbors. Non-forwarding nodes only passively receive the broadcast messages. By using the "*Hello*" messages, each node collects the 1-hop neighbor lists of all its 1-hop neighbors to form its 2-hop neighbors. MPR uses a greedy selection algorithm to select *the forwarding nodes* (or called *MPR nodes*) that cover all the 2-hop neighbors. Each node receives not only the broadcast message, but also the directive to determine if it will act as the forwarding or non-forwarding node. Every forwarding node also runs the selection algorithm to select some of its 1-hop neighbors to be the forwarding nodes for transmitting the broadcast messages in the network.

The original MPR is a source-dependent algorithm that selects a subset of the 1-hop neighbors to fully cover all the 2-hop neighbors. The broadcast messages are sent to all 2-hop neighbors by way of the selected 1-hop forwarding nodes. Now, consider the proposed forwarding node selection scheme in the data discovery phase. We assume that $k = 2$. When we perform the forwarding node selection algorithm, not all but only a small subset of 2-hop neighbors toward the destination node need to be covered by the selected forwarding nodes. Thus, the communication overhead can be reduced because fewer nodes act as the forwarding nodes. The proposed MPR-based forwarding node selection algorithm for the source node and the intermediate nodes shown in Fig. 9 is described as follows.

For source node S, the broadcast messages have to reach all the nodes in its k-group. Let $MPR_j(S)$ be the set of forwarding nodes that are j hops from source node S for $j = 1$ to $k-1$. The procedure for computing $MPR_j(S)$ for $j = k-1$ to 1 is shown in Fig. 9 by setting $z = S$ and $G(z) = k$-hop(S). Lines 2–5 perform the initialization for the target set (TSet), candidate set (CSet), and $MPR_j(S)$. Let CSet $= \{y_1, \dots, y_{|\text{CSet}|}\}$, where |CSet| is the size of CSet. In the while loop, we first compute the *covered set* of y_i denoted by $T(y_i)$ which includes the nodes *TSet* that are covered by y_i. The size of $T(y)$ is denoted by $|T(y)|$. Then, we select $x_{\max}$ in *CSet* such that $|T(x_{\max})| \geq |T(y_i)|$ for any other node y_i in *CSet*. For two nodes y_i and y_j in CSet, if $|T(y_i)| = |T(y_j)|$ and $i < j$, y_i is selected first. Finally, in lines 12–14, $x_{\max}$ is added in $MPR_j(S)$, $x_{\max}$ is removed from *CSet*, and the nodes in $T(x_{\max})$ are removed from *TSet*. After the above process, if *TSet* becomes empty, the process to compute $MPR_j(S)$ is done. Otherwise, the while loop continues.

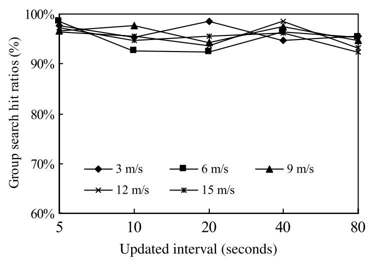
For any intermediate node MH_i on the routing path, MH_i only has to forward the broadcast messages to the nodes that are in the k-group of MH_i, but not in k-group of MH_{i-1}, which is denoted by $G(MH_i) = k\text{-hop}(MH_i) - k\text{-hop}(MH_{i-1})$. Since the i-hop members of MH_i for $i = 1$ to $k-1$ have been reached when MH_{i-1} performed group search, $G(MH_i)$ only includes some of its k-hop group members. Therefore, the procedure for computing $MPR_i(MH_i)$ is similar to that for source nodes except that the target set is $G(MH_i)$ instead of k-hop(MH_i). As a result, we execute the algorithm in Fig. 9 by setting $z = MH_i$ and $G(z) = G(MH_i)$. Consider the example in Fig. 10. MH_1 only needs to select the nodes that cover A, B, C, D, E, F, and MH_4, instead of all the 3-hop neighbors of MH_1. Thus, only nodes a, b, c, d, e, MH_2, and MH_3 are selected as the forwarding nodes.

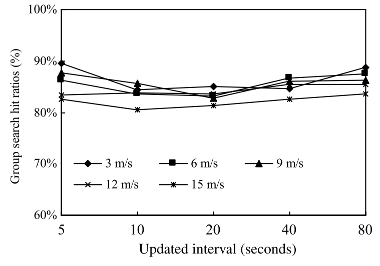
5. Performance evaluation

This section presents the simulation results of the proposed GCC scheme that is compared with SimpleCache, CachePath [27,28], CacheData [27,28], NC [9], ZC [7], COACS [2], and DPIP [8]. The simulation model and environment are first given as follows.

5.1. The simulation model and environment

The experiments are conducted on Network Simulator (NS2) [11] with the CMU wireless and mobility extension. IEEE 802.11 protocol is used as the basis in the MAC layer. The well-known ad hoc on-demand distance vector routing protocol [20] is used as the underlying routing algorithm. The network contains 100 nodes that are randomly distributed over the 1500 m × 1500 m area. The moving pattern of nodes follows the random way point mobility model. Initially, nodes are placed randomly in the network. Each node randomly selects a destination and moves toward it at a speed of 0–2 m/s, also selected randomly. After the node reaches the destination, it pauses for 300 s and repeats the moving pattern.


(a) GCC with $k = 1$.


(b) GCC with $k = 2$.

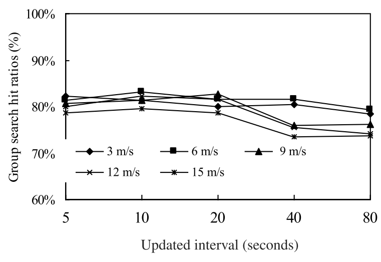

(c) GCC with $k = 3$.

Fig. 12. Group search hit ratios with different node speed and bitmap update interval.

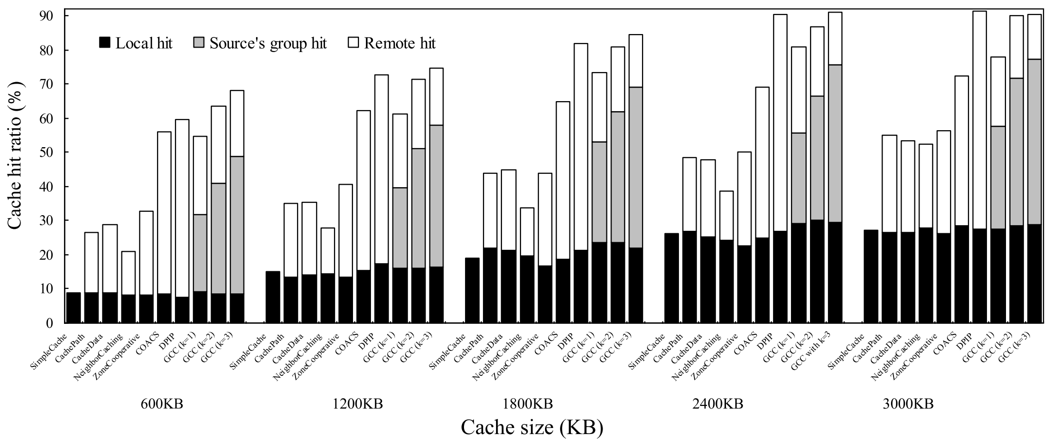


Fig. 13. Cache hit ratios under different cache sizes, 600–3000 KB.

Because of the dynamic property of MANETs, the behavior of nodes joining and leaving the network is simulated by changing the node *join/leave rates* in the range 0.05–0.4 nodes/s. The rate is defined as the number of nodes excluding data sources randomly joining or leaving the network every second. If an MH joins or leaves the network, its cache content is cleared. The detailed parameters are shown in Table 2. In the simulation environment, we assume that the data sources are powerful devices compared with MHs. Thus, the data source does not usually switch to wakeup/sleep mode for saving energy. Also, in the battlefield, an officer in a car carries the data source containing information about everything. The soldiers usually need to obtain information from the data source to help determining the next action, so the data sources can be mobile.

Five data sources (servers) are placed in the network. The data sources can also move arbitrarily in the network at a speed of 0–2 m/s. In the network, there are 4,000 original data items of fixed size 20 KB that are uniformly distributed among them. Therefore, each data source owns 800 data items. When the data source receives a data request from the source node, the *first-come-first-served* policy is applied.

The cache size in each MH is set to 600–3000 KB. In other words, a node can cache 0.75–3.75% of the data items. In the simulation, each pair of source node and data source is selected randomly. A node is randomly selected as the query client whose query rate is set to 0.05 to 1 queries/s. The query pattern for the data object is based on a Zipf-like distribution [3]. In the Zipf-like distribution, which has been frequently used to model a non-uniform distribution, the access probability of the ith ($0 \leqq i \leqq n$) data item is represented as follows:

$$P_i = \frac{1}{i^\theta \sum_{k=1}^{n} \frac{1}{k^\theta}}, \quad \text{where } 0 \leqq \theta \leqq 1.$$

Table 2
Simulation parameters.

Parameter	Default value	Range
Simulator	NS2	
Network size	1500 m × 1500 m	
Bandwidth (MB/s)	2	
Number of mobile hosts	100	
Transmission range (m)	200	
Cache size SC (KB)	600	600–3000
MH join/leave rate (1/s)	0	0.05–0.4
Number of data sources	5	
MH speed (m/s)	0–2 randomly	3–15
Total number of data items	4000	
Size of each data item (KB)	20	
Zipf parameter (θ)	0.8	0–1
Mobility model	Random way point	
Pause time (s)	300	
k-group size for GCC scheme	$k = 1$–3	
Bitmap update interval (s)	80	5–80
Query rate (s)	0.2	0.05–1
TTL (s)	1000	100–2000

When $\theta = 1$, it follows the strict Zipf distribution. When $\theta = 0$, it follows the uniform distribution. Larger θ results in a more "skewed" access distribution. The simulation time is 1500 s. Thus, in total, 1000–30,000 queries (experiments) are measured. For each experiment, a sufficient number of simulation results is obtained and averaged to provide a 90% confidence interval within ±5%.

5.2. Simulation results

5.2.1. Memory usage

In GCC, only a "*Hello*" packet is used to exchange information among the group members. The packet size depends on the k-hop neighbors and the number of data IDs in the networks.

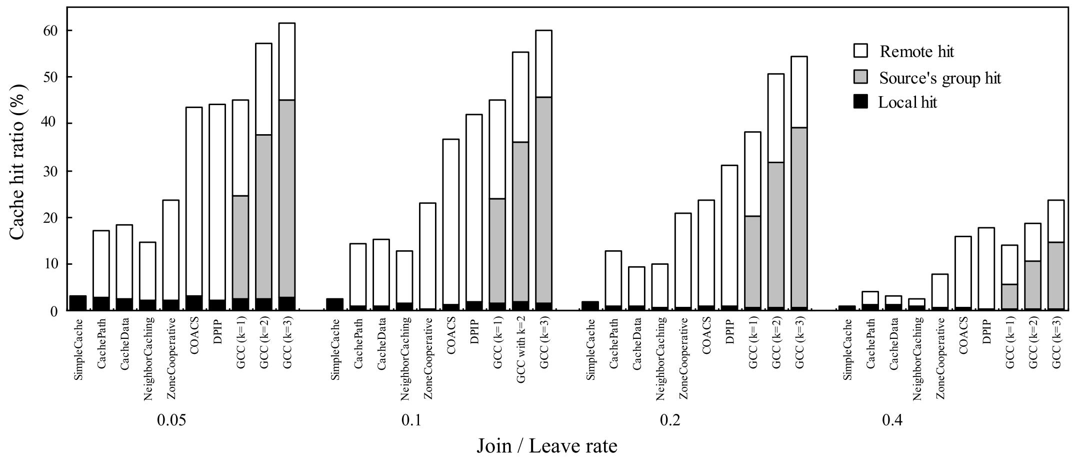


Fig. 14. Cache hit ratios with 600 KB cache and different join/leave rates.

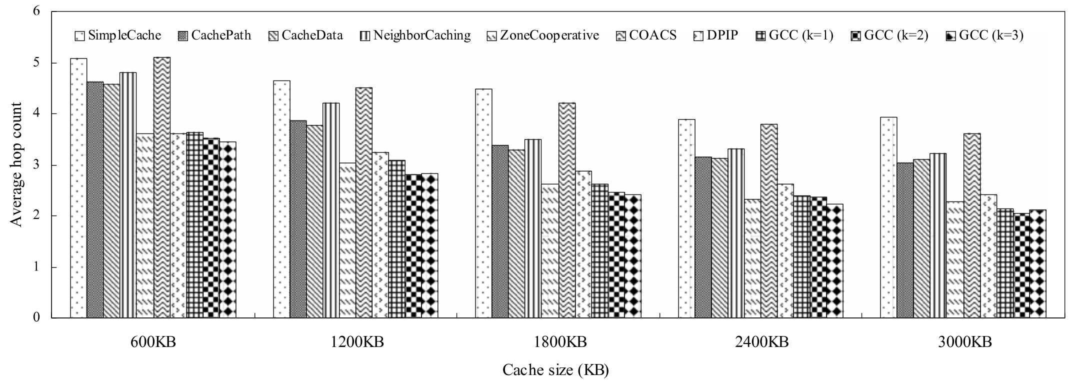


Fig. 15. Average hop counts under caches of size 600–3000 KB.

Table 3
Memory usage in aggregate bitmap scheme with block = 8 bits.

Cache size (KB)	600	1200	1800	2400	3000
# of data items cached in an MH	30	60	90	120	150
# of blocks in level 1			1		
# of blocks in level 2	8	8	8	8	8
# of blocks in level 3	27	43	52	55	58
# of blocks in level 4	30	57	81	106	129
# of blocks in an aggregate bitmap	66	109	142	170	196
Memory usage for all blocks (bits)	528	872	1136	1360	1568
Memory usage for all indices (bits)	350	510	600	630	640
Aggregate bitmap scheme Total memory usage (bits)	878	1382	1736	1990	2208
Original bitmap scheme Total memory usage (bits)			4000		
Memory saved (%)	78	65	56	50	44

The operation of create/search/combine on aggregate bitmaps is simple and easy to implement. However, the memory usage needs to be evaluated because the communication overhead is induced by the size of the aggregate bitmap piggyback in the "*Hello*" packets. Table 3 shows the memory usages of the aggregate bitmap and the original bitmap without compression. An original bitmap needs 4000 bits for the data set of 4000 data items. The number of bits needed for an aggregate bitmap depends on the distribution of the cached data items and the cache size. The simulations are conducted for 600–3000 KB caches and the block size is 8 bits. The number of blocks in each level is also shown in the Table 3. When the cache size varies from 600 to 3000 KB, the memory usage ranges between 878 and 2208 bits. Compared with the original bitmap scheme, saved memory amounts to 44–78%. Although the aggregate bitmap needs to allot additional memory for the index, an efficient bitmap search on the aggregate bitmap is achieved.

5.2.2. *Communication overhead*

To evaluate the communication overhead under the condition of different sizes of "*Hello*" packets, the simulation assumes that the data access has no locality (i.e., the cache miss rate is 100%). Thus, the requesting MH always obtains the data from the data

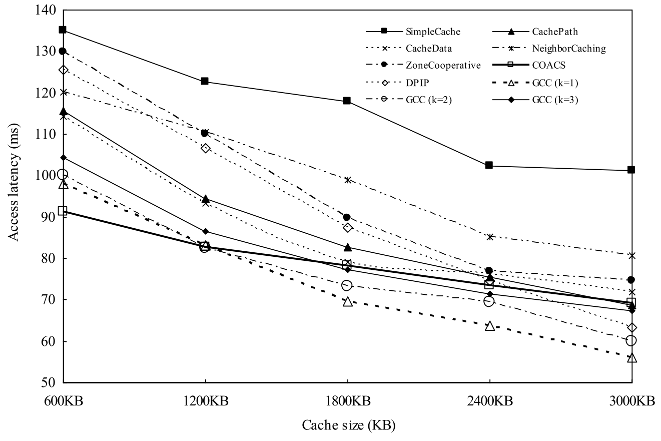


Fig. 16. Comparison of average latency under different cache sizes.

source. The source node and destination node (data source) are randomly selected. We measure the average latencies with and without "*Hello*" messages transferred in the network. The size of the "*Hello*" packet is set to 50–300 bytes, based on the memory usages shown in Table 3. The period P (5–80 s) is the time duration between two "*Hello*" packets sent by source nodes. The average latency for the network without "*Hello*" messages is 131.46 ms. The average latency for the network with "*Hello*" messages varies between 131.54 and 135.36 ms. As a result, the communication overhead does not significantly affect the message latency between two nodes. Assume the latencies in the networks with and without "*Hello*" packets are T_w and T_o. We define the latency increasing ratio to be $(T_w - T_o)/T_o$. Fig. 11 shows the latency increasing ratios under various "*Hello*" packet sizes and periods. The results show that the latency increasing ratio of the proposed scheme is only 0.1–2.9%.

5.2.3. Group hit ratio

In the evaluation of the group hit ratio, five sets of parameters with speeds of 3–15 m/s and different bitmap update intervals (once per 5, 10, 20, 40, and 80 s) are chosen. The high group hit ratio means the requested data can always be found in the group if the data bit of *group_bitmap* is set. In GCC, the degree of mobility (node speed) is low; the cache hit ratio is high because the request node can take the desired data from its group members with high probability. Contrarily, if the degree of mobility is high, the group members could move out of the request node with high probability. The request node could not take back the cached data from the group members. Thus, the cache hit ratio will be low. Also, the average hop count will be long because the data is retrieved from the farther original source node when the cache misses. As a result, the access latency is increased too.

Fig. 12(a) shows the results for GCC with $k = 1$. Whenever the bitmap update interval is remarkably short (once per 5 s) or remarkably long (once per 80 s), and the mobility is low or high, the group search hit ratio can reach at least 92.32%. We also run the same experiments for GCC with $k = 2$ and $k = 3$. The results are shown in Fig. 12(b) and (c). In extreme cases, when the bitmap update interval is long (once per 80 s) and the node speed is 15 m/s, the group search hit ratio reaches 83.65% and 73.76% in Fig. 12(b)

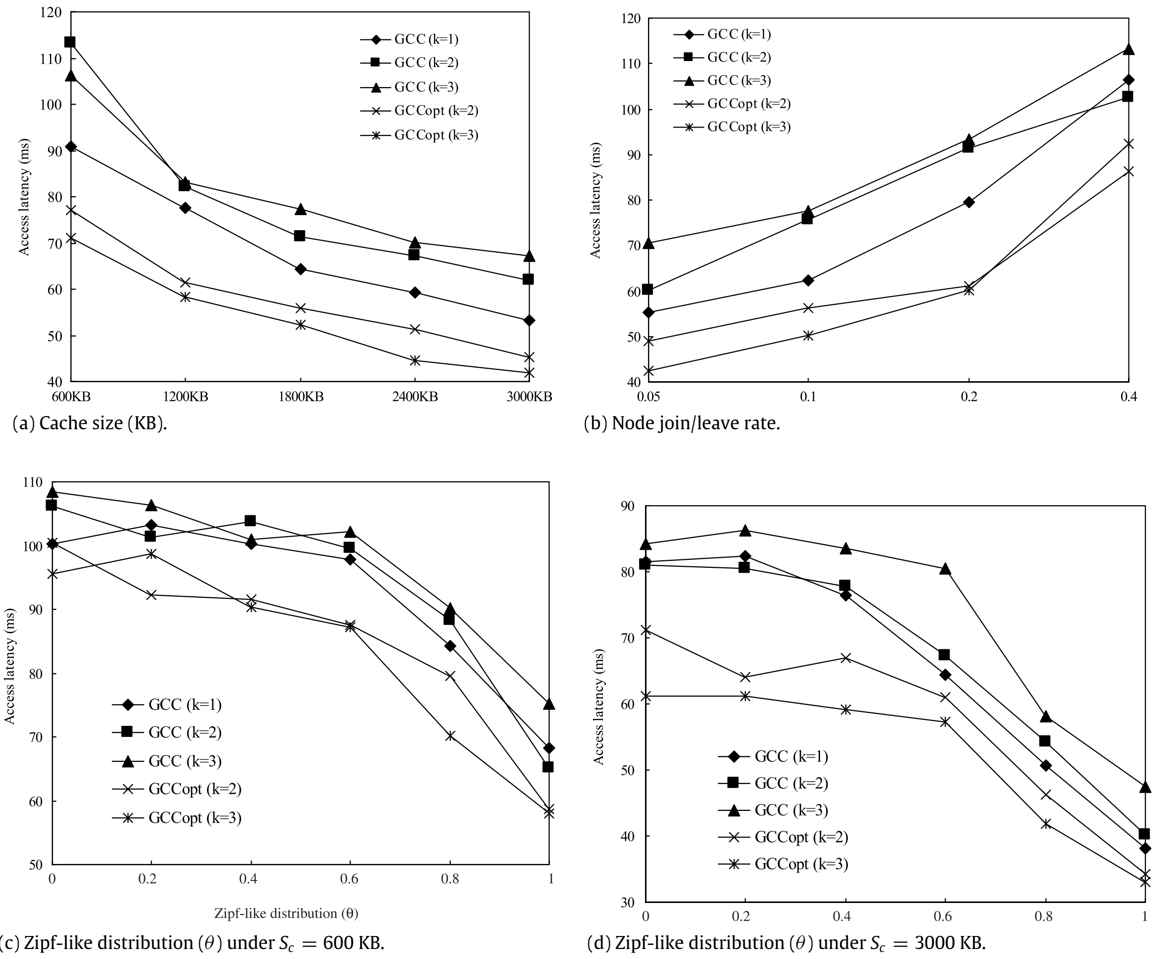
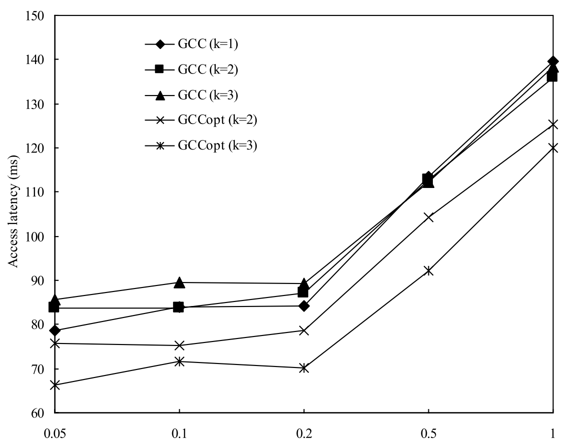
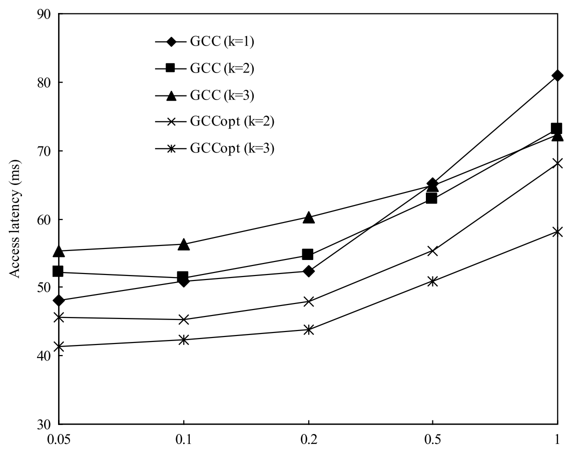


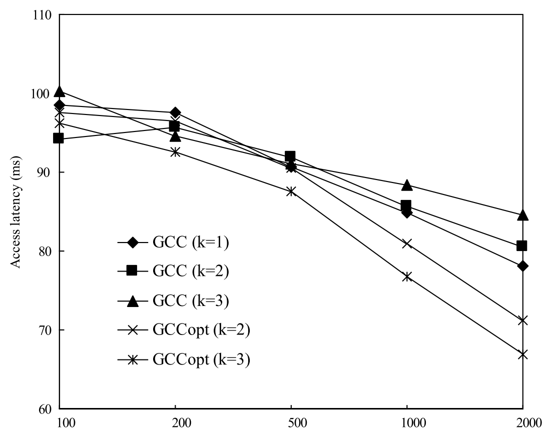
Fig. 17. Access latency of GCC and optimized GCC. (a) Cache size S_c, (b) join/leave rate, (c) θ under $S_c = 600$ KB, and (d) θ under $S_c = 3000$ KB.



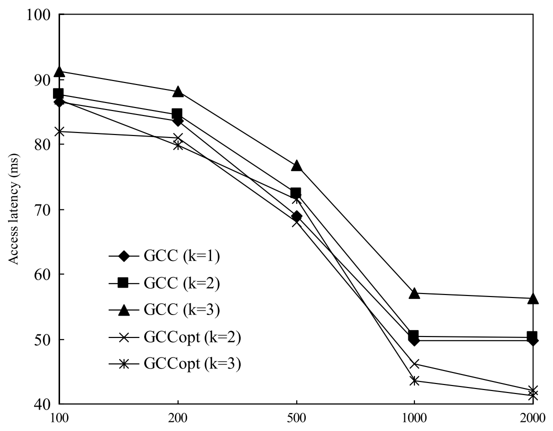
(a) Query rate under $S_c = 600$ KB.



(b) Query rate under $S_c = 3000$ KB.



(c) TTL under $S_c = 600$ KB.



(d) TTL under $S_c = 3000$ KB.

Fig. 18. Access latency of GCC and optimized GCC. (a) Query rate under $S_c = 600$ KB, (b) query rate under $S_c = 3000$ KB, (c) TTL under $S_c = 600$ KB, and (d) TTL under $S_c = 3000$ KB.

and (c), respectively. The reason is that the possibility that group members may move out of the group is high, reaching a speed of 15 m/s in 80 s. The result reveals that GCC has high group hit ratio in most cases.

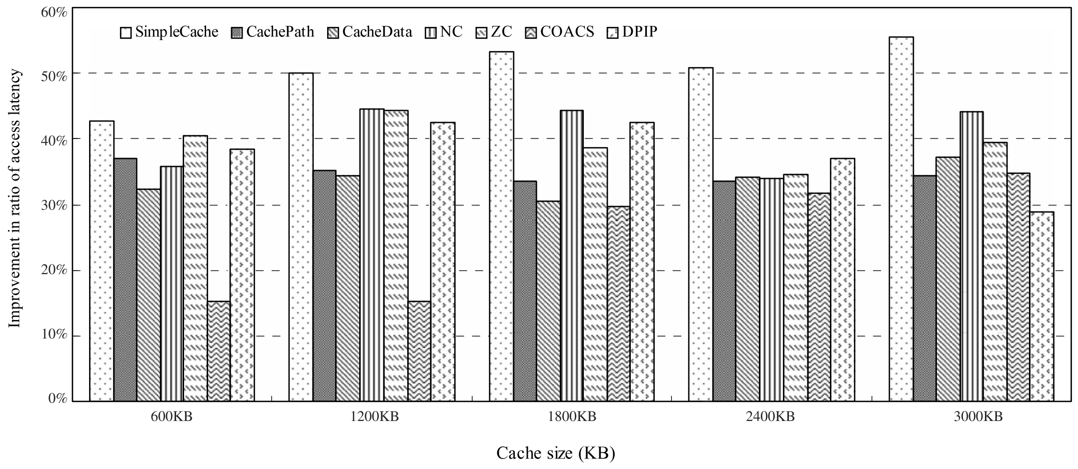
5.2.4. *Cache hit ratios*

The cache hit ratios include the local hit ratio, source group hit ratio, and remote hit ratio. The *local hit ratio* represents the probability of cache hits occurring in the local cache of source node. The *source group hit ratio* represents the probability of cache hits occurring in the group of source nodes in the GCC scheme. The *remote hit ratio* represents the probability of cache hits occurring in caching nodes other than the group of the source, as well as the data source. Fig. 13 shows the cache hit ratio results under caches sized 600–3000 KB. The local hit ratio, shown as the black bars, is not significantly different among all schemes because the common parameters such as cache size, data size, total data set, and data locality are the same. In general, the local hit ratio increases along with cache size increase. In the GCC scheme, when the group size is large, the group hit ratios of the source also increase (shown as the gray bars in Fig. 13). GCC with $k = 3$ has the highest source group hit ratio (i.e., 43% on average for all cache sizes). The average source group hit ratios for GCC with $k = 1$ and 2 are 25% and 35%, respectively. GCC also has the highest cache hit ratio compared with other schemes because the intermediate nodes and their group members can store significantly varied data items in the group. Fig. 14 shows the cache hit ratio under various node join/leave rates (0.05–0.4). If the node join/leave rate is high, the cache utility (cache hit) is decreased because the newly joining node's cache is empty. The request node cannot take any data from the newly joining node. However, in GCC, the entire cache of group members is utilized for the query node. Therefore, if some group members have moved out of the request node, the request node may be able to take the cached data from others.

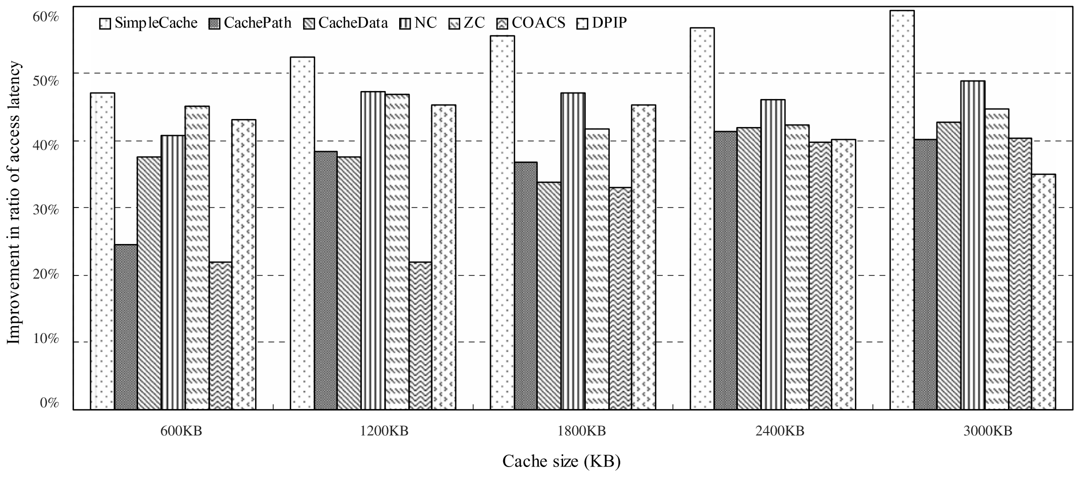
When this rate is as high as 0.4, the cache hit ratio is reduced significantly for SimpleCache, CachePath, CacheData, NC, and ZC schemes. The COACS, DPIP, and GCC schemes reveal better cache hit ratios. GCC with $k = 3$ exhibits the highest cache hit ratio among all the schemes.

5.2.5. *Hop count*

The *average hop count* is the average number of hops between the source node and the node providing the response, that is, the data source or one of the caching nodes. If the average hop count is small, the computation and communication overheads can be reduced when forwarding the replied data. Fig. 15 shows the average hop counts. SimpleCache has the longest average hop count. COACS usually redirects the data request to QDs to find caching nodes. The hop count may be larger when the cache hit ratio is low. DPIP and GCC have fewer hop counts (2.05–3.45) than other schemes because the requested data is usually found in the neighbors of the requester and intermediate nodes. In the case of the cache size of 3000 KB, GCC with $k = 2$ has the smallest hop count (2.05).



(a) GCC_{opt} ($k = 2$).



(b) GCC_{opt} ($k = 3$).

Fig. 19. Improved access latency ratios of GCC_{opt} over existing caching schemes.

5.2.6. *Access latency*

The *average access latency* is the average time between the instance the source node generates the data request and the instance the source node receives the response. Fig. 16 shows the average latency. The latency is affected by the hop count from the source node to the caching node or data source, as well as the process of data request in the data discovery phase. In the case of $S_c = 600$ KB, COACS has shorter latency than others. COACS is a cache-path-based scheme. Thus, the caching node needs to cache the index information, but not the data. In GCC, the caching node needs more cache size to cache data. Thus, when the cache size is too low, GCC's group members are expected not to provide significant help for the cache hit ratio. The latency is increased because the data is taken from the original data source. In the case of $S_c = 1200$ KB, GCC with $k = 2$ has the lowest latency. When the cache size increases, the latency decreases for all schemes. In most cases, GCC with $k = 1$ has the lowest latency. GCC with $k = 3$ has longer average access latency than $k = 1$ and 2. The reason is that, when k is large, the number of forwarding nodes increases in the phase of group search. GCC with $k = 3$ increases the communication overhead for broadcasting in the group, and then causes an additional latency.

5.2.7. *Optimized GCC* (GCC_{opt})

To reduce the communication overhead in group searches, we optimize the proposed GCC using the MPR-based forwarding node selection scheme described in Section 4.2, in which $k = 2$ and 3. GCC_{opt} with $k = 1$ is the same as the original GCC with $k = 1$. We evaluate the average access latency of GCC_{opt} by changing the various parameters including the cache size, join/leave rate, Zipf parameter θ, query rate, and TTL. Default values of parameters are shown in Table 2. Based on our experimental results, the average numbers of forwarding nodes in the group search for GCC_{opt} with $k = 2$ and $k = 3$ are 3.84 and 7.91, respectively. Fig. 17(a) illustrates the results with the cache size varying from 600 to 3000 KB, and Fig. 17(b) illustrates the results with the cache size fixed at 600 KB and node join/leave rates of 0.05–0.4. Fig. 17(c) and (d) illustrate the results with Zipf parameter θ varying from 0 to 1 with the cache size fixed at 600 and 3000 KB, respectively. We can see that GCC_{opt} with $k = 3$ consistently performs best. The performance difference between GCC_{opt} with $k = 2$ and $k = 3$ becomes minimal when the Zipf parameter θ nears 1. When θ is smaller, the query access pattern is similar to the uniform access pattern. Fig. 18(a)–(d) also illustrate the access latency, with the query rate varying from 0.05 to 1, and the TTL varying from 100 to 200 s. When the query rate is small, data traffic is light. Thus, the access latency does not significantly depend on the query rate. When the query rate is increased, the access latency is longer. The weak cache consistency model is used for the cached data. If the TTL is small, the cached data quickly becomes invalid. The caching performance is reduced. Based on the results shown in Figs. 17 and 18, GCC_{opt} with $k = 2$ and 3 obtain more access latency reduction than the original GCC, and GCC_{opt} with $k = 3$ has the lowest access latency. The forwarding node scheme reduces the access latency. Finally, Fig. 19 summarizes the access latency improvement ratios of GCC_{opt} over the existing cache schemes. For GCC_{opt} with $k = 2$, the access latency improvement can reach 15–55% over all the existing schemes. Similarly, the access latency improvement of GCC_{opt} with $k = 3$ is 21–59% over all the existing schemes.

6. Conclusions and future work

Cooperative caching among neighbor nodes is an important issue in MANETs because an MH can always cache a limited number of data objects. Therefore, in this paper, we propose a simple and efficient group-based cooperative caching to integrate the available caching space among the group members. The proposed scheme significantly improves the caching performance by maintaining the cache directory in the group. GCC can easily piggyback the bitmapped data in "*Hello*" messages. The additional communication overhead is minor. In data discovery, the source node and intermediate nodes can search the requested data in their group to avoid a flooding based search. GCC has the highest cache hit ratio compared with existing cooperative caching schemes. Both cache placement and replacement policies consider the mobility and status of the directory of *group_bitmap*. As a result, MHs can cache more data objects and efficiently use the available caching space of group members. In addition, GCC_{opt} reduces the computing overhead and uses the MPR-based forwarding node scheme to reduce the communication overhead in the group. Therefore, GCC_{opt} only consumes minimal traffic overhead. The simulation results show that the cache hit ratios, average hop count, and access latency are the lowest compared with existing cooperative cache schemes. In future studies, the remaining power of group members will be considered. Adjusting the interval of "*Hello*" message dynamically is also an important issue because the communication overhead can be reduced under different network conditions. The evaluation will also consider heterogeneous scenarios and dynamic data management for databases.

References

[1] C. Aggarwal, J. Wolf, P. Yu, Caching on the world wide web, IEEE Trans. Knowl. Data Eng. 11 (1) (1999).
[2] H. Artail, H. Safa, K. Mershad, Z. Abou-Atme, N. Sulieman, COACS: a cooperative and adaptive caching system for MANETs, IEEE Trans. Mobile Comput. 7 (8) (2008) 961–977.
[3] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and Zipf-like distributions: evidence and implications, Proc. IEEE INFOCOM (1999).
[4] J. Broch, D.A. Maltz, D.B. Johnson, Y.C. Hu, J. Jetcheva, A performance comparison of multi-hop wireless ad hoc network routing protocols, in: Proc. of ACM/IEEE Mobicom'98, 1998, pp. 85–97.
[5] T. Camp, J. Boleng, V. Davies, A survey of mobility models for ad hoc network research, Wireless Comm. & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications, pp. 483–502.
[6] G. Cao, L. Yin, C.R. Das, Cooperative cache based data access framework for ad hoc networks, IEEE Comput. (2004) 32–39.
[7] N. Chand, R.C. Joshi, M. Misra, Efficient cooperative caching in ad hoc networks communication system software and middleware, in: First International Conference on Comsware, Jan. 2006, pp. 1–8.
[8] G.-M. Chiu, C.-R. Young, Exploiting in-zone broadcasts for cache sharing in mobile ad hoc networks, IEEE Trans. Mobile Comput. 8 (3) (2009).
[9] J. Cho, S. Oh, J. Kim, H. Ho Lee, J. Lee, Neighbor caching in multi-hop wireless ad hoc networks, IEEE Commun. Lett. 7 (11) (2003) 525–527.
[10] C.-Y. Chow, H.-V. Leong, A. Chan, Peer-to-peer cooperative caching in mobile environments, in: Intl. Conf. on Distributed Computing Systems Workshop, 2004.
[11] K. Fall, K. Varadhan, The NS2 manual, the VINT Project, Apr. 2002. http://www.isi.edu/nsnam/ns/.
[12] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide area web cache sharing protocol, in: Proc. ACM SIGCOMM, 1998, pp. 254–265.
[13] T. Hara, Effective replica allocation in ad hoc networks for improving data accessibility, in: Proc. IEEE INFOCOM 2001, pp. 1568–1576.
[14] T. Hara, Replica allocation methods in ad hoc networks with data update, Mobile Netw. Appl. 8 (2003) 343–354.
[15] C. Imrich, M. Conti, J. Liu, Mobile Ad Hoc Networking: Imperatives and Challenges, in: Ad Hoc Networks, vol. 1, 2003, pp. 13–64.
[16] W.H.O. Lau, M. Kumar, S. Venkatesh, A cooperative cache architecture in supporting caching multimedia objects in MANETs, in: Fifth Int'l Workshop Wireless Mobile Multimedia, 2002.
[17] S. Lim, W.-C. Lee, G. Cao, C.-R. Das, A novel caching scheme for Internet-based mobile ad hoc networks, in: Proc. IEEE Int'l Conf. Computer Comm. and Networks, ICCCN, IEEE Press, 2003, pp. 38–43.
[18] S.-Y. Ni, Y.-C. Tseng, J.-P. Sheu, The broadcast storm problem in a mobile ad hoc network, in: Int'l Conf. on Mobile Computing and Networking, 1999, pp. 151–162.
[19] M. Papadopouli, H. Schulzrinne, Effects of power conservation, wireless coverage and cooperation on data dissemination among mobile devices, in: Proc. MobiHoc, ACM Press, 2001, pp. 117–127.
[20] C. Perkins, E. Belding-Royer, I. Chakeres, Ad hoc on demand distance vector (AODV) routing, IETF Internet draft, draft-perkins-manet-aodvbis-00.txt, Oct. 2003.
[21] A. Qayyum, L. Viennot, A. Laouiti, Multipoint relaying for Flooding broadcast messages in mobile wireless networks, in: Proceedings of the 35th Annual Hawaii International Conference on System Sciences, HICSS'02, Hawaii, 2002.
[22] A. Rousskov, D. Wessels, Cache digests, Comput. Netw. ISDN Syst. 30 (1998) 2155–2168.
[23] F. Sailhan, V. Issarny, Cooperative caching in ad hoc networks, in: International Conference on Mobile Data Management, MDM, 2003, pp. 13–28.
[24] J. Shim, P. Scheuermann, R. Vingralek, Proxy cache algorithms: design, implementation, and performance, IEEE Trans. Knowl. Data Eng. 11 (4) (1999).
[25] Yi-Wei Ting, Yeim-Kuan Chang, A novel cooperative caching scheme for wireless ad hoc networks: groupcaching, in: International Conference on Networking, Architecture, and Storage, NAS 2007, pp. 62–68.
[26] D. Wessels, K. Claffy, ICP and the squid web cache, IEEE J. Sel. Areas Commun. (1998) 345–357.
[27] L. Yin, G. Cao, Supporting cooperative caching in ad hoc networks, IEEE INFOCOM (2004) 2537–2547.
[28] L. Yin, G. Cao, Supporting cooperative caching in ad hoc networks, IEEE Trans. Mobile Comput. 5 (1) (2006) 77–89.

I-Wei Ting received his MS in Computer Science and Information Engineering from Chaoyang University of Technology, Taiwan, R.O.C., in 2004. He is currently working toward his PhD in Computer Science and Information Engineering at National Cheng Kung University, Taiwan, R.O.C. His current research interests include cache design systems, cooperative caching, and broadcasting strategy.

Yeim-Kuan Chang received his MS in Computer Science from the University of Houston at Clear Lake in 1990 and his PhD in Computer Science from Texas A&M University, College Station, Texas, in 1995. He is currently a Professor in the Department of Computer Science and Information Engineering at National Cheng Kung University, Tainan, Taiwan, Republic of China. His research interests include computer architecture, multiprocessor systems, Internet router design, and computer networking.