WIRELESS COMMUNICATIONS AND MOBILE COMPUTING
Wirel. Commun. Mob. Comput. 2010; **10**:436–450
Published online 23 June 2009 in Wiley InterScience
(www.interscience.wiley.com) DOI: 10.1002/wcm.819

Cooperative cache consistency maintenance for pervasive internet access

Yu Huang[1,2]*, Jiannong Cao[3], Beihong Jin[4], Xianping Tao[1,2] and Jian Lu[1,2]

[1]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China*
[2]*Department of Computer Science and Technology, Nanjing University, Nanjing, China*
[3]*Internet and Mobile Computing Lab, Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong*
[4]*Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China*

Summary

Cooperative caching is an important technique to support pervasive Internet access. In order to ensure valid data access, the cache consistency must be maintained properly. However, this problem has not been sufficiently studied in mobile computing environments, especially those with *ad hoc* networks. There are two essential issues in cache consistency maintenance: *consistency control initiation* and *data update propagation*. Consistency control initiation not only decides the cache consistency provided to the users, but also impacts the consistency maintenance cost. This issue becomes more challenging in asynchronous and fully distributed *ad hoc* networks. To this end, we propose the *predictive consistency control initiation* (PCCI) algorithm, which adaptively initiates consistency control based on its online predictions of forthcoming data updates and cache queries. In order to efficiently propagate data updates through multi-hop wireless connections, the *hierarchical data update propagation* (HDUP) algorithm is proposed. Theoretical analysis shows that cooperation among the caching nodes facilitates data update propagation. Extensive simulations are conducted to evaluate performance of both PCCI and HDUP. Evaluation results show that PCCI cost-effectively initiates consistency control even when faced with dynamic changes in data update rate, cache query rate, node speed, and number of caching nodes. The evaluation results also show that HDUP saves cost for data update propagation by up to 66%. Copyright © 2009 John Wiley & Sons, Ltd.

KEY WORDS: consistency control initiation; data update propagation; cooperative caching; *ad hoc* networks

1. Introduction

Pervasive access to the Internet is an essential component of the underlying infrastructure for mobile computing. Internet-based mobile *ad hoc* networks (IMANETs) have received considerable attention due to their potential applications in providing pervasive Internet access [1–5]. For example, in an IMANET shown in Figure 1, mobile hosts close to an access point can directly access Internet, and hence can serve as gateways. Other mobile hosts access Internet *via* multi-hop wireless connections through gateway

*Correspondence to: Yu Huang, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China.
†E-mail: yuhuang@nju.edu.cn

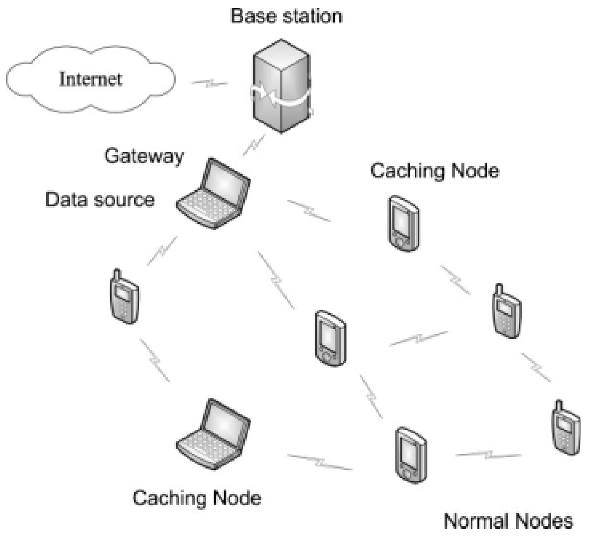

Fig. 1. An IMANET for pervasive Internet access.

nodes. IMANETs can be used in many scenarios, e.g. university campuses, airports, and mobile stores, to provide pervasive Internet access [3,4]. In another example, several commanding officers and a group of soldiers form an IMANET in fulfilling a mission of disaster salvage. The commanding officers access useful information, such as geographic data, *via* satellites. Such information is disseminated to the soldiers *via* the IMANET.

The central problem in IMANET-based Internet access is data dissemination and sharing, which is challenging due to the limited resources (e.g., bandwidth and battery power) and users' mobility. One effective and widely used solution is *cooperative caching*, i.e., to cache frequently accessed data objects at the data source node (gateway node) and a group of caching nodes [2,4,6–8]. Coordination and sharing among the cached data enable mobile users to access required data objects nearby, with reduced traffic overhead and query delay. Cooperative caching also amortizes the overhead of data access among the gateway node and the caching nodes.

In order to ensure valid data access, the cache consistency [3–5], i.e., consistency among the source data owned by the data source node and the cache copies held by the caching nodes, must be maintained properly. We argue that effective cache consistency maintenance requires that the following two essential issues be sufficiently addressed:

- *Consistency control initiation*. Basic consistency control mechanisms include *push* and *pull* [3,4,9–11]. Initiation of push and pull not only decides cache consistency, but also greatly impacts the cost incurred. The asynchronous and fully distributed communication in IMANETs makes consistency control initiation a challenging issue. We propose the *predictive consistency control initiation* (PCCI) algorithm, which achieves adaptability and cost-effectiveness based on its online predictions of forthcoming data updates and cache queries.
- *Data update propagation*. It is desirable to utilize cooperation among the data source node and the caching nodes, in order to facilitate data update propagation over multi-hop wireless connections. We propose the *hierarchical data update propagation* (HDUP) algorithm, which constructs the *source and caching nodes overlay* (SCN-Overlay) network. Based on the SCN-Overlay, the data source node cost-effectively disseminates data updates to all caching nodes, and the caching nodes obtain data updates from caching nodes nearby. HDUP periodically updates the SCN-Overlay to limit the *topology mismatching* [12,13] between the overlay and the physical networks, as well as to cope with node mobility, node failure, network disconnections, etc. We then derive an analytical model to quantify the cost of data update propagation using HDUP.

Extensive simulations are conducted to evaluate PCCI and HDUP. The evaluation results show that PCCI cost-effectively initiates consistency control even when faced with dynamic changes in data update rate, cache query rate, node speed, and the number of caching nodes. The evaluation results also show that HDUP saves the cost for data update propagation by up to 66%.

The rest of this paper is organized as follows. Section 2 provides an overview of the existing work. Sections 3 and 4 describe the design of PCCI and HDUP, respectively. Section 5 presents the analysis on HDUP. Section 6 presents the experimental evaluation. In Section 7, we conclude the paper with a brief summary and the future work.

2. Related Work

Consistency maintenance algorithms for infrastructure-based mobile networks usually employ synchronous push, in which the data source node disseminates data updates at a constant rate [14]. In Reference [15], the frequency of push is dynamically updated based on the data update rate and the cache hit rate. Similar dynamic mechanisms are also adopted on the caching node side [16–18], in which a time to refresh (TTR) value is associated with a cache copy. If the caching node contacts the data source node and finds that the source data have not been updated, the TTR value is increased by a predefined constant value. Otherwise TTR is divided by another constant factor. Initiation of consistency control in the mechanisms above is decided in a static manner. Such mechanisms do not work adaptively in dynamic IMANET environments. PCCI achieves adaptability and cost-effectiveness based on its online predictions of forthcoming data updates and cache queries.

In References [4,10,11], the data source node simply sends the data updates to each caching node *via* unicast, which brings redundant data transmission and imposes most overhead on the data source node. In Reference [5], mobile nodes which are more stable and possessing more cache access rate are elected as *relay peers*. The data source node informs the relay peers of data updates, whereas the caching nodes contact the relay peers to obtain data updates. However, caching nodes usually cannot obtain sufficient information to efficiently elect the relay peers in IMANETs, which is an crucial issue. The role transformation between relay peers and normal caching nodes also induces great overhead. There exist overlay multicast algorithms, which can be employed by the data source node to disseminate data updates. However, overlay multicast algorithms do not enable the caching nodes to efficiently utilize cached data objects nearby. The HDUP algorithm enables efficient cooperation among the source data and the cache copies.

3. Predictive Consistency Control Initiation (PCCI)

In design of PCCI and HDUP, we consider a commonly used system model, in which each data object is associated with a single *data source node*. Only the data source node can update the *source data*. The source data has multiple *cache copies* held by *caching nodes*. Other nodes are called *normal nodes*. Two basic mechanisms for consistency control are *push* and *pull*. Using *push*, the data source node informs the caching nodes of data updates, and the caching nodes can further relay the updates. Using *pull*, the caching node retrieves data updates from other caching nodes or the data source node.

PCCI and HDUP are designed to maintain weak consistency [3,5]. Due to bandwidth and power constraints in *ad hoc* networks, it is too expensive to maintain strong cache consistency, and the weak consistency model is more attractive [2]. Meanwhile, users often have relaxed requirements on data consistency, in order to trade certain cache consistency for reduced data access cost [4].

3.1. Design Rationale

PCCI effectively initiates consistency control taking into account the characteristics of push and pull:

- Using push, the data source node proactively propagates data updates to the caching nodes, which imposes one-way consistency maintenance cost. However, in the asynchronous and fully distributed IMANET environment, the data source node does not know whether the caching nodes need data updates.
- Using pull, the caching nodes obtain data updates upon their requests, but induce round-trip cost.

Given the characteristics of push and pull, we argue that the following principles should be adopted in design of PCCI:

- The data source node pushes only when most caching nodes need the updates.
- The caching nodes pull only when, with high probability, the source data have been updated.

Following the design principles above, the data source node records the time instant of each source data update. It also records time instants of pull queries. Hence, the data source node can obtain the history of pull queries, i.e., the numbers of pull queries between every two sequential updates (e.g., ‘102310’ in Figure 2).

The data source node learns from the pull history and estimates the probability that a caching node needs the data update. The data source node pushes only when this probability is greater than a pre-specified threshold. Users can flexibly tune the threshold value based on the ratio of stale cache hit they can accept.

On the caching node side, each caching node records time instants of data updates and cache queries. Then it calculates the number of updates between sequential cache queries and obtain the history of data updates, also represented by a string of non-negative integers (e.g., ‘120121’ in Figure 3). Upon cache queries, if the

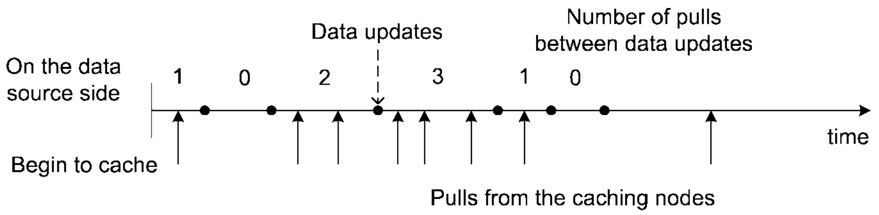
Data updates
Number of pulls between data updates
On the data source side
1 0 2 3 1 0
time
Begin to cache
Pulls from the caching nodes

Fig. 2. Maintaining the history of updates on the source data and pull queries from the caching nodes.

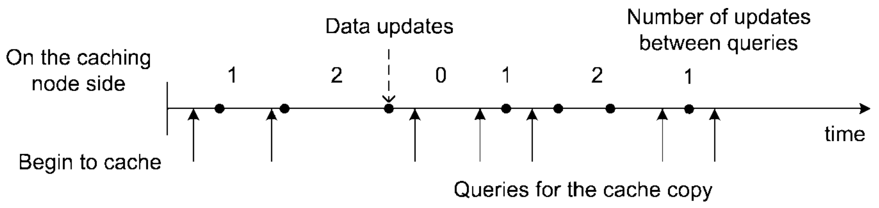

Fig. 3. Maintaining the history of source data updates and queries on the cache copy.

probability that the source data have been updated is less than a pre-specified threshold, the caching node directly serves the query without pull. Here, users can similarly tune the threshold value based on the accuracy of prediction they can accept.

PCCI adopts the LeZi-Update algorithm [19] for its online prediction. LeZi-Update encodes each symbol in the history (e.g., symbols '1', '0', '2' in history of pull queries in Figure 2) and builds a Trie of the symbols. The number of occurrences for each symbol is also recorded in the Trie. Based on the statics of each symbol's occurrence, we can calculate the probability of occurrence of each symbol. We can also calculate the probability of occurrence of a term containing several symbols. With the probability of occurrence of a symbol or a term, we can predict forthcoming data updates and cache queries. We discuss in detail how PCCI use the probability of occurrence to predict data updates and queries in Sections 3.2 and 3.3 below.

PCCI achieves asymptotically accurate prediction with LeZi-Update, i.e., when sufficient history information is encoded, arbitrarily specified accuracy of prediction can be achieved. Note that, the asymptotically accurate prediction enables PCCI to intelligently adapt to changes in patterns of the update and query history. The adaptation needs sufficient history information to achieve certain accuracy of prediction. Specifically, given history size n, LeZi-update is equivalent to a Markov chain of order O(log n). The memory usage of prediction (mainly for storing the Trie of history data) is $O\left(n/\log n - \log\log n\right)$, which ensures that the memory space for maintaining history information increases slowly.

3.2. Design of PCCI on the Data Source Node Side

The data source node records time instant t_u^i of each update in Que_u. Each queue entry, $\text{Que}_u(i)$, contains a pointer to Que_{pl}^i, which stores the time instants of pull queries (denoted by t_{pl}^j) between t_u^i and t_u^{i+1}. Then the data source node can calculate $N_{\text{pl}}[i]$, the number of pull queries between sequential updates t_u^i and t_u^{i+1}.

Each time the source data is updated at t_u^n, the data source node calculates $N_{\text{pl}}[n-1]$ and encodes the single-symbol string '$N_{\text{pl}}[n-1]$' into the Trie for prediction. After encoding the latest pull history, the data source node calculates $P_{\text{pl}}^n(v_i)$, which denotes the probability that $N_{\text{pl}}[n] = v_i$, for each v_i in Set_{val}. Here, Set_{val} contains all distinct symbols in the Trie. It can be formally defined as

$$\text{Set}_{\text{val}} = \{v_1, v_2, \ldots, v_s\}$$

where $v_1, v_2, \ldots, v_s$ are non-negative integers, $v_1 < v_2 < \ldots < v_s$ and $\forall v_i$, $\exists k$, st. $v_i = N_{\text{pl}}[k]$. When the probability $P_{\text{pl}}^n(v_i)$ for each v_i has been obtained, the data source node calculates the expected number of forthcoming pull queries:

$$E_{\text{pl}}^n = \sum_{1\le i\le s} v_i P_{\text{pl}}^n(v_i) + \left(1 - \sum_{1\le i\le s} P_{\text{pl}}^n(v_i)\right) \text{Min}(\text{Set}_{\text{val}}) \quad (1)$$

Here, $\text{Min}(\text{Set}_{\text{val}})$ denotes the minimum non-negative integer which is not in Set_{val}.

Table I. Notations in PCCI on the data source node.

N_{sl}	Number of caching nodes
t_u^i	Time instant of the *i*th source data update
t_{pl}^i	Time instant of the *i*th pull query
Que_u	Queue which stores the time instants of source data updates
Que_{pl}^i	Queue which stores the time instants of pull queries between t_u^i and t_u^{i+1}
$N_{\mathrm{pl}}[i]$	Number of pull queries between t_u^i and t_u^{i+1}
$P_{pl}^i(k)$	Probability that there are *k* pull queries between t_u^i and t_u^{i+1}
$\mathrm{Set}_{\mathrm{val}}$	Set of all unique values of $N_{\mathrm{pl}}[i]$
$\mathrm{Min}(\mathrm{Set}_{\mathrm{val}})$	Minimum positive integer which is not in $\mathrm{Set}_{\mathrm{val}}$
E_{pl}^i	Expected number of pull queries from all caching nodes between t_u^i and (the forthcoming) t_u^{i+1}
τ_{ps}	Threshold value for the data source node, $0 \leq \tau_{\mathrm{ps}} \leq 1$

The rationale behind Equation (1) can be explained by the following example. Assume that the history information is '10221022...', and that the occurrence probabilities of symbols '0', '1', '2' are 20, 20 and 55%, respectively. We can calculate the expected number by $\sum_{1\leq i\leq s} v_i P_{\mathrm{pl}}^n(v_i)$ ((0.20 + 1.20 + 2.55)% in this example). This explains the first term on the right side of Equation (1). Note that, the sum of the occurrence probabilities of all symbols in the Trie will approach 100% when sufficient history information is encoded. However, it will be less than 100%, since there is always the probability that a new symbol appears in the future. The probability that a new symbol appears is $1 - \sum_{1\leq i\leq s} P_{\mathrm{pl}}^n(v_i)$, and we use the minimum number which has not appeared in the history to approximate the new symbol which may appear. This explains the second term on the right side of Equation (1). Note that, the accuracy of Equation (1) is guaranteed by the asymptotically accurate prediction of PCCI.

The data source node calculates the probability that a caching node needs the data update is $E_{\mathrm{pl}}^n/N_{\mathrm{sl}}$. Here, N_{sl} is the number of caching nodes in the IMANET. The user specifies a threshold value based on the ratio of stale hit ratio he can accept. When the maximum acceptable ratio of stale hit is 1-τ_{ps}, τ_{ps} is used as the threshold value for predictive push on the data source node. Notations in PCCI on the data source node are listed in Table I, and pseudo-codes in Algorithm 1.

Algorithm 1 PCCI on the data source node side

```
(1) Record the time instant $t_u^n$ each time the source data are updated
(2) Calculate $N_{\mathrm{pl}}[n-1]$, and encode it
(3) FOR each $v_i$ in $\mathrm{Set}_{\mathrm{val}}$, calculate $P_{\mathrm{pl}}^n(v_i) = P\{N_{\mathrm{pl}}[n] = v_i\}$)
(4) Calculates the expected number of pull $E_{\mathrm{pl}}^n = \Sigma_{1\leq i\leq s} v_i P_{\mathrm{pl}}^n(v_i) + \mathrm{Min}(\mathrm{Set}_{\mathrm{val}})\{1\text{-}\Sigma_{1\leq i\leq s} P_{\mathrm{pl}}^n(v_i)\}$
(5) IF ($E_{\mathrm{pl}}^n/N_{\mathrm{sl}} > \tau_{\mathrm{ps}}$) push
(6) ELSE update the source data without push
```

3.3. Design of PCCI on the Caching Node Side

The caching node records the time instant t_q^i of each query in $\mathrm{Que}_{\mathrm{qry}}$ (the caching node may receive both cache queries from normal nodes, or pull queries from other caching nodes, as discussed in Section 4). Each entry in $\mathrm{Que}_{\mathrm{qry}}(i)$ contains a pointer to $\mathrm{Que}_{\mathrm{ms}}^i$, which stores the time instants of data updates (denoted by t_u^j) between t_q^i and t_q^{i+1}. When the caching node contacts the data source node *via* pull, it obtains the time instants of one or more source data updates and stores them in the corresponding $\mathrm{Que}_{\mathrm{ms}}^i$. The caching node can then calculate $N_u[i]$, the number of data updates between two sequential queries t_q^i and t_q^{i+1}, with $t_q^i < t_q^k$. Here, t_q^k is the time instant of the latest query on the caching node.

After encoding the latest data update history, the caching node calculates the occurrence probabilities of terms consisting of consequent *zeros* '00...0', i.e., the probability that the source date will not be updated for a given number of cache queries. It uses a variable $\mathrm{Flag}_{\mathrm{not\_pull}}$ to decide whether to pull as follows:

$$\mathrm{Flag}_{\mathrm{not\_pull}} = \max\{k | P_{\mathrm{cz}}(k) > t_{\mathrm{pl}}\}$$

where $P_{\mathrm{cz}}(k)$ denotes the probability that $\exists i$, st. the source data are not updated between t_q^i and t_q^{i+k}. The user can specify the threshold value based on the accuracy of prediction they can accept. The acceptable accuracy of τ_{pl} means that when the occurrence probability of *k* consequent zeros is more than τ_{pl}, the user is willing to directly serve the forthcoming *k* queries without pull, in order to trade certain cache consistency for reduced cost.

The caching node directly serves a query if the value of $\mathrm{Flag}_{\mathrm{not\_pull}}$ is greater than *zero*, and then decreases $\mathrm{Flag}_{\mathrm{not\_pull}}$ by one. Notations in PCCI on the caching node side are listed in Table II, and pseudo-codes in Algorithm 2.

4. Hierarchical Data Update Propagation

Following consistency control initiation by PCCI, we present design of the HDUP algorithm. We first

Table II. Notations in PCCI on a caching node.

t_q^i	Time instant of the ith cache query
t_u^j	Time instant of the jth source data update
Que_{qry}	Queue which stores the time instants of queries for the cache copy
Que_{ms}^i	Queue which stores the time instants of source data updates between t_q^i and t_q^{i+1}
$N_u[i]$	Number of source data updates between t_q^i and t_q^{i+1}
$P_{\text{cz}}(k)$	Probability that the source data will not be modified between t_q^i and t_q^{i+k}
τ_{pl}	Threshold value, $0 \le \tau_{\text{pl}} \le 1$
$\text{Flag}_{\text{not\_pull}}$	Number of queries the caching node can directly serve

Algorithm 2 PCCI on the caching node side

```
(1) Record the time instant $t_q^n$ in $\text{Que}_{\text{qry}}$ each time there comes a query for the cache copy;
(2) IF ($\text{Flag}_{\text{not\_pull}} > 0$)
    (2.1) Serve the query without pull;
    (2.2) $\text{Flag}_{\text{not\_pull}} = \text{Flag}_{\text{not\_pull}} - 1$;
(3) ELSE
    (3.1) Pull;
    (3.2) $\text{Flag}_{\text{not\_pull}} = \max\{k | P_{\text{cz}}(k) > \tau_{\text{pl}}\}$;
```

discuss how the SCN-Overlay is constructed and how the data updates are propagated in the overlay. Then we discuss how the SCN-Overlay is maintained in an IMANE environment. Detailed performance analysis is presented in Section 5.

4.1. Overlay Construction

It is the data source node which triggers the SCN-Overlay construction. Initially, the data source node sends an OVL_CONS (overlay construction) message to all its neighbors in the physical network. The OVL_CONS message has a field named *Sender*. Initial value of *Sender* is set to the data source node. Each caching node maintains a field named *Parent*. Initial value of *Parent* of each caching node is set to the data source node. Upon receiving the OVL_CONS message for the first time:

- The caching node will set its *Parent* to value of *Sender* in the OVL_CONS message. It then changes value of *Sender* in the message to itself and rebroadcasts.
- The normal node simply relays the OVL_CONS message.

After the OVL_CONS message is disseminated throughout the IMANET, each caching node knows its parent in the SCN-Overlay. According to Section 4.2, each caching node pulls its parent to obtain data updates. When a caching node receives a pull query, it adds this pulling caching node as its *child* in the SCN-Overlay. In this way, the SCN-Overlay is constructed. Pseudo-codes of the overlay construction and maintenance algorithm are listed in Algorithm 3, and all messages used in Table III.

Algorithm 3 SCN-Overlay construction

```
1. The data source node sends an OVERLAY_CONS message to all its neighbors in the physical network and the Parent field of the message is set to the data source node
2. Upon receiving an OVERLAY_CONS message
    2.1 IF(is a caching node) Set the Parent field of the message to myself
    2.2 IF (receiving the message for the first time) Send the message to all neighbors in the physical network;
3. Upon receiving a PULL message Include the pulling node as my Child in the SCN-Overlay
```

4.2. Data Update Propagation

Data updates are propagated in the SCN-Overlay as follows:

- The data source node pushes data updates to all its children in the SCN-Overlay and the caching nodes relay the updates to their own children.
- Pull query of a caching node will be repeatedly propagated to the parent node, until one of the caching node or (finally) the data source node serves the query.

Note that in a cooperative caching system over an IMANET, the data source node is the gate way node to the Internet, and may hold multiple source data items. The caching nodes are elected through comprehensive evaluation, taking into account their computing capacity, energy, stability of network connection, etc. These

Table III. Messages used in HDUP.

OVL_CONS	Overlay construction message
PUSH	Source data update originating from the data source node
PULL	Pull query from a caching node
PULL_ACK	The reply for a PULL query

caching nodes hold cache copies of each of the source data item. The SCN-Overlay is constructed among the data source node and the caching nodes, and supports dissemination of updates of different data items. Pseudo-codes of the data update propagation algorithm are listed in Algorithm 4.

Algorithm 4 Data update propagation

```
Upon receiving a PUSH message
1. Relay the message to all the children in the SCN-Overlay;

Upon receiving a PULL message
2. IF (local hit according to PCCI)
     Serve the request with a PULL_ACK message;
3. ELSE{
     3.1 Cache the PULL message
     3.2 Forward the PULL message to the parent in
         the SCN-Overlay;
}// else
Upon receiving the PULL_ACK message
4. Find the corresponding PULL message from the
   message cache, and forward the PULL_ACK message
   to sender of the corresponding PULL message;
```

4.3. Maintaining the SCN-Overlay in IMANETs

In IMANETs, maintenance of the SCN-Overlay is essential, in order to cope with the *topology mismatching* [12,13] between the overlay network and the underlying physical network. Moreover, maintenance of the overlay must also take into account node mobility, node failure, network disconnections, etc. In HDUP, the data source node periodically reconstructs the SCN-Overlay to deal with such issues. We argue that the overlay maintenance scheme in HDUP is simple, but robust and effective. The message cost of the overlay construction process is O(n), where n is the number of nodes in the IMANET. The time cost is O(d), where d is the diameter of the IMANET. Moreover, the data source node does not need to maintain extra information for overlay maintenance. We further investigate how the frequency of overlay maintenance impacts the cost-effectiveness of HDUP in Section 6.6.

Even with periodical maintenance of the SCN-Overlay, the caching node may still disconnect from the SCN-Overlay in dynamic IMANETs. When the parent node fails to reply a pull request during data update propagation, the caching nodes directly contact the data source node (based on support from the underlying routing protocol) to fetch the data updates, i.e., becomes children of the data source node in the overlay.

When the caching node cannot find route to the data source node (due to physical network disconnection), it will be included in the SCN-Overlay by our periodical overlay maintenance upon its reconnection. When the data source node fails, the whole cooperative caching system cannot work, and new data source nodes should be elected. Detailed discuss on failure and election of the data source node is not discussed in this work.

5. Analysis on Data Update Propagation Cost

In this section, we first investigate topology of the SCN-Overlay. Then, based on topology of the overlay, we further quantify the cost for data update propagation *via* pull and push. In the analysis, we assume that the source data updates and cache queries follow the *Poisson process* as in References [4,11,20]. Note that due to the dearth of real applications of IMANETs, there is no widely accepted pattern of updates and queries. It is doubted whether there exists a universal model of updates and queries. The Poisson process is widely used to model data updates and cache queries, e.g. in Reference [20]. When updates and queries follow different patterns, the rationale behind our analysis still works, while the specific analysis may be different. The hop count of data transmission is used to measure the consistency maintenance cost [2]. Notations used in the performance analysis are listed in Table IV.

5.1. Topology of the SCN-Overlay

We first study topology of the SCN-Overlay under the assumption that the physical network is connected and then discuss the case of network disconnection below. When the underlying physical network is connected, the SCN-Overlay is also connected. This is because all caching nodes eventually receives the OVL_CONS message, and each caching node gets connected to its parent in the SCN-Overlay. According to *Algorithm* 3, each parent node receives the OVL_CONS message earlier than its child. Thus, we eventually reach the data source node (which triggers the overlay construction at the very beginning) if we keep going upstream from a caching node to its parent. The SCN-Overlay is connected since every caching node is connected to the data source node.

Table IV. Notations used in the performance analysis.

m	Number of caching nodes
V_i	V_0 denotes the data source node; V_i ($1 \leq i \leq m$) denotes the caching nodes
level(V_i)	The shortest path length from V_i to V_0 in the SCN-Overlay, $1 \leq i \leq$ m
$U = \{u_1, u_2, \ldots\}$	Time instants of data updates
λ_u	Data update rate
$Q^s = \{q^s{}_1, q^s{}_2, \ldots\}$	Queries on caching node s
λ_q	Cache query rate
P^s	Time instants of pull queries from caching node s
$P(k,t_1,t_2)$	Probability that there are k queries in period (t_1, t_2)
PAR$^{(i)}(s)$	For a caching node s, PAR$^{(i)}(s) = s$, if $i = 0$; PAR$^{(i)}(s)$ = parent of PAR$^{(i-1)}(s)$ in the SCN-Overlay, if $i > 1$; PAR$^{(i)}$(data source node) = data source node
CPL	Expected pull cost
ω_1	Average edge weight in the SCN-Overlay
ω_2	Average weight of edges between the data source node and a caching node
τ_{push}	Cost-effectiveness of push
CPS	Push cost
CPS$_{\text{MIN}}$	Minimum push cost

We also have that the SCN-Overlay is acyclic. It is because each parent node receives the OVL_CONS message earlier than its children. If any caching node keeps going upstream to the parent node in a circle, it will eventually reach itself, which leads to contradiction. Based on the analysis above, we have that

Theorem 1. *Topology of the SCN-Overlay is Tree, as long as the IMANET is connected.*

Note that the physical network may get disconnected. In this case, our analysis still applies to the connected component of the IMANET which contains the data source node. The SCN-Overlay constructed among the connected component containing the data source node is still a tree. We also need to consider the case where a parent node in the SCN-Overlay gets disconnected from the IMANET, but its child nodes are still connected to the data source node. According to our overlay maintenance scheme (Section 4.3), the child caching nodes will directly connect to the data source node. Thus, the SCN-Overlay is still a tree in this case.

5.2. Expected Cost for Pull Query Propagation

In this section, we first show that the pull queries from a caching node approximately follow the Poisson process of rate λ_u. Then we obtain the expected pull cost, which illustrates the cost-effectiveness of coordination among cache copies.

5.2.1. *Modeling pull queries from a caching node*

A non-leaf caching node in the SCN-Overlay serves two types of queries: cache query from a normal node (named as *normal query*) and pull query from its child caching node in the SCN-Overlay (named as *pull query*). We first define the data updates and queries:

$$U = \{u_1, u_2, \ldots\},\ Q^s = \{q_1^s,\ q_2^s, \ldots\}$$

where u_i denotes time of a data update and $q^s{}_i$, denotes that of a (normal or pull) query on caching node s. Since PCCI achieves asymptotically accurate prediction, as discussed in Section 3.1, we approximately have that only the query right after a data update triggers a pull query, as shown in Figure 4. Thus, the pull queries from caching node s can be defined as

$$P^s = \left\{q_i^s | \exists j, \text{ st. } u_j < q_i^s, \text{ and there is no query in period } \left(u_j,\ q_i^s\right)\right\}$$

According to the discussion above, the number of pull queries in period (t_1, t_2) is decided by whether there are updates and whether there is at least one query after an update in this period. Assume that there is one data update at time t' $(t_1 < t' < t_2)$. For period $(t', t' + \delta)$, $\delta > 0$, the probability that there are no queries in period $(t', t' + \delta)$ is

$$\mathrm{P}(0, t', t' + \delta) = \frac{(\lambda_q^0 \mathrm{e}^{-\lambda_q \delta})}{0!} = \mathrm{e}^{-\lambda_q \delta}$$

P $(0, t', t' + \delta)$ exponentially approaches *zero* when λ_q or δ increases. Hence, when there are frequent cache queries, we approximately have that caching node pulls the data source node soon after a source data update. Since data updates follow the Poisson process of rate λ_u, we have that P^s approximately follows the Poisson process of rate λ_u.

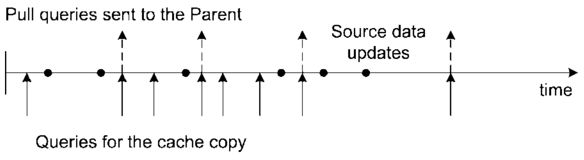

Fig. 4. Pull queries triggered on a caching node.

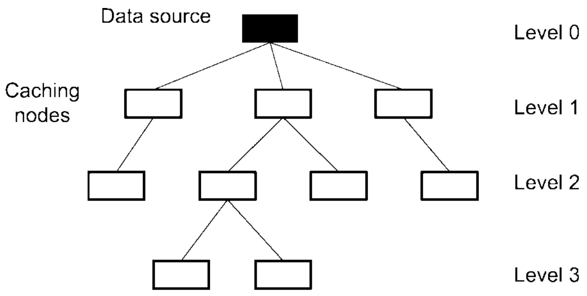

Fig. 5. Caching nodes in different levels.

5.2.2. Obtaining the expected pull cost

We define the *level* of each caching node as its distance from the data source node in the SCN-Overlay (Figure 5). Normal queries on a caching node follow the Poisson process of rate $\lambda_{q.}$ Pull queries from one child-caching node approximately follow the Poisson process of rate λ_u. According to Reference [21], the aggregate process of all normal and pull queries on a caching node also follows the Poisson process of rate:

$$\Lambda_q = \lambda_q + \sum_{1\le i\le k} \lambda_u = \lambda_q + k\lambda_u \tag{2}$$

where k is the number of children of the caching node in the SCN-Overlay, λ_q denotes the rate of normal queries, and $k\lambda_u$ denotes the rate of pull queries from all its children. For a leaf caching node, k is *zero*, and $\Lambda_q = \lambda_q$.

According to the discussion above, only the cache query right after a source data update initiates a pull query. Other cache queries will be directly served. Thus, for one caching node s, the probability of a local query hit is

$$p_{\text{local}} = 1 - \frac{\lambda_u}{\Lambda q}$$

The probability of initiating a pull query is

$$p_{\text{local}} = \frac{\lambda_u}{\Lambda q}$$

For caching node s, $\Lambda_{PAR(k)}$ denotes the rate of aggregate queries on PAR $^{(k)}(s)$. We obtain the expected number of hops the query is relayed upstream toward the data source node in the SCN-Overlay, for caching node s of level h:

$$E(h) = \sum_{1\le i\le h-1} i\frac{\lambda_u^i}{\prod_{0\le j\le i-1}\Lambda_{\text{PAR}(j)}}\left(1 - \frac{\lambda_u}{\Lambda_{\text{PAR}(i)}}\right) + h\frac{\lambda_u^h}{\prod_{0\le j\le h-1}\Lambda_{\text{PAR}(j)}} \tag{3}$$

The first term on the right side calculates the expected cost that the pull query is relayed i $(i < h)$ hops before it is served by a cache copy. The second term calculates the expected cost that the pull query is served by the source data.

We average $E(h)$ among all caching nodes and obtain the average pull cost in the SCN-Overlay:

$$E = \frac{1}{m}\sum_{1\le i\le m} E\left(\text{level}\left(V_i\right)\right)$$

Finally, we get the expected cost of pull using HDUP

$$\text{CPL}_{\text{HDUP}} = E\omega_1$$

We compare the HDUP algorithm with the *simple unicast* (SU) algorithm [4,10]. In SU, the data source node and the caching nodes directly communicate with each other *via* unicast. The expected cost of pull query propagation in SU is

$$\text{CPL}_{\text{SU}} = p_{\text{pull}}\omega_2 = \frac{\lambda_u}{\lambda_q}\omega_2$$

For an IMANET of size 300, we change the number of caching nodes from 50 to 250, and apply the analytical results. We find that the expected pull cost of HDUP is significantly less than that of SU, especially when there are more caching nodes (Figure 6). When there are 250 caching nodes, the cost saved goes up to 62% ((CPL$_{\text{SU}}$ − CPL$_{\text{HDUP}}$)/CPL$_{\text{SU}}$).

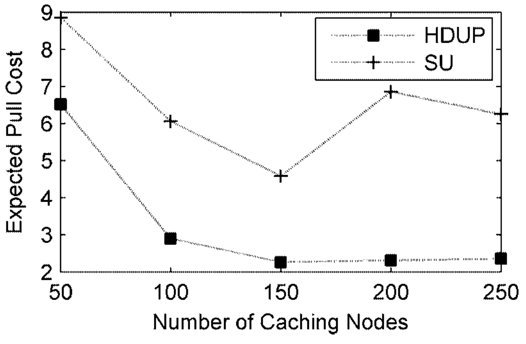

Fig. 6. Expected cost of pull query propagation.

Note that there is an increase in the pull cost of SU. We calculate the cost of SU by averaging path length to the data source node among all the caching nodes. When the number of caching nodes greatly increases (around 70–80% nodes in the network are caching nodes), most caching nodes are far away from the data source node. In SU, there is not coordination among the caching nodes and each caching node independently retrieves data updates from the data source node. Thus the cost of SU increases as the number of caching nodes greatly increases. HDUP enables cooperation among caching nodes, and significantly reduces redundant data transmission. Thus, we do not find notable increase in the cost of HDUP.

5.3. Investigating the Cost-effectiveness of Push

A normalized cost function is defined to measure the push cost. We define the weight of an edge in the overlay as the routing cost between two end nodes in the physical network. We assume that the routing protocol adopts the *least hops* routing metric. The aggregate weight of all edges in the SCN-Overlay measures the push cost. We construct the *minimum spanning tree* (MST) among the data source node and the caching nodes, with the data source node as the root. The normalized ratio τ_{push} measures the cost-effectiveness of push:

$$\tau_{\text{push}} = \frac{\text{CPS}_{\text{MIN}}}{\text{CPS}}, \quad \tau_{\text{push}} \in [0, 1]$$

We also illustrate the cost-effectiveness of push by numerical results, as in Section 5.2.2. We find in Figure 7 that cooperation among the caching nodes greatly improves the cost-effectiveness of push, especially when there are more caching nodes. HDUP achieves more than three times (0.9/0.27) cost-effectiveness, compared with SU, when there are 250 caching nodes.

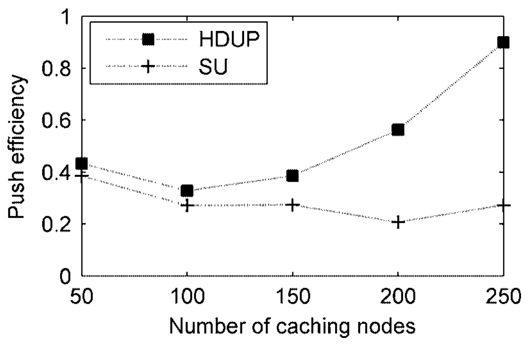

Fig. 7. Normalized cost-effectiveness of push.

6. Experimental Evaluation

In this section, we conduct simulations to study the performance of cooperative cache consistency maintenance with PCCI and HDUP. We first present the experiment methodology and configurations. Then we discuss the evaluation results.

6.1. Experiment Methodology and Configurations

In the evaluation, we first study the adaptability and cost-effectiveness of PCCI by varying the data update rate, cache query rate, the number of caching nodes, and node speed. We compare PCCI with the *pull with dynamic TTR* (DynTTR in short) algorithm [10,16–18]. Both PCCI and DynTTR employ SU for data update propagation. Then, we study the performance of data update propagation. HDUP is compared with SU (PCCI is used for consistency control initiation by both algorithms). We also study impact of the overlay update rate. The following metrics are used in the evaluation:

- *Stale cache hit ratio*: ratio of queries served by stale cache copies.
- *Traffic overhead*: hop count of consistency maintenance message propagation.
- *Query delay*: delay imposed due to consistency maintenance.

In the experiments, we simulate a pedestrian scenario, where all mobile hosts are scattered to a rectangular area of size $200 \times 200\,\text{m}^2$. The transmission range of a mobile host is 15 m. Node speed is varied between 0.5 and 2.0 m/s. Mobile hosts move in the territory following the *random way point* model [22]. Though the mobility model in different scenarios may significantly vary, the random way point model is widely used in pedestrian scenarios.

To simulate node failures, we let randomly chosen nodes crash at random time instants. The ratio of crashing nodes is no more than 10%. Here, we bound the number of crashing nodes. In our experiments, we will specifically study impact of tuning the number of nodes in the network. Message loss rate over one-hop wireless link is set to 5%. Data updates and cache queries are generated following the Poisson process, as assumed

Table V. Experiment configurations.

Network area	$200 \times 200\,\text{m}^2$
Mobility model	Random waypoint
Average velocity of nodes	0.5–2.0 m/s
Transmission range	15 m
Size of network	100–300
Ratio of crashing nodes	10%
Probability of message loss per hop	5%
Update and query model	Poisson process
τ_{ps}	0.1
τ_{pl}	0.5

in our performance analysis (Section 5). Detailed experiment configurations are listed in Table V.

The default configurations of update rate, query rate and number of caching nodes are listed below. We study the impacts of tuning these parameters in the following experiments:

- Average update interval: 20 s.
- Average query interval: 20 s.
- Number of caching nodes: 40.

6.2. Effects of Tuning Data Update Rate and Cache Query Rate

In this experiment, we find that PCCI and DynTTR achieve similar performance in terms of stale cache hit rate (Figures 8 and 11). However, in most cases, PCCI outperforms DynTTR in terms of both traffic overhead and query delay, as shown in Figures 9, 10, 12 and 13. The cost-effectiveness of PCCI is mainly due to its online prediction of forthcoming data updates and cache queries. DynTTR adapts its behavior (by dynamic update of the TTR value) in a simple and pre-defined way. It cannot efficiently adaptive to the dynamic IMANET environment.

Moreover, the online prediction of PCCI achieves more performance improves when faced with frequent

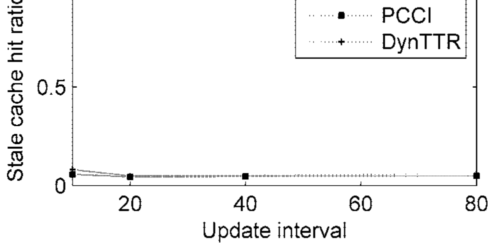

Fig. 8. Stale cache hit ratio, when update interval is increased.

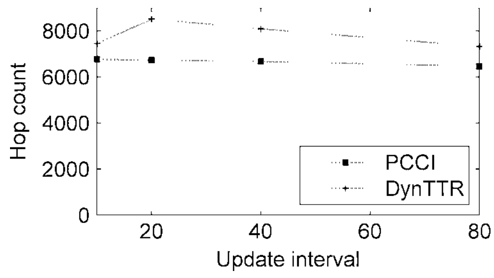

Fig. 9. Traffic overhead, when update interval is increased.

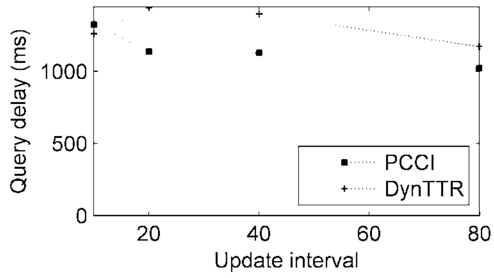

Fig. 10. Query delay, when update interval is increased.

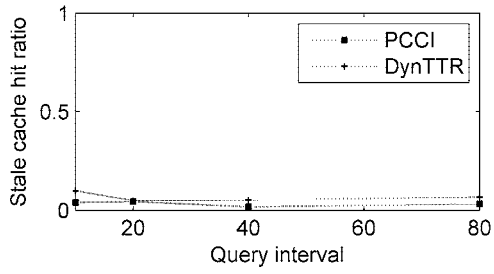

Fig. 11. Stale cache hit ratio, when query interval is increased.

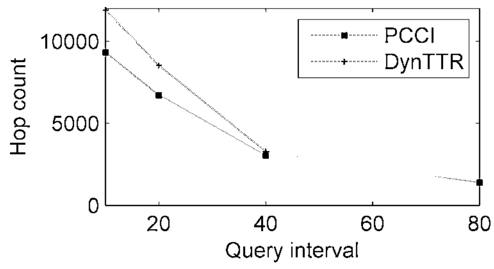

Fig. 12. Traffic overhead, when query interval is increased.

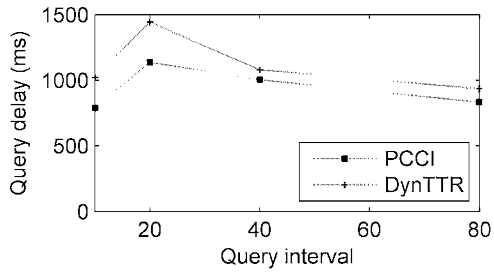

Fig. 13. Query delay, when query interval is increased.

data updates and cache queries. Owing to the asymptotically accurate prediction of PCCI, it achieves more accurate prediction when more update and query information is encoded, thus achieving more performance improvements.

6.3. Effects of Tuning the Number of Caching Nodes

In this experiment, we find that when there are less caching nodes, PCCI is only slightly better than DynTTR. When the number of caching nodes increases, PCCI works more effectively, as shown in Figures 14, 15 and 16. PCCI adapts. PCCI adapts to increase in the number of caching nodes owing to its online predictions, while DynTTR simply tunes the initiation of pull (by tuning the TTR value) based on whether the source data have been updated or not. It does not sufficiently consider changes in the number of caching nodes.

Also note that PCCI becomes less competitive in terms of query delay when there are 80 caching nodes. This is mainly because the cost for encoding and decoding is also increased when there are more caching nodes. The response delay of DynTTR does not increase much since it adapts the TTR value based on a quite simple model.

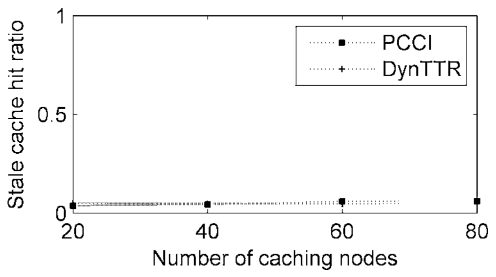

Fig. 14. Stale cache hit ratio, when number of caching nodes is increased.

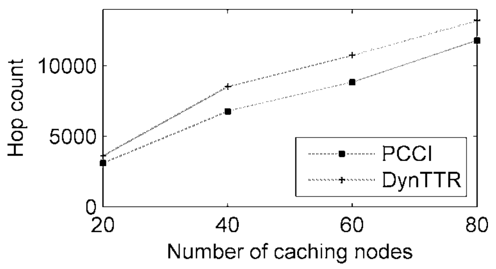

Fig. 15. Traffic overhead, when number of caching nodes is increased.

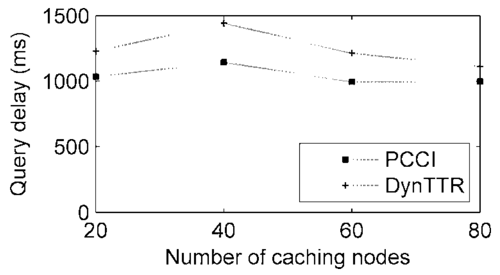

Fig. 16. Query delay, when number of caching nodes is increased.

6.4. Effects of Increasing Node Speed

In this experiment, we study the impact of node speed on performance of both PCCI and DynTTR. As for the stale hit ratio, we find that both PCCI and DynTTR become less efficient (Figure 17). This is mainly because when mobile hosts move more quickly, there are more disconnections in the network. Thus more data updates cannot be propagated in time to the caching nodes, and more cache queries are served by stale cache copies.

As for the traffic overhead, PCCI becomes less cost-effective than DynTTR when the node speed increases (Figure 18). PCCI cannot accurately predict in high-speed scenarios, since the history information cannot be propagated in the IMANET in time, resulting in inaccurate predictions. DynTTR is more effective since it directly tunes the TTR value upon pulling the data source node. When more cache queries are directly served due to disconnections of caching nodes, the traffic overhead imposed by DynTTR slightly decreases.

As for the message delay, PCCI is better since more cache queries are directly served by the caching nodes when there are more network disconnections (Figure 19). Hence, PCCI achieves less query delay at the cost of increasing the stale hit ratio. DynTTR is

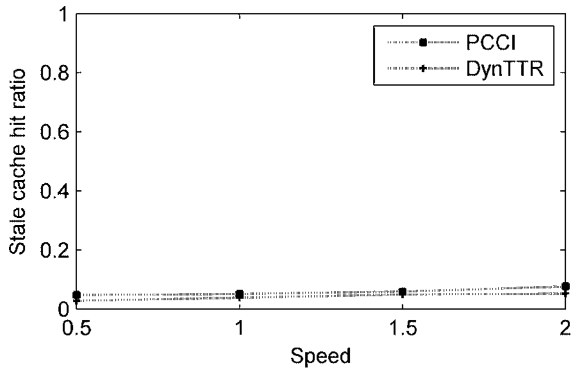

Fig. 17. Stale cache hit ratio, when node speed is increased.

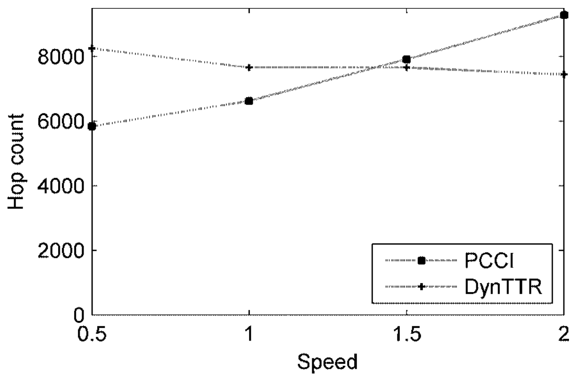

Fig. 18. Traffic overhead, when node speed is increased.

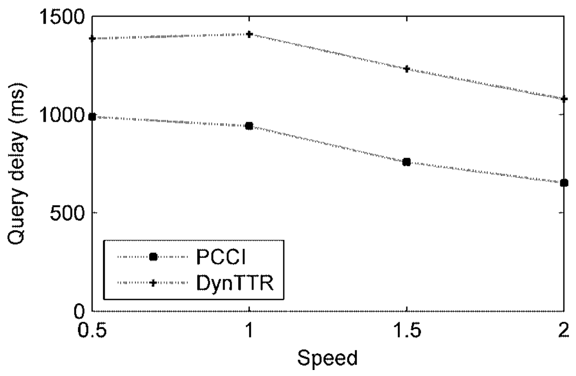

Fig. 19. Query delay, when node speed is increased.

comparatively more stable. The query delay it imposes is many imposed by updating the TTR value, which is less affected by node mobility.

6.5. Cost-effectiveness of Data Update Propagation

In this experiment, we compare the cost for data update propagation using HDUP and SU. We find that in different scenarios (the number of caching nodes increases from 50 to 200), both algorithms propagate

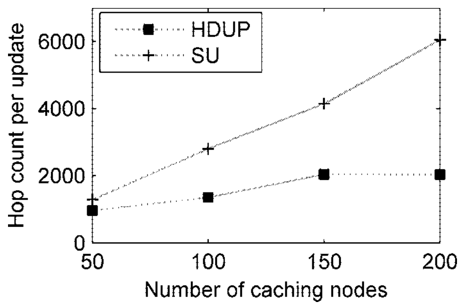

Fig. 21. Cost of data update propagation.

data updates timely and achieve stale cache hit ratio less than 1% (Figure 20). However, HDUP outperforms SU when there are more caching nodes, as shown in Figure 21. When there are 50 caching nodes, HDUP saves 2% (the ratio of the cost saved to the cost imposed by SU) cost, while HDUP saves up to 66% cost when there are 200 caching nodes. We find that when there are more caching nodes, more cost for data update propagation can be saved by sharing cache copies among the caching nodes. Note that, the experiment results are in accordance with those obtained from the analytical model in Section 5.

6.6. Effects of Tuning the Overlay Maintenance Interval

In this experiment, we investigate impact of the overlay maintenance interval. The interval is gradually increased from 30 to 530 s (lifetime of the algorithm is 600 s). We first find that, when the overlay is not frequently updated (interval = 530 in Figure 22), data update propagation in the overlay imposes great traffic overhead. This is mainly due to the topology mismatching between the SCN-Overlay and the physical network. Then we find that frequent maintenance of the SCN-Overlay also imposes great traffic overhead (interval = 30 in Figure 22). The issue here is to efficiently make tradeoffs between the cost for data

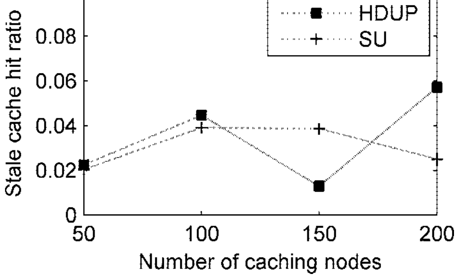

Fig. 20. Stale cache hit ratio.

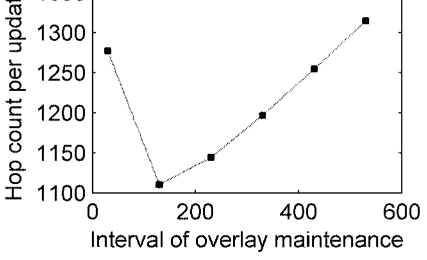

Fig. 22. Tuning the interval of overlay maintenance.

update propagation and that for overlay maintenance. Dedicated heuristics can be developed to decide the frequency of update, and be easily combined with HDUP.

7. Conclusion and Future Work

In this paper, we have studied the problem of cooperative cache consistency maintenance for IMANET-based pervasive Internet access. Toward this objective, our contributions can be described as follows: (1) we propose the PCCI algorithm to initiate cache consistency control in IMANETs by online predictions of forthcoming data updates and cache queries; (2) we propose the HDUP algorithm to enable cooperative data update propagation; we also conduct theoretical analysis on HDUP; and (3) we conducted extensive simulations to evaluate the performance of both PCCI and HDUP.

In our future work, we will work on how to satisfy arbitrary consistency requirements of the users. We also plan to work on consistency maintenance cost-aware cache placement and replacement schemes for cooperative caching over IMANETs.

Acknowledgements

This work is supported by the National 973 Program of China (2009CB320702), the National Natural Science Foundation of China (No. 60736015, 60721002) and the "Climbing" Program of Jiangsu Province, China (BK2008017).

References

1. Corson M, Macker J, Cirincione G. Internet-based mobile ad hoc networking. *IEEE Internet Computing* 1999; **7**: 63–70. DOI: 10.1109/4236.780962
2. Yin L, Cao G. Supporting cooperative caching in ad hoc networks. *IEEE Transactions on Mobile Computing* 2006; **5**: 77–89. DOI: 10.1109/TMC.2006.15
3. Cao J, Zhang Y, Xie L, Cao G. Data consistency for cooperative caching in mobile environments. *IEEE Computer* 2007; **4**: 60–67. DOI: 10.1109/MC.2007.123
4. Huang Y, Cao J, Wang Z, Jin B, Feng Y. Achieving flexible cache consistency for pervasive internet access. In *Proceeding of the 5th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2007; 239–250. DOI: 10.1109/PERCOM.2007.6
5. Cao J, Zhang Y, Xie L, Cao G. Consistency of Cooperative Caching in Mobile Peer-to-Peer Systems Over MANET. *International Journal of Parallel, Emergent, and Distributed Systems* 2006; **21**(3): 151–168. DOI: 10.1080/17445760500356593
6. Yang B, Hurson AR, Jiao Y. On the content predictability of cooperative image caching in ad hoc networks. In *Proceeding of the 7th International Conference on Mobile Data Management (MDM)*, 2006. DOI: 10.1109/MDM.2006.116.
7. Sailhan F, Issarny V. Cooperative caching in ad hoc networks. *IEEE International Conference on Mobile Data Management (MDM)*, 2003; 13–28. DOI: 10.1007/3-540-36389-0 2
8. Lau W, Kumar M, Venkatesh S. A cooperative cache architecture in supporting caching multimedia objects in MANETs. In *Proceedings of Fifth International Workshop Wireless Mobile Multimedia*, 2002; 56–63. DOI: 10.1145/570790.570800
9. Bhide M, Deolasee P, Katkar A, Panchbudhe A, Ramamritham K, Shenoy P. Adaptive push-pull: disseminating dynamic web data. *IEEE Transactions on Computers* 2002; **51**(6): 652–668. DOI: 10.1109/TC.2002.1009150
10. Huang Y, Cao J, Jin B. A predictive approach to achieving consistency in cooperative caching in MANET. In *Proceedings of the First International Conference on Scalable Information Systems, P2PIM Workshop Session*, 2006. DOI: 10.1145/1146847.1146898
11. Huang Y, Jin B, Cao J, Sun G, Feng Y. A selective push algorithm for cooperative cache consistency maintenance in MANETs. In *Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing* (EUC). LNCS 4808 2007: 650–660, DOI: 10.1007/978-3-540-77092-3 56
12. Liu Y, Zhuang Z, Xiao L, Ni LM. A distributed approach to solving overlay mismatching problem. In *International Conference on Distributed Computing Systems (ICDCS)*, 2004; 132–139. DOI: 10.1109/ICDCS.2004.1281576
13. Huang Y, Jin B, Cao J. A distributed approach to construction of topology mismatching aware P2P overlays in wireless ad hoc networks. In *Proceedings of the 14*th *Euro-micro Conference on Parallel, Distributed and Network based Processing (PDP)*, 2006. DOI: 10.1109/PDP.2006.9
14. Tan K, Cai J, Ooi B. An evaluation of cache invalidation strategies in wireless environments. *IEEE Transactions on Parallel and Distributed Systems* 2001; **12**(8): 789–807. DOI: 10.1109/71.946652
15. Barbara D, Imielinksi T. Sleeper and workaholics: caching strategy in mobile environments. In *Proceedings ACM SIGMOD Conf. on Management of Data*, 1994; 1–12. DOI: 10.1145/191843.191844
16. Lan J, Liu X, Shenoy P, Ramamritham K. Consistency maintenance in peer-to-peer file sharing networks. In *The Third IEEE Workshop on Internet Applications*, 2003; 90–94.
17. Urgaonkar B, Ninan A, Raunak M, Shenoy P, Ramamritham K. Maintaining mutual consistency for cached web objects. In *The 21st International Conference on Distributed Computing Systems*, (ICDCS) 2001; 371–380. DOI: 10.1109/ICDSC.2001.918967
18. Sirnivasan R, Liang C, Ramamritham K. Maintaining Temporal Coherency of Virtual Data Warehouses. In *The 19th IEEE Real-Time Systems Symposium*, 1998; 60–70. DOI: 10.1109/REAL.1998.739731
19. Bhattacharya A, Das S. LeZi-update: an information-theoretic framework for personal mobility tracking in PCS networks. Wireless Networks, Vol. 8. Kluwer Academic Publishers: Hingham, MA, USA. 2002; 121–135. DOI: 10.1023/A:1013759724438
20. Hou YT, Pan J, Li B, Panwar SS. On expiration-based hierarchical caching systems. *IEEE Journal on Selected Areas in Communications* 2004; **22**(1): 134–150. DOI: 10.1109/JSAC.2003.818804
21. Ross SM. *Stochastic Processes*. John Wiley & Sons: New York, USA, 1983.
22. Camp T, Boleng J, Davies V. A survey of mobility models for ad hoc network research. *Wireless Communications & Mobile Computing* 2002; **2**(5): 483–502.

Authors' Biographies

Yu Huang received his B.Sc. and Ph.D. degrees from University of Science and Technology of China, Hefei, China, in 2002 and 2007 respectively, both in computer science. From 2003 to 2007, he studied in Institute of Software, Chinese Academy of Sciences, as a co-educated Ph.D. student. He also studied in the Department of Computing, Hong Kong Polytechnic University as an exchange student from 2005 to 2006. He is currently a lecturer in the Department of Computer Science and Technology, Nanjing University, Nanjing China. His research interests include mobile and pervasive computing, software engineering and methodology, and distributed computing. He is a member of IEEE.

Jiannong Cao received his B.Sc. degree in computer science from Nanjing University, Nanjing, China, in 1982, and the M.Sc. and the Ph.D. degrees in computer science from Washington State University, Pullman, WA, USA, in 1986 and 1990, respectively. He is currently a professor in the Department of Computing at Hong Kong Polytechnic University, Hung Hom, Hong Kong. He is also the director of the Internet and Mobile Computing Lab in the department. Before joining Hong Kong Polytechnic University, he was on the faculty of computer science at James Cook University and University of Adelaide in Australia, and City University of Hong Kong. His research interests include parallel and distributed computing, networking, mobile and wireless computing, fault tolerance, and distributed software architecture. He has published over 200 technical papers in the above areas. His recent research has focused on mobile and pervasive computing systems, developing test-bed, protocols, middleware, and applications. Dr. Cao is a senior member of China Computer Federation, a senior member of the IEEE, including Computer Society and the IEEE Communication Society, and a member of ACM. He is also a member of the IEEE Technical Committee on Distributed Processing, IEEE Technical Committee on Parallel Processing, IEEE Technical Committee on Fault Tolerant Computing. He has served as a member of editorial boards of several international journals, a reviewer for international journals/conference proceedings, and also as an organizing/programme committee member for many international conferences.

Beihong Jin received B.S. degree in 1989 from Tsinghua University, M.S. degree in 1992, and Ph.D. degree in 1999 from Institute of Software, Chinese Academy of Sciences, all in computer science. Currently she is a Professor in Institute of Software, Chinese Academy of Sciences. Her research interests include mobile and pervasive computing, middleware and distributed systems. She has published over 50 research papers in these areas and holds one China patent.

Xianping Tao received his B.Sc. from National University of Defence Technology, Changsha, China, and received his M.Sc. and Ph.D. degrees in Nanjing University, Nanjing China, all in computer science. He is currently a professor in the Department of Computer Science and Technology at Nanjing University. His research interests include software engineering and methodology, middleware systems, and pervasive computing. He is a member of IEEE.

Jian Lu received his B.Sc., M.Sc., and Ph.D. degrees in Computer Science from Nanjing University, P.R. China. He is currently a Professor in the Department of Computer Science and Technology and the Director of the State Key Laboratory for Novel Software at Nanjing University. Professor Lu serves on the Board of the International Institute for Software Technology of the United Nations University (UNU-IIST). He also serves as the director of the Software Engineering Technical Committee of the China Computer Federation. His research interests include software methodologies, software automation, software agents, and middleware systems.