# IMPROVING ON-DEMAND DATA ACCESS EFFICIENCY IN MANETS WITH COOPERATIVE CACHING

by

Yu Du

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

ARIZONA STATE UNIVERSITY

November 2005

IMPROVING ON-DEMAND DATA ACCESS EFFICIENCY IN MANETS WITH COOPERATIVE CACHING

by

Yu Du

has been approved

November 2005

APPROVED:

_____, Chair

_____

_____

_____

Supervisory Committee

ACCEPTED:

_____

Department Chair

_____

Dean, Division of Graduate Studies

ABSTRACT

Mobile Ad Hoc Networks (MANETs) provide an attractive solution for networking in the situations where network infrastructure or service subscription is not available. Its usage can further be extended by enabling communications with external networks such as Internet or cellular networks through gateways. However, data access applications in MANETs suffer from dynamic network connections and restricted energy supplies. While most of the research focuses on Media Access Control (MAC) and routing layer solutions, we address these problems by application level cooperation which utilizes the locality principle and commonality in users' interests.

In this dissertation, we explore how to use cooperative caching to improve data access efficiency in MANETs. We devised COOP, a novel cooperative caching scheme for on-demand data access applications in MANETs. The objective is to improve data availability and access efficiency by collaborating local resources of mobile nodes. COOP addresses two basic problems of cooperative caching: cache resolution and cache management, i.e. how to find requested data efficiently and how to manage an individual cache to improve the overall capacity of cooperated caches. To improve data availability and access efficiency, COOP discovers data sources which induce less communication cost by utilizing cooperation zones, historical profiles, and hop-by-hop resolution. For cache management, COOP increases the effective capacity of cooperative caches by minimizing caching duplications within the cooperation zone and accommodating more data varieties. The performance of COOP is studied using mathematical analysis and simulations from the perspectives of data availability, time efficiency, and energy efficiency. The analysis and simulation results show that this

scheme significantly reduces response delay and improves data availability with proper settings of the cooperation zone radius.

This thesis is dedicated to my parents Minge Du and Cuiping Wei, and my husband Chun Fan who have always loved, supported, and inspired me.

ACKNOWLEDGMENTS

The author would like to express acknowledgments to Dr. Sandeep Gupta, Dr. Guoliang Xue, Dr. Arunabha Sen, and Dr. Partha Dasgupta for their supports and advices.

The author would also like to thank Chun Fan, Hai Huang, Bin Wang, Guofeng Deng, Qinghui Tang, Vikram Shankar, Tridib Mukherjee, and Valliappan Anamalai for offering their help in preparing this documents.

Lastly, the author would like to thank NSF (through grants ANI-0196156 and ANI-0086020) and Consortium for Embedded Systems for partially funding this research.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER 1

# Introduction

Mobile Ad Hoc Networks (MANETs) has drawn much attention in recent years due to the flexible networking solution it provides. In such networks, mobile nodes, typically battery-powered, communicate with each other through wireless medium and multi-hop routes. However, data access applications in MANETs have to face lower data availability and higher access cost caused by the special features of MANETs: wireless medium, multi-hop routing, dynamic topologies, and resource constraints. While most of the research efforts focus on Media Access Control (MAC) and routing layer solutions, we address these problems by application level cooperation which utilizes the locality principle and the commonality in users' interests. In this dissertation, a cooperative caching scheme is devised to enable application layer cooperation for MANETs. Resource constraints and dynamic network topologies as well as different application characteristics bring lots of challenges to the cooperative caching scheme design.

### 1.1. MANETs and data access applications

MANETs is composed solely of mobile computing nodes which communicate and forward messages for each other through wireless medium. It is an attractive networking solution in the situations where network infrastructures or service subscriptions are not available. Its usage can further be extended by enabling communications with external networks such as Internet or cellular networks through gateways [1–3]. However, MANETs are limited by intermittent network connections, restricted power supplies, and computing resources etc. These restrictions raise several challenges for data access applications with the respects of data availability and access efficiency. The features of MANETs and the caused problems/issues are summarized as follows:

- Wireless medium. Mobile nodes communicate with each other through wireless medium, which means one transmission can cover all audiences within the transmission range. This broadcast nature is also referred as the Wireless Multicast Advantage(WMA) [4]. This is an advantage for broadcast or multicast traffics, however, for unicast traffic, because of the broadcast nature, the nodes within same transmission/interference range have to compete with each other for the network channel. For data access applications which follows the Client/Server model, unicast is the mostly used communication style. If there are multiple data access sessions going on simultaneously, users may encounter increased communication delay or even packet losses due to the competition and interferences from other sessions.

- Multi-hop routes. In MANETs, multi-hop routes are used when a node communicates with the nodes that are out of its immediate transmission range. As analyzed by Li etc. [5], the optimal throughput for a multi-hop communication is about 1/3 of the throughput of a single hop communication, as a forwarding node cannot transmit and receive at the same time, and the upstream and downstream nodes of the forwarding node cannot transmit at the same time because the packets would conflict at the forwarding node. Their results measured on real hardware show that the throughput drops while the number of hops covered increases. For data access applications, the indication is that as the number of hops between a server and a client increases, the throughput will decrease and the response delay will increase accordingly.

- Dynamic topologies. Network topology may change for MANETs due to node mobility, exhausted power, and interferences etc. Changing of network topologies may break existing routes and cause extra overhead for establishing new routes. For data access applications, if the route to the server is broken, the client/server communication will be delayed if a new route needs to be established, or even get dropped if the new route cannot get established in time. This delay or disconnection has more chances to happen if there are more forwarding nodes involved in the client/server communication path.

- Limited resources. In MANETs, the mobile nodes usually have restricted power supply, limited computing capability, and limited storage space. Due to this fact, data access applications shall be able to function properly with least re-

source possible. That is, data access applications need to be energy-efficient and light-weight (not computing-hungry or memory-hungry).

To solve these problems/issues, most of the research efforts focus on media access control (MAC) and routing layer solutions, while this dissertation studies how to use application level cooperative caching to address these problems by utilizing data locality and the commonality of users' interests. Before explaining how cooperative caching can address these problems, the first thing is to clarify what is cooperative caching.

## 1.2. Cooperative caching

To answer what is cooperative caching, the first thing is to specify what is caching. Generally speaking, caching is to copy a portion of the data from the data provider to a smaller and faster storage device (cache) interposed between the data consumer and the data provider, so that future data accesses can be resolved from the cache with less cost. One thing to point out is that caching is a logical entity instead of a physical entity. A cache may have different physical presentations, but they serve the same logical functionality. In a storage hierarchy shown in Figure 1, each storage layer has a different physical form, but they all function in the same way of serving as the cache for the layer above them. This dissertation focuses on the cache that a computing node uses to store data from a remote data server.

Then what is cooperative caching? Webster's explanation for "cooperate" is "to act or work with another or others: act together; to associate with another or

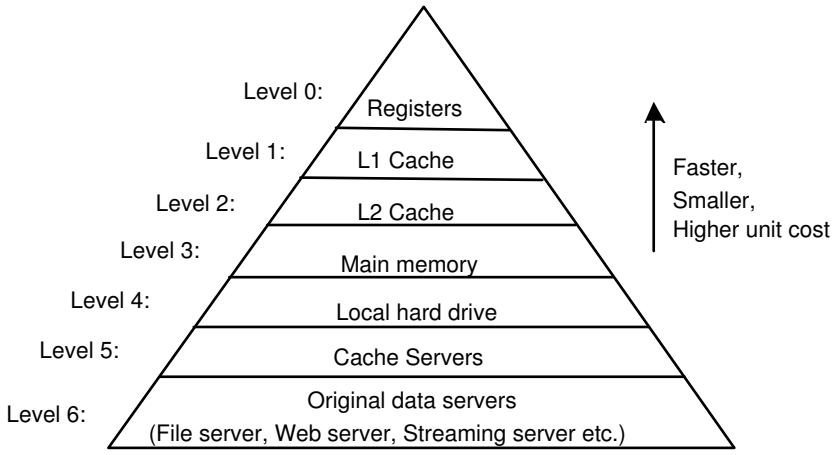


Figure 1. An example storage hierarchy

others for mutual benefit". *Cooperative caching* means multiple caching nodes share and cooperatively manage the cached contents [6–11]. The meaning of cooperation is two-fold.

- A caching node not only serves the data requests from its own users but also the data requests from other nodes.
- A caching node stores data not only on behalf of its own needs, but also on behalf of other nodes' needs.

Correspondingly, a cooperative caching scheme needs to specify:

- How are users' requests processed using the cooperative caching system? That is, once a caching node receives a data request, how does it proceed to resolve the request?
- How does a caching node manage the cached data on behalf of the cooperative caching system?

In the past, cooperative caching has mainly been used in Webcache (Proxy) servers to alleviate server overloading and minimizing client-side response delay on Internet [6, 9–11]. Section 2.1 describes and compares the representative cooperative caching schemes in this paradigm. The difference of Internet cooperative caching and MANET cooperative caching is rooted from the underlying network infrastructure. As stated in Section 1.1, MANETs have special features (such as WMA and dynamic topology) that need to be addressed particularly in the design of MANET cooperative caching schemes.

### 1.3. Data locality and Zipf's law

Previously it has stated what are the problems/issues for data access applications in MANETs and what is cooperative caching. In this section, it is explained how cooperative caching can help solve these problems.

The first fact that supports cooperative caching is the locality principle [12]. The locality principle states that computer programs tend to cluster references to a subset of data/instruction. Because programs repeat referencing the same data/instruction, it is helpful to save the recently used data in a high speed cache to reduce access overhead for future. The locality principle has been observed and utilized in various fields of Computer Science, e.g. processor caches, storage hierarchies, Webcaches, and search engines. This research makes use of the temporal locality observed in data access streams. Temporal locality in an access stream means that the time interval between consecutive accesses of some data items are shorter than

what would occur if all data items were to be accessed with equal probability [13]. It is observed that the traces of Web accesses demonstrate temporal locality and this locality is increased in the aggregated trace as the popular data items tend to occur in each access stream [13].

The second fact that supports cooperative caching is the Zipf's law [14]. Assuming that the data are ranked according to their popularity, Zipf's law basically says the access probability of a data item $P(i)$ is inversely proportional to its rank $i$, i.e. $P(i) \propto \frac{1}{i^{\alpha}}$ ($\alpha$ close to unity. As $\alpha$ increases, the access pattern becomes more concentrated on the popular data items). Another representation of Zipf's law is the 80-20 rule, which basically says that 80% accesses happen on 20% data. Zipf's law is observed in real Web access traces and the parameter $\alpha$ is measured ($\alpha \in [0.64, 0.83]$ according to the studies in [15]). It is also observed from real Web traces that the popular data items tend to occur in each access stream, and the combined stream is generally more skewed than individual streams [13]. This indicates that users have common interests in popular data items. Otherwise if user's interests are randomly distributed, the accesses should be more regularly distributed.

Another fact to notice is that users around the same location tend to share common interests, since people are generally associated by their locations. For example, the people in the same conference room tend to share the same interests on the presentations; the personnel on a rescue site tend to share the interests on the rescue arrangements; an exploration team tend to share the interests on the site they are exploring.

Two examples are used to illustrate how cooperative caching can help improve data access efficiency in MANETs:

- As shown in Figure 2, at a rescue site the personnel communicate with each other's mobile device through a spontaneous MANET due to the lack of network infrastructure support. The devices that have interfaces to external networks serve as gateways to allow other devices communicate with external data servers. Suppose Ashley has retrieved a patient's medical history information from an external data server. Now Bob needs the information for the same patient. If Ashley's device cannot answer Bob's data request, Bob's request will be routed to the external server to retrieve the data. Due to the problems stated in Section 1.1, Bob may encounter a traffic delay or disconnection during this process. On the other hand, if Ashley's device can share the information and answer Bob's request, it may save significant amount of time, bandwidth, and battery consumption for Bob, as the communication cost to the external server is higher than the cost to Ashley's device.

- MANETs can be used to extend the coverage of infrastructure-based networks such as cellular networks [1, 2]. The basic mechanism is to let in-range devices relay packets for out-range devices. The relay can involve multiple intermediate devices until reaching the service area. For example, if tourist Charles is out of range of the service area, his request for the area map can be relayed by other tourists. If the relaying devices are not enabled to answer the request, the request needs to be forwarded to the server for retrieving the map. On the

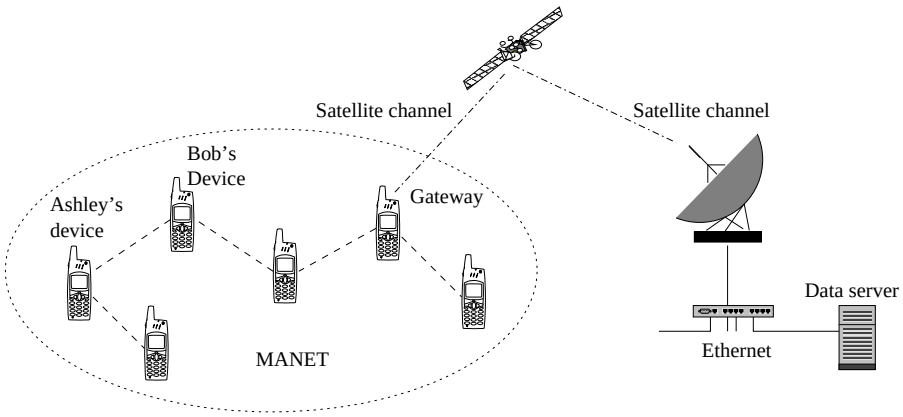


Figure 2. Example of cooperative caching in MANETs.

other hand, if the relaying devices can answer the request, the request could be answered before reaching the server if a tourist on the relay path has already cached the requested information. By enabling the forwarding devices to answer a relayed request, it helps to reduce response time as the communication path is shorter. It also helps to reduce the risk of dropping a request/reply since there are less forwarding nodes and dynamic factors involved in the communication. It helps to reduce the energy consumption of the upstream forwarding nodes because the request may get resolved before reaching these nodes. Finally it may also help to reduce the requester's energy consumption due to the reduction of response waiting time.

From these examples, we can see that for the data of common interests, cooperative caching can help to save time and energy costs as well as reduce the chance of packet dropping in data accesses by localizing communications. For data access applications in MANETs, since the communications between clients typically involve

less hops and less expensive links (like the satellite link in the example) than the communications between client and server, resolving a data request from another client's cache usually causes less time and energy cost than resolving the request from the server. On the other hand, resolving a request from another client's cache may have higher success ratio, as there are less forwarding nodes involved and any failure or change of a forwarding node might lead to dropping of the data request/reply. A problem not taken care of is security and privacy, for example, users may not want to share all the cached contents. This dissertation considers a cache which contains the data publicly available and share-able, such as the vast information provided on Internet. Encryption and authentication mechanisms can be used to further improve privacy and security, but this is out of the focus of this dissertation.

## 1.4. Overview of the dissertation

This dissertation provides a discussion on how to use cooperative caching to improve data access efficiency in MANETs and reports the obtained results. The objective of this research is to devise a cooperative caching scheme that improves the performance of data access applications in MANETs. The assumption is that the node mobility is unknown and unpredictable. The performance consideration comes from the following aspects:

- Data availability: what is the percentage time that the requested data is available to the user.
- Time efficiency: how long it takes to receive the requested data.

- Energy efficiency: how much energy is needed to retrieve the requested data.

The devised cooperative caching scheme, COOP, addresses two basic problems for cooperative caching in MANETs:

- Cache resolution – how does a mobile node decide where to fetch a requested data item.
- Cache management – how does a mobile node decide which data item to place/purge in its cache.

To improve data availability and access efficiency, COOP discovers data sources which induce less communication cost by utilizing cooperation zones, historical profiles, and hop-by-hop resolution. For cache management, COOP minimizes caching duplications within the cooperation zone and increases the effective capacity of cooperative caches to accommodate more data varieties. The performance of COOP is studied using mathematical analysis and simulation results. The evaluation shows that COOP achieves higher data availability and time efficiency with comparable energy cost than other approaches.

The challenges of this research come from the following aspects:

- There is no central coordination point for cooperative caching in MANETs. All the mobile nodes are autonomous, and it is difficult to assign caching tasks to individual nodes in a highly dynamic network.
- Because of energy and bandwidth restrictions, inter-cache communication overhead shall be minimized.

- Caching nodes are dynamic, previous cooperating nodes may move out of range or switch to power-saving mode. How to discover and establish relationship with new cooperating nodes is a question to be answered by the cache resolution scheme.

- The cooperative cache management scheme has to answer questions like whether a node shall cache a data needed by itself or a data needed by the neighborhood.

Our contribution is that we developed a cooperative caching scheme that improves the performance of data access applications in MANETs. With comparable energy cost, our scheme achieves higher data availability and time efficiency than other approaches.

The rest of the dissertation is organized as follows: Chapter 2 provides a discussion on related works and literature study. Chapter 3 describes the system models and problem definition. Chapter 4 gives the overview and formal description of the proposed cooperative caching scheme. Chapter 5 is the mathematical analysis and comparative simulation results of the proposed scheme and existing caching schemes. Finally, conclusions are written in Chapter 6.

CHAPTER 2

# Related Works

A cooperative caching scheme specifies how multiple computing nodes share and manage the cached data in a collaborative manner. This thesis focuses on caching of regular objects such as text files and pictures.

## 2.1. Existing cooperative caching schemes

Cooperative caching has been applied to different contexts, such as Web caches/proxies and file systems. These schemes can be categorized as hierarchical, directory-based, and hashtable-based approaches. Harvest [9] organizes Web caches hierarchically. A user's request is forwarded up the cache hierarchy until cache hits at some level. As a directory-based approach, Summary [10] keeps directory information of which cache has what content. When cache miss happens, the request is forwarded to the caches which contains the requested data potentially. For hashtable-based approach, in Squirrel [11], data items or their location information are cached on the correspondent home nodes, and the home nodes are assigned and located using distributed hash tables. Those schemes have been evaluated and demonstrated

performance improvement for Web accessing. However, these schemes are designed for Internet caching, which generally considers the cooperation between dedicated cache servers with high speed network connections. They impose some kind of structure on the network of cooperative nodes, such as hierarchical, hash-table based, and directory-based etc, to facilitate the search of desired data. But for generic MANETs, its dynamic topology and inefficient multi-hop communications make it extremely difficult to maintain information for traditional structures.

Cooperative caching in MANETs received recent attentions from research efforts. Yin and Cao presented three cache resolution schemes: CacheData, CachePath, and HybridCache in [7]. In CacheData, forwarding node checks the passing-by data requests. If a data item is found to be frequently requested, forwarding nodes cache the data, so that the next request for the same data can be answered by a forwarding node instead of travelling further to the data server. A problem for this approach is that the data could take a lot of caching space in forwarding nodes. CachePath is similar with CacheData. The difference is that forwarding node does not cache the data, instead it records the path information to the closest caching node where the data can be found, and the next request will be redirected to the recorded cache. This scheme saves caching spaces compared to CacheData, but since the caching node is dynamic, the recorded path could become obsolete and this scheme could introduce extra processing overhead. Trying to avoid the weak points of those two schemes, HybridCache decides when to use which scheme based on the properties (size and the Time-to-Live value) of the passing-by data. For example, CacheData is used if the

data size is smaller than the threshold values. Here we question that why the forwarding nodes shall keep record of which data is more popular? Since the forwarding nodes can move to somewhere else in the next second, how much does the previous statistics collected by the forwarding nodes reflect the future needs at their current location? In COOP, we also allows forwarding nodes to reply data requests. But we do not advocate that the forwarding nodes shall collect statistics based on their forwarded requests.

As a summary, the comparison of existing cooperative caching schemes and COOP is listed in Table 1. For other caching schemes in mobile computing environments, please refer to [16].

Table 1. Comparison of cooperative caching schemes

| Schemes | Cache Resolution | Cache management |
|---|---|---|
| Harvest [9] | Hierarchical | No specification |
| Summary [10] | Directory-based | LRU |
| Squirrel [11] | Hash-based | LRU: all accesses, from the local node and from the neighbors, contribute to the weight of a data item. |
| Yin04 [7, 8] | CacheData, CachePath, HybridCache | LRU |
| COOP | A cocktail scheme (Adaptive flooding, Profiled based resolution, Roadside Resolution) | Inter-category and intra-category rules |

Even though the papers [17] and [18] talk about cooperative caching in mobile ad hoc networks, the approaches have a lot of restrictions. For example, both assume the data access frequency is known and fixed, and both use centralized algorithm to

compute cache assignment for their cache resolution scheme. The paper [19] talks about energy efficient caching in ad hoc networks. But they focus on a different problem: the pre-fetching policy. A heuristic algorithm is proposed to decide whether a caching node shall pre-fetch a data item or not. The basic idea is that the prefetching cost should be compensated by future minimization of access delay and energy cost. To conduct this algorithm, the network topology must be static and the data access probability must be already known for all the nodes in the network. A missed part of this paper is the cache resolution scheme, i.e. there is no description about how a requesting node can find the caching nodes which has already pre-fetched the data.

### 2.2. General data searching strategies in MANETs and P2P networks

One important problem cooperative caching deals with is how to find the cache that has the desired data item in the whole cooperative caching system. Expanding ring is widely used in searching for a destination node in MANETs, querying for a requested data item in sensor networks, and searching for a shared file in an unstructured peer-to-peer network. The basic idea is to increase the flooding range if the previous flooding did not find the desired information successfully. The paper [20] provides an algorithm to achieve the optimal average flooding cost if the information distribution is known. While if the distribution is unknown, it provides an algorithm to minimize the worst-case cost. An important difference between cache resolution in cooperative caching and general data searching strategies is that there is a known information source in cooperative caching. On the other hand, information source is

unknown in general data searching strategies. If there exists an information source, it can only be found out after the searching procedure. This difference makes it necessary to consider the relative cost of the searching strategies and getting information directly from the server in cooperative caching.

Another representative data searching strategy in MANETs is location-based data querying used in Data Centric Storage (DCS) [21]. DCS utilizes Distributed Hash Table (DHT) and maps a data item or a data category to a specific geographical location. The queries will be routed to the mapped location for resolution using location-based routing such as GPSR [22]. However, when the whole network is moving (such as a group of people on a trip), this scheme does not work because every node's physical location changes.

The similar problem studied by cache resolution in cooperative caching and Peer-to-Peer (P2P) searching algorithms is that both are trying to find a data item in inherently unstructured networks. P2P systems can be classified into unstructured, structured, and loosely structured systems [23]. In unstructured systems, such as Gnutella and Napster, the data placement is undetermined, and the typical way to find a data item is by flooding or random search. This type of system can easily accommodate a highly transient node population, but lacks of scalability when node population increases. In structured P2P systems, such as Chord and Tapestry, data locations are determined precisely, usually through DHT. This improves the system's scalability, while adds extra overhead processing for nodes joining and leaving. Loosely structured systems, such as Freenet, fall in between the two. Data locations

are determined using routing hints. Since hints are not accurate location information, searches can fail. However, note that P2P searching strategies do not consider energy efficient, which is very important for MANETs.

## 2.3. Data dissemination

Data dissemination is a procedure that distributes information from a source to a set of destinations. What information to distribute and when to distribute the information is not controlled by the demands of an individual destination/client. This is different from the on-demand model studied in COOP. Put this in other words, data dissemination considers a push-based data access model, while COOP studies a pull-based data access model.

Push-based data access is usually seen in distributing system-wide information, such as a new configuration file for a sensor network. Pull-based data accesses are seen in scenarios of getting individual-needed information, such as using a web browser to get an on-line paper.

The study goal of data dissemination is to effectively push the data to all the clients. The algorithms concern with how to construct different dissemination structures, such as multicast trees, grids, and epidemic model used in [24]. For pull-based data access model, it is not explicit that what other clients are interested in the same data, since data access is controlled by different clients and each individual client has his/her own interested data and access pattern. Therefore, it is more difficult to create and maintain structures in this situation than in the pull-based model where

the clients interests are pre-known.

## 2.4. Related classical problems

**2.4.1. $K$-median problem.** The minimum $K$-median problem is a variation of the uncapacitated facility location(UFL) problem. It is well-known NP-hard problems for general graphs and belong to the category of mini-sum locational problems [25].

The minimum $K$-median problem is described as follows. Let $G = (V, E)$ be a connected graph. $V$ is the vertex set and $E$ is the edge set. The shortest distance between vertex $i$ and vertex $j$ is denoted by $w_{i,j}$ ($w_{i,j} \geq 0$). The goal is to select $K$ vertices, called medians, so that the sum of the distance of each vertex to its nearest median is minimized. In the context of this dissertation, the vertices can represent computing nodes, and the medians represent data caches.

For the UFL problem, related research can be dated back to early 60's of last century [26] [27]. Study of the $K$-median problem is recorded at about the same time [28] [29]. Recent works for the $K$-median problem are presented in the papers like [30] [31] [32] [33] [34]. Even though there is no polynomial optimal algorithms for the $K$-median problem, the research force has worked out a few polynomial approximation algorithms for this problem. Lin and Vitter studied the $K$-median problem, and presented an algorithm that provides $1+\epsilon$ approximation, but with $(1+1/\epsilon)(\ln n+1)k$ facilities ($n$ is the total number of vertices). In [31], Charikar et al. presented a $6\frac{2}{3}$ approximation algorithm for $K$-median, and a 3 approximation for the uncapacitated

facility location problem. Next, Jain et al. provided a 6-approximation algorithm for $K$-media with the primal-dual schema in [33]. This approach is further improved by Charikar in [32] to achieve a 4-approximation for $K$-median. In [30] Ayra et al. studied the local search heuristics for the $K$-median problem, which provides a 5-approximation local search algorithm using swapping. Furthermore, if it is permitted to swap $p$ medians at the same time, the approximation can reach the ratio of $3 + \frac{2}{p}$.

Even though this problem is intractable for general graphs, optimal polynomial solutions have been found for some special network topologies. The paper [35] presented polynomial optimal algorithms for $K$-median problem in line and ring networks. Another polynomial optimal algorithm for tree topologies is presented in [36] which can be achieved with the time complexity of $O(|V|^3k^2)$.

**2.4.2. The minimum Steiner tree problem.** The minimum Steiner tree problem is described as follows. For a given graph $G = (V, E)$ and a set $T$, $V$ is the vertex set, $E$ is the edge set, and $T$ is a subset of $V$. The goal is to find a smallest tree connecting all the vertices of $T$. In the context of this dissertation, the cost of the minimum Steiner tree can be used to model the maintaining cost of the cache nodes. A polynomial optimal solution is devised for tree topology in the comprehensive exam. This algorithm finds the cache locations with the maintaining cost (equal to the cost of the Steiner tree) less than a given value and the cache access cost is minimized. For general graphs, the Steiner tree decision problem is NP-complete [37].

**2.4.3. The knapsack problem.** The knapsack problem is described as follows. Given a knapsack with capacity $c$ ($c > 0$). There are $N$ items, each has value $v_i$ and weight $w_i$ ($v_i > 0$ and $w_i > 0$). The goal is to find a selection of items that satisfies $\sum_{i=1}^{N} \delta_i w_i \leq c$ and the total value $\sum_{i=1}^{N} \delta_i v_i$ is maximized ($\delta_i = 1$ if item $i$ is selected, $\delta_i = 0$ if item $i$ is not selected). In the context of this dissertation, the cache size is equivalent to $c$, the size of a data item can be modeled by $w_i$, and the weight of a data item can be modeled by $v_i$. Then solving the knapsack problem is equal to solve the problem of finding a set of data to cache so that the cached items have the maximum sum of weight. The general knapsack problem is NP-hard, and the decision version of the knapsack problem described above is NP-complete.

CHAPTER 3

# Problem Statement

As presented in Introduction, data access applications in MAENTs can experience low data availability and high latency, energy, and bandwidth costs due to multi-hop communication and dynamic network links. Cooperative caching is a feasible solution to improve data access efficiency, because it actually increases the effective cache size for mobile nodes, i.e. mobile nodes can make use of the data stored in another node's cache.

The goal of this research is to devise a cooperative caching scheme to help improve on-demand data access efficiency in MANETs. The proposed scheme is consisted of cache resolution and cache management schemes. The studied network model, data access model, and cache consistency model are specified in the following sections. The performance of the developed approach is studied using mathematical analysis and simulation evaluation. The following metrics are used to evaluate the approaches from the perspectives of data availability, time, bandwidth, and energy efficiency:

- Request success ratio: the percentage of successfully resolved requests. This is

used to reflect the resulted data availability.

- Cache miss ratio: the percentage of requests that have to be forwarded to the server for resolution.

- Average hops covered per successful request: the average traveled hops for a request to get reply. This is used to reflect the time efficiency for cooperative caching protocols.

- Average messages sent per successful request: the average number of sent messages for resolving a request. This is used to reflect the message overhead and energy efficiency of cooperative caching protocols.

## 3.1. Network model

In this thesis, we study a MANET environment where nodes mobility pattern is unknown. To enable connections to external networks such as Internet, the mobile nodes that have interface to external networks serve as gateways between internal and external networks, as shown in Figure 3. A data server can reside inside or outside the MANET.

The distance between two mobile nodes is measured in number of hops. By saying two nodes are 1-hop away, we mean they can reach each other within one transmission. We assume that the communication delay between two nodes is proportional to their distance measured in hops, i.e. $D_{i,j} \propto H_{i,j}$ ($D_{i,j}$ is the communication delay from node $i$ to node $j$; $H_{i,j}$ is the hop count from node $i$ to node $j$). We also assume

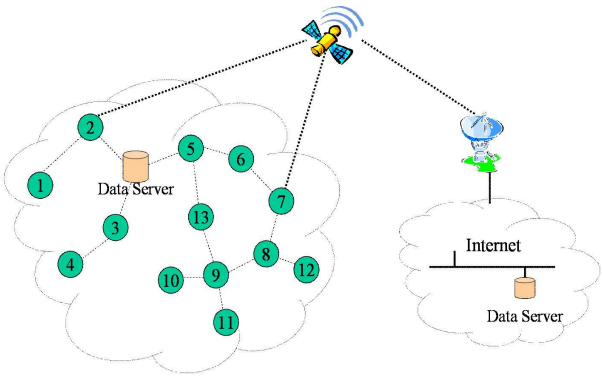


Figure 3. Network Model

symmetric network links, that is $D_{i,j} = D_{j,i}$ and $H_{i,j} = H_{j,i}$. In case a data server is located outside the MANET, we approximate the communication delay between the gateway and the server by integral times of one-hop delay.

### 3.2. Cache consistency model

Cache consistency models specify how to maintain consistency between the original data and the cached copies at the client side. Significant research effort has been carried out and various cache consistency schemes have been developed (see the references [38] [39] [40] etc.).

Based on who initiates data validation or invalidation, cache consistency schemes can be classified into two categories: *pull-based*, and *push-based*. In pull-based schemes, the clients initiate data validation by polling the server if the data has been changed since the caching time. In push-based schemes, the server initiates data invalidation by notifying the clients about the changes. Server invalidation can be announced periodically or upon data changes. According to [38] [39], periodi-

cal invalidations is called synchronous invalidation approach, and invalidation upon changes is called asynchronous approach.

Based on how strict cache consistency is enforced, those schemes can be categorized into *weak consistency* and *strong consistency* models [40] [41]. Weak consistency model allows a stale data to be returned to the user. While strong consistency model does not allow any stale data returned to the user, and always enforces consistency between the original data and the client cached copies. Table 2 shows this two dimensional classification and the representative scheme for each category.

Table 2. A classification of cache consistency schemes

| | pull-based | push-based |
|---|---|---|
| Weak | TTL | Synchronous Invalidation |
| Strong | Lease | Asynchronous Invalidation |

Time-To-Live(TTL) based approaches are typical pull-based weak consistency schemes. The source specifies a Time-To-Live(TTL) value for every data. A cached data whose caching time is shorter than its TTL is considered as valid. After its TTL expired, the data is considered obsolete and the client can use different methods to poll the server to see if the data has been changed. It is a weak consistency scheme, because TTL is an estimated value, and the original data may get updated before the TTL expires for the cached copy. Lease based schemes [42] [43] [44] use similar control mechanisms, except that the server guarantees that the data is not going to be changed before issued leases expire. Under both schemes clients need to poll the server to validate the data after TTL or lease expires.

A widely used approach to assign TTL values is the adaptive TTL approach [40, 45]. It makes use of the fact that file lifetime tends to be bimodal distributed [46]. That is, if a data item has not been modified for a long time, it tends to remain the same for a longer time. On the other hand, if a data item is frequently modified, it is likely to be modified again in the near future. Based on this observation, the adaptive TTL scheme assigns an item's TTL value as a percentage of the time since the item's last modification. This approach is studied in [47] and the results show that the adaptive TTL scheme performs better than other weak consistency schemes.

On the other hand, the representatives for push-based consistency model is synchronous invalidation and asynchronous invalidation for weak and strong consistency respectively. Under synchronous invalidation schemes, server periodically sends out invalidation reports or data updates. It is a weak consistency model, because the client can access a stale data during the invalidation intervals. Under asynchronous invalidation schemes, server pushes invalidation reports to all related clients right after data modifications. The problems with push-based schemes are that it imposes a heavy burden for the server and it is very difficult to ensure that all the clients receive the invalidation reports given wide spread clients and varying network conditions. To alleviate these problems, in [48] they presented a multicast based invalidation approach. The basic idea is to assign each data a multicast group which consists of every cache that requested the data. When the data is changed, the server only needs to send out one invalidation message to the multicast group, and it is up to the routers to ensure that every client receives the invalidation report. In practice, push-based

invalidation schemes are mostly used in cellular networks, where upstream links have very limited bandwidth while the downstream links have much higher communication capacity. Since pull-based schemes require substantial upstream communication, many push-based schemes are proposed and developed for cellular networks.

In this thesis, we assume the TTL(Time-To-Live) scheme is used to maintain consistency between client-side cache and the data server. The TTL scheme is enough and widely used in practice, for example, HTTP 1.1 [49] uses the "max-age" directive to specifies the TTL value of a replied item.

### 3.3. Data access model

According to the message exchange sequences between the original data provider and the data consumer, data access models can be classified into the following three types.

*On-demand data access.* Under this model, the data provider issues a data reply upon receiving a request from a consumer. This is the most popular model used in current network applications, e.g. file transfer, Web browsing, database queries etc. For this type of applications, maintaining cache consistency is not necessary for all the cases, e.g. file downloading. If cache consistency is needed, pull-based consistency model is most popularly used. For instance, HTTP 1.1 uses the "Expires" header or the "max-age" directive to control how long the content can be cached [49], which follows the TTL method described previously. Push-based consistency model can be used for this application type too, but for the MANET environment it introduces a

significant burden on the server given the number of wide spread clients and unreliable network links. Even though multicast can be used to alleviate this problem, it still brings more overhead by maintaining the multicast tree.

*Publisher/Subscriber model.* Under this model, data consumers first register with the data provider what data they are interested in. When that data is available, the publisher delivers the data to the subscribers. For example, an exploration team member can register with the colleagues using a statement like "tell me whenever you see a wild elephant" or "tell me your status every 15 minutes". The most natural way to maintain cache consistency for Publisher/Subscriber applications is using push-based schemes, because either way the publisher needs to deliver the most recent data to the subscribers, which also serves the purpose of consistency validation. The clients may also use TTL, lease etc. pull-based model to maintain cache consistency. But after the specified time expires, a client needs extra message exchanges to verify data consistency from the server.

*Broadcast-like model.* Under this scheme, the data provider broadcasts its data using a certain method, and consumers listen to the broadcast to get the data. Application examples include tourist information, and scrolling news broadcasted on a certain channel. Similarly with the Publisher/Subscriber model, the proper way of maintaining cache consistency is by broadcasting invalidation or updates. Pull-based consistency model can also be used, but we have to pay the price of extra message exchanges.

### 3.4. Data request formats

A user's data request can be categorized into the following three types, based on how the requested data is represented in the request:

*ID-based request.* In this type of requests, the requested data is represented by the data's unique ID which can usually be mapped to the location of the data. The application examples include Web access applications and file transfer applications, in which the user's data request is composed of the server's address and the complete path of the requested data.

*Content-based request.* In this type of requests, the requested data is represented by its attributes, which is also called keyword-based request. Application examples include resource discovery and service location systems, in which the user tells the application what they want instead of where to get. To identify the requested data, the paper [50] presented the design and implementation of an intentional naming system (INS) that names a data based on the attributes.

*Hybrid request.* In this type of requests, the requested data is identified by the server's address as well as the attributes of the data. Application examples include database query applications, where the user tells the application the data server address and the desired attributes of the data.

Content-based request is different from the other two types at that there is no deterministic location of the requested data, and the request may not always get resolved successfully. For this proposal, we focus on ID-based requests and hybrid requests, and the requests will be forwarded to the data server after cache resolution

fails.

In this thesis, we focus on the most popularly used on-demand data access model with TTL cache consistency control. The challenges for designing a cooperative caching scheme for MANET applications include:

- There is no central coordination point for cooperative caching in MANETs. All the mobile nodes are autonomous, and it is difficult to assign caching tasks to individual nodes in a highly dynamic network.

- Because of energy and bandwidth restrictions, inter-cache communication overhead shall be minimized.

- Caching nodes are dynamic, previous cooperating nodes may move out of range or switch to power-saving mode. How to discover and establish relationship with new cooperating nodes is a question to be answered by the cache resolution scheme.

- The cooperative cache management scheme has to answer questions like whether a node shall cache a data needed by itself or a data needed by the neighborhood.

CHAPTER 4

# Proposed Approach

In this section we start with the traditional scheme, explore how it can be improved step by step, and come up with the cocktail cache resolution scheme and zone-based cache management scheme at the end. The code name COOP is referred to the conjunction of the cocktail cache resolution scheme and zone-based cache management scheme.

As described in the system model, we consider a MANET environment where nodes mobility pattern is unknown. To enable connections to external networks such as Internet, the mobile nodes that have interface to external networks(Figure 3) serve as gateways between internal and external networks. A data server can reside inside or outside the MANET. Mobile nodes can read data from the data servers. But only the servers have the privilege to create or update the data. Mobile nodes can cache a copy of the data in its local storage. The TTL(Time-To-Live) scheme is used to maintain consistency between client-side cache and the data server. The basic operation is that each data item is assigned a TTL value by the server, and a cached data item is considered valid as long as the caching time has not exceeded the TTL

value. The TTL scheme is popularly used, for example, HTTP 1.1 [49] uses the "max-age" directive to specifies the TTL value of a replied item.

## 4.1. Overview of COOP

An overview of COOP is illustrated in Figure 4. A running COOP instance receives data requests from user's applications, resolves the requests using the cocktail cache resolution scheme (Section 4.2) and returns the data to user's applications. Besides resolving user's data requests, COOP also stores and manages requested data in a local cache for future reuses. The cache management (Section 4.3) scheme decides what data to place/purge in the local cache of a mobile node. For COOP, the cache management scheme is based on the supply and demand from within its cooperation zone. That is, the cache management scheme evaluates a data item's importance not only based on the local demand but also based on the demand from other nodes in its cooperation zone. The cache management scheme discriminates primary data copy and secondary data copy. A cached item is labeled as secondary copy because there already exists a primary copy of this item in its cooperation zone. We put primary copy at a higher caching priority, so that when needed a secondary copy can be replaced by another item's primary copy. This way the cooperated caches can reduce duplications between each other and accommodate more data varieties to improve the overall performance. For implementation, COOP can be put on the middleware level, acting as a proxy for user's applications, and useing underlying network stack to communicate with COOP instances running on other nodes.

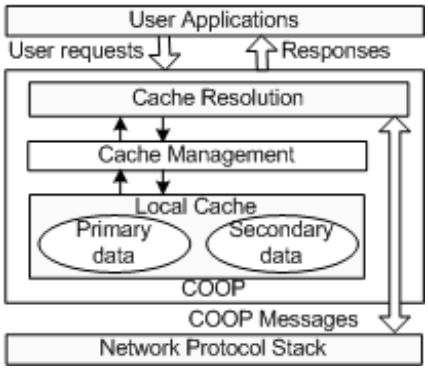


Figure 4. System Architecture

### 4.2. Cache resolution

Cache resolution addresses not only how to resolve a data request with minimal cost of time, energy, and bandwidth, but also how to improve request success ratio. The second part is very important in MANETs since a data request/reply can get dropped easily because of interference, network congestion, or more commonly a forwarding node moving out of range and the path breaks.

The traditional way of resolving a user's data request, called SimpleCache in this thesis, is first to check the local cache. If the local cache has a up-to-date copy of the requested data, the cached copy is returned to user. However, if cache misses, the request is forwarded to the data server to retrieve the data. This is not a problem if the user is connected to the server through reliable high-speed connections. But for MANETs, every hop is a risk because of signal interference and link breaks caused by nodes movement. For example, in Figure 5, if any forwarding node, node 7 in this example, moves out of range, the requests from downstream nodes have to wait for the

reconstruction of a new route to the server. During this waiting period, requests can be dropped because of a full queue. Figure 5 illustrates a scenario where SimpleCache can be easily improved. In this example, since node 3 already has $x$ in its local cache, node 2's request for $x$ should be able to get answer from node 3, way ahead of reaching the data server. The Hop-by-Hop cache resolution scheme can be instrumented to enable this. The Hop-by-Hop scheme is described in Section 4.2.1. However, Hop-by-Hop only enables upstream nodes to resolve data requests for downstream nodes. In this example, if node 4 requests $x$, Hop-by-Hop will not be able to help resolving the request, even though they are immediate neighbors. To this end, we came up with a zone-based cooperative caching scheme. A node's cooperation zone is consisted of the surrounding nodes within $r$-hop range of the node. The parameter $r$ is called the radius of the cooperation zone. In this example, using the zone-based scheme, node 4 will be able to get the requested data $x$ from a node in its cooperation zone, which is node 3 in this case. With the Hop-by-Hop scheme, a downstream node can reuse the cached data in an upstream node, while with the zone-based scheme, a node can reuse any data items cached within its cooperation zone. Section 4.2.2 describes the zone-based scheme. Finally, we combine the Hop-by-Hop scheme and the zone-based scheme and come up with a cocktail scheme to address the cache resolution problem in MANET.

The emphasis of cache resolution in cooperative caching is to answer how nodes can help each other resolve data requests to improve the overall performance. The Hop-by-Hop scheme is an intuitive approach, that is, a data request is checked every

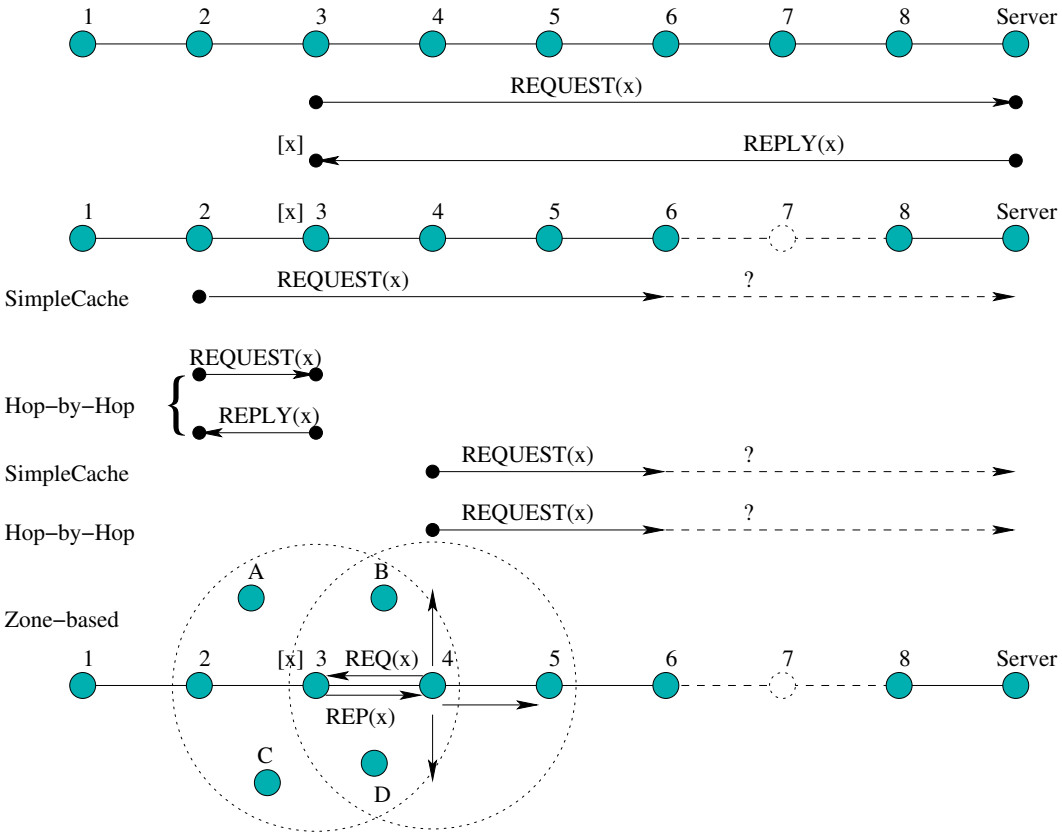


Figure 5. Improving SimpleCache

hop on the forwarding path. If a forwarding node has the requested data, the request gets resolved before reaching the server. But the cooperation is very limited by just using Hop-by-Hop: only between upstream nodes and downstream nodes. Then the zone-based cooperative caching scheme came to mind, to enable higher cooperation among the mobile nodes. But why the surrounding nodes are chosen to compose the cooperation zone? The thought is that cooperating nodes shall have relative faster and more reliable network connections between them, otherwise the cooperation will incur significant communication overhead. In ad hoc networks, throughput and reli-

ability of a network connection heavily depend on the number of hops covered by the connection [5]. As the number of hops increases, the throughput and reliability decreases. In Zone-based cache resolution, we propose to enable cooperation among the nodes within a certain number of hops, so that the cooperating nodes can communicate with each other with less overhead. To resolve data requests from the cooperation zone, nodes can proactively record available data items within its zone or reactively discover available data items by restricted flooding. This will be detailed in Section 4.2.2.

**4.2.1. Hop-by-Hop cache resolution.** For on-demand data access applications, the traditional way of resolving a data request is to check the local cache first and send the request to the server after local cache misses. This scheme is referred to as SimpleCache in this thesis as well as in the paper [8]. This scheme works well as long as the connection to the server is reliable and not too expensive, otherwise, it results in failed data requests or requests time out. To increase data availability and reduce the cost of time and energy, Hop-by-Hop cache resolution allows a node on the forwarding path to serve as a proxy for its received request. If a forwarding node caches an unexpired copy of the requested data, it can send a reply to the requester and stop forwarding the data request. For example, in Figure 5 assuming node 2 issues a request for data $x$, before node 3 relays the request, it checks if it has a valid copy of $x$ in its local cache. If so, node 3 stops forwarding the request and sends the copy back to node 2.

This scheme can be further extended in the following way. If a forwarding node

does not have the data locally but it knows a closer data source (e.g. by previously recorded data request entry from its cooperation zone, Section 4.2.2), it can also redirect the request to that closer data source, which reduces the travel distance of data messages and helps minimize the energy cost and response delay. In case a requesting node does not want the forwarding nodes resolve its requests, it can set a flag in the request message which indicates that Hop-by-Hop resolution is disabled.

By comparing SimpleCache and Hop-by-Hop, we can see another advantage of Hop-by-Hop or disadvantage of SimpleCache. Since Hop-by-Hop provides a chance of resolving a request before reaching the server, the average hop count covered by requests/replies is reduced. As studied in [51], the average traffic and interference received at a node is dependent on the average hop count ($\Lambda = \lambda E[h]$, $\Lambda$ is the expected traffic per node, $E[h]$ is the expected hop count, and $\lambda$ is the mean generated new traffic per node per time interval). This indicates that Hop-by-Hop can help to reduce the average traffic and interference received at a node.

As explained in Section 2, this approach is similar with the approaches in [8] in that both allow forwarding nodes to respond data requests. However, the difference is how forwarding nodes decide what to cache. In [8], forwarding nodes decide what to cache based on the statistics collected from forwarded requests. In COOP, we do not base caching decisions upon statistics of forwarded requests, and the reason is that the forwarding nodes may move to another area and previously collected statistics on forwarding nodes are difficult to reflect future needs.

**4.2.2. Zone-based cache resolution.** Hop-by-Hop improves the Simple-Cache scheme by enable the forwarding nodes to do cache resolution for the requesting nodes. However, Hop-by-Hop only lets upstream nodes to resolve data requests for downstream nodes. In the example illustrated in Figure 5, if node 4 requests $x$, Hop-by-Hop will not be able to help resolving the request, even though node 3 is an immediate neighbor and caches the requested data. To enlarge the cooperation among different nodes, we came up with a zone-based cooperative caching scheme.

A node's cooperation zone is consisted of the surrounding nodes within $r$-hop range of the node. The parameter $r$ is called the radius of the cooperation zone. In Section 5 we will study how the value of $r$ impacts the performance. To resolve a data request from the cooperation zone, both reactive and proactive approaches can be used.

The reactive approach is used when a node does not have information about whether the requested data item is available in the zone or not. In this case, the node can reactively discover this by flooding the request within the zone. To restrict the flooding range, the TTL value of the request is set to the zone radius $r$. If a node receives a broadcasted request, it checks its own cache for the requested data. If the node has an up-to-date copy of the requested item, it replies the item to the requesting node. Otherwise, it records the received request, which provides a location hint if this node requests the same item in future. Note that the broadcasted request not only serves the purpose of discovering the closest cache location, but also can serves as an announcement in the neighborhood. It is possible that a node receives

Table 3. An example for Recent Requests Table (RRT)

| Sender | Time | Target |
|---|---|---|
| 192.168.0.11 | 15:20:59:03:26:2005 | $D_1$ |
| 192.168.0.15 | 15:25:59:03:26:2005 | $D_2$ |
| 192.168.0.18 | 15:26:59:03:26:2005 | $D_3$ |

multiple replies from the broadcast. If this happens, the first reply is processed and the rest of them are discarded. It is also possible that there is no reply received from the broadcast before time out. In this case, the request will be forwarded to the server to retrieve the requested data.

However, since network flooding generates significant amount of traffic, it needs be restricted as much as possible. A proactive approach can be used to reduce the number of flooding and the discovery latency, which we call "Profile-based Resolution". Profile-based Resolution maintains a historical profile of previously received data requests. It is to avoid duplicated flooding for the same data item and to determine the closest data cache based on the profile. In this scheme, previously received data requests are recorded in a table which is called RRT (Recent Requests Table). Each entry of RRT records the information of one previously received request, i.e. the sender, the requested data, and the timestamp of the request. The size of RRT can be specified by the user. If RRT is full, the newest request over-writes the oldest request entry. An example of RRT is shown in Table 3.

In operation, a node checks RRT after its local cache misses and before flooding a request. If matching entries are found, the node compares its network distances to

these matching caches and the original data source, and selects the closest one to send the data request. If the requested data is successfully returned, it means the RRT entry is still good. Otherwise if the data request fails, it means the cache recorded in the RRT entry does not have the requested data. Therefore the RRT entry is removed, and the reactive approach is used to resolve the data request.

This scheme can be further extended by profiling and recognizing reliable long-term cooperative nodes. Unlike P2P networks though, to choose a partner not only needs to consider the common interest, but also needs to consider other factors like communication cost, network link reliability, and storage space constraints.

In a heterogeneous networks, some nodes may be more powerful and have more capacity than other nodes. In this case, these powerful nodes will be more active in answering other nodes' requests, as they can cache more data items in their local cache and be active for longer time. The Profile-based Resolution can be extended by profiling and recognizing these powerful nodes, such that future requests can be directed to the powerful nodes to receive better services.

As explained previously, flooding can help to discover the closest cache around the requester, as well as to serve as an announcement in the cooperation zone such that the surrounding nodes will know who has the data next time. However, network flooding generates significant amount of traffic, and it shall be used on a restricted basis. Section 5 presents a detailed analysis and comparison of the resulted performance from different zone radius. In the following analysis, we simply study the average time cost of using an $x$-hop flooding and determine the optimal flooding range to minimize

this cost.

We assume that the time cost is proportional to the traveled hops of the data request. If a cache for the requested data is discovered with $x$-hop flooding, the cost is proportional to $x$. Otherwise, the data request is forwarded to the original data source after $x$-hop flooding, and the cost is proportional to $(x + L)$, where $L$ is the distance to the original data source in hops. To calculate the average data fetching cost in all situations, the remaining problem is how likely an $x$-hop flooding will discover a satisfying cache? To estimate this possibility, we made the following assumptions:

- $P_d$: the possibility of each node to cache the data $d$. We assume that every node has the same value of $P_d$.
- $\rho$: the average node density. The number of nodes within $x$-hop range of the studied node is computed by $\rho\pi x^2$ .

Then the probability to discover a cache for data $d$ within $x$-hop flooding $(0 \leq x < L)$is $P(x) = 1 - (1 - P_d)^{\rho\pi x^2}$. The average time cost of fetching $d$ is:

$$\overline{C} = x + L(1 - P_d)^{\rho\pi x^2} \tag{4.1}$$

Figure 6 shows the relationship between average time cost $\overline{C}$ and flooding range $x(\rho = 1,\ P_d = 0.1, L = 5, 10, 20, 50, 100)$. We can see that the lowest average cost happens when $x = 2$ for $L = 5$ hops, $x = 3$ for $L = 10, 20$ hops, and $x = 4$ when $L = 50, 100$ hops. As $L$ increases, the optimal flooding range increases very slowly to achieve minimal average cost. However, note that we assume $\rho = 1$ in Figure 6. If node density becomes higher, the optimal flooding range shall be reduced accordingly.

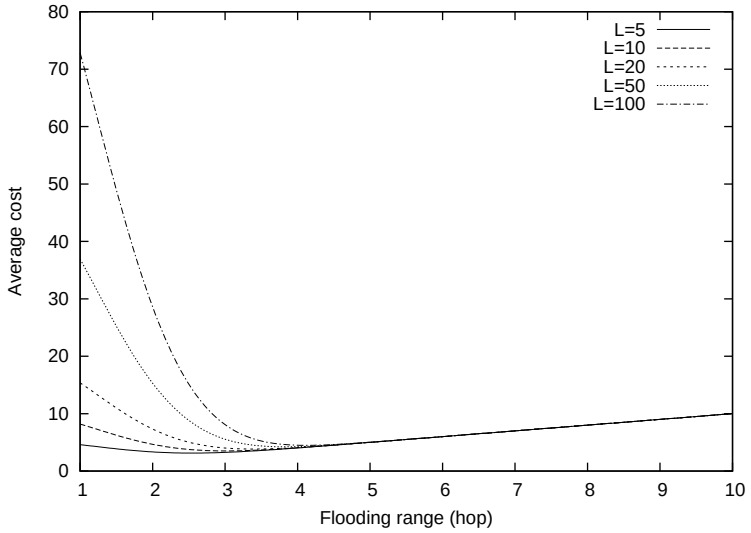


Figure 6. Average data fetching cost.

The conclusion of this analysis is that limited flooding helps to minimize average data fetching cost. The flooding range is dependent on the cost of fetching data from the original data source which is proportional to the distance of the data source. For most of the cases, the flooding range shall be restricted within 4 hops.

**4.2.3. The cocktail resolution scheme – COOP.** These schemes described previously benefit different phases of cache resolution. COOP uses a cocktail approach based on these basic schemes. As shown in Figure 7, COOP uses Profile-based Resolution after the local cache misses. If no matching cache is found or the request fails, COOP uses reactive approach to discover the data in its cooperation zone. If this again fails, COOP forwards the data request to the data server, and Hop-by-Hop resolution is used to resolve the request along the forwarding path. The formal description of this cocktail scheme is written in Section 4.4.

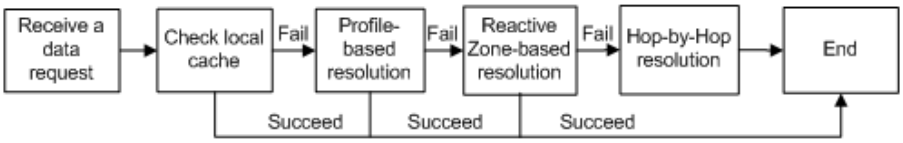


Figure 7. The Cocktail Resolution Scheme

In the cocktail resolution scheme, the ordering of the basic schemes is based on the feasibility consideration. For example, if the Hop-by-Hop scheme is put in front of the Zone-based resolution, it means that the request is first sent to the server and if none of the forwarding nodes or the server can answer the request, the request will be broadcasted and resolved within the cooperation zone. This different ordering will cancel the benefits of the Zone-based resolution, which tends to avoid or reduce long-distance communications with the server and the reliance on expensive links. If the reactive Zone-based resolution is put in front of the Profile-based resolution, it means that a request is first flooded in the cooperation zone regardless whether this request has been flooded before and then check who else has flooded the same request and send requests to the previous requester. This altered ordering cancels the meaning of using Profile-based resolution, which is to avoid duplicated flooding.

### 4.3. Cache management

Cache management studies how to manage the cached data items and how to perform cache replacement if there is not enough cache space when new item comes. The goal of cache management is to increase cache hit ratio, which largely depends on the capacity of the cache. For cooperative caching to achieve this goal, the emphasis

of cache management is how to manage an individual cache not only from the local node's point of view, but also from the view of the cooperative caching system, that is, try to maximize the effective cache size of the whole system. To maximize the capacity of cooperative caches, COOP tries to reduce duplicated caching within the cooperation zone, such that the cache space can be used to accommodate more distinct data items.

We categorize cached data copies based on whether they are already available in the cooperation zone or not. A data copy is primary if it is not available within the zone. Otherwise, the data copy is secondary. The reason of discriminating primary and secondary data is that cache miss cost is proportional to the travel distance of a data request, and primary data usually incur higher cache miss cost than secondary data. The inter-category and intra-category rules are used to decide caching priorities of primary and second data, which are described in the following.

**4.3.1. The inter-category rule.** The idea of inter-category rule is to put primary items at a priority level, i.e., secondary items are purged to accommodate primary items, but not vice versa. The problem in implementation is how to determine whether a data item is primary or secondary. We can do this by broadcasting a request in the cooperation zone, and those who have the data item sends a reply so that we know if the item is available in the neighborhood or not. But this would add extra overhead and probably is not worthwhile since the cooperation zone is composed of dynamic nodes.

Here we use a simplified approach to address this problem. After a node

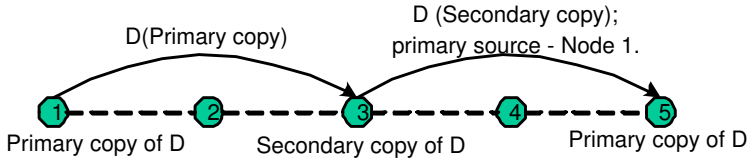


Figure 8. Primary and secondary copy

receives its requested item, it labels the item as primary copy if the item comes from a node beyond the zone radius. Otherwise, if a data item comes from within the zone radius, we need to further consider whether the data provider labels the item as primary or secondary. If the provider already labels its copy as primary, the new copy would be secondary since we do not intend to have duplicated primary copies for the same cooperation zone. On the other hand, if the provider tags its own copy as secondary, the provider needs to attach the information of the primary copy holder. If the primary copy holder is beyond the zone radius, the new copy is primary copy, otherwise, the new copy is a secondary copy.

An example is illustrated in Figure 8. We assume that the zone radius is 3 hops and node 1 holds the primary copy of data $D$ initially. When node 3 requests $D$, node 1 replies and indicates that it has the primary copy of $D$. Since $D$ already has a primary copy on node 1, which is within 3 hops, node 3 labels it copy of $D$ as secondary copy. Later, when node 5 requests $D$, node 3 replies and indicates it has a secondary copy and the primary copy is on node 1. Even though $D$ comes from within the zone, node 5 labels its copy of $D$ as a primary copy because the original primary copy is beyond the zone radius for node 5.

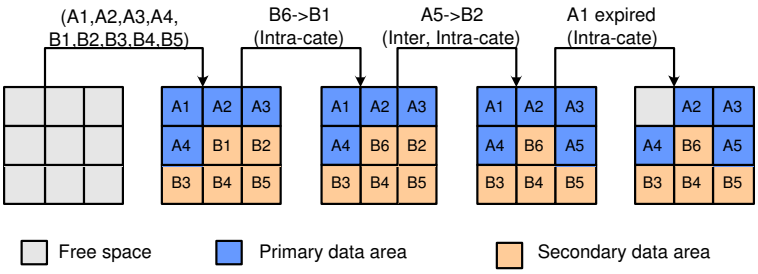


Figure 9. Cache management example

**4.3.2. The intra-category rule.** The intra-category rule is used to evaluate the data items within the same category. For this purpose, we simply adopt the LRU (Least Recently Used) algorithm. LRU is a widely used cache replacement algorithm in practice. It is also shown to have the best performance consistently among many tested cache replacement algorithms [12, 52].

Figure 9 shows an example of the application of the inter-category and intra-category rules. When the node first initializes, all its cache space is free. Then the primary copies $A1$ - $A4$, and secondary copies $B1$-$B5$ are entered to the cache one by one, at when all the space is taken. When another secondary copy $B6$ comes, $B1$, the secondary copy which is least recently used, is replaced by $B6$. When the primary copies $A5$ and $A6$ come, since there is no spare space, they took the space of the least recently used secondary copy $B2$ and $B3$. Finally when the TTL of $A1$ expires, its space is released and become spare space again.

## 4.4. Formal description of the proposed approach

In this section, we use pseudocode and flow charts to describe the data structures and event processing procedures for COOP.

**4.4.1. Data structures.** Each data item has a unique identifier. The notation $D_x$ is used to denote the data item whose identifier is $x$. The data structures in Figure 10 are maintained at each mobile node.

- *COOP_RANGE*: the radius of a node's cooperation zone in hops.
- *COOP_cache*: the local cache maintained by a node. The size of the cache is defined by the field *max_size*. Each cache entry is of the type *COOP_cache_entry*. It records the following information of a cached data item:
  - *dataID*: the identifier of the cached data item.
  - *timestamp*: the last update time of the cached item. This information is recorded from the data reply message.
  - *last_read*: the last read time of the cached item. The read could be originated from the node itself or from another node.
  - *ttl*: the Time-to-Live value of the cached item. The cached item is supposed to expire at the time $timestamp + ttl$.
  - *importance*: this field denote the cached item is a primary or secondary item.

  - *notes*: this field contains auxiliary information about the cached data item. For example, if the cached data item is a secondary copy, this field contains the node address where the primary copy is located.

- *RRT*: the recent request table maintained by a node. It uses a circular array to record the most recently received data requests. The size of RRT is defined by the field *RRT_SIZE*. Each RRT entry is of the type *RRT_entry*. It records the following information of a received request:

  - *dataID*: the identifier of the requested data item.
  - *requester*: which node requested this data item.
  - *timestamp*: the time when the request was sent.

**4.4.2. Message header.** The format of the message header is shown in Figure 11. The fields are described as follows:

- *type*: identifies the type of the message. If $type = 1$, it is a request message; if $type = 2$, it is a reply message.
- *dataID*: if it is a request message, this field contains the identifier of the requested data item.
- *hops*: the number of hops covered by this message so far.
- *serverAddr*: the data server's address.

- *seq[32]*: the message's sequence number, which is a unique identifier for a message.

- *notes[64]*: auxiliary information. For example, in a data reply message this field contains the data's timestamp and TTL information.

**4.4.3. Event processing.** Once the COOP program is started on a mobile node, it keeps listening to the assigned port, monitors and handles the following events. Figure 12 shows how COOP handles a data request from user's application. Figure 13 shows how COOP handles received messages.

- *COOP receives a data request from user's application.* COOP first checks the local cache. If the local cache has the requested data item and the TTL is not expired, this local cached copy is returned to user's application. Otherwise, COOP checks if this data item has been recorded in the RRT table. If there are one or more RRT entries found, COOP sends the data request to the closest source recorded. If there is no matching RRT entry, COOP broadcasts this data request within the cooperation zone by setting the TTL field of the broadcast message to the cooperation zone radius. At this time, COOP schedules a broadcast waiting timer to wait for the response. If a reply is received before the timer expires, the timer is canceled and the reply is sent to user's application. If no reply has been received when the timer expires, the data request is sent to the data server.

- *COOP receives a broadcasted data request.* COOP first checks if this broadcast is a duplicated broadcast. Note that COOP records broadcasted requests in the RRT table, and to check the duplication COOP only needs to check if this request has been recorded in the RRT table. If this is a duplicated broadcast, COOP drops this request and continues listening. Otherwise, COOP inserts the request in RRT and checks if the requested item is in the local cache. If the local cache does not have the data item, no reply is sent. Otherwise, COOP replies the data to the requester and update its cache information (the last_read field for the data item).

- *COOP receives a data request dedicated to it.* COOP receives this message because it is recorded in the requester's RRT table. After receiving this message, COOP checks if the requested data item is still in its local cache. If so, COOP replies the data to the requester. Otherwise, COOP forwards the data request to the server. COOP also sends a failure notice to the requester, so that the requester can remove the stale entry from its RRT table.

- *COOP receives a failure notice.* As just said, this message is received because the source recorded in the RRT table failed to serve the data request. Upon receiving this failure notice, COOP removes the corresponding RRT entry.

- *COOP receives a data request dedicated to another node.* This is because this COOP node is on the forwarding path between the requester and the data source. After receiving this type of message, COOP checks if it has the requested data item in its local cache. If found, COOP stops forwarding the request and

replies the data back to the requester. If not, COOP forwards the request to the next hop on the path.

- *COOP receives a data reply.* Once receiving a data reply, COOP checks if there is a broadcast timer that is waiting on this reply. If so, cancel the broadcast timer since there is no need to schedule other attempts for this data item. COOP then checks which data source this data item comes from. If this COOP node does not have a route to the data source or the data source is out of the cooperation zone range, the received data item is labeled as primary copy. Otherwise, if the data source is reachable and within the cooperation zone, the received data item is labeled secondary.

After receiving a data reply, COOP uses the following procedure to decide whether to cache the received item and how cache replacement is done if there is not enough cache space. We assume the data items have the same size in this algorithm. The flowchart is shown in Figure 14.

1. If the received item is a newer copy of a cached data item, replace the existed data item with the newly received one and return. Otherwise, go to the next step.

2. If there is spare space in the cache, insert the received item and return. Otherwise, go to the next step.

3. Check if there is any data item that has expired. If there exists such item, throw out the expired item, and insert the newly received item. Otherwise, go to the

next step.

4. If all cached items are up to date and there is no space left in cache, find the secondary data item that has the most ancient last_read field. Replace this item with the newly received item. However, if there is no secondary data item in the cache, go to the next step.

5. The procedure reaches here because the cache is full and all cached items are up-to-date primary copy. Now if the newly received item is a secondary copy, drop the newly received item and return. Otherwise if the data item is also a primary copy, replace the most ancient primary copy with the newly received item.

```
int COOP_RANGE;

struct COOP_cache_entry {
        u_int32_t   dataID;
        double      timestamp;
        double      last_read;
        int         ttl;
        u_int8_t    importance;
        char        notes[64];
};

class COOP_cache {
        public:
                COOP_cache();
                ~COOP_cache();
                void    cache_init(int ms);
                int     add(COOP_cache_entry newEntry);
                int     addAll(u_int32_t dataID[], int size);
                COOP_cache_entry get(int index);
                int     indexOf(u_int32_t dataID);
                int     set(int index, COOP_cache_entry newEntry);
                int     getTop();
                int     setTop(int inTop);
        protected:
                int     max_size;
                COOP_cache_entry *data;
                int     top;
};

struct RRT_entry {
        u_int32_t   dataID;
        nsaddr_t    requester;
        double      timestamp;
};

class RRT {
        public:
                RRT();
                int         getCurrent();
                int         setCurrent(int c);
                int         indexOf(RRT_entry e);
                RRT_entry   get(int index);
                int         set(int index, RRT_entry e);
                int         add(RRT_entry e);
                int         lookup(u_int32_t dataID);
        protected:
                RRT_entry requests[RRT_SIZE];
                int current;
};
```

```
struct hdr_coop {
        u_int8_t        type;
        u_int32_t       dataID;
        int             hops;
        nsaddr_t        serverAddr;
        char            seq[32];
        char            notes[64];
};
```

Figure 11. COOP message header

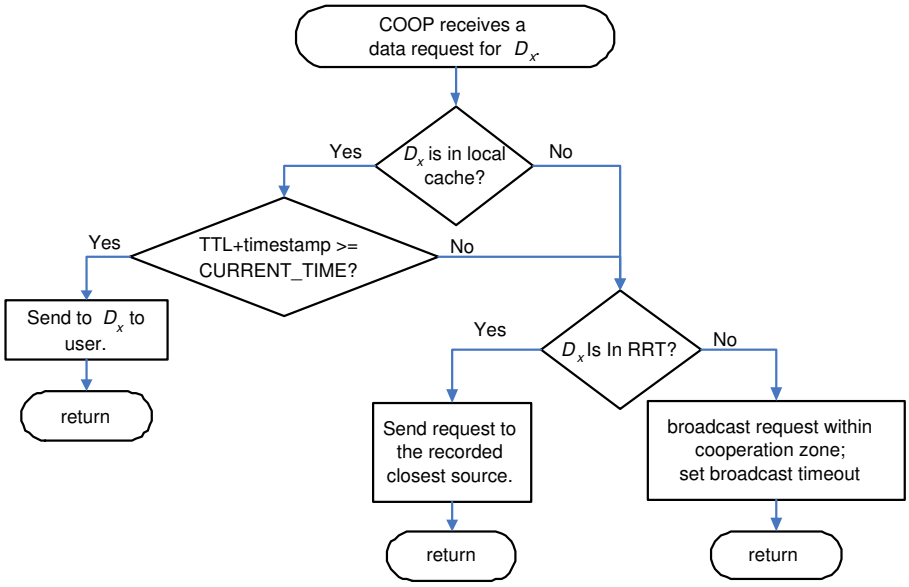


Figure 12. Processing a request from user's application

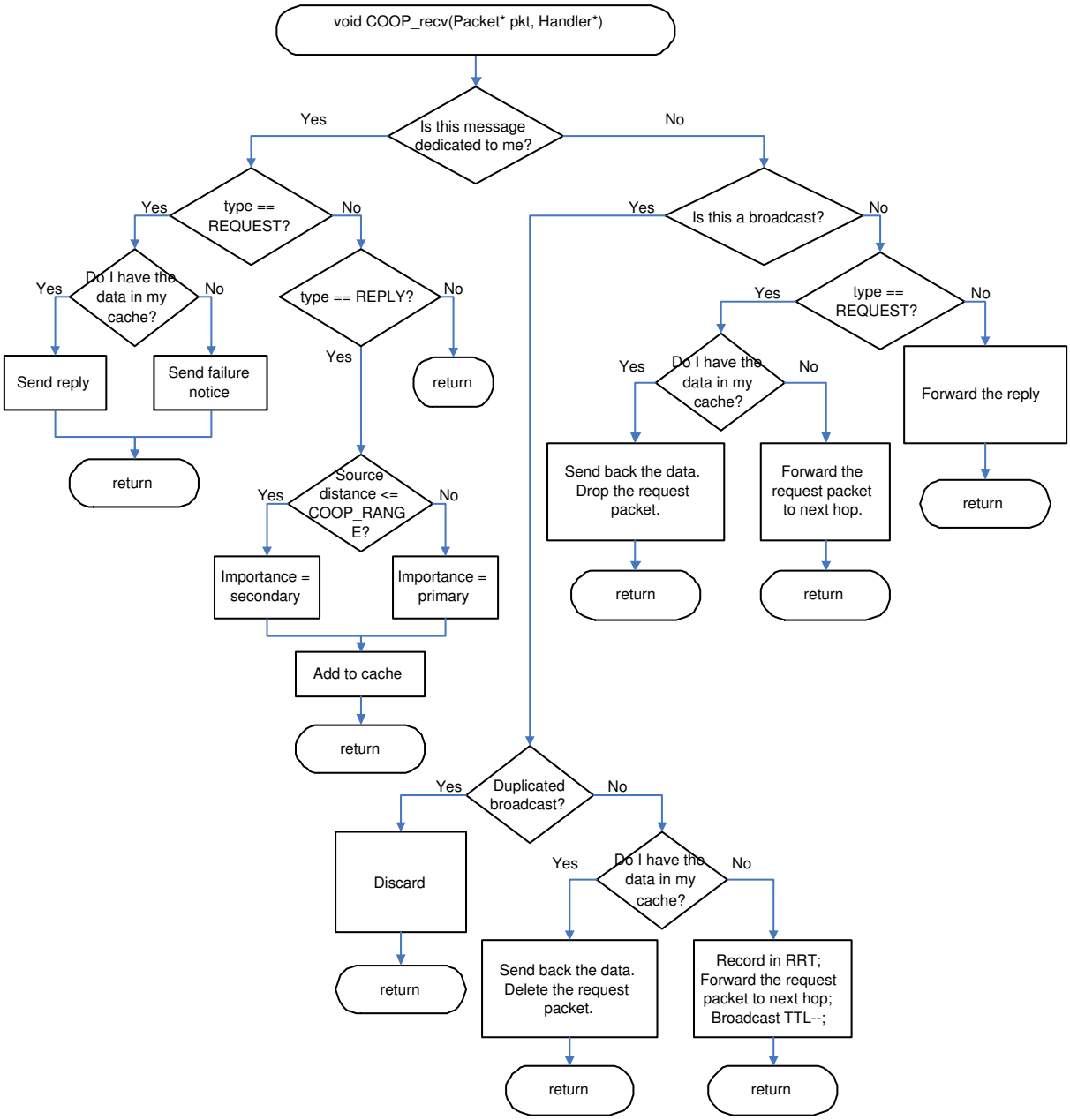


Figure 13. Processing received messages

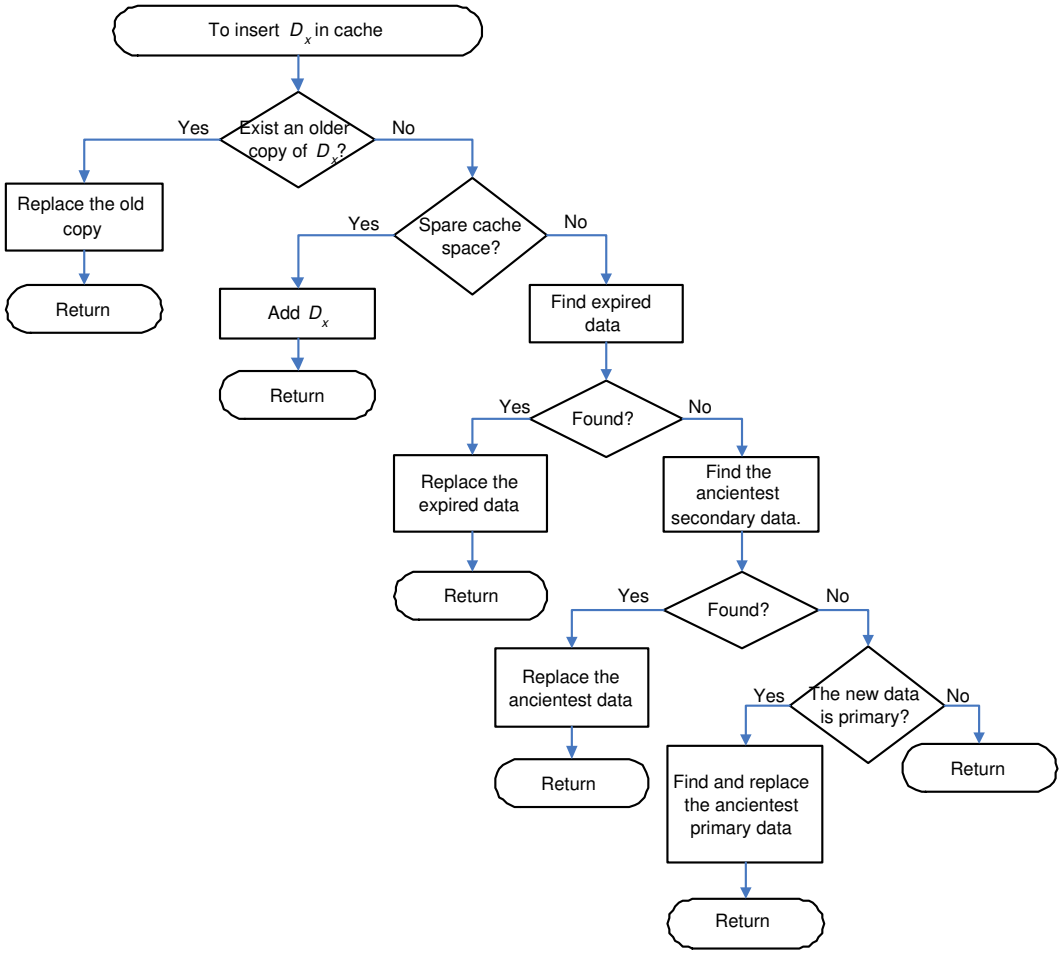


Figure 14. Adding a received data item to cache

CHAPTER 5

# Mathematical Analysis and Simulation Evaluation

## 5.1. Mathematical analysis

In this section, we set up a mathematical model to analyze the cooperative caching schemes described in the previous section. The goal of the analysis is to show the impact of the network environment on these approaches from the perspectives of response delay and energy consumption. The impact can come from different factors, e.g. node density, node velocity, and the data access pattern etc. While node velocity is an important factor for the stability of the network topology and hence impacts the performance, it is difficult to quantitate the impact and include it in the analysis. However, we will show the impact of node velocity in the simulation section.

**5.1.1. Performance metrics.** Two metrics are adopted in this mathematical analysis: average response delay and average energy cost per request, which reflects time efficiency and energy efficiency respectively. The definitions are:

- Average response delay: the average delay from sending a request to receiving the corresponding response.

- Average energy cost: the average energy consumed while sending, receiving, and forwarding a data request and the corresponding reply.

Data availability is another important performance metric. However, since the measurement of data availability heavily depends on the factor of node velocity or link breakage rate which is not included in the mathematical analysis, we will show its evaluation in the simulation section.

**5.1.2. Assumptions and notations.** The following assumptions are made for this analysis:

- The response delay and energy cost is 0 if local cache hits. This is reasonable because the local cache access cost can be ignored when comparing to inter-node communications.

- If a data request cannot be satisfied from the local cache, we assume the response delay is proportional to the travel distance (in hops) of the data request, that is, $Response\_delay = \alpha \times Travel\_distance$ and $\alpha$ is a constant. In the analysis we ignore the constant $\alpha$ since this does not change the analysis and comparison results. This assumption is based on the observation of the results from simulation and implementation – we found each hop takes approximately the same amount of time and the response delay is proportional to the number of hops covered by the request.

- If a data request cannot be satisfied from the local cache, we assume the energy cost of a data request is proportional to the number of messages trigged by

the request, that is, $Energy\_cost = \beta \times Number\_of\_messages$. Similarly, we ignore the constant $\beta$ since it does not change the results. We ignore the energy consumed during processing a data request or reply. This is based on previous research results that the power consumed during instruction processing can be ignored comparing to the power consumed during data transmission/reception.

Notations used in this analysis are listed in Table 4. $P_d$ is the average probability of a node caches a copy of data $d$. It can be estimated by the percentage of caching nodes in the network from history profiles. $P_d$ can be different for different data items, but we suppose for the same data item every node has the same $P_d$. $P'_d$ is the probability of a node in the cooperation zone of the requesting node caches $d$ locally. We suppose $P'_d \geq P_d$, because the cache replacement algorithm prioritize the data items used by the nodes in the cooperation zone. $P''_d$ is the probability of a forwarding node outside the cooperation zone of the requesting node resolves the data request in the Extended-Zone-based approach. We also suppose $P''_d \geq P_d$, since the forwarding nodes can resolve a data request not only from its own cache, but also from other nodes in its zone. $L$ is the distance between the requesting node and the data server. The distance is represented in number of hops (in case there is a bottle neck link between the client and the server, we suppose the bottleneck link is equal to a multi-hop link). The average node density $\rho$ is used to estimate the total number of nodes in a two-dimensional space. For example, the number of nodes within $r$-hop range of the requesting node is estimated by $\rho\pi r^2$.

Table 4. Notations

| Notation | Description |
|---|---|
| $D_0$ | average response delay of the SimpleCache approach. |
| $D_1$ | average response delay of the Hop-by-Hop approach. |
| $D_2$ | average response delay of the Zone-based approach. |
| $D_3$ | average response delay of the Extended-Zone-based approach. |
| $E_0$ | average energy cost of the SimpleCache approach. |
| $E_1$ | average energy cost of the Hop-by-Hop approach. |
| $E_2$ | average energy cost of the Zone-based approach. |
| $E_3$ | average energy cost of the Extended-Zone-based approach. |
| $L$ | the distance (in hops) between the requesting node and the data server. |
| $P_d$ | the average probability of a node caches a copy of data $d$ locally. |
| $P'_d$ | the average probability of a node in the cooperation zone of the requesting node caches the requested data $d$ locally. |
| $P''_d$ | the average probability of a forwarding node outside the cooperation zone of the requesting node resolves the request for $d$ in the Extended-Zone-based approach. |
| $r$ | the radius of the cooperation zone of the requesting node. |
| $\rho$ | the average node density. |

**5.1.3. Performance analysis.** In this section, functions are derived for previously presented cooperative caching schemes (namely, SimpleCache, Hop-by-Hop, Zone-based, and Extended Zone-based schemes) to show their average response delay and energy cost based on different node density, access pattern, and server distance etc. The SimpleCache approach is served as a baseline in the performance analysis.

*Performance analysis of the SimpleCache approach.* For the SimpleCache approach, a data request is forwarded to the server after local cache misses. In this case,

the average response delay is:

$$D_0 = 0 * P_d + L(1 - P_d) \tag{5.1}$$

The average energy cost is :

$$E_0 = 0 * P_d + L(1 - P_d) \tag{5.2}$$

Formula 5.1 gives average response delay and Formula 5.2 gives average energy cost for SimpleCache. The average value is calculated based on two possible situations. One is that a data request is satisfied from the local cache. In this situation, the response delay and energy cost is 0 and the probability of occurance is $P_d$. The other possible situation is when the local cache misses and the data request is forwarded to the server. The probability of this occurs is $1 - P_d$. Since we can omit the constant coefficients, the response delay is equal to the travel distance $L$, and the energy cost is equal to the number of trigged messages which is also $L$.

The functions show that the response delay and energy cost increases as the server distance $L$ increases and decreases as $P_d$ increases. This is easy to understand. The further a data server is, the more time and energy it costs in fetching a data item from the server. The function shows that the increase is linear, but the increase is actually faster with the interference caused by other simultaneous transmissions in practice. On the other hand, if $P_d$ increases, it means that there is a better chance of getting the desired data from the local cache, which leads to smaller time and energy cost. To increase $P_d$, we need to either increase the size of effective cache space or adopt better cache replacement algorithms.

*Performance analysis of Hop-by-Hop resolution.* For the Hop-by-Hop cache resolution scheme, each forwarding node checks the forwarded data request and replies the request if the data is in the local cache. This actually enlarges the effective cache space, which is one of the optimization directions for the SimpleCache approach. For this approach, the average response delay is:

$$\begin{aligned} D_1 &= 0 * P_d + 1 * (1 - P_d)P_d + 2 * (1 - P_d)^2 P_d + \ldots + (L-1)(1 - P_d)^{L-1} P_d + L(1 - P_d)^L \\ &= \sum_{i=0}^{L-1} iP_d(1 - P_d)^i + L(1 - P_d)^L \\ &= \frac{(1 - P_d) - (1 - P_d)^L}{P_d} \end{aligned} \tag{5.3}$$

The energy cost is:

$$\begin{aligned} E_1 &= 0 * P_d + 1 * (1 - P_d)P_d + 2 * (1 - P_d)^2 P_d + \ldots + (L-1)(1 - P_d)^{L-1} P_d + L(1 - P_d)^L \\ &= \sum_{i=0}^{L-1} iP_d(1 - P_d)^i + L(1 - P_d)^L \\ &= \frac{(1 - P_d) - (1 - P_d)^L}{P_d} \end{aligned} \tag{5.4}$$

Formula 5.3 gives average response delay and Formula 5.4 gives average energy cost for Hop-by-Hop resolution. The average value is calculated based on $L + 1$ possible situations, i.e. a data request can be resolved at 0, 1, 2, ..., $L$-hop away. The probability of resolving a request at $i$-hop ($i = 0, 1, 2, \ldots, L - 1$) away is $(1 - P_d)^i P_d$, which equals to the probability of previous $i$ nodes fail while the $i+1$-th node satisfies

the request. Same as the analysis of SimpleCache, we ignore the constant coefficient. Therefore average response delay is equal to the average travel distance of a data request, and average energy cost is equal to average number of messages trigged by a request which is also equal to the average travel distance of a data request.

Figure 15 and 16 compared the average travel distance of a data request using Hop-by-Hop and SimpleCache schemes. Figure 15 shows that as $L$ increases, average travel distance of a request increases for both schemes. However, using Hop-by-Hop resolution the average travel distance is always smaller, and the increasing of average travel distance is slower than using the SimpleCache scheme. The explanation is that the worst case cost of Hop-by-Hop resolution is equal to the cost of using SimpleCache, where no forwarding nodes have the requested data and the request can only be satisfied from the server. Other than this, Hop-by-Hop resolution can resolve a request from some forwarding node and results in a shorter travel distance than SimpleCache. As $L$ increases, the cost of SimpleCache increases because a request has to reach the serve for resolution after local cache misses. For Hop-by-Hop resolution, however, the increase of $L$ also means the increase of the number of forwarding nodes, and this gives a better chance of resolving a data request before reaching the server. Moreover, the average travel distance of Hop-by-Hop resolution is actually bounded by $\frac{1-P_d}{P_d}$ no matter how large $L$ becomes (the limit of 5.3 and 5.4 is $\frac{1-P_d}{P_d}$ when $L$ approaches infinity). As $L$ and hence the number of forwarding nodes increases, the probability of resolving a data request on the forwarding path increases (unless $P_d = 0$ which means no node would ever cache $d$). When this probability equals 1 which means a

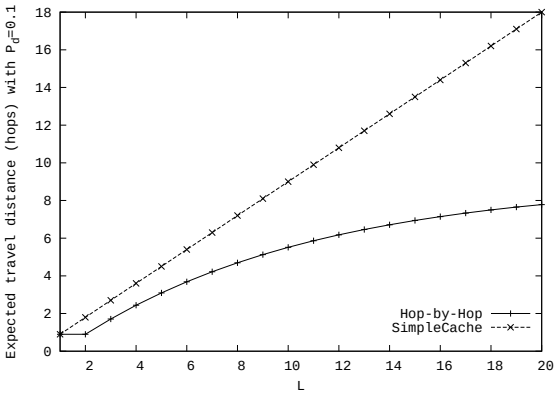


Figure 15. Expected travel distance with different $L$

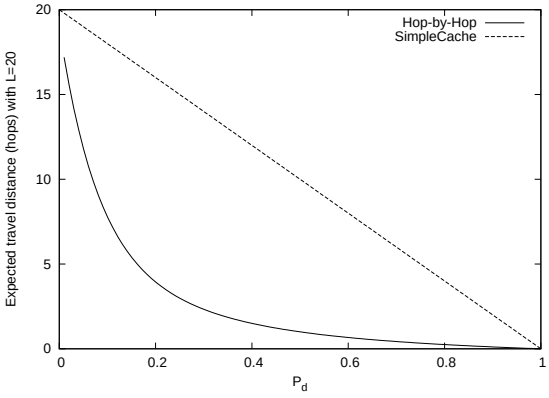


Figure 16. Expected travel distance with different $P_d$

request can always get resolved from some forwarding node, average travel distance no longer depends on how large $L$ is.

Figure 16 shows that as $P_d$ increases, the average travel distance of a data request decreases for both SimpleCache and Hop-by-Hop resolution. Using Hop-by-Hop resolution the decreasing is much faster than using SimpleCache. This is because SimpleCache only uses a node's local cache while Hop-by-Hop also uses other caches along the forwarding path, therefore the increase of cache probability $P_d$ exhibits a stronger effect in Hop-by-Hop resolution.

*Performance analysis of the Zone-based approach.* In Zone-based cache resolution, after local cache miss, the first attempt is to resolve a data request within the zone by using previously recorded information or broadcast. The TTL value of the broadcast message is set to the zone radius of the requesting node to control the flooding range. If no node answers the request, the request is forwarded to the server for resolution. Coupled with Zone-based cache resolution, Zone-based cache manage-

ment is used to manage a node's local cache with the aim of improving intra-zone cooperation. Let $r$ be the radius of the cooperation zone, and $P'_d$ the probability of a node in the zone caches $d$ locally. We expect $P'_d$ is no less than $P_d$, because Zone-based cache management prioritizes the data items used by the nodes in the cooperation zone. To derive the functions for average response delay and energy cost for Zone-based cooperative caching, we need to calculate the costs and probabilities of different possible scenarios. First of all, the probability of getting the requested data item $d$ from within the cooperation zone is:

$$P(r) = 1 - (1 - P'_d)^{\rho\pi r^2} \tag{5.5}$$

The average response delay is calculated based on three possible situations: a request can be satisfied from the local cache, from the broadcast within the cooperation zone, or from the data server. The probabilities of these three scenarios are $P'_d$, $P(r)$, and $1 - P(r)$ respectively. Correspondingly, the response delay is 0, $r$, and $r + L$ for those three situations. Therefore, the average response delay is:

$$\begin{aligned} D_2 &= 0 * P'_d + rP(r) + (r + L)(1 - P(r)) \\ &= r + L(1 - P(r)) \\ &= r + L(1 - P'_d)^{\rho\pi r^2} \end{aligned} \tag{5.6}$$

For average energy cost, if local cache hits, the energy cost is 0. If local cache misses and the data is found from within the zone, the energy cost is $\rho\pi r^2$. In this case, we assume the number of messages triggered by the broadcast is equal to the

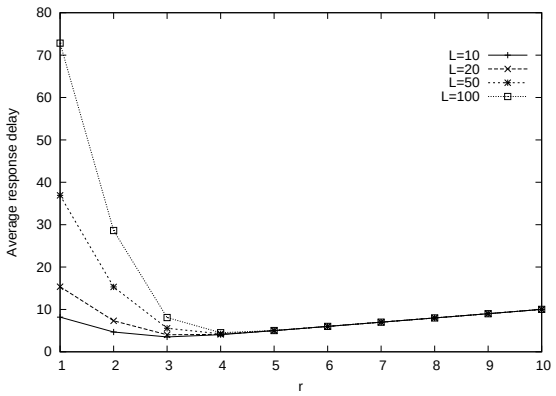


Figure 17. Plot of $D_2$ with different $L$

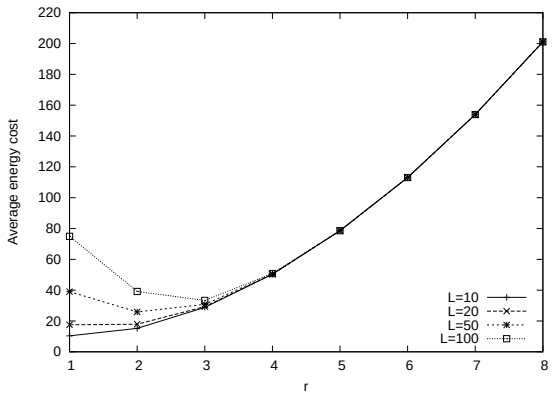


Figure 18. Plot of $E_2$ with different $L$

number of nodes in the cooperation zone. In a practical system, a node may receive multiple copies of the broadcast message. There are discussions about how to alleviate the broadcast storm problem [53]. In this analysis we will ignore the technical details, and simply assumes that a broadcast with TTL value of $k$ will reach all neighbors within $k$ hops of the originating node, and the number of messages is equal to the number of nodes within the broadcast range $\rho\pi k^2$. In the third scenario where the data is not found within the zone, the energy cost is $\rho\pi r^2 + L$ and the part of $\rho\pi r^2$ represents the cost of the broadcast which failed to find the desired data. The average energy cost is:

$$\begin{aligned} E_2 &= 0 * P'_d + \rho\pi r^2 P(r) + (\rho\pi r^2 + L)(1 - P(r)) \\ &= \rho\pi r^2 + L(1 - P(r)) \\ &= \rho\pi r^2 + L(1 - P'_d)^{\rho\pi r^2} \end{aligned} \tag{5.7}$$

Figure 17, 19, and 21 show the relationship between the cooperation zone radius $r$ and the average response delay $D_2$ for Zone-based resolution (except explicitly

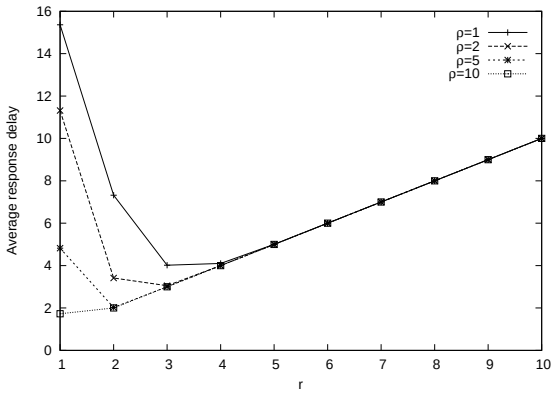


Figure 19. Plot of $D_2$ with different $\rho$

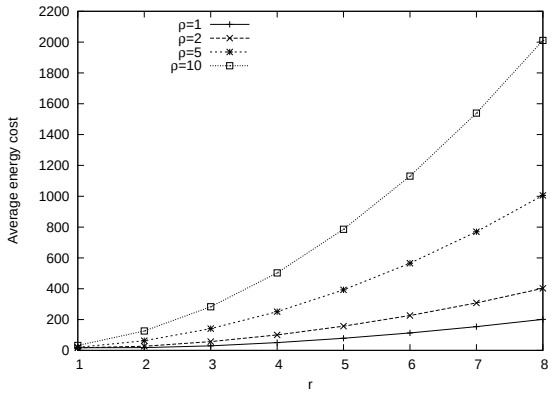


Figure 20. Plot of $E_2$ with different $\rho$

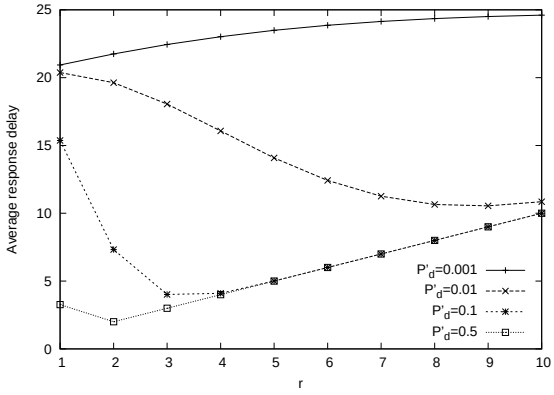


Figure 21. Plot of $D_2$ with different $P'_d$

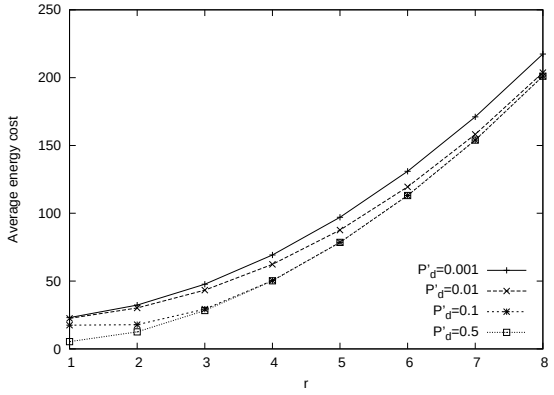


Figure 22. Plot of $E_2$ with different $P'_d$

indicates, the parameter values are $L = 20, P_d = 0.1, \rho = 1$). The lowest points of $D_2$ corresponds the optimal $r$ value that achieves the minimal average response delay. As we can see from the figures, the optimal $r$ varies for different parameters $L$, $\rho$, and $P_d$. Figure 17 shows that the distant the data server is, the larger value of $r$ is preferred. Figure 19 shows that optimal $r$ decreases while node density increases. The reason behind is that a smaller cooperation zone with higher node density can cover same number of nodes of a larger zone with a lower node density. Figure 21 shows that a small $r$ is appropriate for the data with extreme low (such as $P'_d = 0.001$) or extreme high access probability (such as $P'_d = 0.5$). One common observation from the these figures is that the optimal $r$ is no larger than 4 for most of the situations.

Figure 18, 20, and 22 show the relationship between the zone radius $r$ and the average energy cost $E_2$ for Zone-based resolution. The lowest points of $E_2$ corresponds to the optimal $r$ that achieves minimal average energy cost. For Zone-based cache resolution, both proactive and reactive approaches can be used to discover a data item within the zone. The worst case is using restricted flooding for in-zone data discovery, which has the energy cost of $O(\rho\pi r^2)$. This is exhibited in Figure 20. Figure 18 and 22 shows that unless $L$ is relatively large (such as 50), the energy cost increases as $r$ increases. One common observation from the these figures is that to achieve minimal energy cost $r$ shall be kept as small as possible; for most of the situations $r$ shall be restricted within 3 hops .

*Performance analysis of the Cocktail approach.* The Cocktail approach com-

bines Hop-by-Hop resolution and cooperation zone resolution. It first tries to resolve a data request in the zone. If this fails, the request is sent out to the server. On the forwarding path, a node can resolve the request if it has the data locally or it knows a closer data source. Let $P''_d$ be the probability of a forwarding node resolves the data request. $P''_d$ is no less than $P_d$, since the forwarding nodes can resolve a data request not only from its own cache, but also from some node in its zone. Using this approach, the average delay is:

$$\begin{aligned} D_3 &= 0 * P'_d + rP(r) + (2r+1)(1-P(r))P''_d + (2r+2)(1-P(r))(1-P''_d)P''_d \\ &\quad + (2r+3)(1-P(r))(1-P''_d)^2P''_d + \ldots + (r+L-1)(1-P(r))(1-P''_d)^{L-r-2}P''_d \\ &\quad + (r+L)(1-P(r))(1-P''_d)^{L-r-1} \\ &= rP(r) + (r+L)(1-P(r))(1-P''_d)^{L-r-1} \\ &\quad + P''_d(1-P(r)) \sum_{i=1}^{L-r-1} (2r+i)(1-P''_d)^{i-1} \end{aligned} \tag{5.8}$$

The average energy cost is:

Table 5. Response delay and energy cost for the Cocktail approach

| Possible situations | Probabilities | Response delay | Energy cost |
|---|---|---|---|
| The request is resolved locally | $P'_d$ | $O(1)$ | $O(1)$ |
| The request is resolved within the cooperation zone of the requesting node | $P(r)$ | $O(r)$ | $O(\rho\pi r^2)$ |
| The request is resolved at $i$ ($i = 1, 2, \ldots, L-r-1$) hop beyond the cooperation zone | $(1 - P(r))P''_d(1 - P''_d)^{i-1}$ | $O(2r + i)$ | $O(\rho\pi r^2 + r + i)$ |
| The request is resolved at the data server | $(1 - P(r))(1 - P''_d)^{L-r-1}$ | $O(r + L)$ | $O(\rho\pi r^2 + L)$ |

$$
\begin{aligned}
E_3 &= 0 * P'_d + \rho\pi r^2 P(r) + (\rho\pi r^2 + r + 1)(1 - P(r))P''_d + (\rho\pi r^2 + r + 2)(1 - P(r))(1 - P''_d)P''_d \\
&\quad + (\rho\pi r^2 + r + 3)(1 - P(r))(1 - P''_d)^2 P''_d + \ldots + (\rho\pi r^2 + L - 1)(1 - P(r))(1 - P''_d)^{L-r-2} P''_d \\
&\quad + (\rho\pi r^2 + L)(1 - P(r))(1 - P''_d)^{L-r-1} \\
&= \rho\pi r^2 P(r) + (\rho\pi r^2 + L)(1 - P(r))(1 - P''_d)^{L-r-1} \\
&\quad + P''_d(1 - P(r)) \sum_{i=1}^{L-r-1} (\rho\pi r^2 + r + i)(1 - P''_d)^{i-1}
\end{aligned}
\tag{5.9}
$$

Formula 5.8 gives average response delay and Formula 5.9 gives average energy cost for the Cocktail approach. The average value is calculated based on the possible situations listed in Table 5.

Figure 23, 25, and 27 show the relationship between the cooperation zone radius $r$ and the average response delay $D_3$ for the Cocktail approach (except explicitly indicates, the parameter values are $L = 20, P'_d = P''_d = 0.1, \rho = 1$). The lowest points

of $D_3$ corresponds the optimal $r$ value that achieves the minimal average response delay. As we can see from the figures, the optimal $r$ varies for different parameters $L$, $\rho$, and $P'_d$. Figure 23 shows that the optimal zone radius is 3 for $L = 10, 20, 50, 100$. Compared to Figure 17, the optimal radius is smaller than Zone-based resolution for large values of $L$, since the Cocktail approach added Hop-by-Hop resolution. Figure 25 shows that optimal $r$ decreases while node density increases. Same reason with Figure 19, it is because a smaller cooperation zone with higher node density can cover same number of nodes of a larger zone with a lower node density. Similar with Figure 21, Figure 27 shows that a small $r$ is appropriate for the data with very low (such as $P'_d = P''_d = 0.001$) or very high access probability (such as $P'_d = P''_d = 0.5$). One common observation from the these figures is that the optimal $r$ is no larger than 3 for most of the situations.

Figure 24, 26, and 28 show the relationship between the zone radius $r$ and the average energy cost $E_3$ for the cocktail approach. The lowest points of $E_3$ corresponds to the optimal $r$ that achieves minimal average energy cost. All the figures show that average energy cost increases as $r$ increases. Therefore to achieve minimal energy cost, $r$ shall be kept as small as possible for the Cocktail approach.

## 5.2. Simulation evaluation

**5.2.1. Simulation model.** The simulation of COOP is implemented by extending the popularly adopted network simulator NS-2 [54] (version 2.28).

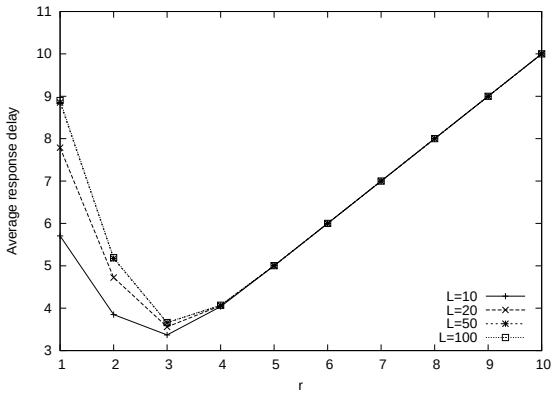


Figure 23. Plot of $D_3$ with different $L$

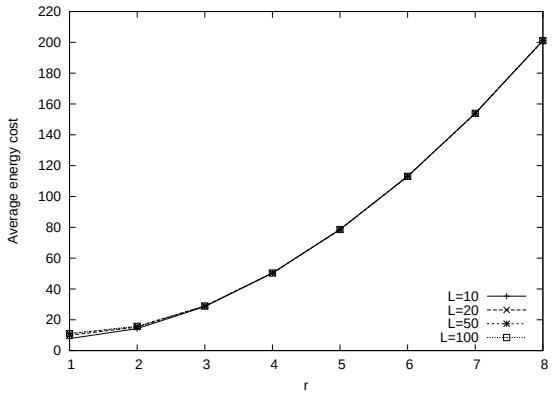


Figure 24. Plot of $E_3$ with different $L$

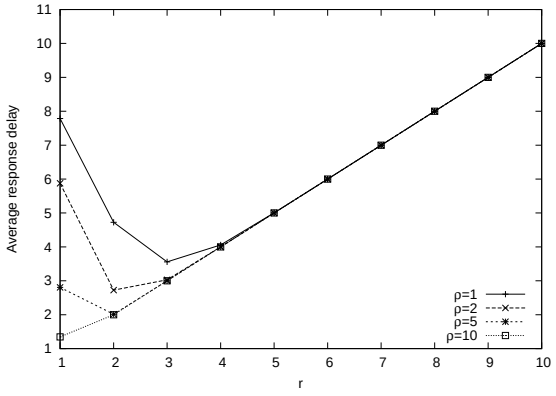


Figure 25. Plot of $D_3$ with different $\rho$

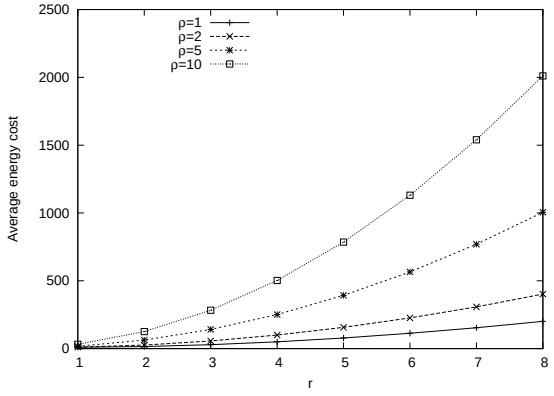


Figure 26. Plot of $E_3$ with different $\rho$

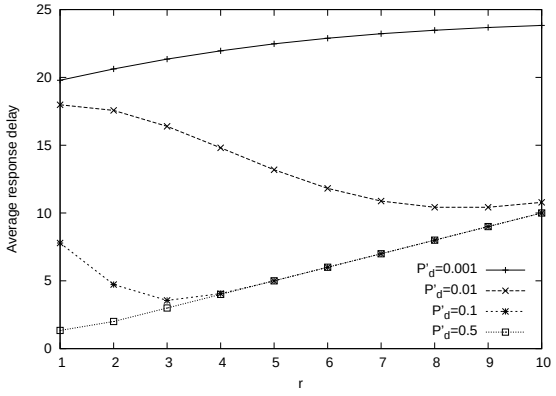


Figure 27. Plot of $D_3$ with different $P'_d$

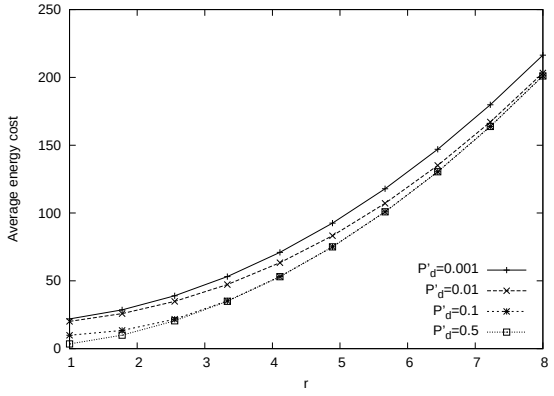


Figure 28. Plot of $E_3$ with different $P'_d$

5.2.1.1. *Mobility model.* The simulation models nodes' movement using the random way point model in a simulation area of 2500m×500m, which is similar with [7, 8]. In the random way point model [55], nodes are initially randomly distributed in the simulation area, each node has a random destination and moves toward the destination with a randomly selected speed from (0m/s, $v_{max}$m/s). The value of $v_{max}$ ranges from 0m/s to 30m/s in the simulations, which covers the typical pedestrian or automobile velocity. Once a node reaches its destination, it pauses for a period of time $T_p$ and continues moving to the next randomly selected destination. The value of $T_p$ ranges from 60s to 300s in the simulations.

5.2.1.2. *Client query model.* To simulate user's requests, we assume that each node generates a sequence of data requests with exponentially distributed time intervals. The request pattern follows Zipf-like distribution [14]. Zipf-like distribution has been used to model various behaviors such as the Web page request pattern. The normalized mathematical representation is $P(i) = \frac{1}{i^{\alpha} \sum_{k=1}^{n} \frac{1}{k^{\alpha}}}$, which is used to calculate the access probability of the $i$-th popular data item. In this formula, the parameter $n$ is the total number of data items, and the parameter $\alpha$ indicates how skew is the distribution. The larger the value of $\alpha$, the more concentration the requests put on popular data items. The value of $\alpha$ varies for different applications. For Web page accesses, it is in the range of [0.64, 0.83] according to the studies in [15]. In the simulations, the value of $\alpha$ ranges from [0.6, 0.9], which covers the observed values from Web page accesses.

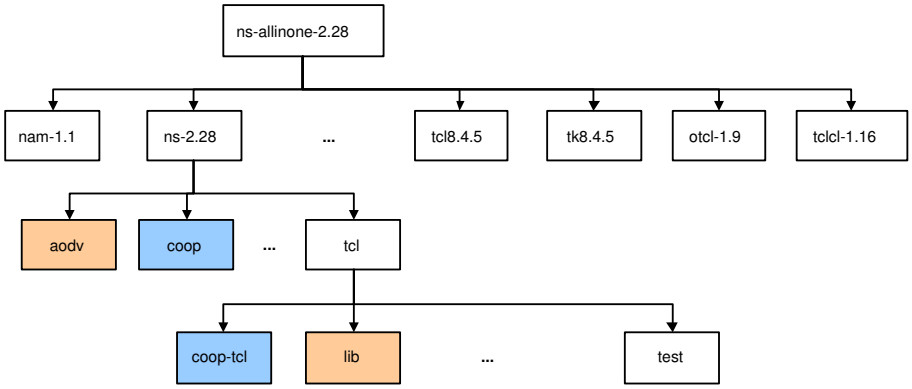


Figure 29. The directory structure

5.2.1.3. *Data access model.* A data server is placed at the corner of the simulation area, which is similar to the setup in [7, 8]. The server stores $N$ items, each of the items has a unique ID. To access a data item stored on the server, a client sends a request with the requested item ID to the server. The server processes the requests in the first-come-first-serve (FCFS) manner. In the simulation, the server can host 2000 items, and a client can cache 10 to 50 items.

5.2.1.4. *Directory structure.* The directory structure of this implementation is shown in Figure 29.

The directory ns-allinone-2.28/ns-2.28/coop contains the C++ source files for COOP. The C++ source code implements the function logics for COOP. It includes the following important classes and structures:

- COOPAgent: the C++ implementation of COOP's logic. This class contains the methods to process users' requests and the messages from other nodes.
- COOP_cache: the C++ implementation of COOP's cache management scheme.

This class has the methods to add, delete, and lookup an item from COOP's cache.

- hdr_coop: the message frame of COOP. It has the fields to identify the type of the message, the referenced data ID, and the referenced server ID etc.

- RRT: the C++ implementation of Recent Request Table. This class has the methods to add, delete, and lookup a received requests from the Recent Request Table.

The directory ns-allinone-2.28/ns-2.28/tcl/coop contains the Tcl source files and example scripts for COOP. The examples illustrate how to initialize COOP agent, COOP cache, data server, and Zipf's parameter $\alpha$ etc.

The implementation of AODV (under the directory ns-allinone-2.28/ns-2.28/aodv) is changed so that the routing agent can forward certain type of packets to the upper-level agent, which is implemented by COOPAgent.

In the directory ns-allinone-2.28/ns-2.28/tcl/lib, COOP's default packet size is specified and COOP's message header is included in the ns packet header list.

The function logics are described in Section 4.4 and the source code for COOP is attached in Appendix A.

**5.2.2. Simulation setup.** As previously stated, the simulation is based on the NS-2 [54] which is popularly adopted in networking research. In the simulation, COOP is implemented in ns-2.28 on a Linux platform. COOP uses AODV as the underlying routing protocol which is included in the simulator. To mitigate the impact

of different routing protocols, we represent the time efficiency using average travel distance (hops) for a request to get reply, instead of measuring time latency directly. The reason is that for same number of hops, different routing protocols (or conditions) may result in different time delay, thus making the time delay an unstable metric which can change because of many factors other than the effective cooperative caching protocol. On the other hand, the number of covered hops by a request mostly depends on where the requested item is found, which then relies on the cooperative caching protocol. The following metrics are used to evaluate COOP from the perspectives of data availability, time efficiency, and energy efficiency:

- Request success ratio: the percentage of successfully resolved requests. This is used to reflect the resulted data availability.

- Cache miss ratio: the percentage of requests that have to be forwarded to the server for resolution.

- Average hops covered per successful request: the average traveled hops for a request to get reply. This is used to reflect the time efficiency for cooperative caching protocols.

- Average messages sent per successful request: the average number of sent messages for resolving a request. This is used to reflect the message overhead and energy efficiency of cooperative caching protocols.

The settings of important simulation parameters are presented in Table 6. The mean values of exponentially distributed parameters are noted in the parentheses.

Table 6. Simulation Parameters

| Parameters | Settings |
|---|---|
| Simulation area | 500m*2500m |
| Total #nodes | 50 – 100 |
| Bandwidth | 2Mb/s |
| Radio range | 250m |
| Mobility pattern | random two way point |
| Speed | 1 – 30m/s |
| Pause time | 60 – 300s |
| Cooperation zone radius | 1 – 3 hops |
| Request trace | zipf-distributed ($\alpha = 0.6 - -0.9$) |
| Request interval | exponentially distributed (10 s) |
| Total #data items | 2000 |
| Client cache size | 10 – 50 items |
| TTL | exponentially distributed (5000 sec) |

**5.2.3. Results.** The results show the comparisons of cache miss ratio, request success ratio, average travel distance, and average number of generated messages under different situations: various zipf request pattern, cache size, number of nodes, node velocity, and pausing time. The comparisons are for the following schemes:

- The SimpleCache (SC) scheme [8]. In SimpleCache, each node has a local cache, and user's requests are forwarded to the original data source if local cache misses.
- Hop-by-hop (HBH) caching, which is similar to the schemes proposed in [8]. In this scheme, the requested data item is checked at every forwarding nodes. If a forwarding node has the requested data in its local cache, it stops forwarding and sends back the reply.

- The cocktail approach with 1 hop cooperation range (CT-1).
- The cocktail approach with 2 hop cooperation range (CT-2).
- The cocktail approach with 3 hop cooperation range (CT-3).

Each data point represents the average value of a number of simulations, such that the confidence interval is less than 10% on the 95% confidence level. That is, 95% of the results (assuming normal distribution) are within the range $[10\%M, 110\%M]$ ($M$ is the average value of the measure results).

Figure 30 shows the cache miss ratio (how many requests are forwarded to server for resolution) when the zipf parameter $\alpha$ changes from 0.5 to 0.9. From this figure, we can see that as the value of $\alpha$ increases, the cache miss ratio decreases for all the schemes. The reason is that the increase of $\alpha$ makes the requests more concentrated on a smaller percentage of the data set. This then leads to a higher chance of reusing the cached data items for all the schemes. For different $\alpha$ values we have tried, the cocktail approaches perform better than SimpleCache and Hop-by-Hop approach. The reason is that the cocktail approach not only resolves a request from local cache and forwarding nodes, but also resolves from neighboring caches, thus making more opportunities for cache hit. Comparing SimpleCache and the cocktail approach with 1-hop cooperation range(CT-1), we can see an improvement of 15% to 40%. Comparing Hop-by-Hop and CT-1, we can see an improvement of 5% to 8%. As the cooperation range increases, the cache miss ratio decreases. For example, the improvement ratio from Hop-by-Hop to CT-3 is 19% to 22%.

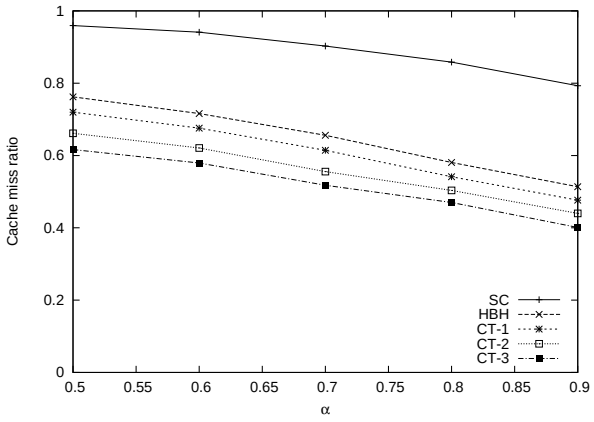


Figure 30. Plot of cache miss ratio with different $\alpha$

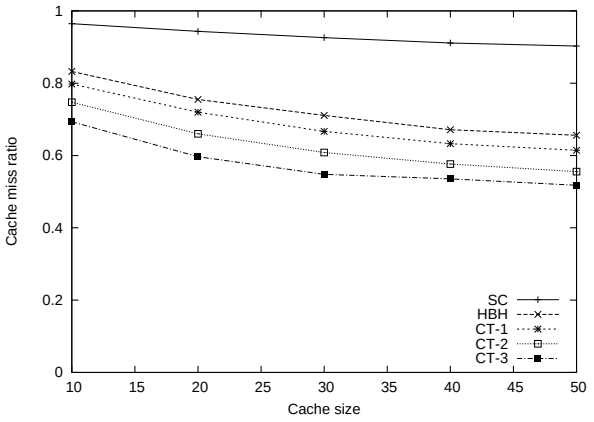


Figure 31. Plot of cache miss ratio with different cache size

Figure 31 shows the cache miss ratio (how many requests are forwarded to server for resolution) when the size of client cache changes from 10 to 50 item capacity. From this figure, we can see that as the client cache size increases, the cache miss ratio decreases for all the schemes. The reason is that the increase of client cache size makes it possible to accommodate more data items in the cache. This then leads to a higher chance of finding the requested data item in the cache for all the schemes. For different client cache sizes we have tried, the cocktail approaches perform better than SimpleCache and Hop-by-Hop approach. Comparing SimpleCache and the cocktail approach with 1-hop cooperation range(CT-1), we can see an improvement of 17% to 32%. Comparing Hop-by-Hop and CT-1, we can see an improvement of 4% to 7%. As the cooperation range increases, the cache miss ratio decreases. For example, the improvement ratio from Hop-by-Hop to CT-3 is 16% to 23%.

Figure 32 shows the average travel distance of a request (average response delay) when the zipf parameter $\alpha$ changes from 0.5 to 0.9. From this figure, we can

see that as the value of $\alpha$ increases, the average travel distance decreases for all the schemes. The reason is that the increase of $\alpha$ makes the requests more concentrated on a smaller percentage of the data set. This then leads to higher reuse of cached data items, and then leads to less travel distance for all the schemes. For different $\alpha$ values we have tried, the cocktail approaches perform better than SimpleCache and Hop-by-Hop approach. The reason is that the cocktail approach has lower cache miss ratio (see Figure 30), which means that the cocktail approach uses more short-distance data transmission (with the cache nodes) to substitute long-distance transmission to/from the data server. Comparing SimpleCache and the cocktail approach with 1-hop cooperation range(CT-1), we can see an improvement of 13% to 16%. Comparing Hop-by-Hop and CT-1, we can see an improvement of 4% to 11%. As the cooperation range increases from 1 hop to 2 hops, the average travel distance decreases 5% to 15%. While when the cooperation range increases from 2 hops to 3 hops, the average travel distance decreases only 1% to 6%.

Figure 33 shows the average travel distance of a request (average response delay) when the size of client cache changes from 10 to 50 item capacity. From this figure, we can see that as the client cache size increases, the average travel distance decreases for all the schemes. The increase of client cache size leads to an decrease of cache miss ratio, which then leads to less travel distance for all the schemes. For different client cache size we have tried, the cocktail approaches perform better than SimpleCache and Hop-by-Hop approach. The reason is that the cocktail approach has lower cache miss ratio (see Figure 31), which means that the cocktail approach

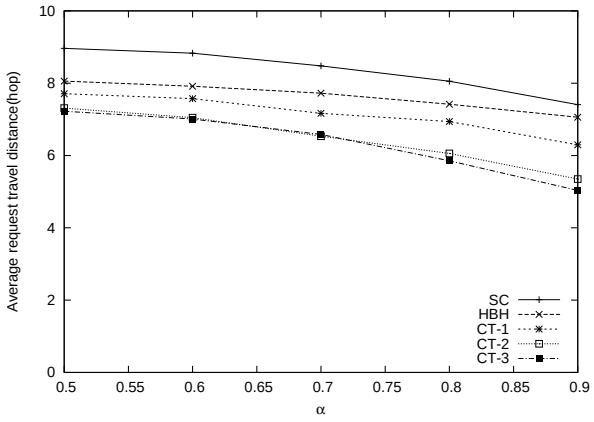


Figure 32. Plot of average request travel distance with different $\alpha$

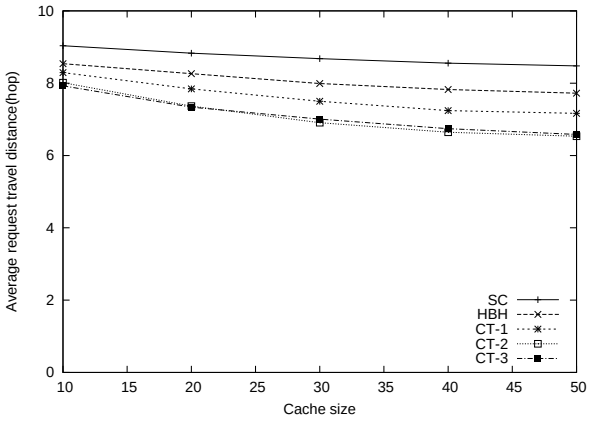


Figure 33. Plot of average request travel distance with different cache size

uses more short-distance data transmission (with the cache nodes) to substitute long-distance transmission to/from the data server. Comparing SimpleCache and the cocktail approach with 1-hop cooperation range(CT-1), we can see an improvement of 8% to 16%. Comparing Hop-by-Hop and CT-1, we can see an improvement of 2% to 8%. As the cooperation range increases from 1 hop to 2 hops, the average travel distance decreases 3% to 9%. While when the cooperation range increases from 2 hops to 3 hops, the average travel distance decreases only 1% to 2%.

Figure 34 shows the average number of messages generated for a successful request (average response delay) when the zipf parameter $\alpha$ changes from 0.5 to 0.9. From this figure, we can see that as the value of $\alpha$ increases, the average number of messages decreases for all the schemes. The reason is that the increase of $\alpha$ makes the requests more concentrated on a smaller percentage of the data set. This then leads to higher reuse of cached data items, and then leads to less number of messages for fetching a data item. For different $\alpha$ values we have tried, the cocktail approach with

3-hop cooperation range generates the most number of messages. This is attributed to the biggest broadcast range it has. The cocktail approach with 2-hop cooperation range generates similar amount of messages with the SimpleCache scheme. And the message amount is in the middle in the compared schemes. The cocktail approach with 1-hop cooperation range generates similar amount of messages with Hop-by-Hop resolution scheme, which has the least average messages per successful request.

Figure 35 shows the average number of messages generated for a successful request (average response delay) when the size of client cache changes from 10 to 50 item capacity. From this figure, we can see that as the client cache size increases, the average number of messages per successful request decreases for all the schemes. The increase of client cache size leads to an decrease of cache miss ratio, which then leads to less number of generated messages for all the schemes. For different client cache size we have tried, the cocktail approach with 3-hop cooperation range generates the most number of messages. This is attributed to the biggest broadcast range it uses for cooperation zone. The cocktail approach with 2-hop cooperation range generates similar amount of messages with the SimpleCache scheme. And the message amount is in the middle in the compared schemes. The cocktail approach with 1-hop cooperation range generates similar amount of messages with Hop-by-Hop resolution scheme, which has the least average messages per successful request.

Figure 36 shows the request success ratio when the number of nodes changes from 50 to 100. As node number increases, the request success ratio increases slightly. The reason for that is there are more backup routes available as node number in-

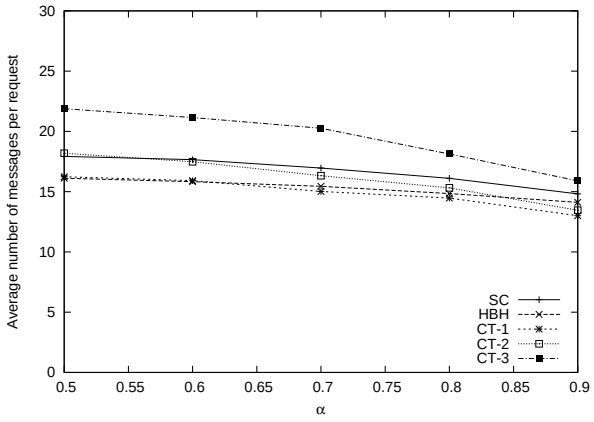


Figure 34. Plot of average number of messages with different $\alpha$

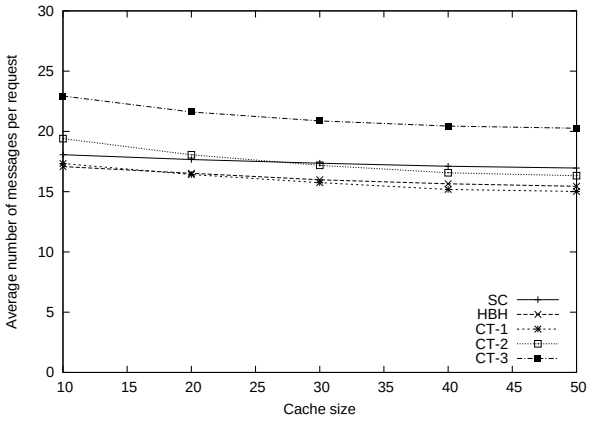


Figure 35. Plot of average number of messages with different cache size

creases. For different number of nodes we have tried, the cocktail approaches outperform other approaches. Since Hop-by-Hop and SimpleCache use relative longer transmission path, any forwarding node that moves out of range will lead to a path breakage and raises the risk of dropping a request. On the other hand, the cocktail approach resolves many requests with broadcast covering only a few hops around the request node, which has less chance to fail. Comparing SimpleCache and the cocktail approach with 2 hop cooperation range, the success ratio is improved by 2% to 25%. Comparing Hop-by-Hop and the cocktail approach with 2 hop cooperation range, the success ratio is improved by 2% to 24%.

Figure 40 shows the request success ratio when the pause time changes from 60 to 300. As pause time increases, the request success ratio increases. The reason for that is network links live longer when the pause time increases. Thus link breakage is less frequent and less requests are dropped. For different pause time we have tried, the cocktail approaches outperform other approaches. The reason is as explained for

Figure 36. Since Hop-by-Hop and SimpleCache use relative longer transmission path, any forwarding node that moves out of range will lead to a path breakage and raises the risk of dropping a request. On the other hand, the cocktail approach resolves many requests with broadcasts covering only a few hops around the request node, which has less chance to fail. Comparing SimpleCache and the cocktail approach with 2 hop cooperation range, the success ratio is improved by 8% to 27%. Comparing Hop-by-Hop and the cocktail approach with 2 hop cooperation range, the success ratio is improved by 8% to 24%.

Figure 44 shows the request success ratio when the node velocity changes from 1 to 30. As node velocity increases, the request success ratio decreases. The reason for that is network links live shorter when the node velocity increases. Thus link breakage is more frequent and more requests are dropped. For different pause time we have tried, the cocktail approaches outperform other approaches. The reason is as explained for Figure 36. Since Hop-by-Hop and SimpleCache use relative longer transmission path, any forwarding node that moves out of range will lead to a path breakage and raises the risk of dropping a request. On the other hand, the cocktail approach resolves many requests with broadcasts covering only a few hops around the request node, which has less chance to fail. Comparing SimpleCache and the cocktail approach with 2 hop cooperation range, the success ratio is improved by 7% to 12%. Comparing Hop-by-Hop and the cocktail approach with 2 hop cooperation range, the success ratio is improved by 5% to 11%.

Figure 37 shows the cache miss ratio (how many requests are forwarded to server for resolution) when the number of nodes changes from 50 to 100. For SimpleCache and Hop-by-Hop, there is no obvious relationship between the number of nodes and cache miss ratio. The reason is that even though the node number increases, the average number of hops to the server is not increasing. Therefore, there are still similar number of caches for request resolution. For the cocktail approach with 1 hop cooperation range, cache miss ratio decreases slightly when the node number increases, since there are more neighboring caches that can be used for resolving requests. For the cocktail approach with 2 hop cooperation range, this relationship is not obvious. The reason is that even with smaller number of nodes such as 50, 2-hop range already covered enough caches; the increase of neighbor nodes cannot have obvious contribution to the reduce cache miss ratio. For all the node numbers we have tried, the cocktail approaches perform better than SimpleCache and Hop-by-Hop. Comparing SimpleCache and the cocktail approach with 2-hop cooperation range(CT-2), we can see an improvement of 44% to 48%. Comparing Hop-by-Hop and CT-2, we can see an improvement of 27% to 35%. As the cooperation range increases, the cache miss ratio decreases. The improvement ratio from CT-1 to CT-2 is 5% to 15%.

Figure 41 shows the cache miss ratio (how many requests are forwarded to server for resolution) when the pause time changes from 60 to 300. There is no obvious relationship between the pause time and cache miss ratio. But for all the pause time we have tried, the cocktail approaches perform better than SimpleCache and Hop-

by-Hop. Comparing SimpleCache and the cocktail approach with 2-hop cooperation range(CT-2), we can see an improvement of 43% to 48%. Comparing Hop-by-Hop and CT-2, we can see an improvement of 31% to 36%. As the cooperation range increases, the cache miss ratio decreases. The improvement ratio from CT-1 to CT-2 is 7% to 12%.

Figure 45 shows the cache miss ratio (how many requests are forwarded to server for resolution) when the node velocity changes from 1 to 30 (m/s). There is no obvious relationship between the node velocity and cache miss ratio. But for all the node velocity we have tried, the cocktail approaches perform better than SimpleCache and Hop-by-Hop. Comparing SimpleCache and the cocktail approach with 2-hop cooperation range(CT-2), we can see an improvement of 43% to 49%. Comparing Hop-by-Hop and CT-2, we can see an improvement of 25% to 36%. As the cooperation range increases, the cache miss ratio decreases. The improvement ratio from CT-1 to CT-2 is 2% to 13%.

Figure 38 shows the average travel distance of a request when the number of nodes changes from 50 to 100. There is no obvious relationship between the number of nodes and the average travel distance. But for all the node numbers we have tried, the cocktail approaches perform better than SimpleCache and Hop-by-Hop. Comparing SimpleCache and the cocktail approach with 2-hop cooperation range(CT-2), we can see an improvement of 50% to 58%. Comparing Hop-by-Hop and CT-2, we can see an improvement of 39% to 48%. As the cooperation range increases, the cache miss ratio decreases. The improvement ratio from CT-1 to CT-2 is 13% to 17%.

Figure 42 shows the average travel distance of a request when the pause time changes from 60 to 300. There is no obvious relationship between the pause time and the average travel distance. But for all the node numbers we have tried, the cocktail approaches perform better than SimpleCache and Hop-by-Hop. Comparing SimpleCache and the cocktail approach with 2-hop cooperation range(CT-2), we can see an improvement of 52% to 57%. Comparing Hop-by-Hop and CT-2, we can see an improvement of 41% to 48%. As the cooperation range increases, the cache miss ratio decreases. The improvement ratio from CT-1 to CT-2 is 10% to 18%.

Figure 46 shows the average travel distance of a request when the node velocity changes from 1 to 30. There is no obvious relationship between the node velocity and the average travel distance. But for all the node numbers we have tried, the cocktail approaches perform better than SimpleCache and Hop-by-Hop. Comparing SimpleCache and the cocktail approach with 2-hop cooperation range(CT-2), we can see an improvement of 54% to 58%. Comparing Hop-by-Hop and CT-2, we can see an improvement of 43% to 47%. As the cooperation range increases, the cache miss ratio decreases. The improvement ratio from CT-1 to CT-2 is 11% to 19%.

Figure 39 shows the average number of messages generated for a request when the number of nodes changes from 50 to 100. Figure 43 shows the average number of messages generated for a request when the pause time changes from 60 to 300. Figure 47 shows the average number of messages generated for a request when the node velocity changes from 1 to 30. As the number of nodes increases, the number of messages generated by CT-2 increases significantly. This is because the broadcast

in the cooperation zone covers more nodes while the node number/density increases. CT-1 generates similar amount of messages with Hop-by-Hop. Since the cooperation zone is only 1 hop and one broadcast message is enough to cover this zone.

From the above simulation results, the conclusion is that the cocktail approach outperforms the other approaches from the perspectives of request success ratio, cache miss ratio, and average request travel distance. That is, the cocktail approach results in higher data availability and less data travel distance. But the price is that the cocktail approach usually generates more messages than other approaches. However, the number of messages generated by the cocktail approach with 1 hop cooperation range is comparable to that of the Hop-by-Hop approach, which generates the least number of messages among all the compared approaches.

### 5.3. Reconciliation of mathematical analysis and simulation results

The simulation results are consistent with the mathematical analysis. For average time cost which is reflected by the average travel distance in the simulations, the analysis shows that the improvement ratio of time cost decreases as the zone radius increases. This is also seen in the simulation results. For example, Figure 32 and Figure 33 show that the improvement ratio can reach 10% by increasing zone radius from 1 to 2, but the improvement ratio is less than 2% by increasing zone radius from 2 to 3. This reflects the saturation points observed in previous analysis (Figure 23).

For average energy cost which is reflected by the average number of messages

in the simulations, the analysis shows that the energy cost increase gets faster when zone radius increases (Figure 24 28 and 26). This trend is also shown in the simulation results (Figure 34 and 35). From the simulation results, the average message number (energy cost) increases around 10% when zone radius increases from 1 to 2, and this increase is much faster, over 20%, when zone radius increases from 2 to 3.

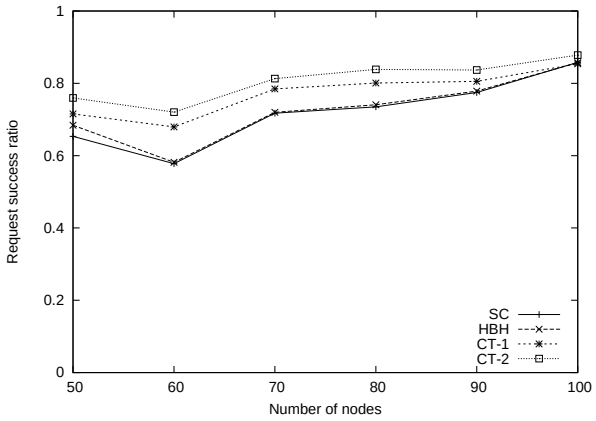


Figure 36. Plot of success ratio with different node number

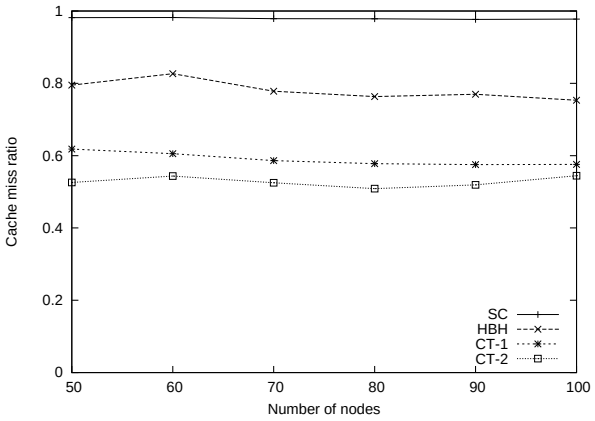


Figure 37. Plot of cache misss ratio with different node number

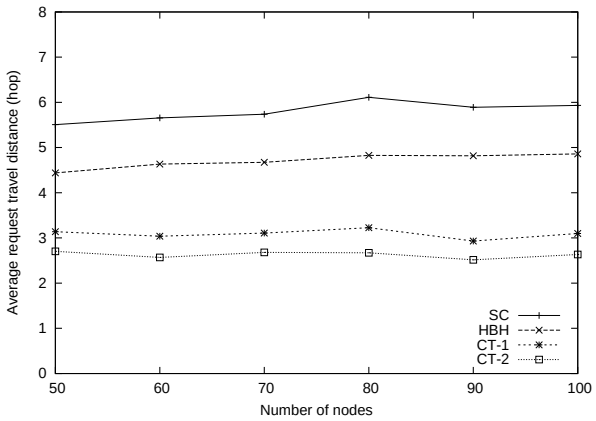


Figure 38. Plot of average request travel distance with different node number

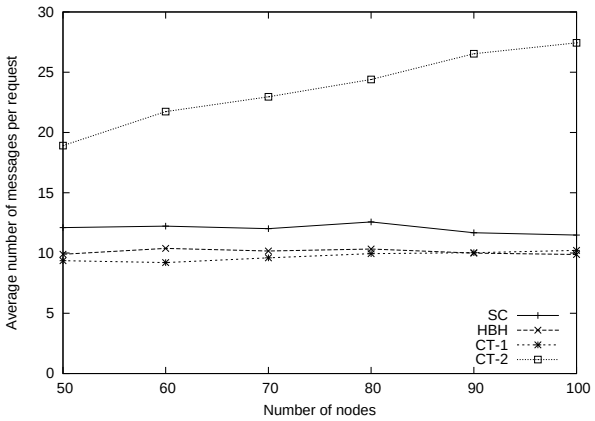


Figure 39. Plot of average number of messages with different node number

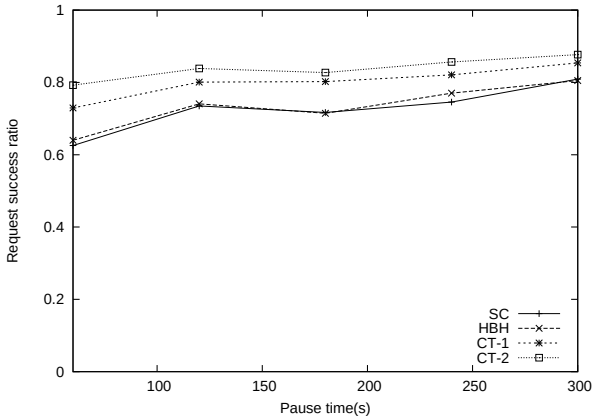


Figure 40. Plot of success ratio with different pause time

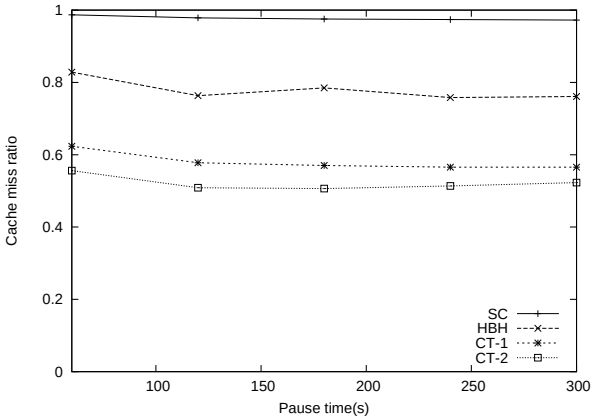


Figure 41. Plot of cache miss ratio with different pause time

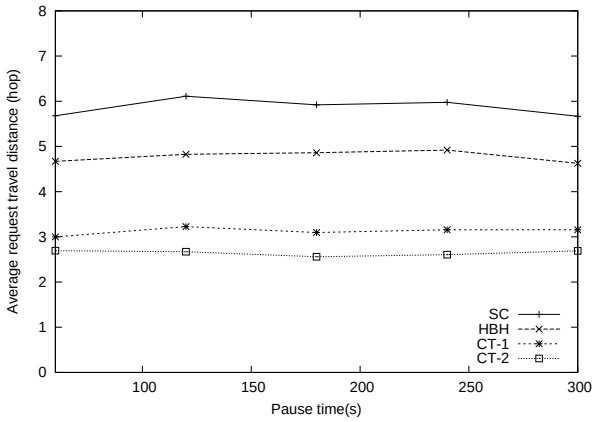


Figure 42. Plot of average request travel distance with different pause time

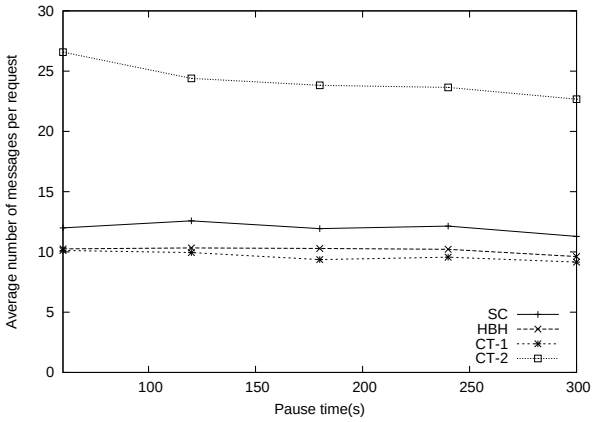


Figure 43. Plot of average number of messages with different pause time

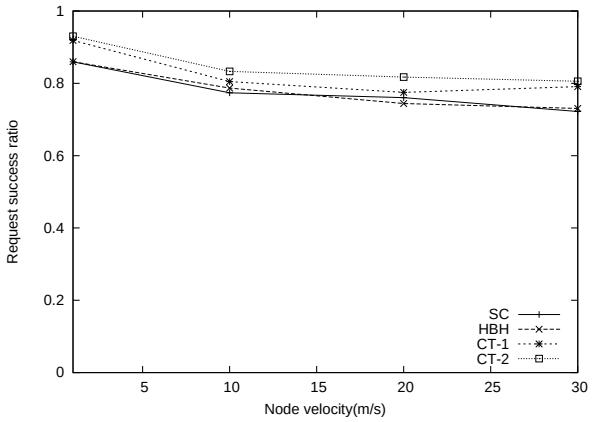


Figure 44. Plot of success ratio with different node velocity

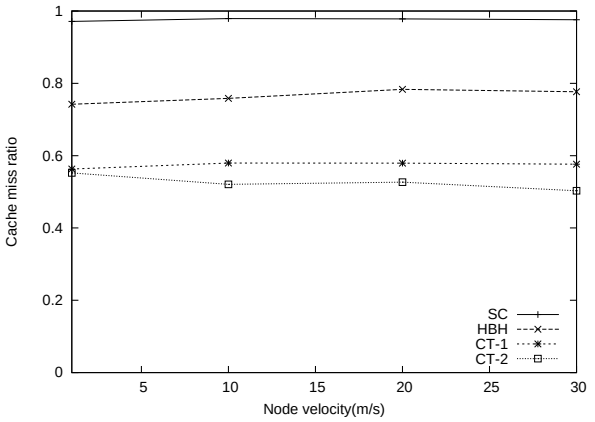


Figure 45. Plot of cache miss ratio with different node velocity

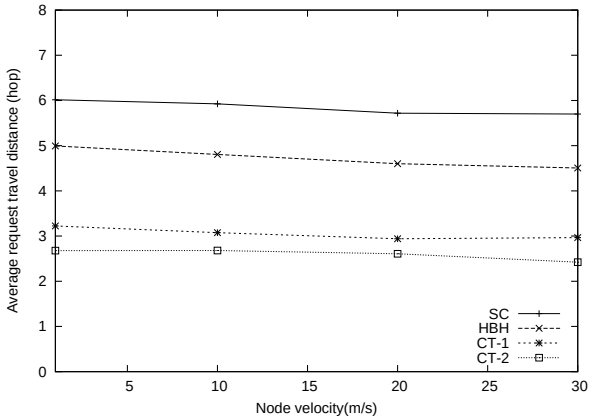


Figure 46. Plot of average request travel distance with different node velocity

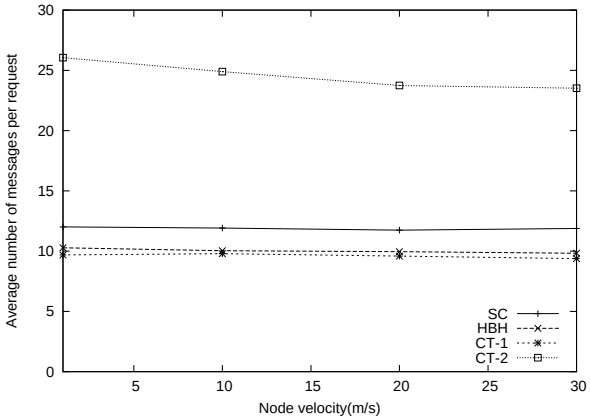


Figure 47. Plot of average number of messages with different node velocity

CHAPTER 6

# Conclusions and Future Works

This thesis has explored how to use cooperative caching to improve data access efficiency in MANETs and reported the obtained results. The proposed cooperative caching scheme, COOP, addresses two basic problems of cooperative caching.

For cache resolution, COOP uses a cocktail approach. This cocktail approach outperforms the other approaches from the perspectives of request success ratio, cache miss ratio, and average request travel distance. For data availability, comparing COOP (with 2-hop cooperation zone) to SimpleCache and Hop-by-Hop, the improvement ratio is up to 27% and 24% respectively. For data travel distance which representing response delay, the improvement of COOP from SimpleCache and Hop-by-Hop is up to 48% and 58%. But the price is that COOP can generate more messages than SimpleCache and Hop-by-Hop when the cooperation zone increases its radius. However, the number of generated messages is similar (within 3%) for Hop-by-Hop and COOP with 1 hop cooperation range, which generate the least number of messages among all the compared approaches. The number of messages generated by COOP with 2 hop cooperation range is largely dependent on the node density.

For cache management, COOP uses the inter-category and intra-category rules to minimize caching duplications between the nodes within a same cooperation zone. This improves the overall effective capacity of cooperated caches. The simulation results show that the cache miss ratio has been greatly reduced by using this cache management scheme. The cache miss ratio of COOP with 2-hop cooperation range is decreased up to 49% compared to the SimpleCache scheme and up to 36% compared to the Hop-by-Hop scheme.

### 6.1. Dynamic adapting cooperation range

The simulation and analysis results provide a guideline for dynamically adapting cooperation zone range. There are tradeoffs in adjusting cooperation zone range. As the zone radius increases, the cache hit ratio and request success ratio increases and response delay decreases, but it introduces higher energy consumption. We also observe a saturation point of cooperation zone radius, after which the cache hit ratio and request success ratio no longer increases and response delay stops decreasing.

In order to choose an optimal zone radius, the cooperative caching scheme needs to evaluate user's current requirements and priorities on energy consumption, data availability, and response delay. For example, if a user specifies that the first priority is data availability and no restrictions on energy consumption, then the zone radius may be enlarged step by step until reaching the specified data availability threshold or the highest availability possible. On the other hand, if the first priority is to save energy consumption as much as possible with a minimum data availability, the cooperation

zone radius can be confined to a minimum value that satisfies the minimum data availability. In practice, the real application scenarios may be more complex, and future work is needed to find out how to efficiently adapt to the application needs.

## 6.2. Cooperation structure

In this dissertation, cooperation zone is used to partition the network and manage the cooperation structure. This is because the assumption is that the node mobility is unknown and unpredictable. Under such assumption, the cooperative caching scheme presented in this dissertation has a good performance, as studied in Chapter 5. The data availability by up to 27%, and the response delay is improved up to 59% compared to the SimpleCache scheme.

Existing cooperative caching structures such as DHT has potential problems under this assumption. DHT maps each data item to a caching node with a distributed hashing table. Since the hashing function does not consider the network topology, the hashed caching node may have a long network distance from the requesting node. This will introduce higher response delay and energy consumption to retrieve the data from the caching node. This may even cause packet loss due to the movements and status changes of the forwarding nodes.

However, DHT has the advantage of reducing searching overhead, as the caching node is more deterministic. By incorporating DHT into the intra-zone cooperation, i.e. using DHT to determine which in-zone node cache what data, it is possible to achieve better performance. Future research needs to study what type of

DHT scheme can be used and how to use it in COOP.

### 6.3. Enforcing fairness in cooperative caching

Cooperative caching on MANETs has the nature of P2P systems. One important problem of P2P systems is that the participants may have different incentives and they cannot be trusted to work toward the common benefit of the whole system. Take the world famous P2P software BitTorrent [56] as an example, some users may only want to download files from others but do not want share their own files. If this situation continues, there will be no user providing shared files in the BitTorrent world. Similarly in cooperative caching, if some caching node refuses to send the request data for other nodes while keeps asking others for the data it needs, eventually no node can obtain desired data from its peer nodes. To prevent this scenario from happening, fairness shall be enforced in the cooperative caching protocol. Currently, COOP enables the same cooperation policy on every node, which requires that a node shall answer a peer's request if it has the requested data in the local cache. This policy ensures the minimum level of fairness that no node can deny service to others. But it cannot ensure the fairness if some user can change the implementation of COOP in a malicious way, as the user may drop other nodes' requests in his own implementation. Thus, another future work for MANET cooperative caching is to enforce fairness at a higher level.

# REFERENCES


[1] D. Tacconi and C. Saraydar and Ş. Tekinay, "Ad hoc enhanced routing in umts for increased packet delivery rates," in *IEEE WCNC*, 2004.

[2] H. Wu and C. Qiao and S. De, "Integrated cellular and ad hoc relaying systems: icar," *IEEE JSAC*, vol. 19, no. 10, pp. 450–455, 2001.

[3] Y. Sun and E. M. Belding-Royer and C. E. Perkins, "Internet connectivity for ad hoc mobile networks," *International Journal of Wireless Information Networks*, vol. 9, no. 2, 2002.

[4] J. E. Wieselthier, G. D. Nguyen, and A. Ephremides, "Resource-limited energy-efficient wireless multicast of session traffic," in *34th Annual Hawaii Int'l Conf. System Sciences*, Maui, Hawaii, 2001.

[5] J. Li and C. Blake and D. Couto and H. Lee and R. Morris, "Capacity of ad hoc wireless networks," in *Mobile Computing and Networking*, 2001.

[6] J. F. Kurose and K. W. Ross, *Computer networking: a top-down approach featuring the Internet*, 1st ed. Addison-Wesley, 2000.

[7] G. Cao and L. Yin, "Cooperative cache based data access in ad hoc networks," *IEEE Computer*, vol. 37, no. 2, pp. 32–39, 2004.

[8] L. Yin and G. Cao, "Supporting cooperative caching in ad hoc networks," in *INFOCOM*, 2004.

[9] A. Chankhunthod and P. B. Danzig and C. Neerdaels and M. F. Schwartz and K. J. Worrell, "A hierarchical internet object cache," in *USENIX Annual Technical Conference*, 1996.

[10] L. Fan and P. Cao and J. Almeida and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[11] S. Iyer and A. Rowstron and P. Druschel, "Squirrel: A decentralized peer-to-peer web cache," in *PODC*, 2002.

[12] P. Denning, "The locality principle," *Communications of the ACM*, vol. 48, no. 7, pp. 19–24, 2005.

[13] R. Fonseca and V. Almeida and M. Crovella, "Locality in a web of streams," *Communications of the ACM*, vol. 48, no. 1, pp. 82–88, 2005.

[14] G. Zipf, *Human behavior and the principle of least effort.* Addison-Wesley, 1949.

[15] L. Breslau and P. Cao and L. Fan and G. Phillips and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *INFOCOM*, 1999.

[16] Y. Du and S. Gupta, *Handbook of Mobile Computing.* CRC Press, 2004, ch. 15, pp. 337–360.

[17] T. Hara, "Effective replica allocation in ad hoc networks for improving data accessibility," in *INFOCOM*, 2001.

[18] ——, "Replica allocation in ad hoc networks with periodic data update," in *the 3rd International Conference on Mobile Data Management*, 2002.

[19] P. Nuggehalli and V. Srinivasan and C. Chiasserini, "Energy-efficient caching strategies in ad hoc wireless networks," in *MobiHoc*, 2003.

[20] N. Chang and M. Liu, "Revisiting the ttl-based controlled flooding search: optimality and randomization," in *MobiCom*, 2004.

[21] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker, "Ght: A geographic hash table for data-centric storage in sensornets," in *the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002.

[22] B. Karp and H. T. Kung, "GPSR: greedy perimeter stateless routing for wireless networks," in *Mobile Computing and Networking*, 2000.

[23] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer file sharing technologies," white paper, Electornic Trading Research Unit (ELTRUN), Athens University fo Economics and Business, 2002.

[24] P. T. Eugster, R. Guerraoui, A. Kermarrec, and L. Massoulieacute, "Epidemic information dissemination in distributed systems," *IEEE Computer*, vol. 37, no. 5, pp. 60–67, 2004.

[25] P. B. Mirchandani and R. L. Francis Ed, "Discrete location theory."

[26] M. L. Balinksi and P. Wolfe, "On benders decomposition and a plant location problem," working paper ARO-27. Mathematica Inc. 1963.

[27] A. A. Kuehn and M. J. Hamburger, "Discrete location theory," management Science 9, 1963.

[28] S. L. Hakimi, "Optimum locations of switching centers and the absolute centers and medians of a graph," operations Research 12, 1964.

[29] F. Hwang, D. Richards, and P. Winter, "Applications of mathematical methods to wheat harvesting," pp. 77–91, chinese Mathematics 2, 1962.

[30] V. Arya and N. Garg and R. Khandekar and A. Meyerson and K. Munagala and V. Pandit, "Local search heuristics for k-median and facility location problems," in *ACM Symposium on Theory of Computing*, 2001.

[31] M. Charikar and S. Guha and E. Tardos and D. B. Shmoys, "A constant-factor approximation algorithm for the k-median problem," in *the 31st Annual ACM Symposium on Theory of Computing*, 1999.

[32] M. Charikar and S. Guha, "Improved combinatorial algorithms for the facility location and $k$-median problems," in *the 40th Annual Symposium on Foundations of Computer Science*, 1999.

[33] K. Jain and V. Vazirani, "Primal-dual approximation algorithms for metric facility location and k-median problems," manuscript, 1999.

[34] J. H. Lin and J. S. Vitter, "E-approximation with minimum packing constraint violation," in *24th ACM Symp. on Theory of Computing*, 1992.

[35] P. Krishnan, D. Raz, and Y. Shavitt, "The cache location problem," *IEEE/ACM Transactions on Networking*, vol. 8, no. 5, pp. 568–582, 2000.

[36] B. Li and M. J. Golin and G. F. Italiano and X. Deng, "On the optimal placement of web proxies in the internet," in *INFOCOM*, 1999.

[37] F. Hwang, D. S. Richards, and P. Winter, *The Steiner tree problem.* North-Holland, 1992.

[38] D. Barbara and T. Imielinski, "Sleepers and workaholics: Caching strategies for mobile environments," in *ACM SIGMOD Conference on Management of Data*, 1994.

[39] ——, "Sleepers and workaholics: Caching strategies for mobile environments (extended version)," *The VLDB Journal - the International Journal on Very Large Data Bases*, vol. 4, no. 4, pp. 567–602, 1995.

[40] P. Cao and C. Liu, "Maintaining strong cache consistency in the world wide web," *IEEE Transcations On Computers*, vol. 47, no. 4, pp. 445–457, 1998.

[41] L. Y. Cao and M. T. Özsu, "Evaluation of stroing consistency web caching techniques," *World Wide Web*, vol. 5, no. 2, pp. 95–123, 2002.

[42] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *the 12th ACM Symposium on Operating Systems Principles*, 1989.

[43] J. Yin and L. Alvisi and M. Dahlin and C. Lin, "Using leases to support server-driven consistency in large-scale systems," in *the 18th IEEE International Conference on Distributed Computing Systems*, 1998.

[44] ——, "Volume leases for consistency in large-scale systems," *Knowledge and Data Engineering*, vol. 11, no. 4, pp. 563–576, 1999.

[45] V. Cate, "Alex – a global file system," in *USENIX File System Workshop*, 1992.

[46] M. Baker and J. H. Hartman and M. D. Kupfer and K. W. Shirriff and J. Ousterhout, "Measurements of a distributed file system," in *ACM Symposium on Operating Systems Principles*, 1991.

[47] J. Gwertzman and M. Seltzer, "World-wide web cache consistency," in *USENIX Technical Conference*, 1996.

[48] M. Rabinovich and O. Spatscheck, *Web Caching and Replication*. Addison-Wesley, 2002.

[49] R. Fielding, J. Gettys, and J. Mogul, "Rfc 2616: Hypertext transfer protocol – http/1.1," 1999. [Online]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616.html

[50] W. Adjie-Winoto and E. Schwartz and H. Balakrishnan, "The design and implementation of an intentional naming system," in *Symposium on Operating Systems Principles*, 1999.

[51] R. Hekmat and P. Van Mieghem, "Interference in wireless multi-hop ad-hoc networks," in *Med-hoc-net 2002 Conference*, 2002.

[52] L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[53] S. Ni and Y. Tseng and Y. Chen and J. Sheu, "The broadcast storm problem in a mobile ad hoc network," in *MobiCOM*, 1999.

[54] K. Fall and K. Varadhan Ed, "The ns manual." [Online]. Available: http://www.isi.edu/nsnam/ns/doc/index.html

[55] J. Broch and D. Maltz and D. Johnson and Y. Hu and J. Jetcheva, "A performance comparison of multi-hop wireless ad hoc network routing protocols," in *MobiCom*, 1998.

[56] B. Cohen, "Incentives build robustness in bittorrent," in *the 1st Workshop on the Economics of Peer-to-Peer Systems*, 2003.

# Appendix A. Simulation code for COOP

This appendix contains the main part of COOP's simulation code written for the simulator NS-2. Two files are included in this appendix: coop.h and coop.cc.

```
/*
 * File name: coop.h
 * Copyright (c) 2004-2005 Arizona State University.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this program is
 * allowed as long as the following conditions are met:
 * 1. The copyright notice and the permission notice is retained in
 * all copies of the program, derivative works, or modified version.
 * 2. Credit is given to the author and Arizona State University in
 * all publications reporting on direct or indirect use of this
 * program or its derivatives.
 * 3. Neither the name of the University nor of the author may be used
 * to endorse or promote products derived from this program without
 * specific prior written permission.
 *
 * THIS PROGRAM IS PROVIDED IN ITS ``AS IS'' CONDITION, AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL ARIZONA STATE
 * UNIVERSITY OR THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
 *
 * The code for COOP is developed by Yu Du during her Phd study in
 * Arizona State University.
 */


#ifndef ns_coop_h
#define ns_coop_h

#include "packet.h"

// Max number of cached data items:
//#define CACHE_SIZE    5
// Size of the RRT table:
#define RRT_SIZE    50
//double
#define CURRENT_TIME    Scheduler::instance().clock()
#define COOP_PORT       222

// how much time it takes to broadcast one-hop, based on observation
#define ONE_HOP     0.005


// Zipf generator parameters
// #define  RAND_SEED   1   // An integer greater than 0
#define ALPHA       0.8 // zipf parameter
//#define   DATA_NUM    2000    // total number of data items

// Mean time interval (exponentially distributed) between requests
//#define MEAN_INT  10
//#define SERVERID  0


struct COOP_cache_entry
{
    u_int32_t       dataID;
    double  timestamp;
    double  last_read;
    int  ttl;
    double  weight;
    u_int8_t        importance; // primary or secondary : 1 or 2
```

```
};//COOP_cache_entry

class COOP_cache
{
    public:
        COOP_cache();
        COOP_cache(int maxsize);
        ~COOP_cache();

        int add(COOP_cache_entry newEntry);
        int addAll(u_int32_t dataID[], int size);
        COOP_cache_entry get(int index);
        int     indexOf(u_int32_t dataID);
        int set(int index, COOP_cache_entry newEntry);

        int getTop();
        int setTop(int inTop);
        int getMax_size();
        int setMax_size(int ms);

        void    cache_init(int ms);

    protected:
        int max_size;
        COOP_cache_entry *data;
        int top;

}; //COOP_cache

struct RRT_entry
{
    u_int32_t dataID;
    nsaddr_t requester;
    double timestamp;

}; // RRT_entry

class RRT
{
    public:
        RRT();
        int getCurrent();
```

```
        int setCurrent(int c);
        int indexOf(RRT_entry e);
        RRT_entry get(int index);
        int set(int index, RRT_entry e);
        int add(RRT_entry e);
        int lookup(u_int32_t dataID);

    protected:
        RRT_entry requests[RRT_SIZE];
        int current;

}; // class RRT

struct hdr_coop {
    // 1 - request, 2 - reply, 3 - request route info
    u_int8_t    type;
    u_int32_t   dataID;
    nsaddr_t    serverAddr; //int32_t
    char        msg_[64];

    char        seq[32];
    char        hFlag;

    static int offset_;
    inline static int& offset() { return offset_; }
    inline static hdr_coop* access(const Packet* p) {
        return (hdr_coop*) p->access(offset_);
    }

    /* per-field member functions */
    char* msg() { return (msg_); }
    int maxmsg() { return (sizeof(msg_)); }
};

#endif

/*
 * File name: coop.cc
 * Copyright (c) 2004-2005 Arizona State University.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this program is
 * allowed as long as the following conditions are met:
```

```
 * 1. The copyright notice and the permission notice is retained in
 * all copies of the program, derivative works, or modified version.
 * 2. Credit is given to the author and Arizona State University in
 * all publications reporting on direct or indirect use of this
 * program or its derivatives.
 * 3. Neither the name of the University nor of the author may be used
 * to endorse or promote products derived from this program without
 * specific prior written permission.
 *
 * THIS PROGRAM IS PROVIDED IN ITS ``AS IS'' CONDITION, AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL ARIZONA STATE
 * UNIVERSITY OR THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * The code for COOP is developed by Yu Du during her Phd study in
 * Arizona State University.
 */

#include "agent.h"
#include "random.h"
#include "coop.h"
#include "ip.h"
#include <aodv/aodv_rtable.h>

int COOP_seq = 0; // COOP message sequence number
int COOP_time = 2500; // Max COOP simulation time
int COOP_RANGE = 2;
int DATA_NUM = 2000;
int MEAN_INT = 10;
int SERVERID = 0;
double ALPHA = 0.7;

int total_req = 0;
int local_hit = 0;
```

```
int zone_hit = 0;
int path_hit = 0;
int server_hit = 0;
int total_sent = 0;

int COOP_wu_time = 500;

int start_a_log = 0;
int a_total_req = 0;
int a_local_hit = 0;
int a_zone_hit = 0;
int a_path_hit = 0;
int a_server_hit = 0;
int a_total_sent = 0;
FILE* COOP_log;

int hdr_coop::offset_;

static class COOPHeaderClass : public PacketHeaderClass {
public:
    COOPHeaderClass() : PacketHeaderClass("PacketHeader/COOP",\
                        sizeof(hdr_coop))
    {
        bind_offset(&hdr_coop::offset_);
    }
} class_coophdr;

//======================== Zipf related ============================

double rand_val(int seed)
{
    const long  a =      16807;  // Multiplier
    const long  m = 2147483647;  // Modulus
    const long  q =     127773;  // m div a
    const long  r =       2836;  // m mod a
    static long x;               // Random int value
    long        x_div_q;         // x divided by q
    long        x_mod_q;         // x modulo q
    long        x_new;           // New x value

    // Set the seed if argument is non-zero and then return zero
    if (seed > 0)
```

```
    {
        x = seed;
        return(0.0);
    }

    // RNG using integer arithmetic
    x_div_q = x / q;
    x_mod_q = x % q;
    x_new = (a * x_mod_q) - (r * x_div_q);
    if (x_new > 0)
        x = x_new;
    else
        x = x_new + m;

    // Return a random value between 0.0 and 1.0
    return((double) x / m);
}

int zipf(double alpha, int n)
{
    static int first = TRUE;      // Static first time flag
    static double c = 0;          // Normalization constant
    // Uniform random number (0 < z < 1)
    double z;
    double sum_prob;              // Sum of probabilities
    // Computed exponential value to be returned
    int zipf_value;
    int    i;                     // Loop counter

    // Compute normalization constant on first call only
    if (first == TRUE)
    {
        for (i=1; i<=n; i++)
            c = c + (1.0 / pow((double) i, alpha));
        c = 1.0 / c;
        first = FALSE;
    }

    // Pull a uniform random number (0 < z < 1)
    do
    {
        z = rand_val(0);
```

```
    }
    while ((z == 0) || (z == 1));

    // Map z to the value
    sum_prob = 0;
    for (i=1; i<=n; i++)
    {
        sum_prob = sum_prob + c / pow((double) i, alpha);
        if (sum_prob >= z)
        {
            zipf_value = i;
            break;
        }
    }

    // Assert that zipf_value is between 1 and N
    assert((zipf_value >=1) && (zipf_value <= n));

    return(zipf_value);
}

//=========================COOP_cache related=========================

COOP_cache::COOP_cache()
{
    top = 0;
    data = NULL;

}//COOP_cache::COOP_cache()


COOP_cache::COOP_cache(int ms)
{
    top = 0;
    max_size = ms;
    data = new COOP_cache_entry[max_size];

}//COOP_cache::COOP_cache()


COOP_cache::~COOP_cache()
{
```

```
    delete [] data;
}

int COOP_cache::addAll(u_int32_t dataID[], int size)
{
    struct COOP_cache_entry newentry;
    int status = 0;

    for (int i = 0; i < size; i++)
    {
        newentry.dataID = dataID[i];
        if (status = add(newentry))
            break;
    }
    return status;

}//int COOP_cache::addAll(u_int32_t dataID[], int size)

/*  If out of space, return 1;
    if new, add, top ++, return 0;
    If existing, update, keep top, return 0;
*/

int COOP_cache::add(COOP_cache_entry newentry)
{
    struct COOP_cache_entry *entry;
    int index;

    // existing data -> replace the old copy
    if ((index = indexOf(newentry.dataID)) >= 0)
    {
        data[index].timestamp = newentry.timestamp;
        data[index].ttl = newentry.ttl;
        data[index].last_read = CURRENT_TIME;
        return 0;
    }

    // new data, see if still has spare space
    if (top < max_size)
    {

        data[top].dataID = newentry.dataID;
```

```
    data[top].timestamp = newentry.timestamp;
    data[top].ttl = newentry.ttl;
    data[top].weight = newentry.weight;
    data[top].importance = newentry.importance;
    data[top].last_read = CURRENT_TIME;
    top ++;
    return 0;
}

// No spare space, see if any data has expired?
index = 0;
while (index < top && (data[index].timestamp \
+ data[index].ttl > CURRENT_TIME))
    index ++;

if (index < top)
{
    data[index].dataID = newentry.dataID;
    data[index].timestamp = newentry.timestamp;
    data[index].ttl = newentry.ttl;
    data[index].weight = newentry.weight;
    data[index].importance = newentry.importance;
    data[index].last_read = CURRENT_TIME;
    return 0;
}

// No expired data, replace the ancientest secondary data

double a_time = CURRENT_TIME;
int a_index = -1;

for (index = 0; index < top; index++)
{
    if (data[index].importance == 2 \
    && data[index].last_read < a_time)
    {
        a_time = data[index].last_read;
        a_index = index;
    }
}

if (a_index >= 0)
```

```
{
    data[a_index].dataID = newentry.dataID;
    data[a_index].timestamp = newentry.timestamp;
    data[a_index].ttl = newentry.ttl;
    data[a_index].weight = newentry.weight;
    data[a_index].importance = newentry.importance;
    data[a_index].last_read = CURRENT_TIME;
    return 0;
}

// No secondary data found, see if the new data is primary

if (newentry.importance == 2)
    return 0; // do nothing

// The new data is primary, found the ancientest primary

a_time = CURRENT_TIME;
a_index = -1;

for (index = 0; index < top; index++)
{
    if (data[index].last_read < a_time)
    {
        a_time = data[index].last_read;
        a_index = index;
    }
}

if (a_index >= 0)
{
    data[a_index].dataID = newentry.dataID;
    data[a_index].timestamp = newentry.timestamp;
    data[a_index].ttl = newentry.ttl;
    data[a_index].weight = newentry.weight;
    data[a_index].importance = newentry.importance;
    data[a_index].last_read = CURRENT_TIME;
    return 0;
}

return 1;
```

```
}//int add(COOP_cache_entry newentry)

COOP_cache_entry COOP_cache::get(int index)
{
    COOP_cache_entry result;

    if (index >= top)
        return result;


    result.dataID = data[index].dataID;
    result.timestamp = data[index].timestamp;
    result.ttl = data[index].ttl;
    result.weight = data[index].weight;
    result.importance = data[index].importance;
    result.last_read = data[index].last_read;

    return result;
}//COOP_cache_entry COOP_cache::get(int index)

int COOP_cache::indexOf(u_int32_t dataID)
{
    int index = top-1;

    while (index >= 0 && data[index].dataID != dataID)
    {
        index--;
    }

    return index;
}//int COOP_cache::indexOf(u_int32_t dataID)

int COOP_cache::set(int index, COOP_cache_entry newEntry)
{
    if (index >= top)
        return 1;
    else
    {
        data[index].dataID = newEntry.dataID;
        data[index].timestamp = newEntry.timestamp;
        data[index].ttl = newEntry.ttl;
        data[index].last_read = newEntry.last_read;
```

```
        return 0;
    }
}//int COOP_cache::set(int index, COOP_cache_entry newEntry)

int COOP_cache::getTop()
{
    return top;
}

int COOP_cache::setTop(int inTop)
{
    if (inTop < 0 || inTop >= max_size)
        return 1;

    top = inTop;
    return 0;
}

int COOP_cache::getMax_size()
{
    return max_size;
}

int COOP_cache::setMax_size(int ms)
{
    max_size = ms;
    return 0;
}

void COOP_cache::cache_init(int ms)
{
    max_size = ms;
    data = new COOP_cache_entry[ms];
}

//============================RRT related==========================

RRT::RRT()
{
    current = 0;
}
```

```
int RRT::getCurrent()
{
    return current;
}

int RRT::setCurrent(int c)
{
    current = c;
    return 0;
}

// return the entry which has same dataID and requester.

int RRT::indexOf(RRT_entry e)
{
    // if current < RRT_SIZE, search 0 .. current-1
    // if current >= RRT_SIZE, search 0 .. RRT_SIZE -1

    int i;
    i = (current < RRT_SIZE ? current-1 : RRT_SIZE-1);

    while ((requests[i].dataID != e.dataID \
    || requests[i].requester != e.requester) && i >= 0)
        i--;

    return i;

}

RRT_entry RRT::get(int index)
{

    if (index < RRT_SIZE && index >= 0)
    {
        return requests[index];
    }

    RRT_entry e;
    return e;
}
```

```
int RRT::set(int index, RRT_entry e)
{
    if (index < RRT_SIZE && index > 0)
    {
        requests[index] = e;
        return 0;
    }
    return 1;
}

int RRT::add(RRT_entry e)
{
    requests[current % RRT_SIZE] = e;
    current++;
    return 0;
}

int RRT::lookup(u_int32_t in_dataID)
{
// search data from (current-1)%RRT_SIZE, the most recent record:

    int i, size;
    size = (current < RRT_SIZE ? current : RRT_SIZE);

    i = 1;
    while (requests[(current - i) % RRT_SIZE].dataID != in_dataID\
    && i <= size)
        i++;

    if (i > size)
        return (-1);
    else
        return ((current - i) % RRT_SIZE);

}
//============================COOPAgent related=====================

class COOPAgent;

class COOPRequestTimer : public TimerHandler {
public:
    COOPRequestTimer(COOPAgent* a) : TimerHandler() {agent = a;}
```

```
protected:
    virtual void expire(Event *e);
    COOPAgent    *agent;
};

class COOPBcTimer : public TimerHandler {
public:
    COOPBcTimer(COOPAgent* a) : TimerHandler() {agent = a;}
protected:
    virtual void expire(Event *e);
    COOPAgent    *agent;
};

class COOPAgent : public Agent {

    friend class COOPRequestTimer;
    friend class COOPBcTimer;

    public:
        COOPAgent();
        int command(int argc, const char*const* argv);
        void recv(Packet*, Handler*);
        int broadcast(u_int32_t dataID, int ttl);
        int coopreq(nsaddr_t serverAddr, u_int32_t dataID);
        void timeout();
        void btimeout();

    protected:
        COOP_cache  cache;
        RRT     rrt;
        aodv_rtable *rtp;
        // data ID of the pending request,
        // -1 if no request pending;
        int     pendingID;
        char        pendingSeq[32];

        COOPRequestTimer    rtimer;
        COOPBcTimer     btimer;
};

static class COOPClass : public TclClass {
public:
```

```
    COOPClass() : TclClass("Agent/COOP") {}
    TclObject* create(int, const char*const*) {
        return (new COOPAgent());
    }
} class_coop;

void COOPRequestTimer::expire(Event*)
{
        agent->timeout();
}

void COOPBcTimer::expire(Event*)
{
    agent->btimeout();
}

COOPAgent::COOPAgent() : Agent(PT_COOP), rtimer(this), btimer(this)
{
    bind("packetSize_", &size_);
    pendingID = -1;
}

void COOPAgent::recv(Packet* pkt, Handler*)
{
    hdr_coop* mh = hdr_coop::access(pkt);
    struct hdr_cmn* ch = HDR_CMN(pkt);
    struct hdr_ip* ih = HDR_IP(pkt);


    printf("%f r %d %d %s %d %d %d \n", CURRENT_TIME, addr(), \
    mh->type, mh->seq, mh->dataID, ih->saddr(), ih->daddr());

    mh->hops++;

    if (ih->daddr() == addr())
    {
        // I am the server getting a request
        if (mh->type == 1)
        {

            int index=cache.indexOf(mh->dataID);
            if (index == -1)
```

```
{
    // I am the server
    if (mh->serverAddr == addr())
    {
        printf("No entry found in the\
         server! \n");
        Packet::free(pkt);
        return;
    }
    else // I am another source
    {
        printf("No entry found in the\
        source, forwarding the request\
         to the server");
        ih->daddr() = mh->serverAddr;
        ch->direction() = hdr_cmn::DOWN;
        total_sent ++;
        if (start_a_log)
            a_total_sent ++;

        send(pkt, 0);


        printf("%f s %d %d %s %d %d %d\
         \n", CURRENT_TIME, addr(), \
         mh->type, mh->seq,\
         mh->dataID, ih->saddr(),\
         ih->daddr());

        return;
    }
}

server_hit ++;
if (start_a_log)
{
    a_server_hit++;
    fprintf(COOP_log, "%s \t %d \t %d\n", \
    mh->seq, mh->hops, addr());
}
Packet* rpkt = allocpkt();
hdr_coop* rmh = hdr_coop::access(rpkt);
```

```
    hdr_ip* rih = hdr_ip::access(rpkt);

    rmh->type = 2; //reply
    rmh->dataID = mh->dataID;
    strcpy(rmh->seq, mh->seq);

    COOP_cache_entry tempEntry= cache.get(index);

    //timestamp and ttl
    sprintf(rmh->msg(), "%lf %d", \
    tempEntry.timestamp, tempEntry.ttl);

    rih->saddr() = addr();
    rih->sport() = COOP_PORT;
    rih->daddr() = ih->saddr();
    rih->dport() = COOP_PORT;

    printf("%f : %d sends reply to %d at port \
    %d\n", CURRENT_TIME, addr(), rih->daddr(),\
    rih->dport());
    Packet::free(pkt);

    total_sent++;
    if (start_a_log)
        a_total_sent++;

    send(rpkt, 0);
    printf("%f s %d %d %s %d %d %d \n", \
    CURRENT_TIME, addr(), rmh->type, rmh->seq,\
    rmh->dataID, rih->saddr(), rih->daddr());

    return;
}
else if (mh->type == 2) // I am getting my reply
{
    COOP_cache_entry tempEntry;

    tempEntry.dataID = mh->dataID;
    sscanf(mh->msg(), "%lf %d", \
    &(tempEntry.timestamp), &(tempEntry.ttl));
    aodv_rt_entry *rtep = \
    rtp->rt_lookup(ih->saddr());
```

```
            if (rtep == 0)
            {
                tempEntry.importance = 1;

            }
            else
            {
                if (rtep->get_rt_hops() > COOP_RANGE)
                    tempEntry.importance = 1;
                else
                    tempEntry.importance = 2;

            }

            cache.add(tempEntry);

            if (pendingID == mh->dataID)
            {
                pendingID = -1;
                strcpy(pendingSeq, "null");
            }
            Packet::free(pkt);
            return;
        }
        // I am getting the route information.
        else if (mh->type == 3)
        {
            sscanf(mh->msg(), "%p", &rtp);
            Packet::free(pkt);
            return;
        }

    } // if (ih->daddr() == addr())
    // I am receiving a broadcasted message,
    // note only request is broadcasted in the scheme...
    else if (ih->daddr() == IP_BROADCAST)
    {

        RRT_entry rrt_entry;
        rrt_entry.dataID = mh->dataID;
        rrt_entry.requester = ih->saddr();
```

```
rrt_entry.timestamp = ch->timestamp();

int rrt_index = rrt.indexOf(rrt_entry);
// duplicated broadcast, if I've heard it
// or I'm the broadcaster
if (rrt_index != -1 || ih->saddr() == addr())
{
    printf("duplicated broadcast! \n");
    Packet::free(pkt);
    return;
}
else // non-duplicated broadcast
{
    rrt.add(rrt_entry);

    int index = cache.indexOf(mh->dataID);
    if (index != -1)
    {
        COOP_cache_entry tempEntry;
        Packet* rpkt = allocpkt();
        hdr_coop* rmh = hdr_coop::access(rpkt);
        hdr_ip* rih = hdr_ip::access(rpkt);

        tempEntry = cache.get(index);
        rmh->dataID = tempEntry.dataID;
        rmh->type = 2;
        strcpy(rmh->seq, mh->seq);
        sprintf(rmh->msg(), "%lf %d", \
        tempEntry.timestamp, tempEntry.ttl);

        rih->saddr() = addr();
        rih->sport() = COOP_PORT;
        rih->daddr() = ih->saddr();
        rih->dport() = COOP_PORT;


        zone_hit ++;
        if (start_a_log)
        {
            a_zone_hit++;
            fprintf(COOP_log, "%s \t %d \t\
             %d\n", mh->seq, mh->hops, \
```

```
            addr());
        }

        Packet::free(pkt);

        total_sent ++;
        if (start_a_log)
            a_total_sent ++;

        send(rpkt, 0);
        printf("%f s %d %d %s %d %d %d \n", \
        CURRENT_TIME, addr(), rmh->type, \
        rmh->seq, rmh->dataID, rih->saddr(),\
        rih->daddr());

        return;

    }
    else // no data, forward request
    {

        rrt.add(rrt_entry);
        //for some unknown reason, ih->ttl_
        //is 1 when the message reaches the
        //expected ending point
        if (ih->ttl_ > 1)
        {
            ch->direction() = hdr_cmn::DOWN;
            total_sent ++;
            if (start_a_log)
                a_total_sent ++;

            send(pkt, 0);
            printf("%f s %d %d %s %d %d %d\
             \n", CURRENT_TIME, addr(), \
             mh->type, mh->seq, mh->dataID,\
              ih->saddr(), ih->daddr());

        }
        else
        {
            Packet::free(pkt);
```

```
            }
            return;
        }
    }// if non-duplicated broadcast

    return;
}//else if (ih->daddr() == IP_BROADCAST)

else // I am forwarding a unicasted message
{
    if (mh->type == 1) // I am forwarding a request
    {
        int index = cache.indexOf(mh->dataID);
        if (index == -1)
        {
            ch->direction() = hdr_cmn::DOWN;
            total_sent ++;
            if (start_a_log)
                a_total_sent++;

            send(pkt, 0);
            printf("%f s %d %d %s %d %d %d \n", \
            CURRENT_TIME, addr(), mh->type, \
            mh->seq, mh->dataID, ih->saddr(), \
            ih->daddr());

            return;
        }
        else
        {
            COOP_cache_entry tempEntry;
            Packet* rpkt = allocpkt();
            hdr_coop* rmh = hdr_coop::access(rpkt);
            hdr_ip* rih = hdr_ip::access(rpkt);

            tempEntry = cache.get(index);
            rmh->dataID = tempEntry.dataID;
            sprintf(rmh->msg(), "%lf %d", \
            tempEntry.timestamp, tempEntry.ttl);
            strcpy(rmh->seq, mh->seq);

            rih->saddr() = addr();
```

```
            rih->sport() = COOP_PORT;
            rih->daddr() = ih->saddr();
            rih->dport() = COOP_PORT;

            path_hit++;
            if (start_a_log)
            {
                a_path_hit++;
                fprintf(COOP_log, "%s \t %d \t \
                %d\n", mh->seq, mh->hops, \
                addr());
            }
            Packet::free(pkt);
            total_sent ++;
            if (start_a_log)
                a_total_sent ++;
            send(rpkt, 0);
            printf("%f s %d %d %s %d %d %d \n", \
            CURRENT_TIME, addr(), rmh->type, \
            rmh->seq, rmh->dataID, rih->saddr(), \
            rih->daddr());

            return;

        }
    }// if (mh->type == 1) I am forwarding a request
    else // I am forwarding a reply
    {
        ch->direction() = hdr_cmn::DOWN; // necessary!

        total_sent ++;
        if (start_a_log)
            a_total_sent++;
        send(pkt, 0);
        printf("%f s %d %d %s %d %d %d \n", \
        CURRENT_TIME, addr(), mh->type, mh->seq, \
        mh->dataID, ih->saddr(), ih->daddr());

        return;
    }

}// I am forwarding a unicasted message
```

```
} //void COOPAgent::recv(Packet* pkt, Handler*)

int COOPAgent::broadcast(u_int32_t dataID, int ttl)
{

    Packet *pkt = allocpkt();
    hdr_coop *mh = hdr_coop::access(pkt);
    hdr_ip *ih = hdr_ip::access(pkt);
    Tcl& tcl = Tcl::instance();
    char info[128];

    strcpy(mh->msg(), "bcast");
    mh->dataID = dataID;
    mh->type = 1; // req

    ih->saddr() = addr();
    ih->sport() = COOP_PORT;
    ih->daddr() = IP_BROADCAST;
    ih->dport() = COOP_PORT;
    ih->ttl_ = ttl;
    total_sent ++;
    if (start_a_log)
        a_total_sent++;

    send(pkt, 0);
    sprintf(info, "puts \"%f : %d bcast %s\"", CURRENT_TIME, \
    addr(), mh->msg());
    tcl.eval(info);
    return (TCL_OK);

} // void COOPAgent::broadcast(char msg[], int ttl)


int COOPAgent::coopreq(nsaddr_t serverAddr, u_int32_t dataID)
{
    int index = cache.indexOf(dataID);
    aodv_rt_entry *server_entry;
    aodv_rt_entry *peer_entry;

    // local cache hit, check the ttl
```

```
total_req ++;
if (start_a_log)
{
    a_total_req ++;
}
if (index != -1)
{
    COOP_cache_entry ce = cache.get(index);
    if (ce.ttl + ce.timestamp > CURRENT_TIME)
    {// local cache hit
        ce.last_read = CURRENT_TIME;
        cache.set(index, ce);
        printf("%f : Get data %d from local cache.\n",\
         CURRENT_TIME, ce.dataID);
        local_hit ++;
        if (start_a_log)
        {
            a_local_hit ++;
            fprintf(COOP_log, "%d-%d-%d \t %d \t \
            %d\n", addr(), dataID, total_req, 0,\
             addr());
        }
        return (TCL_OK);
    }
}
// local cache miss, see if it is in rrt:

index = rrt.lookup(dataID);
printf("%f : rrt lookup result : %d.\n", CURRENT_TIME, index);

// It is in rrt, send the req to the rrt source or the server,
// whichever is closer
if (index != -1)
{
    // the default src is server addr
    nsaddr_t src_addr = serverAddr;
    server_entry = rtp->rt_lookup(serverAddr);
    // try to decide which src to use
    if (server_entry == 0)
    {
        // No route to server
        src_addr = rrt.get(index).requester;
```

```
}
else
{
    peer_entry = \
    rtp->rt_lookup(rrt.get(index).requester);

    if (peer_entry == 0)
    {
        // Has route to server,
        // but no route to pre-requester
        src_addr = rrt.get(index).requester;
    }
    else
    {
        // has route to server,
        // and has route to pre-requester
        if (peer_entry->get_rt_hops() \
        < server_entry->get_rt_hops())
            src_addr = \
            rrt.get(index).requester;
    }
} // try to decide which src to use


Packet* pkt = allocpkt();
hdr_coop* mh = hdr_coop::access(pkt);
hdr_ip *ih = hdr_ip::access(pkt);

mh->type = 1;
mh->hops = 0;
// let the serverAddr be the original server,
// so that the receiving side can tell.
mh->serverAddr = serverAddr;
mh->dataID = dataID;
strcpy(mh->msg(), "req");
sprintf(mh->seq, "%d-%d-%d", addr(), dataID, total_req);

//COOP_seq++;

ih->daddr() = src_addr;
ih->dport() = COOP_PORT;
ih->saddr() = addr();
```

```
    ih->sport() = COOP_PORT;

    total_sent ++;
    if (start_a_log)
        a_total_sent++;
    send(pkt, 0);

    printf("%f s %d %d %s %d %d %d \n", CURRENT_TIME, \
    addr(), mh->type, mh->seq, mh->dataID, ih->saddr(), \
    ih->daddr());

}// if (index != -1) it is in rrt

if (index == -1)
{
    // if the data is not in the rrt table, send broadcast:
    //int ret = broadcast(mh->dataID, COOP_RANGE);
    //return ret;
        Packet *bpkt = allocpkt();
        hdr_coop *bmh = hdr_coop::access(bpkt);
        hdr_ip *bih = hdr_ip::access(bpkt);

        strcpy(bmh->msg(), "bcast");
        bmh->dataID = dataID;
    bmh->hops = 0;
        bmh->type = 1; // req
    sprintf(bmh->seq, "%d-%d-%d", addr(), dataID, \
    total_req);
    //COOP_seq++;

        bih->saddr() = addr();
        bih->sport() = COOP_PORT;
        bih->daddr() = IP_BROADCAST;
        bih->dport() = COOP_PORT;
        bih->ttl_ = COOP_RANGE;

    total_sent ++;
    if (start_a_log)
        a_total_sent++;

        send(bpkt, 0);
    printf("%f s %d %d %s %d %d %d \n", CURRENT_TIME, \
```

```
        addr(), bmh->type, bmh->seq, bmh->dataID, \
        bih->saddr(), bih->daddr());

        pendingID = dataID;
        strcpy(pendingSeq, bmh->seq);

        btimer.resched(ONE_HOP*COOP_RANGE*4);
    }
    return (TCL_OK);

}// int COOPAgent::coopreq(u_int32_t dataID, nsaddr_t serverAddr)

void COOPAgent::timeout()
{
        //printf("%f : rtimeout.\n", CURRENT_TIME);
    if (CURRENT_TIME < COOP_time && pendingID == -1)
    {
        int dataID = zipf(ALPHA, DATA_NUM);
        printf("%f : reqest dataID = %d.\n", \
        CURRENT_TIME, dataID);

        if (CURRENT_TIME > COOP_wu_time)
            start_a_log = 1;

            int ret = coopreq(SERVERID, dataID);

        double r_val = rand_val(0);
        double exp_val = -MEAN_INT * log(r_val);
        printf("%f : r_val = %lf ; exp_val = %lf \n", \
        CURRENT_TIME, r_val, exp_val);

            rtimer.resched(exp_val);
    }
}

void COOPAgent::btimeout()
{
    // btimer expires, to unicast the request to the server
    if (pendingID != -1)
    {
                Packet* pkt = allocpkt();
                hdr_coop* mh = hdr_coop::access(pkt);
```

```
                hdr_ip *ih = hdr_ip::access(pkt);

                mh->type = 1;
                // let the serverAddr be the original server,
                // so that the receiving side can tell.
                mh->serverAddr = SERVERID;
                mh->dataID = pendingID;
                strcpy(mh->msg(), "req");
                strcpy(mh->seq, pendingSeq);

                ih->daddr() = SERVERID;
                ih->dport() = COOP_PORT;
                ih->saddr() = addr();
                ih->sport() = COOP_PORT;

        total_sent++;
        if (start_a_log)
            a_total_sent++;
                send(pkt, 0);

        pendingID = -1;
        strcpy(pendingSeq, "null");

                printf("%f s %d %d %s %d %d %d \n", CURRENT_TIME,\
                 addr(), mh->type, mh->seq, mh->dataID, ih->saddr(),\
                 ih->daddr());

    }
}

/*
 * $proc handler $handler
 * $proc send $msg
 */
int COOPAgent::command(int argc, const char*const* argv)
{
    Tcl& tcl = Tcl::instance();
    char info[128];

    switch (argc)
    {
    case 2: //$proc cache-content
```

```
if (strcmp(argv[1], "cache-content") == 0)
{
    COOP_cache_entry tempEntry;
    int i = cache.getTop();
    sprintf(info, "puts \"dataID \t timestamp \t\
    ttl \t last_read \t importance\"");
    tcl.eval(info);
    while (i--)
    {
        tempEntry = cache.get(i);
        sprintf(info, "puts \"%d \t %f \t %d \
        \t %f \t %d\"", tempEntry.dataID, \
        tempEntry.timestamp, tempEntry.ttl, \
        tempEntry.last_read, \
        tempEntry.importance);
        tcl.eval(info);
    }
    return (TCL_OK);
}// $proc cache-content
// $proc rt - request information from routing agent.
else if (strcmp(argv[1], "rt-init") == 0)
{
    Packet* pkt = allocpkt();
    hdr_coop* mh = hdr_coop::access(pkt);
    hdr_ip* ih = hdr_ip::access(pkt);

    mh->type = 3;
    mh->serverAddr = addr();
    mh->dataID = 0;
    strcpy(mh->msg(), "rt-init");

    ih->daddr() = addr();
    ih->dport() = RT_PORT;
    ih->saddr() = addr();
    ih->sport() = COOP_PORT;

    printf("%f : %d sent request to routing \
    agent.\n", CURRENT_TIME, addr());
    send(pkt, 0);

    return (TCL_OK);
}
```

```
    else if (strcmp(argv[1], "rrt-content") == 0)
    {
        RRT_entry tempEntry;
        int i, size;
        size = (rrt.getCurrent() < RRT_SIZE ? \
        rrt.getCurrent() : RRT_SIZE);

        i = 1;
        printf("nodeID \t index \t dataID \t \
        requester \t timestamp\n");
        while (i <= size)
        {
            //print current-i % RRT_SIZE
            tempEntry = rrt.get((rrt.getCurrent()\
            -i)%RRT_SIZE);
            printf("%d \t %d \t %d \t %d \t %f\n",\
            addr(), (rrt.getCurrent()-i)%RRT_SIZE,\
            tempEntry.dataID, tempEntry.requester,\
            tempEntry.timestamp);
            i++;
        }

        return (TCL_OK);
    }// $proc cache-content
// "$proc send msg" - must connect src-dst before using this
case 3:
    if (strcmp(argv[1], "send") == 0)
    {
        Packet* pkt = allocpkt();
        hdr_coop* mh = hdr_coop::access(pkt);
        const char* s = argv[2];
        int n = strlen(s);
        if (n >= mh->maxmsg()) {
            tcl.result("message too big");
            Packet::free(pkt);
            return (TCL_ERROR);
        }
        strcpy(mh->msg(), s);
        send(pkt, 0);
        sprintf(info, "puts \"%f : %d send %s\"", \
        CURRENT_TIME, addr(), mh->msg());
        tcl.eval(info);
```

```
    return (TCL_OK);
}
// $proc add-data dataID
else if (strcmp(argv[1], "add-data") == 0)
{
    COOP_cache_entry newEntry;
    newEntry.dataID = (u_int32_t)atoi(argv[2]);
    newEntry.timestamp = CURRENT_TIME;
    newEntry.ttl = 1000;
    newEntry.weight = 0;
    newEntry.importance = 1;
    //printf("dataID = %d\n", newEntry.dataID);
    cache.add(newEntry);
    return (TCL_OK);
}
// $proc rt-info nodeAddr
else if(strcmp(argv[1], "rt-info") == 0)
{
    aodv_rt_entry *rt_entry;

    rt_entry = rtp->rt_lookup(atoi(argv[2]));
    if (rt_entry == 0)
    {
        printf("%f : COOPAgent : No route to\
         %s!\n", CURRENT_TIME, argv[2]);
    }
    else
    {
        printf("%f : COOPAgent : the distance \
        to %s is %d.\n", CURRENT_TIME, argv[2],\
        rt_entry->get_rt_hops());
    }
    return (TCL_OK);

}
// $proc cache-init ms
else if(strcmp(argv[1], "cache-init") == 0)
{
    cache.cache_init(atoi(argv[2]));
    printf("%f : Node %d init cache size to %d.\
    \n", CURRENT_TIME, addr(), atoi(argv[2]));
    return (TCL_OK);
```

```
}
// $proc zipf-init seed
else if(strcmp(argv[1], "zipf-init") == 0)
{
    rand_val(atoi(argv[2]));
    printf("%f : init zipf seed to %d. \n", \
    CURRENT_TIME, atoi(argv[2]));
    return (TCL_OK);
}
// $proc log-init logFileName
else if(strcmp(argv[1], "log-init") == 0)
{
    COOP_log = fopen(argv[2], "w");
    if(COOP_log == NULL)
    {
        perror("log-init logFileName");
        exit(1);
    }
    //fprintf(COOP_log, "file opened.\n");
    return (TCL_OK);
}
// $proc log-close logFileName
else if(strcmp(argv[1], "log-close") == 0)
{
    fclose(COOP_log);
    fprintf(stderr, "Total req : %d\n", total_req);
    fprintf(stderr, "Local hit : %d\n", local_hit);
    fprintf(stderr, "Server hit : %d\n", \
    server_hit);
    fprintf(stderr, "Zone hit : %d\n", zone_hit);
    fprintf(stderr, "Path hit : %d\n", path_hit);
    fprintf(stderr, "Total sent : %d\n",\
    total_sent);

    fprintf(stderr, "\nAdjusted total req : %d\n",\
    a_total_req);
    fprintf(stderr, "Adjusted local hit : %d\n",\
    a_local_hit);
    fprintf(stderr, "Adjusted server hit : %d\n", \
    a_server_hit);
    fprintf(stderr, "Adjusted zone hit : %d\n", \
    a_zone_hit);
```

```
    fprintf(stderr, "Adjusted path hit : %d\n", \
    a_path_hit);
    fprintf(stderr, "Adjusted total sent : %d\n", \
    a_total_sent);
    return (TCL_OK);
}
// $proc simulation-time time
else if (strcmp(argv[1], "simulation-time") == 0)
{
    COOP_time = atoi(argv[2]);
    return (TCL_OK);
}
// $proc coop-range int
else if (strcmp(argv[1], "coop-range") == 0)
{
    COOP_RANGE = atoi(argv[2]);
    return (TCL_OK);
}
// $proc data-num int
else if (strcmp(argv[1], "data-num") == 0)
{
    DATA_NUM = atoi(argv[2]);
    return (TCL_OK);
}
// $proc mean-int int
else if (strcmp(argv[1], "mean-int") == 0)
{
    MEAN_INT = atoi(argv[2]);
    return (TCL_OK);
}
// $proc serverid int
else if (strcmp(argv[1], "serverid") == 0)
{
    SERVERID = atoi(argv[2]);
    printf("The server ID is set to %d.\n", \
    SERVERID);
    return (TCL_OK);
}
// $proc start-req timeOffset
else if(strcmp(argv[1], "start-req") == 0)
{
    rtimer.sched(10.123 + atoi(argv[2])*2);
```

```
            return (TCL_OK);

        }
    case 4:// "$proc bcast dataID ttl"
        if (strcmp(argv[1], "bcast") == 0)
        {
            return  broadcast(atoi(argv[2]), atoi(argv[3]));
        }

        // $proc req serverAddr dataID
        else if (strcmp(argv[1], "req") == 0)
        {

            int ret = coopreq(atoi(argv[2]), atoi(argv[3]));
            printf("ret value = %d \n", ret);
            return ret;
        }
    default:
        return (Agent::command(argc, argv));

    }// switch

}
```