Cooperative Caching in Wireless P2P Networks: Design, Implementation, and Evaluation

Jing Zhao, *Student Member*, *IEEE*, Ping Zhang,
Guohong Cao, *Senior Member*, *IEEE*, and Chita R. Das, *Fellow*, *IEEE*

Abstract—Some recent studies have shown that cooperative cache can improve the system performance in wireless P2P networks such as ad hoc networks and mesh networks. However, all these studies are at a very high level, leaving many design and implementation issues unanswered. In this paper, we present our design and implementation of cooperative cache in wireless P2P networks, and propose solutions to find the best place to cache the data. We propose a novel asymmetric cooperative cache approach, where the data requests are transmitted to the cache layer on every node, but the data replies are only transmitted to the cache layer at the intermediate nodes that need to cache the data. This solution not only reduces the overhead of copying data between the user space and the kernel space, it also allows data pipelines to reduce the end-to-end delay. We also study the effects of different MAC layers, such as 802.11-based ad hoc networks and multi-interface-multichannel-based mesh networks, on the performance of cooperative cache. Our results show that the asymmetric approach outperforms the symmetric approach in traditional 802.11-based ad hoc networks by removing most of the processing overhead. In mesh networks, the asymmetric approach can significantly reduce the data access delay compared to the symmetric approach due to data pipelines.

Index Terms—Wireless networks, P2P networks, cooperative cache.

1 Introduction

Wireless P2P networks, such as ad hoc network, mesh networks, and sensor networks, have received considerable attention due to their potential applications in civilian and military environments. For example, in a battlefield, a wireless P2P network may consist of several commanding officers and a group of soldiers. 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 has accessed a data item from the data center, it is quite possible that nearby soldiers access the same data some time later. It will save 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 far away data center. As another example, people in the same residential area may access the Internet through a wireless P2P network, e.g., the Roofnet [3]. After one node downloads a MP3 audio or video file, other people can get the file from this node instead of the far away Web server.

Through these examples, we can see that if nodes are able to collaborate with each other, bandwidth and power can be saved, and delay can be reduced. Actually, *cooperative caching* [5], [16], [23], [24], which allows the sharing and coordination of cached data among multiple nodes, has been applied to improve the system performance in wireless P2P networks. However, these techniques [5], [16], [23], [24] are only evaluated by simulations and studied at a very high level, leaving many design and implementation issues unanswered.

There have been several implementations of wireless ad hoc routing protocols. In [22], Royer and Perkins suggested modifications to the existing kernel code to implement AODV. By extending ARP, Desilva and Das [7] presented another kernel implementation of AODV. Dynamic Source Routing (DSR) [12] has been implemented by the Monarch project in FreeBSD. This implementation was entirely in kernel and made extensive modifications in the kernel IP stack. In [2], Barr et al. addressed issues on system-level support for ad hoc routing protocols. In [13], the authors explored several system issues regarding the design and implementation of routing protocols for ad hoc networks. They found that the current operating system was insufficient for supporting on-demand or reactive routing protocols, and presented a generic API to augment the current routing architecture. However, none of them has looked into cooperative caching in wireless P2P networks.

Although cooperative cache has been implemented by many researchers [6], [9], these implementations are in the Web environment, and all these implementations are at the system level. As a result, none of them deals with the multiple

- *J. Zhao and P. Zhang are with the Department of Computer Science & Engineering, The Pennsylvania State University, 344 IST Building, University Park, PA 16802. E-mail: {jizhao, pizhang}@cse.psu.edu.*
- *G. Cao is with the Department of Computer Science & Engineering, The Pennsylvania State University, 354G IST Building, University Park, PA 16802. E-mail: gcao@cse.psu.edu.*
- *C.R. Das is with the Department of Computer Science & Engineering, The Pennsylvania State University, 354F IST Building, University Park, PA 16802. E-mail: das@cse.psu.edu.*

Manuscript received 7 Aug. 2008; revised 29 Jan. 2009; accepted 2 Mar. 2009; published online 13 Mar. 2009.
Recommended for acceptance by J.C.S. Lui.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-08-0299.
Digital Object Identifier no. 10.1109/TPDS.2009.50.

hop routing problem and cannot address the on-demand nature of the ad hoc routing protocols. To realize the benefit of cooperative cache, intermediate nodes along the routing path need to check every passing-by packet to see if the cached data match the data request. This certainly cannot be satisfied by the existing ad hoc routing protocols.

In this paper, we present our design and implementation of cooperative cache in wireless P2P networks. Through real implementations, we identify important design issues and propose an asymmetric approach to reduce the overhead of copying data between the user space and the kernel space, and hence to reduce the data processing delay.

Another major contribution of this paper is to identify and address the effects of data pipeline and MAC layer interference on the performance of caching. Although some researchers have addressed the effects of MAC layer interference on the performance of TCP [10] and network capacity [17], this is the first work to study this problem in the context of cache management. We study the effects of different MAC layers, such as 802.11-based ad hoc networks and multi-interface-multichannel-based mesh networks, on the performance of caching. We also propose solutions for our asymmetric approach to identify the best nodes to cache the data. The proposed algorithm well considers the caching overhead and adapts the cache node selection strategy to maximize the caching benefit on different MAC layers. Our results show that the asymmetric approach outperforms the symmetric approach in traditional 802.11-based ad hoc networks by removing most of the processing overhead. In mesh networks, the asymmetric approach can significantly reduce the data access delay compared to the symmetric approach due to data pipelines.

The rest of the paper is organized as follows: Section 2 presents our design and implementation of cooperative cache for wireless P2P networks. In Section 3, we present our prototype and experimental results. Section 4 extends our cooperative cache design to a large-scale network and presents extensive simulation results based on various MAC layers. Section 5 concludes the paper.

2 Design and Implementation of Cooperative Caching

In this section, we first present the basic ideas of the three cooperative caching schemes proposed in [24]: CachePath, CacheData, and HybridCache. Then, we discuss some design issues and present our design and implementation of cooperative cache in wireless P2P networks.

2.1 Cooperative Caching Schemes

Fig. 1 illustrates the CachePath concept. Suppose node N_1 requests a data item d_i from N_0. When N_3 forwards d_i to N_1, N_3 knows that N_1 has a copy of the data. Later, if N_2 requests d_i, N_3 knows that the data source N_0 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. Many routing algorithms (such as AODV [20] and DSR [12]) provide the hop count information between the source and destination. Caching the data path for each data item reduces bandwidth and power consumption because nodes can obtain the data using fewer hops. However, mapping data items and caching nodes increase routing overhead, and the following techniques are used to improve CachePath's performance.

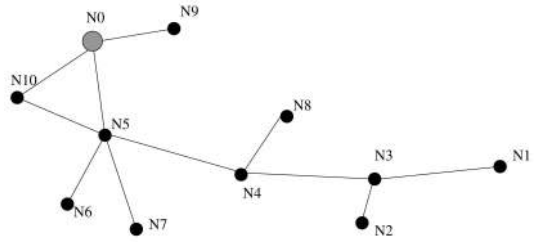


Fig. 1. A wireless P2P network.

In CachePath, a node need not record the path information of all passing data. Rather, it only records the data path when it is closer to the caching node than the data source. For example, when N_0 forwards d_i to the destination node N_1 along the path $N_5 - N_4 - N_3, N_4$ and N_5 won't cache d_i path information because they are closer to the data source than the caching node N_1. In general, a node caches the data path only when the caching node is very close. The closeness can be defined as a function of the node's distance to the data source, its distance to the caching node, 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 is much shorter than its distance to the data source, then the routing node should cache the data path.

In CacheData, the intermediate node caches the data instead of the path when it finds that the data item is frequently accessed. For example, in Fig. 1, if both N_6 and N_7 request d_i through N_5, N_5 may think that d_i is popular and cache it locally. N_5 can then serve N_4's future requests directly. Because the CacheData approach needs extra space to save the data, it should be used prudently. Suppose N_3 forwards several requests for d_i to N_0. The nodes along the path N_3, N_4, and N_5 may want to cache d_i as a frequently accessed item. However, they will waste a large amount of cache space if they all cache d_i. To avoid this, CacheData enforces another rule: A node does not cache the data if all requests for the data are from the same node.

In this example, all the requests N_5 received are from N_4, and these requests in turn come from N_3. With the new rule, N_4 and N_5 won't cache d_i. If N_3 receives requests from different nodes, for example, N_1 and N_2, it caches the data. Certainly, if N_5 later receives requests for d_i from N_6 and N_7, it can also cache the data.

CachePath and CacheData can significantly improve system performance. Analytical results [24] show that CachePath performs better when the cache is small or the data update rate is low, while CacheData performs better in other situations. To further improve performance, we can use HybridCache, a hybrid scheme that exploits the strengths of CacheData and CachePath while avoiding their weaknesses. Specifically, when a node forwards a data item, it caches the data or path based on several criteria discussed in [24].

2.2 Design Issues on Implementing Cooperative Cache

In this paper, we focus on design and implementation of the CacheData scheme discussed in the above section. To realize the benefit of cooperative cache, intermediate nodes along the routing path need to check every passing-by

packet to see if the cached data match the data request. This certainly cannot be satisfied by the existing ad hoc routing protocols. Next, we look at two design options.

2.2.1 Integrated Design

In this option, the cooperative cache functions are integrated into the network layer so that the intermediate node can check each passing-by packet to see if the requested data can be served. Although this design sounds straightforward, several major drawbacks make it impossible in real implementation.

The network layer is usually implemented in kernel, and hence, the integrated design implies a kernel implementation of cooperative cache. However, it is well known that kernel implementation is difficult to customize and then it is difficult for handling different application requirements. Second, kernel implementation will significantly increase the memory demand due to the use of CacheData. Finally, there is no de facto routing protocol for wireless P2P networks currently. Implementing cooperative cache at the network layer requires these cache and routing modules to be tightly coupled, and the routing module has to be modified to add caching functionality. However, to integrate cooperative cache with different routing protocols will involve tremendous amount of work.

2.2.2 Layered Design

The above discussions suggest that a feasible design should have a dedicated cooperative cache layer resided in the user space; i.e., cooperative cache is designed as a middleware lying right below the application layer and on top of the network layer (including the transport layer).

There are two options for the layered design. One naive solution uses cross-layer information, where the application passes data request (search key) to the routing layer, which can be used to match the local cached data. However, this solution not only violates the layered design, but also adds significant complexity to the routing protocol which now needs to maintain a local cache table. Further, if an intermediate node needs to cache the data based on the cooperative cache protocol, it has to deal with fragmentation issues since some fragments of the data may not go through this node. Thus, this naive solution does not work in practice.

Another solution is to strictly follow the layered approach, where the cooperative cache layer is on top of the network layer (TCP/IP). Fig. 2 shows the message flow (dashed line) in the layered design. In the figure, N_5 sends a request to N_0. Based on the routing protocol, N_5 knows that the next hop is N_4 and sends the request to N_4 encapsulating the original request message. After N_4 receives the request, it passes the message to the cache layer, which can check if the request can be served locally. This process continues until the request is served or reaches N_0. After N_0 receives the request, it forwards the data item back to N_5 hop by hop, which is the reverse of the data request, as shown in Fig. 2b. Note that the data has to go up to the cache layer in case some intermediate nodes need to cache the data.

Although this solution can solve the problems of the naive solution, it has significant overhead. For example, to avoid caching corrupted data, reliable protocols such as TCP are needed. However, this will significantly increase the overhead, since the data packets have to move to the TCP layer at each hop. Note that the data packets only need to go to the routing layer if cooperative cache is not used. Further, this solution has a very high context switching overhead. At each intermediate node, the packets have to be copied from the kernel to the user space for cache operations, and then reinjected back to kernel to be routed to the next hop.

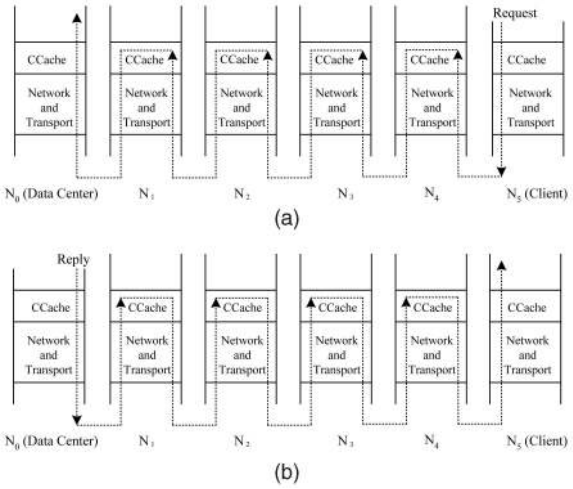


Fig. 2. Layered design. (a) The request packet flow and (b) the reply packet flow.

The pipeline effect. Another problem of the layered design is the lack of data pipeline. Normally, the transport layer can fragment a large data item into many small data packets, which are sent one by one to the next hop. If there are multihops between the sender and the receiver, these small packets can be pipelined and the end-to-end delay can be reduced.

In cooperative cache, the caching granularity is at the data item level. Although a large data item is still fragmented by the transport layer, there is no pipeline due to the layered design. This is because the cache layer is on top of the transport layer, which will reassemble the fragmented packets. Since all packets have to go up to the cache layer hop by hop, the network runs like "stop and wait" instead of "sliding window." This will significantly increase the end-to-end delay, especially for data with large size.

2.3 The Asymmetric Cooperative Cache Approach

To address the problem of the layered design, we propose an asymmetric approach. We first give the basic idea and then present the details of the scheme.

2.3.1 The Basic Idea

In our solution, data requests and data replies are treated differently. The request message still follows the path shown in Fig. 2a; however, the reply message follows a different path. If no intermediate node needs to cache the data, N_0 sends the data directly to N_5 without going up to the cache layer. Suppose N_3 needs to cache the data based on the cooperative cache protocol, as shown in Fig. 3. After N_3 receives the request message, it modifies the message and notifies N_0 that the data should be sent to N_3. As a result, the data are sent from N_0 to N_3 through the cache layer, and then sent to N_5. Note that the data will not go to the cache layer in intermediate nodes such as N_1, N_2, and N_4 in this example. In this way, the data only reach the routing layer for most intermediate nodes, and go up to the

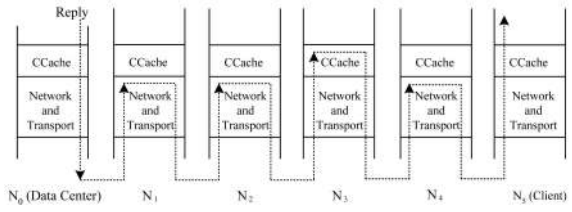


Fig. 3. In the asymmetric approach, the data reply only goes up to the cache layer at the intermediate nodes that need to cache the data.

cache layer only when the intermediate node needs to cache the data. Although the request message always needs to go up to the cache layer, it has a small size, and the added overhead is limited.

If the requested data item is large, this asymmetric approach allows data pipeline between two caching nodes, and hence reduces the end-to-end delay. The cache layer processing overhead, especially data copying between kernel and user spaces, is also minimized because the data item is not delivered to the cache layer at the nodes that are unlikely to cache the data. Next, we discuss the details of our asymmetric approach.

2.3.2 The Asymmetric Approach

Our asymmetric caching approach has three phases:

Phase 1: Forwarding the request message. After a request message is generated by the application, it is passed down to the cache layer. To send the request message to the next hop, the cache layer wraps the original request message with a new destination address, which is the next hop to reach the data server (real destination). Here, we assume that the cache layer can access the routing table and find out the next hop to reach the data center. This can be easily accomplished if the routing protocol is based on DSR or AODV. In this way, the packet is received and processed hop by hop by all nodes on the path from the requester to the data server.

For example, in Fig. 2a, when N_5 requests d_i from N_0, it adds a new header where the destination of the data request becomes N_4, although the real destination should be N_0. After N_4 receives and processes the packet, it changes the destination to be N_3, and so on, until the request packet arrives at N_0.

When an intermediate node receives the request message and delivers to the cache layer, the cache manager performs two tasks: First, it checks if it has the requested data in its local cache; if not, it adds its local information to the request packet. The local information includes the access frequency (number of access requests per time unit) of the requested data, channel used, and throughput of the link where the request is received. Its node id will also be added to $Path\_List$, which is a linked list encapsulated in the cache layer header. When the request message reaches the node who has the data, $Path\_List$ in the message will include all the intermediate nodes along the forwarding path.

Phase 2: Determining the caching nodes. When a request message reaches the data server (the real data center or the intermediate node that has cached the requested data), the cache manager decides the caching nodes on the forwarding path, which will be presented in Section 2.3.3. Then, the ids of these caching nodes are added to a list called $Cache\_List$, which is encapsulated in the cache layer header.

Phase 3: Forwarding the data reply. Unlike the data request, the data reply only needs to be processed by those nodes that need to cache the data. To deliver the data only to those that will cache the data, tunneling techniques [8] are used. The data reply is encapsulated by the cache manager and tunneled only to those nodes appearing in $Cache\_List$. As shown in Fig. 3, suppose the intermediate node N_3 needs to cache data d_i. Then, N_3 and N_5 are the nodes to process the data at the cache layer. N_0 includes N_3 and N_5 in the cache header of the data item d_i, and first sets the destination address of d_i to be N_3. When N_3 receives any fragmented packet of d_i, the routing layer of N_3 will deliver the packet upward to the transport layer and then to the cache layer. After the whole data item d_i has been received by N_3, it caches the data item, sets the next destination using the next entry in $Cache\_List$, which is N_5, and then passes the data down to the routing layer. After N_5 receives the data, it delivers the data to the application layer.

2.3.3 Determining the Caching Nodes

When a request reaches the data server, the cache manager examines the $Path\_List$ and determines which nodes in the $Path\_List$ to cache the data. One advantage of letting the data server make the decision is because the data server can use global information to achieve better performance.

The data server needs to measure the benefit of caching a data item on an intermediate node and use it to decide whether to cache the data. After an intermediate node (N_i) caches a data item, N_i can serve later requests using the cached data, instead of forwarding the requests to the data server, saving the communication overhead between N_i and the data center. However, caching data at N_i increases the delay of returning the data to the current requester, because it adds extra processing delay at N_i, and the data reassembly at N_i may affect possible pipelines.

Suppose the data server (N_0) receives a request from a client (N_n). The forwarding path $(N_0, N_1, \ldots, N_{n-1}, N_n)$ consists of $n-1$ intermediate nodes, where N_i is the ith node from N_0 on the path. To compute the benefit of caching a data item, the data server needs the following parameters:

- *Excluded data access frequency* (f_i): the number of access requests for a given data item received by N_i per time unit, but excluding the requests forwarded from other nodes on the forwarding path. To compute f_i, a node can first count the total requests it receives for the given data item, including the requests generated by itself, and those forwarded from its neighbors, then divide by the amount of time since the node starts to participate caching. This computes the total data request frequency (a_i). The node can attach the value of a_i to the forwarding request. When the data server receives the data request and gets the value of a_i for each node along the forwarding path, it can compute f_i as

$$f_i = \begin{cases} a_i - a_{i+1}, & \text{if } 1 \le i \le n-1, \\ a_n, & \text{if } i = n. \end{cases}$$

- *Tunneling delay* ($d_{i,j}(S)$): the delay to forward a data item with size S from the cache layer of N_i to the cache layer of N_j, without handing the data up to the cache layer of any intermediate nodes. $d_{i,j}(S)$ is hard to measure because it is affected by many factors, such as transmission interference, channel diversities, node mobility, etc. We assume $d_{i,j}(S)$ is known at this point to introduce our optimal placement model. We will present heuristics to calculate it later.

Optimal cache placement. We first define a new term called *aggregate delay*, which includes the time to serve the current client request and the delay to serve future data requests coming from the same path. We can also assign different weights to these two parts of the delay based on the optimization objective, e.g., giving less weight to the future access delay if the current request has strict delay requirement. For simplicity, we assign equal weight for the current and future access delay in this paper. Below, we formally define the optimal cache placement problem.

Definition. *Given a n-hop forwarding path $N_0, N_1, \ldots N_n$, where N_0 is the data server and N_n is the client, the problem of optimal cache placement is to find a cache node set $P = \{N_{c_1}, \ldots, N_{c_m} | 0 < c_1 < c_2 < \cdots < c_m < n\}$, which has the minimum aggregate delay for a given period of time Δt.*

In this definition, c_i refers to the subscript of the node N_{c_i} on the forwarding path, which implies that node N_{c_i} is c_i hops away from the data server. For instance, if only N_2 and N_4 are selected as cache node in a forwarding path $N_0, N_1, \ldots, N_{n-1}, N_n, c_1 = 2$ and $c_2 = 4$.

Given a cache placement $P = \{N_{c_1}, N_{c_2}, \ldots, N_{c_m}\}$, the aggregate delay is computed by considering both the latency to return the current reply (L_c) and the latency to reply future data requests (L_f). We have

$$L_c = \sum_{i=1}^{m} h(S_D) + d_{0,c_1}(S_D) + \sum_{i=1}^{m-1} d_{c_i,c_{i+1}}(S_D) + d_{c_m,n}(S_D), \tag{1}$$

where S_D is the data size and $h(S_D)$ is the data processing delay. Since the data are cached at m intermediate nodes, $\sum_{i=1}^{m} h(S_D)$ considers the cache processing delay at these nodes. As the data are reassembled at m nodes, the forwarding path is cut into $m+1$ pieces. The rest part of (1) considers the total forwarding delay as the summation of the tunneling delays on each piece.

$$\begin{aligned} L_f = &\sum_{j=1}^{c_1-1} f_j(d_{0,j}(S_R) + d_{0,j}(S_D))\Delta t \\ &+ \sum_{i=1}^{m-1} \sum_{j=c_i}^{c_{i+1}-1} f_j(d_{c_i,j}(S_R) + d_{c_i,j}(S_D))\Delta t \\ &+ \sum_{j=c_m}^{n-1} f_j(d_{c_m,j}(S_R) + d_{c_m,j}(S_D))\Delta t, \end{aligned} \tag{2}$$

where f_j is the excluded data access frequency at node j, S_R and S_D are the size of the request message for the data request and the data itself, respectively. Equation (2) computes the future delay by assuming that any future request going through the node on the current forwarding path will be replied by the first caching node met by the request. The replied data will be sent to the client without handing up to the cache layer of the intermediate node. Thus, the cache placement set P is optimal if it can minimize the weighted aggregate delay L, which is given by

$$\text{minimize } L = L_c + L_f. \tag{3}$$

Heuristics to calculate $d_{i,j}(S)$. We first calculate the throughput between two caching nodes N_i and N_j, denoted as $T_{i,j}$. Each node passively estimates its link throughput to its one-hop neighbors. N_i can estimate the throughput to N_{i+1} (i.e., $T_{i,i+1}$) by using the request message size divided by the link transmission delay. The multihop throughput is computed recursively by considering the node interference and channel diversity on the forwarding path. Let l_i denote the channel used between N_i and N_{i+1}, and assume that the interference range is p hops. $T_{i,j+1}$ ($j > i$) is computed as

$$T_{i,j+1} = \begin{cases} \min\{T_{i,j}, T_{j,j+1}\}, & \text{if } l_j \neq l_{j-1}, l_{j-2}, \ldots, l_{j-p}, \\ 1/\left(\frac{1}{T_{i,j}} + \frac{1}{T_{j,j+1}}\right), & \text{otherwise.} \end{cases} \tag{4}$$

In (4), if the channel used by link l_j is different from the links ($l_{j-1}, \ldots, l_{j-p}$) within its interference range, link l_j can transmit the packet concurrently with any of those links. So after adding link l_j, the end-to-end throughput can be better sustained, which is given by the link throughput of l_j or the already obtained throughput, whichever is lower. Otherwise, link l_j interferes with the previous added links and reduces the throughput.

Given $T_{i,j}$ computed by (4) and B as the size of MTU, $d_{i,j}(S)$ can be computed as

$$d_{i,j}(S) = \frac{(\lceil \frac{S}{B} \rceil - 1) \cdot B}{T_{i,j}} + \sum_{k=i}^{j-1} \frac{B}{T_{k,k+1}}. \tag{5}$$

Equation (5) considers that a data item may be fragmented into multiple packets, with the maximum packet size of B. In the formula, the first term of the summation approximately computes the waiting time for the last packet to be sent, which is after all the previous packets being injected to the forwarding path. The second term estimates the delay to transmit the last packet from the data server to the client.

When a forwarding node N_i receives a request message from the link, it can attach the link throughput $T_{i,i+1}$ and the channel used on this link l_i to the request packet header and keeps forwarding. So given a data item of size S, based on the link channel and throughput information, the data server can compute $d_{i,j}(S)$ ($0 \leq i, j \leq n$) by using (4) and (5).

A greedy cache placement algorithm. To get the optimal cache placement, the data server needs to compute the aggregate delay for every possible cache placement set. Since there are 2^n (n is the length of the forwarding path) possible ways to choose cache placement set, it takes $\Theta(2^n)$, which is too expensive. Therefore, we propose a greedy heuristic to efficiently compute the optimal cache placement.

Let $P^*(k)$ be the optimal cache placement for a forwarding path when only the nearest k hops from the data server are considered as possible cache nodes. With the same condition, let $L^*(k)$ be the aggregate delay of the optimal placement $P^*(k)$, and $p^*(k)$ be the hop distance of the farthest cache node from the data server in $P^*(k)$.

Notations

- P: cache placement set.
- L: aggregate delay.
- pos: hop distance of the farthest cache node from the data server.
- $f[i]$: Excluded data access frequency on N_i.
- $d_D[i,j]$: delay of forwarding the data item from N_i to N_j.
- $d_R[i,j]$: delay of forwarding the data request from N_j to N_i.
- S_D: size of the data item.
- S_R: size of the data request.
- $R[i]$: link throughput between nodes N_i and N_{i+1}.
- $T[i,j]$: link throughput between nodes N_i and N_j.
- $l[i]$: channel used for the link between nodes N_i and N_{i+1}.
- $h(S)$: cache processing cost for the data size of S.
- Δt: the expiration time of the data item.

Cache Placement Algorithm

```
1: input: f[], R[], l[], Δt, S_R, S_D.
2: output: P.
3: for i = 0 to n − 1 do
4:    T[i, i + 1] = R[i];
5: end for
6: for i = 0 to n do
7:    for j = i + 1 to n do
8:       compute T[i, j], d_D[i, j] and d_S[i, j] using Equ. 4 and
         5;
9:    end for
10: end for
11: L ← d_D[0, n];
12: P ← ∅;
13: pos ← 0;
14: for i = 0 to n − 1 do
15:    L ← L + f[i](d_D[0, i] + d_R[0, i])Δt;
16: end for
17: for k = 0 to n − 2 do
18:    ΔL_c ← h(S_D) + d_D[pos, k + 1] + d_D[k + 1, n] −
       d_D[pos, n];
19:    ΔL_f ← 0;
20:    for i = k + 1 to n − 1 do
21:       ΔL_f ← ΔL_f + (d_D[pos, i] + d_R[pos, i] − d_D[k +
          1, i] − d_R[k + 1, i])f[i]Δt;
22:    end for
23:    L' ← L + ΔL_c − ΔL_f;
24:    if L' < L then
25:       L ← L';
26:       P ← P ∪ N_{k+1};
27:       pos ← k + 1;
28:    end if
29: end for
```

Fig. 4. The greedy algorithm to determine the cache placement.

When $k = 0$, no cache node is between the data server and the client, and then the data server N_0 transmits the data directly to the client N_n without reassembling at any intermediate node. All future requests from N_i need to get data from the data server. Therefore, $P^*(0) = \emptyset, p^*(k) = 0$, and

$$L^*(0) = d_{0,n}(S_D) + \sum_{i=1}^{n-1} f_i \cdot (d_{0,i}(S_D) + d_{0,i}(S_R)) \cdot \Delta t.$$

Given $L^*(k), P^*(k)$, and $p^*(k)$, we check whether to select the intermediate node N_{k+1} as a cache node. If N_{k+1} is selected, we have

$$\begin{aligned} L(k+1) = L^*(k) + h(S_D) + (d_{p^*(k),k+1}(S_D) + d_{k+1,n}(S_D) \\ - d_{p^*(k),n}(S_D)) - \sum_{i=k+1}^{n-1} (d_{p^*(k),i}(S_D) + d_{p^*(k),i}(S_R) \\ - d_{k+1,i}(S_D) - d_{k+1,i}(S_R)) f_i \Delta t. \end{aligned}$$

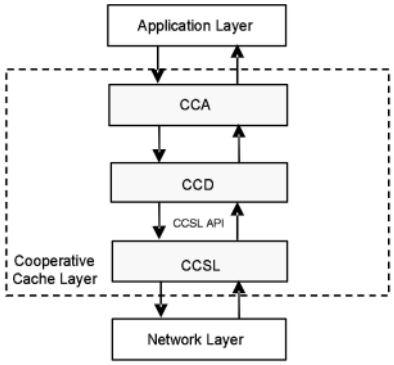


Fig. 5. System architecture.

If $L(k+1)$ is smaller than $L^*(k), N_{k+1}$ is selected as the cache node since it reduces the aggregate delay. The optimality is updated: $L^*(k+1) = L(k+1), P^*(k+1) = P^*(k) \cup N_{k+1}$, and $p^*(k+1) = N_{k+1}$; otherwise, $P^*(k+1)$, $L^*(k+1)$, and $p^*(k+1)$ keep unchanged from $P^*(k), L^*(k)$, and $p^*(k)$. Fig. 4 shows the detail of the algorithm. The algorithm has complexity of $\Theta(n^2)$, which is much more efficient than the optimal algorithm.

2.4 System Implementation

2.4.1 Architecture Overview

Fig. 5 shows the architecture of our cooperative cache middleware, which consists of three parts: Cooperative Cache Supporting Library (CCSL), Cooperative Cache Daemon (CCD), and Cooperative Cache Agent (CCA).

CCSL is the core component to provide primitive operations of the cooperative cache, e.g., checking passing-by packets, recording data access history, and cache read/write/replacement primitives. A data cache buffer is maintained at every node to store the cached data items. There is an interface between CCSL and the routing daemon, from which CCSL obtains the routing distance to a certain node. All these primitive cache operations are enclosed as CCSL API to provide a uniform interface to the upper layer.

CCD is the component that implements different cooperative cache mechanisms, namely, CacheData, CachePath, and HybirdCache. There is one cache daemon for each cooperative cache scheme. It extends the basic CCSL API to accomplish the characteristic of each scheme.

CCA is the module that maps application protocol messages to corresponding cooperative cache layer messages. It is a connector between CCD and user applications. There is one CCA for each user application.

2.4.2 The Cooperative Cache Supporting Library (CCSL)

CCSL is implemented as a user-space library. It implements common functions that cooperative cache schemes need and provides APIs to the Cooperative Cache Daemon (CCD).

Fig. 6 illustrates the software architecture of CCSL. As shown in the figure, the *Cache table* is used to record data access. It is a hash table keyed by *data id*. Data items are cached in the *data cache*. Besides these two components, a list of recently received requests is maintained to detect duplicate data requests. If the data request is not a

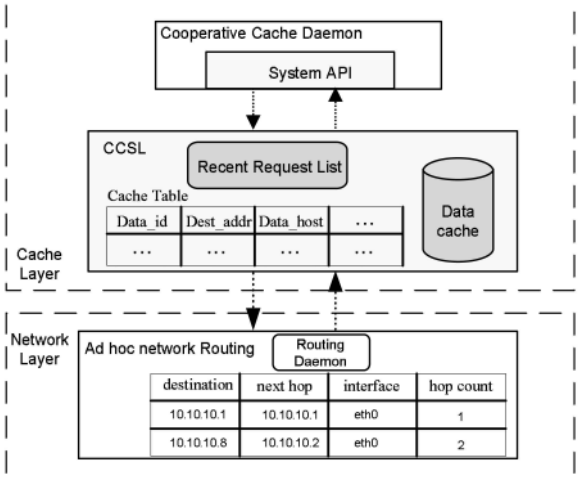


Fig. 6. CCSL design.

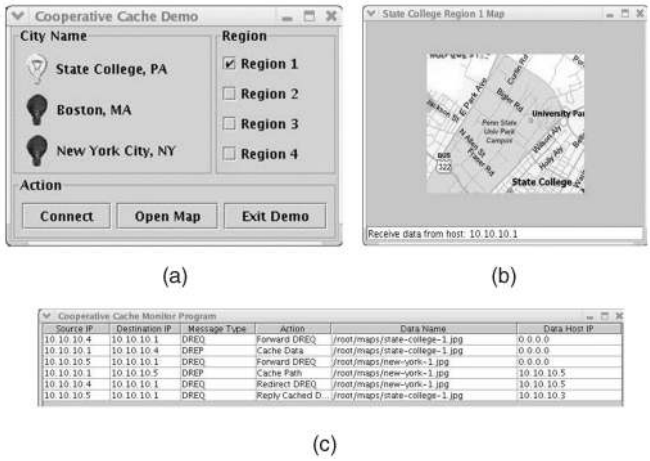


Fig. 8. A map quest application demo. (a) Map client, (b) map displayed by N_4, and (c) message monitor.

duplicate, it will be passed to the Cooperative Cache Daemon (CCD). An interface is provided between CCSL and the routing daemon. It enables CCSL to get the routing information which is used for transmitting cooperative cache layer packets.

CCSL encapsulates the complex mechanisms of the cooperative cache to provide simple interfaces to CCD. For example, when a data request is issued, CCD constructs a data request packet and calls $send\_packet()$ to send it. $send\_packet()$ reads the destination address of this packet, consults routing daemon for the next hop address, and sends a packet containing the received data request to the next hop. Another example is $cache\_data()$. When $cache\_data()$ is called by CCD, it checks the data cache for some space and then saves the data item. If there is not enough space, cache replacement is used to find enough space.

3 The Prototype and Experimental Results

To evaluate the performance of the cooperative cache implementation, we set up an ad hoc network as shown in Fig. 7. Five nodes are Dell Pentium laptops with 1.6 GHz CPU and 512 MB memory. Each Laptop has a Dell TrueMobile 802.11 wireless cards configured in ad hoc mode. The dashed circle around each node indicates the transmission range. Any two nodes connected with a solid line are direct neighbors. The routing protocol is based on AODV [20]. The implementation is based on Linux kernel version 2.4.21.

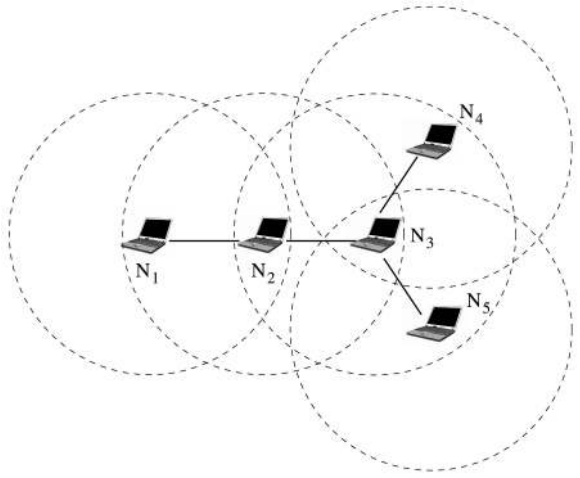


Fig. 7. Topology of the testbed.

3.1 A Map Quest Application

We wrote a simple map quest application to demo our cooperative cache functions. This map quest application can be divided into two parts: map center and map client. All maps are originally saved in the map center. Once the map center receives a map query request, the requested map is retrieved and sent back to the requester.

Fig. 8a shows the graphic user interface (GUI) of a map client. After the user clicks the map request, the request message is sent to a specially designed Cooperative Caching Java Agent (CCJA) at this node. As a protocol translator, the CCJA translates the received map request message to the format used in Cooperative Cache Daemon (CCD). If the requested map is returned back to the CCD of the requesting node, the CCD forwards the received reply message together with the replied map to the corresponding CCJA. Then, the CCJA constructs a map reply message and sends it to the map client. Fig. 8b shows a map displayed by a client.

Fig. 8c shows the message flows at intermediate nodes. For each message that the CCD receives, the message monitor displays information such as source and destination address, message type, map name, CCD's processing result, and the address of the actual node from which the requested map is received.

3.2 Experimental Results

In this section, we compare the performance of the symmetric and asymmetric cooperative cache approaches. In the symmetric approach, both data request and data reply go up to the cache layer. As shown in Fig. 7, N_1 is the data center which stores 100 test files of sizes 0.9 KB, 1.3 KB, 1.9 KB, 3.2 KB, and 6.4 KB. N_4 and N_5 randomly choose files and initiate data requests.

Data from Table 1 shows that the Asymmetric approach reduces the data access latency by 20-23 percent compared to

TABLE 1
Data Access Delay (in Milliseconds)

	0.9KB	1.3KB	1.9KB	3.2KB	6.4KB
SimpleCache	28.56	31.19	42.12	60.64	118.26
Symmetric	24.87	27.13	36.97	49.30	102.38
Asymmetric	22.56	24.21	32.36	46.09	93.70

TABLE 2
Packet Processing Time at the Cache Layer

Packet type	Packet processing time (ms)				
	0.9KB	1.3KB	1.9KB	3.2KB	6.4KB
Request	0.217	0.218	0.215	0.217	0.219
Reply	1.483	1.514	1.836	2.294	3.132

noncooperative cache (SimpleCache), and the symmetric approach reduces the data access latency by 12-18 percent compared to SimpleCache. This is because cooperative cache helps the requester to get data from nearby nodes when the data are not locally cached by the requester. For these two cooperative cache approaches, the asymmetric approach experiences on average 10 percent less data access delay compared to the symmetric approach. This delay reduction is achieved by reducing the number of times that the data item is passed from the network layer to the cooperative cache layer. In the symmetric approach, for any intermediate node, the received data item has to be passed to the cooperative cache layer which is in the user space. If this intermediate node does not need to cache the data, this context switch is not necessary. The asymmetric approach reduces the access delay by removing these unnecessary context switches.

The small-scale prototype has several limitations which make in-depth performance evaluation hard to perform. First, due to the small scale, the distance between the source and the destination is short, and hence, the advantage of cooperative caching is not as much as that shown in [24]. Second, N_4 and N_5 are the only two requesters in the network, and N_3 is the only node selected by the algorithm to do cooperative caching. A data item will be put into the cache of N_3 after it is accessed by either N_4 or N_5 (let's say N_4), and the cached data can only help N_5 once. Later both N_4 and N_5 can access this data from their local caches. All the cached data at N_3 can at most serve one request, thus the utilization of the cooperative cache is very low in this prototype. Fig. 9 shows the cache hit ratio in our experiment, which confirms the above findings. Third, since each node only has one wireless interface, due to interference, it is difficult to test the pipeline effect identified in Section 2.2.2. This also explains why the difference between symmetric and asymmetric approaches is relatively small, as the asymmetric approach only saves the processing delay at N_3.

Although our prototype has some limitations, it can be used to verify our simulation testbed, which will be shown in the next section. Further, it verifies one important delay factor: the data processing delay. To better evaluate the system performance such as the pipeline effect, we collect real data from the prototype and use the data to tune the simulation testbed to better simulate the real system performance.

In order to accurately evaluate the processing delay at the cache layer, we use results from our prototype, which are shown in Table 2. The cache layer processing delay of our simulator is strictly tuned to follow these values. The data processing delay is generally not considered in most network simulators, but it is a very important factor in our system design.

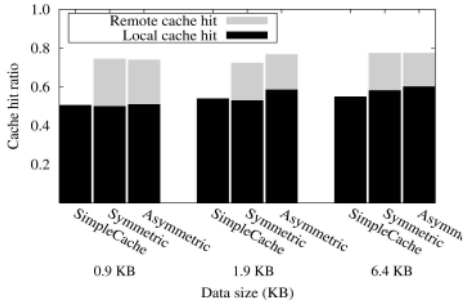


Fig. 9. Cache hit ratio of the three approaches at different data sizes.

4 Performance Evaluations

To evaluate our design and implementation in a large-scale network, and to evaluate how different factors affect the system performance, we perform extensive simulations. We also compare our approach with various design options.

4.1 Simulation Model

The simulation is based on *ns-2* [18]. The implementation of the cooperative cache layer is ported from the real system implementation, but simplified to fit the simulator.

At the transport layer, we set the MTU (Maximum Transmission Unit) to be 500 bytes. When the data packet received from the upper layer exceeds the size of MTU, it breaks the packet into fragments and passes the fragments to the routing layer. For the packets passed from the routing layer, the transport layer needs to reassemble the IP fragments.

The MAC layer. We simulate several MAC and physical layer options, and compare the performance of various cache designs. Table 3 shows the complete list of MAC and physical layer options. The basic wireless interface follows 802.11b standard. The radio transmission range is 250 m and the interference range is 550 m. We use the existing 802.11 MAC implementation included in the original ns-2 package as our single-interface single-channel MAC layer.

For the multi-interface multichannel MAC protocol, we assume each node is equipped with multiple 802.11b compatible interfaces. These interfaces can be tuned to multiple orthogonal channels [14], [15], [19], [21]. In this way, it is possible for a single node simultaneously sending and receiving packets using two independent radios, whereas neighboring nodes can also simultaneously transmit packets

TABLE 3
Wireless Interface Setup

Wireless Interface	Channel Bandwidth	
single-interface single-channel	2 Mbps	5Mbps
multi-interface multi-channel	2M bps	5Mbps

TABLE 4
Simulation Parameters

Parameter	Value
Simulation area	$4500m \times 600m$
Number of nodes	100
Communication range	250m
Interference range	550m
Query generate interval	$T_{query} = 5s$
MTU size	500B
Client cache size	800KB
Database size	1000 items
Data item size	$s_{min} = 100B$, $s_{max} = 7KB$

at other noninterfering channels. Since the standard ns-2 does not support multichannel, we add the multi-interface and multichannel functionality based on the techniques provided in [1].

The client query model. The client query model is similar to what has been used in previous studies [24]. 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, which has been frequently used [4] to model nonuniform distribution. In the Zipf-like distribution, the access probability of the ith $(1 \leq i \leq n)$ data item is represented as follows:

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

where $0 \leq \theta \leq 1$. When $\theta = 1$, it follows the strict Zipf distribution. When $\theta = 0$, it follows the uniform distribution. Larger θ results in more "skewed" access distribution. We choose θ to be 0.8 according to studies on real Web traces [4].

The access pattern of the wireless 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 $0, 1, 2, \ldots 19$ in a columnwise 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 has a range 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.

Most system parameters are listed in Table 4. For each workload parameter (e.g., the data size), the mean value of the measured data is obtained by collecting a large number of samples such that the confidence interval is reasonably small. In most cases, the 95 percent confidence interval for the measured data is less than 10 percent of the sample mean.

TABLE 5
Simulated Data Access Delay Using the Five-Node Topology

	0.9KB	1.3KB	1.9KB	3.2KB	6.4KB
SimpleCache	27.21	30.51	41.77	59.92	117.67
Symmetric	23.82	26.46	36.05	48.67	101.90
Asymmetric	11.78	24.03	32.01	45.09	94.24

We first verify our simulation testbed by configuring it with our five-node experimental topology. As shown in Table 5, the simulation results match that in the prototype experiment. Next, we increase the scale of our network using parameters listed in Table 4 to collect more results.

4.2 Simulation Results

In this section, we compare the performance of the SimpleCache approach, the Symmetric Cooperative Cache (SCC) approach, and the Asymmetric Cooperative Cache (ACC) approach in various network environments. Simple Cache is the traditional cache scheme that only caches the received data at the query node. We also compare these schemes to an Ideal Cooperative Cache (ICC) approach, which does not have processing delay at the cache layer. Further, upon receiving each packet, the cache manager makes a copy of the packet and buffers it, and then forwards the original one immediately. Thus, an intermediate node can immediately forward the packet without waiting until the whole data item is received, which can maximize the pipeline effect. It is easy to see that ICC sets up a performance upper bound that a cooperative cache scheme may achieve.

4.2.1 Comparisons in Traditional 802.11 Networks

Fig. 10a shows the average data access delay of different designs in transitional 802.11 ad hoc networks. The benefit of cooperative caching is easy to see when the channel bandwidth is low (2 Mbps) regardless of the cache design options. Cooperative cache increases the chance of getting data with less number of hops, and hence can reduce the access delay compared to the SimpleCache approach no matter how they are designed.

Fig. 11 provides a closer view to compare the performance of these three cooperative cache schemes (SCC, ACC, and ICC). As shown in the figure, the ACC approach is quite close to the performance of the ICC approach. The advantage of ACC and ICC over SCC is about 10-25 percent. Based on the results of Section 3, most of this performance gain comes from the processing delay but not from the pipeline effect. This is because the spatial channel reuse of 802.11 is extremely limited. Since a node can only receive or send packets at one time with one interface, it is impossible to have a perfect pipeline. The best we can achieve is to have a partial pipeline as the two-hop neighbor of the sending node can also send. However, this is not exactly true when considering the 802.11 DCF mechanism [11].

As shown in Fig. 12, a chain of nodes are positioned with 250 meters between neighboring nodes. Based on the 802.11b configuration in ns-2, the transmission range is 250 m, and the interference range is 550 m. Even with only

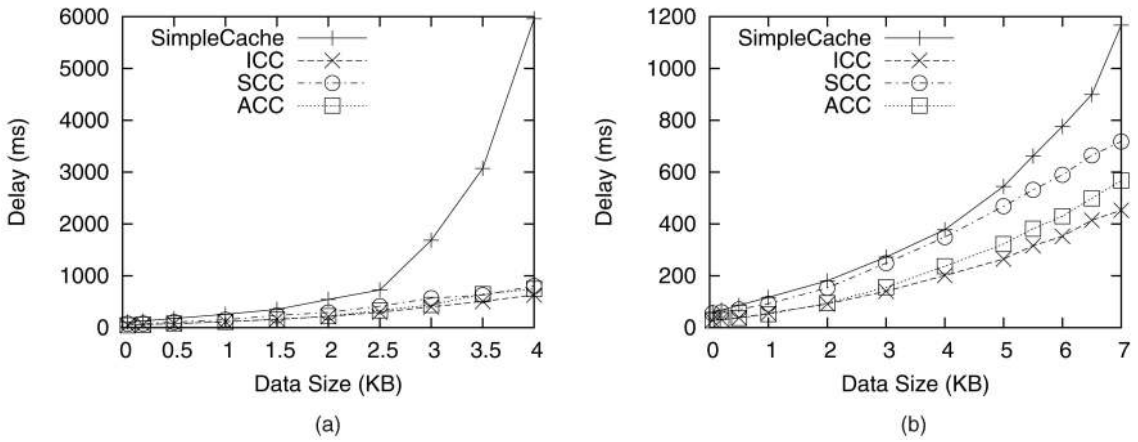


Fig. 10. Comparison of the data access delay in 802.11 networks. (a) 2 M bandwidth and (b) 5 M bandwidth.

one data flow going through N_1 to N_9, when node N_5 is transmitting a packet to N_6, the nearest possible concurrent transmission is between N_1 and N_2 because N_5's interference range covers nodes N_3, N_4, and N_7. Although N_2 is outside of the interference range of N_5, node N_3 is an exposed terminal of the on-going transmission $N_5 \rightarrow N_6$, so it cannot response to the RTS message from node N_2. Also node N_8 is a hidden terminal since it cannot decode N_6's CTS, and it cannot sense N_5's data transmission since N_5 is out of N_8's 550 m carrier sensing range. Thus, when node N_8 transmits to N_9, it will disrupt the on-going transmission from N_5 to N_6. Therefore, even with a perfect scheduling mechanism which can evenly spread out the packets along the path to maximize the packets pipeline, the maximum spatial reuse can be achieved is just $1/4$ of the chain length. In a real network where nodes are not perfectly positioned, other factors such as interference between concurrent flows, ACK packet in the reverse path when using TCP, and the actual spatial reuse with 802.11 are even worse. Similar problems have been identified in [10], [17], although in different context.

From Fig. 10a, we can also see that the average data access delay increases linearly to the data size. The data access delay of the SimpleCache approach significantly increases when the data size is larger than 3.5 KB. This is due to network congestion. In the SimpleCache approach, each data request needs to travel more hops to be served compared to that in the cooperative cache schemes. As a result, each data request uses more network bandwidth, and the chance of network congestion is higher. In case of network congestion, the data access delay significantly increases.

By increasing the transmission rate to 5 Mbps, as shown in Fig. 10b, the network capacity increases, and there is no network congestion in SimpleCache even when the data size increases to 7 KB. From Fig. 10b, we can also see that the SCC approach does not have too much advantage over the SimpleCache approach. There are two reasons: First, as the data transmission rate increases, the processing overhead of the SCC approach becomes significant. Second, as the data transmission rate increases, it starts to have some pipelines, but the SCC approach does not allow any pipelines. The ACC approach does not have these disadvantages and hence still has much better performance compared to the SimpleCache approach.

The Ideal Cooperative Cache (ICC) approach allows pipeline and has no processing overhead. Hence, it has the lowest data access delay. The delay of the ACC approach is quite close to optimal, which verifies that the asymmetric approach is quite effective on mitigating the cache layer overhead. It has almost the same delay as the ideal cooperative cache approach when the data size is not much larger than MTU, and there are normally not enough packets to fill in the "pipe" along the forwarding path. As the data size increases, the ACC approach has a little bit longer delay than the ICC approach, since the caching nodes stop the pipeline. But it is still much better than the SCC approach.

Fig. 13 further explains why the cooperative cache schemes can reduce the data access delay. To get this figure, we set the average data size to be 5 KB and change the cache size. We use a similar cache replacement algorithm as that in [24]. As cache size increases, the remote data hit ratio of the cooperative cache scheme (ACC, ICC, or SCC) increases.

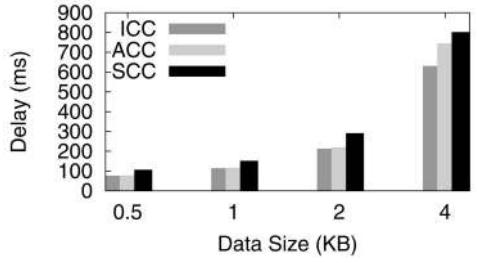


Fig. 11. A close view of query latency for different cooperative cache schemes in single-interface-single-channel 802.11 2 M ad hoc networks.

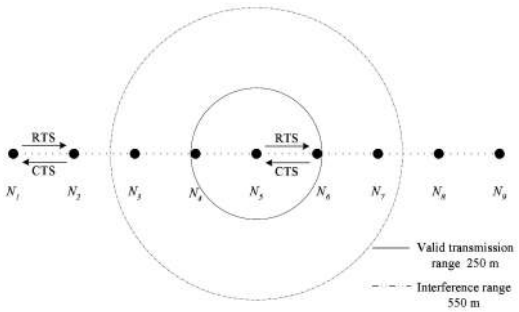


Fig. 12. 802.11 MAC layer interference in a chain topology.

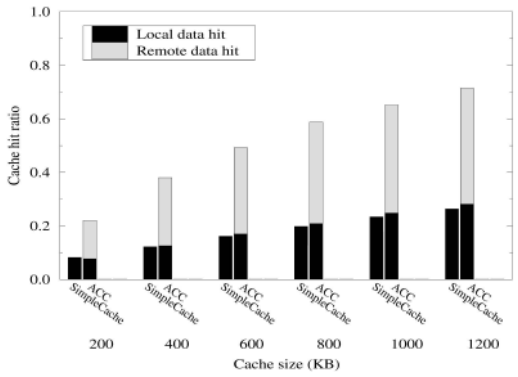


Fig. 13. Comparison of cache hit ratio between SimpleCache and ACC (in 802.11 2 M ad hoc networks).

Although the local data hit ratio is similar between SimpleCache and ACC, ACC can get data through remote data hit and, hence, save some hops of transmission delay. When pipeline is not very effective, especially in Fig. 10a, the cooperative cache approach can significantly reduce the delay. The comparison between Fig. 13 and Fig. 9 (see Section 3.2) also explains why the delay reduction using cooperative cache in the simulation is much higher than that in our prototype. As the network size increases, the possibility of accessing data from the cache of other nodes greatly increases for ACC. Also, the local cache hit ratio decreases when the data set becomes larger.

4.2.2 Comparisons in Wireless Mesh Networks

Multi-interface multichannel wireless mesh network is designed to increase the bandwidth utilization and allow neighbor nodes communicate concurrently. As a result, it is easier for the nodes to take advantage of the pipeline effect. When a large data item is transmitted in network, it is fragmented into small packets. These packets can be pipelined along the forwarding path to maximize the throughput and reduce the data access delay. As shown in Fig. 14, due to data pipeline, the SimpleCache approach may outperform the SCC approach. This is because, as discussed in the last section, the SCC approach has high processing overhead and it is lack of pipeline.

In Fig. 14a, when the data size is larger than 6 KB, the SimpleCache approach still runs into severe congestion due to excessive packets injected to the network. As shown in Fig. 14b, the performance improvement of ACC over ICC drops as the data size increases. This can be explained as follows. The major benefit of cooperative caching is to reduce the hop distance of data access. This will be translated into the reduction of data access delay in 802.11-based network. However, this is not exactly true in high-bandwidth multichannel mesh networks. In such networks, as long as the data item is large enough for a full pipeline, the hop distance becomes less important to the data access delay. Although caching in the intermediate node can reduce the hop distance for future data access, this delay reduction is less important. Further, it is at the cost of shortening the pipeline due to caching in the intermediate node. Even considering these constraints, the ACC approach outperforms the SimpleCache approach and is very close to the ideal cooperative cache approach.

From Fig. 14b, we can see that the delay advantage of the cooperative cache approaches is not that significant in high-bandwidth multichannel mesh networks. This is because the network has enough bandwidth to support all the nodes. However, as the nodes increase the query rate or access data of larger size, the delay of the SimpleCache becomes much higher. Similar results have been shown in Fig. 14a. Although the pipeline can reduce the delay, the SimpleCache approach still generates more traffic, which may result in a network congestion and longer delay. As shown in Fig. 15, the cooperative cache schemes (ICC, SCC, ACC) generate 30-50 percent less data traffic than the SimpleCache approach because cooperative cache can reduce the number of hops to get the data.

4.2.3 The Effect of Cache Placement

In this section, we evaluate the greedy cache node selection algorithm proposed in Section 2.3.3. In ACC, the data server uses this algorithm to determine which node should cache the data on the forwarding path. We compare our algorithm with the existing caching node selection algorithm in [24], which relies on the intermediate forwarding nodes to make the caching decision independently. More specifically, the existing algorithm in [24] is solely based on a node's local information, and it relies on a benefit threshold to decide

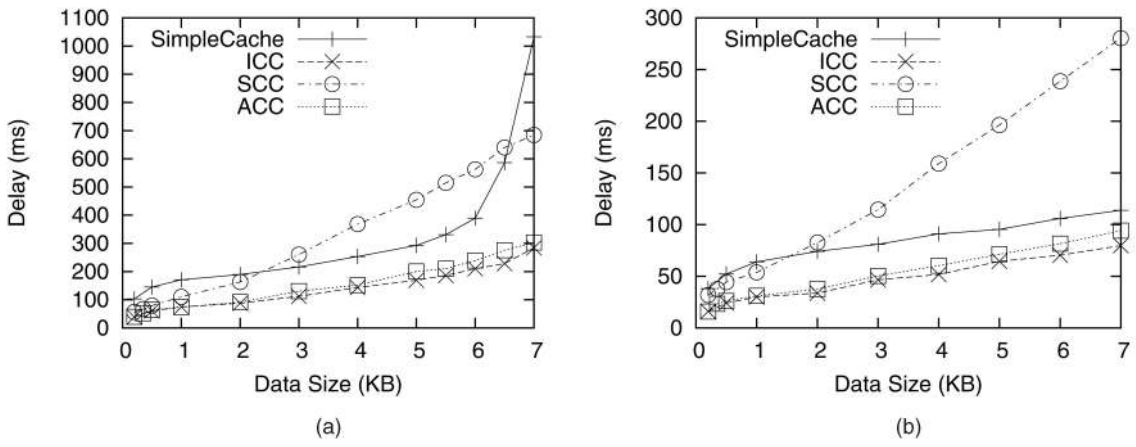


Fig. 14. Comparison of the data access delay in wireless mesh networks. (a) 2 M bandwidth and (b) 5 M bandwidth.

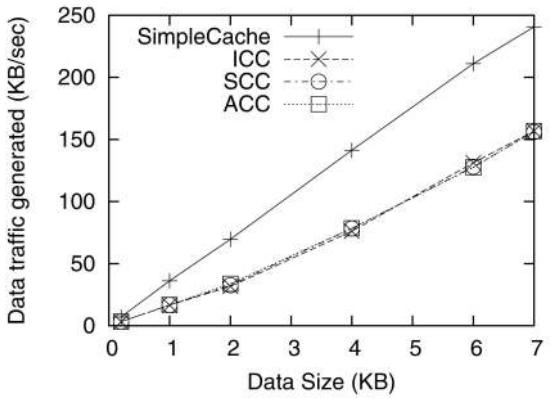


Fig. 15. Comparison of the data traffic generated in 5 M mesh networks.

whether to cache the data. Two versions of such algorithm are chosen for the purpose of comparison: 1) Local Aggressive: the benefit threshold is set low so that forwarding nodes are more likely to cache the data; and 2) Local Lazy: the benefit threshold is set high so that the forwarding nodes are less likely to cache the data. We compare their performance in both 802.11 networks and multi-interface multichannel mesh networks with 5 M bandwidth. The results are based on the data size of 4 KB.

Figs. 16 and 17 show the average data access hop distance and data access delay, respectively. In both 802.11 and mesh networks, Local Aggressive generates more cache copies for data, so the average data access hop distance is the smallest among the three. In contrast to Local Aggressive, much less nodes are qualified to cache data with Local Lazy algorithm. So, Local Lazy has the longest hop distance. Intuitively, data access should have shorter delay when data are cached by more nodes, and Local Aggressive should be more favorable. This is only true in 802.11 networks (as shown in Fig. 17). However, in mesh networks, overly caching data may cause adverse effects since it can increase the cache processing delay and eliminate the pipeline effect, as we have discussed previously. Local Lazy only picks a small amount of nodes which receive request messages most frequently to cache the data. In this way, it reduces the caching overhead and obtains less delay than Local Aggressive in mesh networks.

Our algorithm considers the trade-off between reducing hop distance and the caching overhead in different network structures. In 802.11 networks, it tries more aggressively to cache data because the gain of caching is more than the

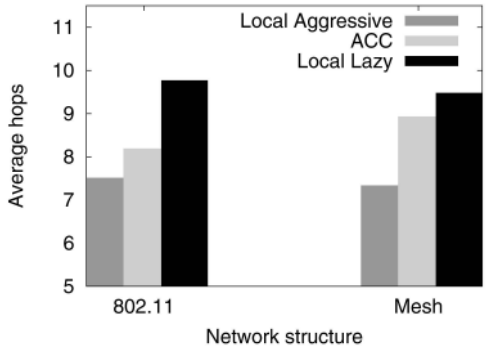


Fig. 16. The effect of cache node selection algorithms on the data access hops.

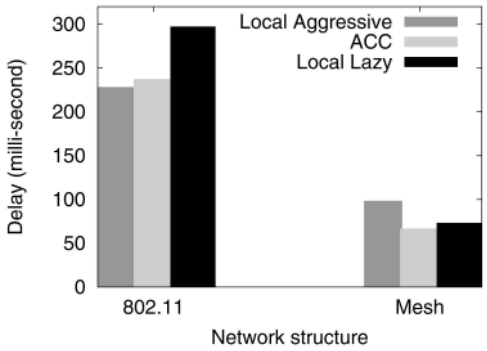


Fig. 17. The effect of cache node selection algorithms on the data access delay.

overhead of caching. Thus, the average data access hop distance of ACC is much less than that of the Local Lazy algorithm (in Fig. 16), and its delay is similar to Local Aggressive (in Fig. 17). In mesh networks, caching overhead becomes more significant, so data caching should be used more prudently. ACC adapts its caching strategy to select less number of caching nodes. Its data access hop distance increases, and it achieves similar data access delay as Local Lazy, which is much less than Local Aggressive.

Although Local Aggressive and Local Lazy algorithms can achieve good performance in the simulated 802.11 and mesh networks, respectively, they are unable to determine a proper strategy (e.g., calculate a proper benefit threshold) by themselves. Particularly in a practical network where different degrees of channel interference exist, it is impossible to have a threshold only based on local information and still obtain good performance. ACC considers the network condition of the whole data forwarding path, and therefore can select proper nodes to cache data and achieve better performance.

5 Conclusions

In this paper, we presented our design and implementation of cooperative cache in wireless P2P networks, and proposed solutions to find the best place to cache the data. In our asymmetric approach, data request packets are transmitted to the cache layer on every node; however, the data reply packets are only transmitted to the cache layer on the intermediate nodes which need to cache the data. This solution not only reduces the overhead of copying data between the user space and the kernel space, but also allows data pipeline to reduce the end-to-end delay. We have developed a prototype to demonstrate the advantage of the asymmetric approach. Since our prototype is at a small scale, we evaluate our design for a large-scale network through simulations. Our simulation results show that the asymmetric approach outperforms the symmetric approach in traditional 802.11-based ad hoc networks by removing most of the processing overhead. In mesh networks, the asymmetric approach can significantly reduce the data access delay compared to the symmetric approach due to data pipelines.

To the best of our knowledge, this is the first work on implementing cooperative cache in wireless P2P networks, and the first work on identifying and addressing the effects of data pipeline and MAC layer interference on cache management. We believe many of these findings will be valuable for making design choices.

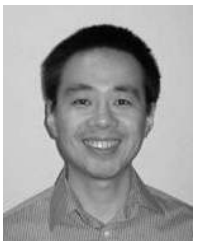
ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers whose insightful comments helped us improve the presentation of the paper. A preliminary version [25] of the paper appeared in IEEE ICDCS '08. This work was supported in part by the US National Science Foundation (NSF) under grant number CNS-0721479, and by Network Science CTA under grant W911NF-09-2-0053.

REFERENCES

[1] R. Agüero and J.P. Campo, "Adding Multiple Interface Support in NS-2," Jan. 2007.
[2] B. Barr, J. Bicket, D. Dantas, B. Du, T. Kim, B. Zhou, and E. Sirer, "On the Need for System-Level Suppport for Ad Hoc and Sensor Networks," *ACM Operating System Rev.*, vol. 36, no. 2, pp. 1-5, Apr. 2002.
[3] J. Bicket, D. Aguayo, S. Biswas, and R. Morris, "Architecture and Evaluation of an Unplanned 802.11b Mesh Network," *Proc. ACM MobiCom*, 2005.
[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," *Proc. IEEE INFOCOM*, 1999.
[5] G. Cao, L. Yin, and C. Das, "Cooperative Cache-Based Data Access in Ad Hoc Networks," *Computer*, vol. 37, no. 2, pp. 32-39, Feb. 2004.
[6] M. Cieslak, D. Foster, G. Tiwana, and R. Wilson, "Web Cache Coordination Protocol v2.0," IETF Internet Draft, 2000.
[7] S. Desilva and S. Das, "Experimental Evaluation of a Wireless Ad Hoc Network," *Proc. Ninth Int'l Conf. Computer Comm. and Networks*, 2000.
[8] H. Eriksson, "MBONE: The Multicast Backbone," *Comm. ACM*, vol. 37, no. 8, pp. 54-60, 1994.
[9] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide Area Web CAche Sharing Protocol," *Proc. ACM SIGCOMM*, pp. 254-265, 1998.
[10] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, and M. Gerla, "The Impact of Multihop Wireless Channel on TCP Throughput and Loss," *Proc. IEEE INFOCOM*, 2003.
[11] IEEE, *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec*, IEEE 802.11 Standard, 1999.
[12] D. Johnson and D. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Network," *Mobile Computing*, pp. 153-181, Kluwer Academic Publishers, 1996.
[13] V. Kawadia, Y. Zhang, and B. Gupta, "System Services for Ad-Hoc Routing: Architecture, Implementation and Experiences," *Proc. Int'l Conf. Mobile Systems, Applications and Services (MobiSys)*, 2003.
[14] M. Kodialam and T. Nandagopal, "Characterizing the Capacity Region in Multi-Radio, Multi-Channel Wireless Mesh Networks," *Proc. ACM MobiCom*, 2005.
[15] P. Kyasanur and N.H. Vaidya, "Routing and Link-Layer Protocols for Multi-Channel Multi Interface Ad Hoc Wireless Networks," *ACM SIGMOBILE Mobile Computing and Comm. Rev.*, vol. 10, no. 1, pp. 31-43, 2006.
[16] W. Lau, M. Kumar, and S. Venkatesh, "A Cooperative Cache Architecture in Supporting Caching Multimedia Objects in MANETs," *Proc. Fifth Int'l Workshop Wireless Mobile Multimedia*, 2002.
[17] J. Li, C. Blake Douglas, S.J. De Couto, H.I. Lee, and R. Morris, "Capacity of Ad Hoc Wireless Networks," *Proc. ACM MobiCom*, 2001.
[18] ns Notes and Documentation, http://www.isi.edu/nsnam/ns/, 2002.
[19] J. Padhye, R. Draves, and B. Zill, "Routing in Multi-radio, Multi-hop Wireless Mesh Networks," *Proc. ACM MobiCom*, 2004.
[20] 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.
[21] A. Raniwala and T. Chiueh, "Architecture and Algorithms for an IEEE 802.11-Based Multi-Channel Wireless Mesh Network," *Proc. IEEE INFOCOM*, 2005.
[22] E. Royer and C. Perkins, "An Implemenatation Study of the AODV Routing Protocol," *Proc. IEEE Wireless Comm. and Networking Conf.*, 2000.
[23] B. Tang, H. Gupta, and S. Das, "Benefit-Based Data Caching in Ad Hoc Networks," *IEEEE Trans. Mobile Computing*, vol. 7, no. 3, pp. 289-304, Mar. 2008.
[24] L. Yin and G. Cao, "Supporting Cooperative Caching in Ad Hoc Networks," *IEEE Trans. Mobile Computing*, vol. 5, no. 1, pp. 77-89, Jan. 2006.
[25] J. Zhao, P. Zhang, and G. Cao, "On Cooperative Caching in Wireless P2P Networks," *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS)*, pp. 731-739, June 2008.

Jing Zhao received the BS degree from Peking University, Beijing, China. He is currently pursuing the PhD degree in computer science and engineering at the Pennsylvania State University. His research interests include distributed systems, wireless networks, and mobile computing, with a focus on mobile ad hoc networks. He is a student member of the IEEE.



Ping Zhang received the MS degree from the Pennsylvania State University. His research interests include wireless networks and mobile computing, with a focus on mobile ad hoc networks.

Guohong Cao received the BS degree from Xian Jiaotong University, Xian, China. He received the MS and PhD degrees in computer science from the Ohio State University in 1997 and 1999, respectively. Since then, he has been with the Department of Computer Science and Engineering at the Pennsylvania State University, where he is currently a full professor. His research interests are wireless networks and mobile computing. He has published more than 100 papers in the areas of sensor networks, wireless network security, data dissemination, resource management, and distributed fault-tolerant computing. He has served on the editorial board of the *IEEE Transactions on Mobile Computing* and *IEEE Transactions on Wireless Communications*, and has served on the program committee of many conferences. He was a recipient of the US National Science Foundation CAREER Award in 2001. He is a senior member of the IEEE.

Chita R. Das received the MSc degree in electrical engineering from the Regional Engineering College, Rourkela, India, in 1981, and the PhD degree in computer science from the Center for Advanced Computer Studies, University of Louisiana, Lafayette, in 1986. Since 1986, he has been with Pennsylvania State University, where he is currently a professor in the Department of Computer Science and Engineering. His main areas of interest are parallel and distributed computer architectures, cluster computing, mobile computing, Internet QoS, multimedia systems, performance evaluation, and fault-tolerant computing. He has served on the editorial boards of the *IEEE Transactions on Computers* and *IEEE Transactions on Parallel and Distributed Systems*. He is a fellow of the IEEE.