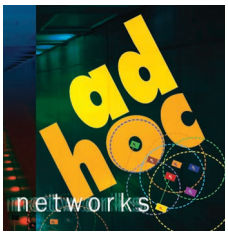
Cooperative Cache-Based Data Access in Ad Hoc Networks



A cooperative cache-based data access framework lets mobile nodes cache the data or the path to the data to reduce query delays and improve data accessibility.

Guohong Cao
Liangzhong Yin
Chita R. Das
Pennsylvania State University

Mobile ad hoc networks have potential applications in civilian and military environments such as disaster recovery efforts, group conferences, wireless offices, mobile infostations (in tourist centers, restaurants, and so on), and battlefield maneuvers, making them a focus of current research.

A battlefield ad hoc network might consist of several commanding officers and a group of soldiers. The soldiers could access officers' information centers for detailed geographic information, information about the enemy, new commands, and so on. Because neighboring soldiers tend to have similar missions and thus common interests, several soldiers might need to access the same data at different times. Having a nearby soldier serve later accesses to this data instead of the faraway information center saves battery power, bandwidth, and time.

In ad hoc networks, mobile nodes communicate with each other using multihop wireless links. Due to a lack of infrastructure support, each node acts as a router, forwarding data packets for other nodes. Most previous research in ad hoc networks focused on the development of dynamic routing protocols that can efficiently find routes between two communicating nodes. Although routing is an important issue, the ultimate goal of ad hoc networks is to provide mobile nodes with access to information.

If mobile users around infostations, which have limited coverage, form an ad hoc network, a mobile user who moves out of the range of a particular infostation can still access the data it contains. If one of the nodes along the path to the data source has a cached copy of the requested data, it can forward the data to the mobile user, saving bandwidth and power. Thus, if mobile nodes can work as request-forwarding routers, they can save bandwidth and power and reduce delays.

Cooperative caching, in which multiple nodes share and coordinate cached data, is widely used to improve Web performance in wired networks. The "Related Work in Cooperative Caching" sidebar provides additional information about recent research focusing on cooperative caching approaches for wired networks. However, resource constraints and node mobility have limited the application of these techniques in ad hoc networks.

Our proposed caching techniques—CachePath, CacheData, and HybridCache—use the underlying routing protocols to overcome these constraints and further improve performance by caching the data locally or caching the path to the data to save space.

To increase data accessibility, mobile nodes should cache different data items than their neighbors. Although this increases data accessibility, it also can increase query delays because the nodes might have to access some data from their neighbors instead of accessing it locally. In addition, replicating data from the server could create security problems.

As Figure 1 illustrates, the cooperative cache-

Published by the IEEE Computer Society
0018-9162/04/$20.00 © 2004 IEEE

Related Work in Cooperative Caching

Existing cooperative caching schemes for the Web environment can be classified as message-based, directory-based, hash-based, or router-based.

Duane Wessels and Kim Claffy introduced the standardized and widely used Internet cache protocol.[1] As a message-based protocol, ICP supports communication between caching proxies using a simple query-response dialog.

Directory-based protocols for cooperative caching—such as cache digests[2] and summary cache[3]—let caching proxies exchange information about cached content.

The cache array routing protocol is the most notable hash-based cooperative caching protocol. The rationale behind CARP constitutes load distribution by hash routing among Web proxy cache arrays.

As a router-based protocol, the Web cache coordination protocol transparently distributes requests among a cache array. Because these protocols usually assume fixed network topology and often require high computation and communication overhead, they might be unsuitable for ad hoc networks.

To tolerate network partitions and improve data accessibility, Takahiro Hara proposed several replica allocation methods for ad hoc networks.[4] In Hara's schemes, a node maintains replicas of data that is frequently requested. The data replicas are relocated periodically based on three criteria: access frequency, neighbor nodes' access frequency, or overall network topology. Later, Hara proposed schemes to deal with data updates. Although data replication can improve data accessibility, significant overhead is associated with maintaining and redistributing the replicas, especially in ad hoc networks.

Maria Papadopouli and Henning Schulzrinne[5] proposed a 7DS architecture similar to cooperative caching, which defines two protocols to share and disseminate data among users experiencing intermittent Internet connectivity. It operates on a prefetch mode to gather data for serving the user's future needs or on an on-demand mode to search for data on a single-hop multicast basis. The 7DS architecture focuses on data dissemination instead of cache management. Further, it focuses on a single-hop rather than a multihop environment.

References

1. D. Wessels and K. Claffy, "ICP and the Squid Web Cache," *IEEE J. Selected Areas in Comm.*, Mar. 1998, pp. 345-357.
2. A. Rousskov and D. Wessels, "Cache Digests," *Computer Networks and ISDN Systems*, vol. 30, 1998, pp. 2155-2168.
3. L. Fan et al., "Summary Cache: A Scalable Wide Area Web Cache Sharing Protocol," *Proc. ACM SIGCOMM*, ACM Press, 1998, pp. 254-265.
4. T. Hara, "Effective Replica Allocation in Ad Hoc Networks for Improving Data Accessibility," *Proc. IEEE Infocom*, IEEE CS Press, 2001, pp. 1568-1576.
5. M. Papadopouli and H. Schulzrinne, "Effects of Power Conservation, Wireless Coverage and Cooperation on Data Dissemination among Mobile Devices," *Proc. MobiHoc*, ACM Press, 2001, pp. 117-127.

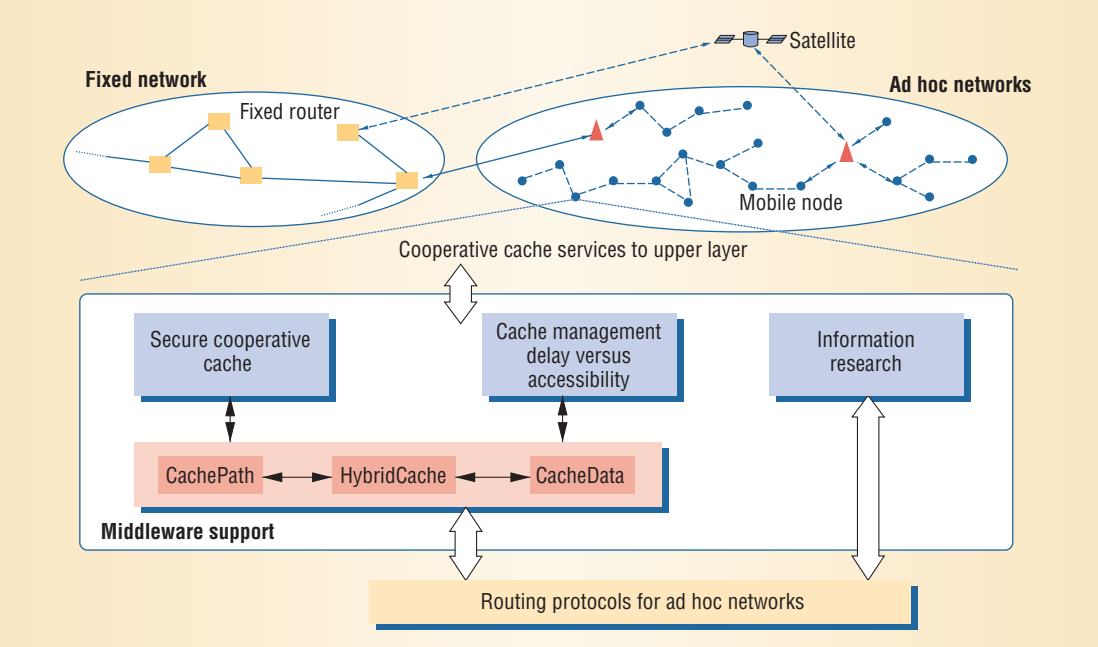


Figure 1. Cooperative caching in ad hoc networks. Middleware support mechanisms provide secure cooperative caching, cache management, and information search.

based framework stays on top of the routing protocols. It relies on several components, such as secure cooperative caching, cache management, and information search to provide services to the upper layer.

ROUTER SUPPORT FOR COOPERATIVE CACHING

Suppose that in the ad hoc network in Figure 2, N_{11} is a data source containing a database of n items d_1, d_2, …, and d_n. N_{11} might be a connecting

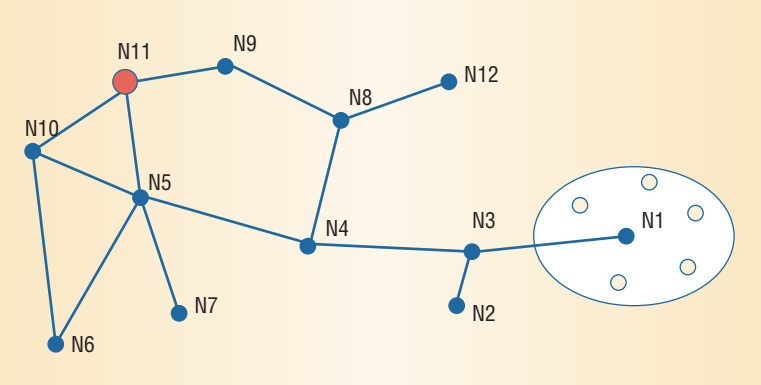


Figure 2. Ad hoc network. Node N_{11} is a data source and the blue nodes are router nodes. Node N_1 is a cluster head surrounded by mobile nodes.

node to the wired network with the database. The blue nodes are router nodes, which can be cluster heads if a cluster-based routing protocol is used; otherwise, they are just mobile nodes. Around each cluster head (as illustrated by node N_1) are several mobile nodes, or cluster members. To reduce bandwidth and power consumption, the number of hops between the data source and the requesting node should be as small as possible.

Routing protocols can help reduce bandwidth and power consumption to a limited degree. Our proposed caching techniques use the underlying routing protocols to further improve performance.

CachePath and CacheData concepts

Figure 2 illustrates the CachePath concept. Suppose node N_1 requests a data item d_i from N_{11}. 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_{11} is three hops away whereas N_1 is only one hop away. Thus, N_3 forwards the request to N_1 instead of N_4. Many routing algorithms provide the hop count information between the source and destination. Caching the data path for each data item reduces bandwidth and power because nodes can obtain the data using fewer hops. However, mapping data items and caching nodes increases routing overhead. We propose various optimization techniques to improve CachePath's performance.

In CachePath, a node need not record the path information of all passing data. Rather, it only records the data path when it's closer to the caching node than the data source. For example, when N_{11} 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's path information because they're closer to the data source than the caching node N_1.

In a mobile network, the node caching the data might move or it might replace the cached data because of cache size limitations. Consequently, the node modifying the route should reroute the request to the original data source after discovering that the node moved or replaced the data. Thus, the cached path might be unreliable, and using it can increase the overhead.

To deal with this issue, in our approach 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, the routing node should cache the data path.

In CacheData, the router node caches the data instead of the path when it finds that the data is frequently accessed. For example, in Figure 2, if both N_6 and N_7 request d_i through N_5, N_5 might 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_{11}. The nodes along the path—N_3, N_4, and N_5—might want to cache d_i as a frequently accessed item. However, they'll 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 were from N_4, and those requests in turn came 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. If the requests all come from N_1, N_3 won't cache the data but N_1 will, or the requesting node in N_1's cluster will cache the data if it's the only requesting node. Certainly, if N_5 later receives requests for d_i from N_6 and N_7, it can also cache the data.

Maintaining cache consistency. To handle cache consistency, CachePath and CacheData use a simple weak consistency model based on the time-to-live mechanism. In this model, a routing node considers a cached copy up-to-date if its TTL hasn't expired. If the TTL has expired, the node removes the map from its routing table (or removes the cached data). As a result, the routing node forwards future requests for this data to the data source. We optimize this model by allowing nodes to refresh a cached data item if a fresh copy of the same data passes by. If the fresh copy contains the same data but a newer TTL, the node updates only the cached data's TTL field. If the data center has updated the data item, the node replaces both the cached data item and its TTL with the fresh copy. When strong cache consistency is needed, we adopt techniques reported in earlier work.[1]

Locating cached data within a cluster. To save power, many routing protocols divide areas into clusters (or grids), with only one node in the cluster active while others sleep. To avoid network partitioning and maintain fairness, all nodes in the cluster alternate the role of the cluster head, or *coordinator*. Nodes can also leave or join the cluster. In such an environment, to ensure that all nodes can share the cached data, the cluster head needs to know which node caches which data.

A simple solution has each node send its cache data IDs to the cluster head. Most data IDs are very long, so sending the cached data IDs can consume a lot of bandwidth and power. *Cache digests*—a lossy compression of all cache keys with a lookup capability—offer a low overhead option that facilitates data searching in cluster-based ad hoc networks.[2] A node can check another node's digests to discover (with some uncertainty) whether it holds a given data item. When a node joins a cluster, it sends its cache digests to the cluster head. During the query reply phase, the cluster head calculates the cache digests based on the data ID and updates relevant information accordingly. If the current cluster head needs to sleep, it sends the cache digests to the new cluster head.

By caching the data or the data path, a nearby node can serve requests instead of the distant data center. This reduces query latency as well as bandwidth and power consumption because fewer nodes are involved in the query process. In addition, because the data center handles fewer requests, the workload is spread over the network, reducing the load on the data center.

The HybridCache approach

CachePath and CacheData can significantly improve system performance. Our analysis showed that CachePath performs better when the cache is small or the data update rate is low, while CacheData performs better in other situations.[3]

To further improve performance, we propose HybridCache, a hybrid scheme that exploits the strengths of CacheData and CachePath while avoiding their weaknesses. Specifically, when a mobile node forwards a data item, it caches the data or path based on some criteria. These criteria include the data item size s_i and the TTL time TTL_i.

For a data item d_i, we use the following heuristics to decide whether to cache data or the path:

- If s_i is small, CacheData is optimal because the data item only needs a small part of the available cache; otherwise, CachePath is preferable because it saves cache space. We denote the data size threshold as $\mathcal{T}_s$.
- If TTL_i is small, CacheData is preferable. Because the data item might soon be invalid, using CachePath can result in chasing the wrong path and having to resend the query to the data center. For large TTL_is, however, CachePath is acceptable. We denote the TTL threshold value as $\mathcal{T}_{TTL}$.

```
(A) When a data item di arrives:
        if di is the requested data by the current node
        then cache di
        else /* Data passing by */
        if there is a copy of di in the cache
        then update the cached copy if necessary
        else if si < Ts or TTLi < TTTL then
            cache data item di and the path;
        else if there is a cached path for di, then
            cache data item di;
            update the path for di;
        else
            cache the path of di;

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

Figure 3. HybridCache concept. The hybrid caching scheme caches the data or path based on criteria such as data item size and TTL time.

To achieve better performance, the threshold values used in these heuristics must be set carefully.[3] Figure 3 shows the algorithm applying these heuristics in the HybridCache scheme.

In CachePath, caching a data path only requires saving a node ID in the cache, which has a very small overhead. As a result, in HybridCache, when a node caches a data item d_i using CacheData, it also caches d_i's path. Later, if the cache replacement algorithm decides to remove d_i, it removes the cached data but keeps the path. Thus, CacheData degrades to CachePath for d_i. Similarly, CachePath can upgrade to CacheData again when data item d_i passes by.

When TTL expires, some cached data can be invalidated. Usually, the node removes such invalid data from the cache. However, invalid data items can be useful. For example, caching the data indicates the mobile node's interest in it.

When forwarding a data item, if a mobile node finds an invalid copy of that data in its cache, it caches the new copy for future use. To save space, when a cached data item expires, the mobile node removes the item from the cache, keeping the data's path in invalid state to indicate its interest. Because the mobile node's interest can change, it should not keep the expired path in the cache forever. In our design, if an expired path or data item has not been refreshed for the duration of its original TTL time

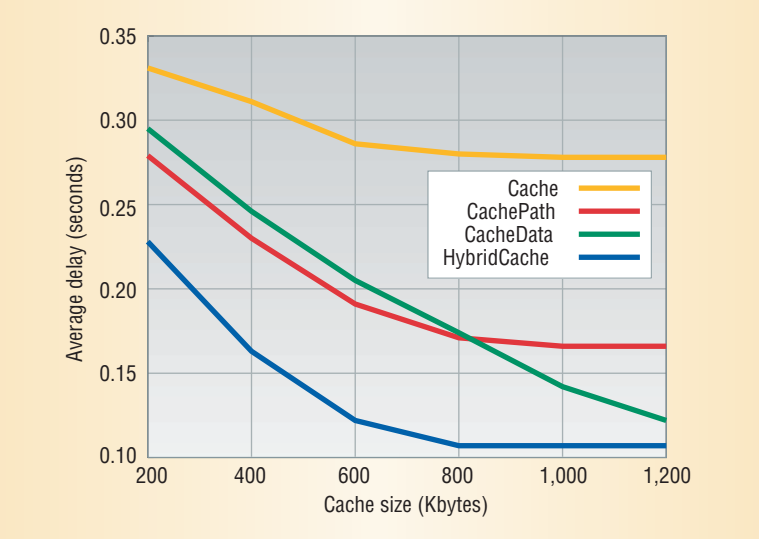


Figure 4. The average query delay as a function of the cache size. HybridCache performs better than both CacheData and CachePath because it combines the strengths of the two approaches.

(set by the data center), the node removes it from the cache.

Experimental results

To compare the performance of CachePath, CacheData, HybridCache, and traditional caching, in which a mobile node only caches the data it requests from the data center, we developed a simulation platform based on a simulator called ns-2[3].

Figure 4 shows the impact of cache size on average query delay. When the cache is small, CachePath performs better than CacheData. When the cache is more than 800 Kbytes, CacheData performs better because it uses more cache space to save passing data.

HybridCache performs better than either approach because it applies either CacheData or CachePath to different data items. HybridCache dynamically switches between CacheData and CachePath at the data item level, not at the database level.

PROACTIVE COOPERATIVE CACHING

With HybridCache, router nodes help other mobile nodes get the requested data quickly. A mobile node doesn't know whether the data source or some other nodes serve its request. If multiple data sources exist, or if the mobile node doesn't know where the data source is, HybridCache might not be a good option. In addition, caching nodes outside the path between the requesting node and the data source might not be able to share cache information with the requesting node.

Proactive cooperative caching, in which the requesting node actively searches for data from other nodes, is a possible solution. Rather than locating data within a cluster, the information search can go through multiple hops.

In proactive cooperative caching, the requesting node broadcasts a request to its neighbor nodes. If a node receiving the request has the data in its local cache, it sends an acknowledgment (ACK) to the requesting node; otherwise, it forwards the request to its neighbors. In this way, a request is flooded to other nodes and eventually acknowledged by the data source or a node with the cached copy.

The requesting node uses the arriving order of ACKs from the mobile nodes to select a path to the requested data. For example, if the requesting node gets an ACK from N_i earlier than it gets one from N_j, it requests the data from N_i. To improve performance, if the data size is small, it can be piggybacked with the ACK.

Flooding can create problems such as redundancy, contention, and collision—collectively referred to as the *broadcast storm* problem.[4] When a node receives multiple broadcast requests, many of its neighbors have already sent out broadcast requests, and an extra broadcast might not add much additional coverage. However, this extra broadcast can increase network contention and collision, thus the broadcast might be unnecessary. Techniques used to reduce the broadcast storm problem can be applied to information search to reduce flooding overhead. Other techniques such as dominant pruning[5] and dominating sets[6] can be used to further reduce collision, contention, and redundancy problems.

Because of flooding overhead, most information searches should be limited to some range instead of the entire network. One simple solution is based on the hop counter concept used in routing protocols. When the hop counter reaches some threshold, nodes stop flooding.

Setting up the hop counter is challenging. A hop counter that is too small reduces network traffic, but at the cost of not finding the data. A hop counter that is too large increases the probability of finding the data, but it also increases network traffic.

Intuitively, if the data size is very large or if the requesting node is closer to the caching node than the data source, the information search overhead might pay off. Otherwise, limiting the search range is preferable. Thus, the hop counter's value is related to the data size, flooding message size, distance to the caching node, and distance to the source node.

Information search is always valuable in network partitions, because the data may be found through information search when the data center is not accessible. However, because information search creates a large amount of network traffic, it should be used only when the data source is unreachable, using the basic cooperative cache scheme otherwise.

CACHE MANAGEMENT

Cache management is more complex in cooperative caching because deciding what to cache can also depend on the node's neighbors. Cooperative caching presents two problems: cache replacement and cache admission control.

Cache replacement algorithms

When the cache is full, cache replacement algorithms can find a suitable subset of data items to evict from the cache. Cache replacement algorithms have been extensively studied in operating systems, virtual memory management, and database buffer management. However, these algorithms might be unsuitable for ad hoc networks for several reasons:

- Because the data item size is not fixed in wireless environments, the least recently used policy must be extended to handle data items of varying sizes.
- The data item's transfer time might depend on the item's size and the distance between the requesting node and the data source (or cache). Consequently, the cache hit ratio might not be the most accurate measurement of a cache replacement algorithm's quality.
- The cache replacement algorithm should also consider cache consistency—that is, data items that tend to be inconsistent earlier should be replaced earlier. For example, one item is accessed 1 percent of the time at a particular client and is also updated 1 percent of the time.[7] A second item is accessed 0.5 percent of the time at the client, but updated only 0.1 percent of the time. In this example, the LRU algorithm would replace the second item and keep the first. However, keeping the second item might result in better performance.

Most cache replacement algorithms designed for the Web are function-based, employing factors such as time since last access, the item's entry time in the cache, transfer time cost, and item expiration time. Most of these cost functions are valid only for the traces used, however, and are not generic enough for insightful observations. These algorithms work relatively well under a certain goal—for example, response time. When the goal changes, they must produce another experience function.

Environments in which data accessibility is a concern require an enhanced cache replacement policy. Here, we consider two factors in selecting the data item (victim) to be replaced. The first factor is the distance (δ)—the number of hops away from the data source or caching node. Because δ is closely related to latency, if the node selects the data item with a higher δ as the victim, access latency will be high. Therefore, it selects the data item with the lowest δ.

Node mobility means the network topology can constantly change.

The second factor is data access frequency. Node mobility means the network topology can constantly change. As the topology varies, the δ value becomes obsolete. Therefore, we use a parameter τ that captures the time elapsed since the last δ update. We obtain τ by $1/(t_{cur} - t_{update})$, where t_{cur} and t_{update} are δ's current and last updated times for the data item. If τ is close to 1, δ has recently been updated. If τ is close to 0, the update gap is long. Thus, we use τ as an indicator of δ to select a victim. Using these two factors, we can design different cache replacement algorithms. For example, the victim can be the item with the least value of $(\delta + \tau)$ or $(\delta \times \tau)$.

Some preliminary results show that different values for these parameters affect the tradeoffs between data accessibility and query delay.[8]

Cache admission control

Cache admission control decides whether a data item should be brought into the cache. Inserting a data item into the cache might not always be favorable because it can lower the probability of cache hits. For example, replacing a data item that will be accessed soon with a data item that won't be accessed in the near future degrades performance. We can use the cache replacement value function to implement cache admission control simply by comparing the requested item's cost to the cached item with the highest cost.

To increase data accessibility, a node cannot cache data that some neighbors already cache; rather, it uses its local space to cache more data. For example, if the requesting node is less than a (a system parameter) hops away from a node that has cached the data, it won't cache the data. Thus, the same data item is cached at least a hops apart.

A tradeoff exists between query latency and data accessibility. With a small a, the number of replicas for each data item is high, and the access delay for this data is low. However, with a fixed amount of cache space, mobile nodes can cache a limited number of distinct data items.

If a network partition exists, many nodes might not be able to access this data. On the other hand, with a large a, each data item has a small number of replicas, and the access delay can be a little longer. On the positive side, mobile nodes can cache more distinct data items and still serve

The server can prevent nodes from caching some data or limit the number of nodes that can cache it.

requests when the data source is not accessible.

Depending on the application, *a* can take different values. Accessibility can outweigh access latency when network partitions occur frequently. Instead of waiting for the network topology to change, the nodes should maintain high probability of finding the requested data items. A large *a* lets a node distribute more distinct data items over the entire cache due to admission control, increasing the number of accessible data items and thereby improving overall data accessibility. Note that when *a* reduces to 1, the node can cache the data with minimal latency.

SECURE COOPERATIVE CACHE

Caching nodes can replicate data from the server. Because of security concerns, the owners of some sensitive data might want to restrict access to the data, preventing its duplication. We define different levels of data security with regard to duplication and storage in node caches. The data server can specify the level of security for each data item. Depending on the security level, the server can prevent nodes from caching some data or limit the number of nodes that can cache it. For most sensitive data, the data server sends the encrypted version to a few trusted nodes, which decrypt the data using a shared key.

Future research should focus on mechanisms that let the data owner control the caching scope without undermining its flexibility. Such mechanisms should maintain a balance between security strength and system performance because encrypting or limiting the distribution of the most sensitive data can reduce cooperative caching's benefits.

With cooperative caching, mobile nodes can return the cached data or modify the route and forward the request to the caching node; hence, the mobile nodes should not be able to modify the data maliciously. With data authentication, a receiver can ensure that the received data is *authentic*—that is, it originated from the source and was not modified on the way—even when none of the other data receivers is trusted.

Authenticating the data source is more complicated. Appending each packet with a message authentication code calculated using a shared key does not work because any receiver with the shared key can forge the data and impersonate the sender. Consequently, we use solutions based on asymmetric cryptography, namely digital signature schemes. After the data source signs the data with its private key, mobile nodes can verify the data's integrity using the data source's public key.

Because digital signatures have high overhead in terms of both time to sign and verify and bandwidth, we focus on reducing authentication overhead. For example, if the data has gone through nodes with good reputations, the receiver might not need to verify the signature. This trades some security strength for system performance. Periodically, the receiver might want to verify the signature and change a node's credit rating based on the verification results. A mobile node can verify the signature if the data is important or the node has enough computation power. This allows the user to choose the proper tradeoffs between security and performance—for example, trading security strength for performance if the data is not very important and requires less computation power.

Although ad hoc networks have attracted many researchers, most previous research in this area focuses on routing, with little work on data access. We anticipate that our work will stimulate further research on cooperative cache-based data access. For example, how can we reduce the broadcast overhead during information search? How can we maintain cache consistency and deal with attacks on cache consistency? Comparing cooperative cache-based data access with data replication techniques might also be interesting. We are currently setting up a prototype to test the proposed protocols and address the implementation issues. ■

Acknowledgments

This work was supported in part by the US National Science Foundation through career grant CCR-0092770 and grant ITR-0219711.

References

1. G. Cao, "A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments," *IEEE Trans. Knowledge and Data Eng.*, vol. 15, no. 5, 2003, pp. 1251-1265.
2. A. Rousskov and D. Wessels, "Cache Digests," *Computer Networks and ISDN Systems*, vol. 30, 1998, pp. 2155-2168.
3. L. Yin and G. Cao, "On Supporting Cooperative Cache in Ad Hoc Networks," to appear in *Proc. IEEE Infocom*, IEEE CS Press, 2004.
4. Y. Tseng, S. Ni, and E. Shih, "Adaptive Approaches to Relieving Broadcast Storms in a Wireless Multi-

hop Mobile Ad Hoc Network," *Proc. IEEE Int'l Conf. Distributed Computing Systems*, IEEE Press, 2001, pp. 481-488.
5. W. Lou and J. Wu, "On Reducing Broadcast Redundancy in Ad Hoc Wireless Networks," *IEEE Trans. Mobile Computing*, vol. 1, no. 2, 2002, pp. 111-123.
6. I. Stojmenovic and J. Wu, "Broadcasting and Activity Scheduling in Ad Hoc Networks," to appear in *Ad Hoc Networking*, S. Basagni et al., eds., IEEE Press, 2004.
7. G. Cao, "Proactive Power-Aware Cache Management for Mobile Computing Systems," *IEEE Trans. Computers*, vol. 51, no. 6, 2002, pp. 608-621.
8. S. Lim et al., "A Novel Caching Scheme for Internet-Based Mobile Ad Hoc Networks," *Proc. IEEE Int'l Conf. Computer Comm. and Networks* (ICCCN), IEEE Press, 2003, pp. 38-43.

Guohong Cao *is an assistant professor of computer science and engineering at Pennsylvania State University. His research interests are wireless networks, mobile computing, and distributed fault-tolerant computing. Cao received a PhD in computer science from Ohio State University. Contact him at gcao@cse.psu.edu.*

Liangzhong Yin *is a PhD candidate at Pennsylvania State University. His research interests include wireless/ad hoc networks and mobile computing. Yin received an ME in computer science and engineering from the Southeast University, Nanjing, China. Contact him at yin@cse.psu.edu.*

Chita R. Das *is a professor in the Department of Computer Science and Engineering at Pennsylvania State University. His research interests include parallel and distributed computing, cluster computing, mobile computing, performance evaluation, and fault-tolerant computing. Das received a PhD in computer science from the University of Louisiana, Lafayette. He is a Fellow of the IEEE. Contact him at dsag@cse.psu.edu.*