View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Texas A&M Repository

ON DATA CACHING FOR MOBILE CLOUDS

A Thesis

by

YING FENG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Alexander Sprintson
Co-Chair of Committee,	Radu Stoleru
Committee Member,	I-Hong Hou
Head of Department,	Chanan Singh

May 2014

Major Subject: Computer Engineering

Copyright 2014 Ying Feng

ABSTRACT

Recent advances in smart device technologies have enabled a new computing paradigm in which large amounts of data are stored and processed on mobile devices. Despite the available powerful hardware, the actual capabilities of mobile devices are rather limited as they are often battery powered. This work explores data caching for k-out-of-n computing in mobile cloud environments, with the goal of distributing data in a way that the expected future energy consumption for nodes to retrieve data is minimized, while preserving reliability. More specifically, we propose to place data caches (in addition to the originally stored data) based on the actual data access patterns and the network topology. Consequently, we formulate the cache placement optimization problem and propose a centralized caching framework that optimally solves the problem and a distributed solution that approximates the optimal solution. The distributed caching framework *(DC)* learns data access patterns by sniffing packets and informing a resident cache daemon about popular data items. Extensive evaluations are carried out through both simulations and a proof-of-concept hardware implementation. The results show that our proposed *D-C* effectively improves the energy efficiency by up to 70% when compared with a no-caching framework, and even outperforms the centralized framework when taking the overhead into account.

DEDICATION

To My Parents,

Teachers That Were,

Teachers That Are

and

Teachers To Be

ACKNOWLEDGEMENTS

I wish to thank my advisor Dr. Radu Stoleru, without whom this work would have been an unrealized dream. His methods of intuitive thinking, research methodology and outlook on life will be something I will treasure and learn from in the years to come. I would like to thank him for accepting me into the LENSS lab, which provided me with the necessary exposure to the field of mobile cloud. I would also like to thank Dr. Alexander Sprintson and Dr. I-Hong Hou for being part of my committee.

I also wish to thank all the members in the Laboratory for Embedded Networked Sensor Systems, especially Jay Chen, for their support and help during the course of my study. This work would not have been possible without their constructive advice and criticism.

I wish to thank my parents for their constant love, support and encouragement. Last, but not the least, I would like to acknowledge my friends, for making my stay in College Station memorable and fun-filled.

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGEMENTS iv

TABLE OF CONTENTS v

LIST OF FIGURES vii

LIST OF TABLES viii

1. INTRODUCTION 1

2. STATE-OF-ART 5

3. PRELIMINARIES AND PROBLEM FORMULATION 8

   3.1 Overview and General Definitions 8
   3.2 Cache Placement Formulation 9
   3.3 Cache Placement Policy 10

4. CENTRALIZED/IDEAL CACHING FRAMEWORK 12

5. DISTRIBUTED CACHING FRAMEWORK 14

   5.1 System Architecture 14
   5.2 Statistics Collection 16
   5.3 Distributed Cache Placement 17
      5.3.1 When will a fragment cache be created? 17
      5.3.2 How to coordinate the cache placement 18
      5.3.3 How to select a cache agent 19
   5.4 Distributed Cache Replacement 19
   5.5 Integrated Solution 20

6. SIMULATION RESULTS 22

   6.1 Effect of Requests Number 22
   6.2 Effect of Buffer Size 24
   6.3 Effect of Nodes Number 25

7. SYSTEM IMPLEMENTATION AND EVALUATION 29

8. CONCLUSIONS AND FUTURE WORK 34

REFERENCES 35

LIST OF FIGURES

FIGURE Page

1.1 An overview of data allocation for k-out-of-n computing framework. . . 2

5.1 System architecture of cross-layer design for proposed distributed caching framework 15

5.2 An example of cache placement in distributed caching framework . . 17

6.1 Effect of requests number on (a). Energy Consumption; (b). Retrieval Rate; (c). Prefetching Overhead; (d). Total Caches. The test scenario is based on 14 nodes, 12 files, and the buffer size is set to be holding up to 24 fragments. 24

6.2 Effect of buffer size on (a). Energy Consumption; (b). Retrieval Rate; (c). Prefetching Overhead; (d). Total Caches. The test scenario is based on 14 nodes, 12 files, and the number of requests is fixed to 600. 26

6.3 Effect of nodes number on (a). Energy Consumption; (b). Retrieval Rate; (c). Prefetching Overhead; (d). Total Caches. The test scenario is based on 12 files, 600 requests, and the buffer size is set to be holding up to 24 fragments. 28

7.1 Router deployment and network topology 31

LIST OF TABLES

TABLE Page

3.1 Summary of Notations 9

5.1 Features and Sniffing Specifics of Packets 16

6.1 Simulation Parameters and Basic Setting 23

6.2 Energy Savings under the Effect of Requests Number 25

6.3 Energy Savings under the Effect of Buffer Size 27

6.4 Energy Savings under the Effect of Nodes Number 27

7.1 Performance Metrics for Proof-of-Concept Evaluation 33

1. INTRODUCTION

Mobile clouds provide an alternate solution for cloud computing (e.g., big data storage and processing) in environments where internet or high performance computers are unavailable. When the infrastructure network is damaged or unavailable in scenarios such as disaster responses [25] and battlefields [24], an infrastructureless mobile cloud formed by mobile devices becomes an attractive option. However, data access in a mobile cloud encounters several challenges such as intermittent connection, mobility, unreliable devices, and limited energy resources. Depending on the dynamic nature of the network, the route between nodes may change or it may be unstable over time. Nodes in the network can be inaccessible due to energy depletion, software/hardware failure, or mobility, leading to more broken links. Consequently, these must be taken into account when planning to allocate data or provide services in a mobile cloud.

As nodes in a wireless network may become inaccessible, additional mechanisms for ensuring data reliability must be employed. A k-out-of-n system [8] is a widely used and well-studied technique in many engineering fields when developing a fault-tolerant system. It describes an n-component system that can function properly as long as k ($k \leq n$) or more of the n components function properly. The k-out-of-n concept is also applied to distributed storage system where each file is encoded into n fragments by erasure coding and stored to n different nodes, called service centers *(SC)*. When a client node needs to access a file, it retrieves k fragments from k different SCs and reconstruct the file locally. In such a manner, the functionality of the system is guaranteed as long as k or more SCs are accessible. Figure 1.1 shows an overview of data allocation in a k-out-of-n computing framework, in which $n = 5$,

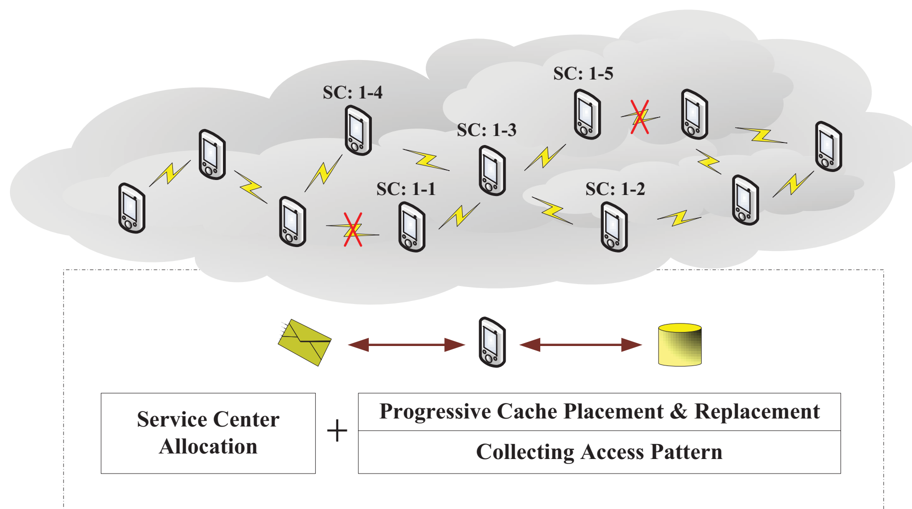

Figure 1.1: An overview of data allocation for k-out-of-n computing framework.

$k = 3$, and each fragment is represented as $fileId - fragId$.

Because each node has a different failure probability and different distances to the client nodes, the locations of service centers directly affect the energy efficiency and data availability of the system. The k-out-of-n distributed storage system in [5] places the service centers in a way to minimize the expected energy consumption for client nodes for accessing the service centers. They assumed the network is homogeneous and all nodes have equal probability to request each file. However, in reality, not all nodes request all files and some files may be requested only by a small portion of nodes. For instance, given a network of rectangle shape, if the files are requested only by client nodes located at the shorter edges of the rectangle, it will be extremely energy inefficient to place service centers at the center of the network. Additionally, for security concerns, client nodes are not allowed to keep the decoded files locally and nodes always need to retrieve the data fragments from the service centers whenever a file is needed for reading. This security constraint causes an unavoidable high energy consumption and heavy network traffics. To address these challenges, we propose to cache some "popular" data fragments in the network and

allow client nodes to retrieve data fragments from nearby caching nodes instead of always going to the farther service centers.

Our caching strategy is designed based on two observations: temporal locality of file access and the group mobility exhibited by nodes. Temporal locality of file access means that a file recently accessed by a node is likely to be accessed again by the same node in the near future. Thus collecting statistics, i.e., how files were accessed by nodes in the past, lays ground for predicting the future. Group mobility exhibited by nodes indicates that nodes often move as a group instead of moving individually. As a result, placing cached data within a group of nodes that tend to move together can also greatly improve the performance.

To the best of our knowledge, we are the first to investigate the data caching for k-out-of-n computing. Our objective is to determine the data to be cached and to select the caching nodes in a mobile network such that the expected energy consumption for nodes to access the data is minimized. Our proposed solutions monitor the file request activities and make caching decisions based on the past statistics and the failure probabilities of nodes. We first formulate the problem as an Integer Linear Programming (ILP) problem and solve it using a centralized caching algorithm (CC). We then propose a lightweight and distributed caching framework in which nodes learn the files' popularity in a distributed manner and cooperate with each other to decide the cache placement. Finally, as the cache buffer is finite, the least frequently used *(LFU)* algorithm is adopted for buffer management.

The proposed solutions are evaluated through a real hardware implementation and extensive simulations. For hardware implementation, a daemon that performs the cache placement and replacement is implemented as a Linux kernel module. A modified Kernel-AODV module [22] was used as the routing protocol. A cross-layer communication allows the network layer to pass the sniffed information to our

middleware (above the transport layer, and below the application layer). Both the cache daemon and Kernel-AODV are running on the RouterBoard 433UAH hardware and evaluated in a network of 8 nodes. Through extensive simulations, we evaluate the impacts from parameters such as the number of requests, buffer size, and network size. We compare our distributed caching framework with a no-caching scheme, centralized caching and ideal-caching in terms of energy consumption and data availability.

The rest of this thesis is organized as follows. Section 2 presents the related works. Section 3 formulates the optimal cache placement problem for k-out-of-n system and provides an effective placement policy. In Section 4, two caching frameworks, a centralized caching and an ideal caching are presented. In Section 5, a distributed cache framework is proposed. Section 6 presents performance evaluation results from simulations. Section 7 presents the hardware implementation and evaluation. Finally, section 8 concludes the thesis and discusses future work.

2. STATE-OF-ART

Data caching has been widely used in Internet to enhance the performance of web services [15] [14] [19]. However, there have not been many research efforts dedicated to mobile environments.

Dowdy and Foster [12] were among the first to study the cache placement for cooperative networks. The optimality of this problem in terms of access cost has its root in the multi-facility location problem [3] [1], which is NP hard. Baev et al. [2] proposed a 10-approximation algorithm for placing replicas in arbitrary networks by taking into account data access frequency and node storage capacity. Tang et al. [26] improved this by delivering a 4-approximation (2-approximation for uniform-size data items) solution. Both algorithms considered cache placement for multiple data items, but they became inapplicable in situations where data did not come in batches. Jin et al. [20] mathematically proved that the number of replicas of each item in the optimal solution is proportional to $p^{2/3}$, where p is the access probability of the item, and verified its huge performance gains when compared with the proportional replication strategy. Taking advantage of these results, our work also integrate the peculiar characteristics of mobile environment, i.e. unstable links, node mobility, and energy constraint, into our model.

Yin et al. [27] proposed three caching schemes: CacheData, CachePath, and HybridCache. The idea is to analyze passing-by data and cache either data or path to a known cache node. However, the design was focused on the system point of view and did not take into account the complexity of on-the-fly caching. A less aggressive caching scheme is to maintain caches only on the client sides, to which we refer as *cache-on-clients*. COOP is an example of such schemes for mobile ad-hoc networks

proposed by Du et al. [13]. To avoid overflowing active clients' buffer, COOP applied both inter and intra category rules to reduce duplicates within the cooperation zone. Another similar scheme, COCA presented by Chow et al. [7], employed a different strategy for buffer control. Two types of mobile clients, low activity mobile clients (LAM) and high activity mobile clients (HAM) were identified. A centralized server replicates appropriate data items to LAMs so that HAMs can make use of them. Building upon COCA, GroCoCa [6] introduced the concept of tightly-coupled group (TCG), defined as a set of peers pursuing a similar movement pattern and exhibiting a similar data affinity. Cache cooperation was then performed within TCGs. The major drawback of *cache-on-clients* schemes was that their performance degrades when servicing multiple highly-active clients concurrently.

The aforementioned research implicitly or explicitly assumed a group-based mobility [17] model. Hara [16] quantified the impact of node mobility on data availability in mobile ad-hoc networks. The result revealed that the Reference Point Group Mobility model had larger partition sizes and higher connectivity compared to other mobility models. As a result, cooperative caching is intrinsically suitable for Group Mobility model. Our work also assumes a group-mobility model, but should be categorized as an "in-between" algorithm between the *cache-on-the-fly* and the *cache-on-clients*. We allow caches to be placed along the paths from clients to the data source, but the caching activity happens after the observation of popular data. Additionally, more factors have been considered for cache agent selection, including the distance to the clients, failure probability, buffer availability and data security.

When considering the reliability of a distributed storage system, Dimakis et al. proposed several erasure coding algorithms, together with their maintenance schemes for distributed storage [10] [9] [11]. Erasure coding was essentially the theory behind the k-out-of-n data storage in mobile computing. MDFS [18] was the first work

to create a distributed file system on mobile devices. Chen et al. [4] [5] studied the service center allocation problem in mobile cloud and introduced the "expected distance" by taking into account the nodes' failure probability. Yadi et al. [23] proposed a caching scheme, named CAROM, that combines data replication and erasure codes to improve data availability and responsiveness. However, CAROM did not consider how to optimally place the encoded and replicated data. Compared with the traditional data caching, caching for k-out-of-n computing framework is a much more complicated problem. Facing the challenges that did not appear in these previous works, our caching algorithm considers issues such as dual-request resolution and fragments coordination for a file.

3. PRELIMINARIES AND PROBLEM FORMULATION

3.1 Overview and General Definitions

This work builds on the service center allocation algorithm proposed in [5]. [5] considered a mobile ad-hoc network consisting of N nodes. Each node v_i is associated with a failure probability P_{f_i} and the expected distance D_{ij} is obtained by estimating the "expected hop count" between node v_i and node v_j, with their failure probabilities considered. Under the assumption that data transmission/reception is the major source of energy consumption, the objective of service center allocation problem is to minimize the expected distance from client nodes to their closest k service centers.

On the other hand, each newly created file is encoded into n fragments and distributed to n selected service centers. Any subset of k fragments is able to recover the original file. As the access pattern of the file is unknown at the file creation time, the service centers are only selected based on the network topology. Without considering the access pattern, some service centers may be used much more frequently than others, leading to network hot spots.

To overcome this, we combine caching with service center allocation. The items to be cached are fragments of the stored files. The cache placement and replacement decision are made based on the collected file access patterns. We call the nodes that hold caches as *cache agents* (service centers included). There are several characteristics of the cache agents. From the "resource" perspective, each cache agent v_i is associated with a buffer of capacity L_i. As time elapses, the availability of a caching agent may vary with the number of cached items, A_i. From the "demand" perspective, the client nodes U request fragments from the cache agents and the agents learn the request frequency r of each file. Intuitively, the more popular the file is (i.e., with

Table 3.1: Summary of Notations

Symbol	Description
V, v_i	collection of nodes, $V = \{v_1, v_2, \cdots, v_N\}$
F, F^w, f_i^w	collection of files and their fragments, $F = \{F^w\}$, $F^w = \{f_1^w, f_2^w, \cdots, f_n^w\}$
P_{f_i}	failure probability of v_i
D_{ij}	expected distance between v_i and v_j
s_i^w	service center for item with fileId of w and fragId of i
L_i	buffer capacity on node v_i
A_i	the amount of buffer that has been used on node v_i
U^w, u_i^w	collection of interested user for file F^w, $U^w = \{u_i^w\}$
r_i^w	access frequence of user u_i for file F^w
K_w	total number of fragment caches for file F_w

higher requests frequency r), the larger the number of caches we need to maintain for that file. We use K_w to represent the total number of fragment caches that will be created for file F_w.

Table 3.1 presents all the notations we have so far defined.

3.2 Cache Placement Formulation

Now we are ready to formulate the cache placement optimization problem. The objective of the problem is to minimize the total expected distance from every potential user to its k cache agents. For convenience, we omit the file index w and represent the file as F in the problem formulation. Based on the previous definition, two mapping variables are defined as follows:

x_i^l: a binary variable indicating whether v_i is a cache agent for f_l.

y_{ij}^l : a binary variable indicating whether v_j is assigned to v_i for retrieving f_l.

The following Integer Linear Program (ILP) then expresses our cache placement problem.

$$\text{Minimize} \quad \sum_{l \in F} \sum_{i \in V} \sum_{j \in U} D_{ij} y_{ij}^l r_j \tag{3.1}$$

$$\text{s.t.} \quad \sum_{l \in F} \sum_{i \in V} x_i^l \leq K \tag{3.2}$$

$$\sum_{l \in F} \sum_{i \in V} y_{ij}^l \geq k, \forall j \in U \tag{3.3}$$

$$x_i^l \geq y_{ij}^l, \forall i \in V, \forall j \in U, \forall l \in F \tag{3.4}$$

$$x_{s_l}^l = 1, \forall l \in F \tag{3.5}$$

$$\sum_{l \in F} x_i^l \leq \min\{k-1, L_i - A_i\}, \forall i \in V \tag{3.6}$$

$$x_i^l, y_{ij}^l \in \{0, 1\}, \forall i \in V, \forall j \in U, \forall l \in F \tag{3.7}$$

The first constraint (Eq. 3.2) indicates that up to K fragment copies will be placed on the cache agents for this file. The second constraint (Eq. 3.3) ensures that each potential user has accesses to at least k different fragment caches. The third constraint (Eq. 3.4) makes sure that if a potential user is assigned to a node for a particular fragment, then the node must be a cache agent for that fragment. Eq. 3.5 ensures that the service centers are also cache agents. Eq. 3.6 creates a buffer limit on each cache agent. Also, for security purposes, less than k cached fragments can be created for each file. The last constraint (Eq. 3.7) is the binary requirement for the decision variables.

3.3 Cache Placement Policy

We adopt the findings from [20] to help determine the number of caches for each file, K_w, given the file's popularity and the nodes' buffer size. In Eq. 3.8 below, n is the number of service centers selected when a file is created, ϕ represents the correlation between a file's popularity and the total number of its cached fragments,

and r_w is the request frequency of file F_w. The minimum number of caches for each file is n because each file is encoded and distributed to n service centers at the creation time. Eq. 3.9 defines a user-configured variable η to represent the percentage of occupancy allowed on cache agents' buffer. Combining Eq. 3.8 with Eq. 3.9 and configuring a proper η, we can then solve for ϕ.

$$K_w = \max\{n, \phi \cdot (\sum_{i \in U} r_i)^{2/3}\} \tag{3.8}$$

$$\eta = \frac{\textit{total \# of fragment copies of all files}}{\textit{overall buffer size of all nodes}} = \frac{\sum_w K_w}{\sum_i L_i} \tag{3.9}$$

Based on this, files with higher popularity are given higher priority when selecting the cache placement. In specific, given a collection of files $\{F_w\}$, the cache placements for each file is determined one by one based on its K_w value. The process repeats until all files are associated with a specific K_w.

4. CENTRALIZED/IDEAL CACHING FRAMEWORK

The basic idea of centralized caching framework is to collect the global information and to solve the optimization problem on a master node. According to temporal locality and group mobility, the learned statistics in the near past can very well predict the future activities. To be more concrete, the master node collects collects information regarding on network topology and file access pattern by exchanging control messages, and determines the optimal cache placement in the future based on these learned statistics.

As time goes on, the network topology changes and the cached items may become obsolete. Therefore, the algorithm is executed periodically to adapt to these dynamics. Each round of the algorithm gives an updated solution for cache placement, which may or may not differ from the previous placement. Given an updated placement, the newly chosen fragment caches are immediately fetched by the cache agents, and the old caches can be kept as long as they do not violate the capacity or security constraints. If a cache buffer is full, we replace its content using the least frequently used (LFU) policy.

We refer to the framework described above as **centralized caching** (*CC*) and define a simple variation of *CC* as **ideal caching** (*IC*). The only difference between them is that *CC* uses the previous time slot for cache placement optimization while *IC* "foresees" the future access pattern and topology. *IC* simply serves as the true optimal or the ground truth of our caching framework and no other caching algorithms could outperform *IC*.

Although the centralized solution provides an optimal solution, it is computationally infeasible in a large scale network and has single node failure disadvantage. As

a result, in the next section, we propose a lightweight distributed caching framework to approximate the optimal solution.

5. DISTRIBUTED CACHING FRAMEWORK

The goal of distributed caching framework (*DC*) is to allow each individual n-ode to make its own caching decision without the need of global information. The distributed algorithm does not collect the topology information or all files' access pattern, and is robust to node failures.

5.1 System Architecture

Figure 5.1 describes the system architecture of our cross-layer design for *DC*. We add our middleware, which includes the cache daemon (CDaemon) on top of the transport layer. A cross-layer communication channel is built between the network layer (where Kernel-AODV resides) and our middleware. The CDaemon actively interacts with the network layer: CDaemon collects the access pattern information, such as *reqId* and their counters for each requested fragment from the network layer; and the network layer also looks up cache information, such as cache ID (*cacheId*) and reference number (*refs*) maintained in CDaemon. A step-by-step explanation of the framework based on a file request example is also given in Figure 5.1. Suppose A is a file requester, and B is an intermediate node or the destination node.

- Step 1: To request for a file, client node A broadcasts a file request (*fileReq*) packet containing the file ID, the requester ID, and an initial hop count value (set to 0).

- Step 2: Upon receiving *fileReq*, B examines whether the request has been seen before. If it is a new request, B updates the hop count maintained in this *fileReq* and rebroadcasts the packet. Node B then uses *cachedId* to check with CDaemon to see if it has the desired fragment. If yes, it replies A with a file

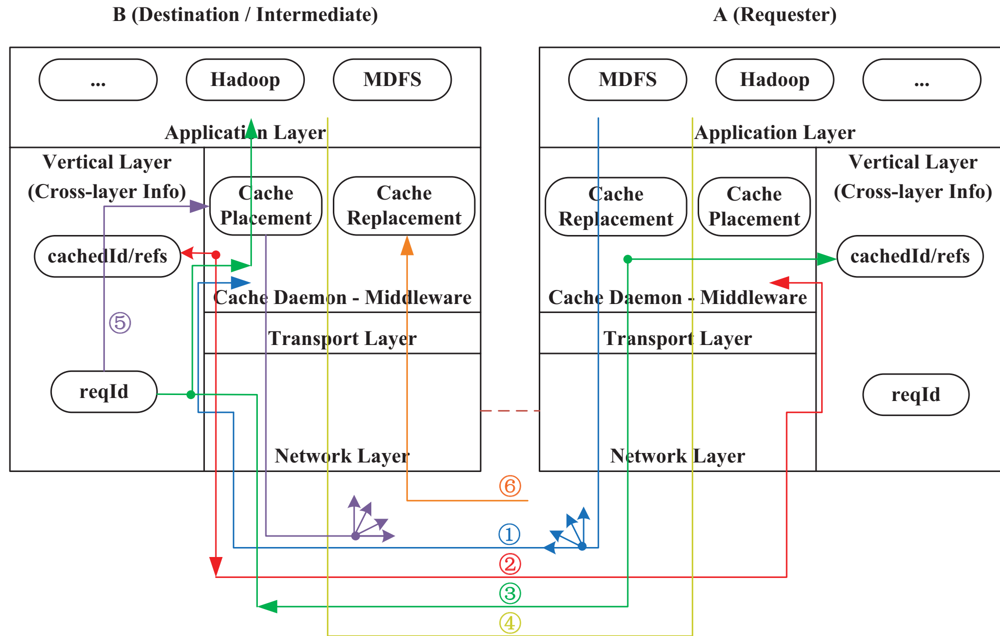

Figure 5.1: System architecture of cross-layer design for proposed distributed caching framework

reply (*fileRep*) including the hop count information. (Note that sending back a *fileRep* may require a route discovery in reactive routing protocol.)

- Step 3: Upon receiving all the *fileReps*, the CDaemon in A decides which are the closest cached fragments and unicasts fragment requests (*fragReq*) to these cache agents. (Note that unicasting the *fragReq* may also require a route discovery.)

- Step 4: When *fragReq* reaches the destination, a TCP session is established for reliable data transmission.

Step 5 and *Step 6* are related to the cache placement/replacement of DC and will be explained in the following subsections.

Table 5.1: Features and Sniffing Specifics of Packets

Packets	Protocol	Dst Port	Dst Addr	Sniff? and Actions
fileReq	UDP/IP	CDaemon	broadcast	yes, check *cachedId*
fileRep	UDP/IP	CDaemon	unicast	no, –
fragReq	UDP/IP	CDaemon	unicast	yes, update *reqId*
data	TCP/IP	MDFS	unicast	no, –

5.2 Statistics Collection

To learn the file access pattern, the network layer sniffs the passing-by packets and delivers the packets of interest to the middleware. In this way, CDaemon learns the file request frequency and whom the file is requested by. As cross-layer communication also introduces computation overhead, only the packets that are necessary should be passed to the middleware. Table 5.1 summarizes the packets defined in our framework and the packets that CDaemon is interested in. *fileReq* is sniffed so that CDaemon can check if any cached fragment is available locally and replies if necessary. *fragReq* is sniffed at intermediate nodes so that nodes can learn about the popularity and request frequency of the file.

The process of sniffing packets is done efficiently at network layer by several filters. The network layer simply examines a very small part of each arrival packet and determines weather to pass it to the middleware. The fields that are checked in each packet are shown below:

- *Protocol Type*: All the control packets are sent via UDP, and from which we can rule out the unnecessary *data packets*;
- *Destination Port*: There may be other UDP packets, and we are only interested in those sent to the CDaemon port;
- *Message Type*: MDFS, CDaemon, and the network layer should have an a-

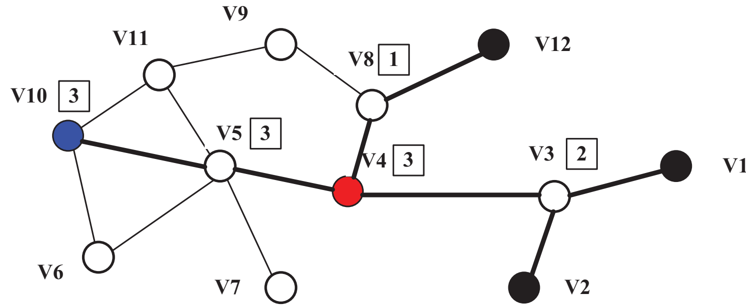

Figure 5.2: An example of cache placement in distributed caching framework

greement on the payload structure. Therefore, the network layer can easily recognize the packets such as *fileReq*, *fileRep*, and *fragReq* by reading the first few bytes of the payload.

5.3 Distributed Cache Placement

Once nodes have identified the popular fragments needed to be cached, *DC* algorithm needs to determine where to place the cache fragments to maximize the energy saving. The intuition is to select the nodes that are closest to the file requestors in terms of the hop-count. We assume that the node that first observes the popular fragment is the closest one to the users group. Nodes on the route from the file requestor to the service centers cooperate to determine the best cache agent. An example is illustrated in Figure 5.2.

5.3.1 When will a fragment cache be created?

A new fragment cache is added when we find the counter associated with the fragment exceeds a predefined threshold θ. Suppose θ is set to 3 in Figure 5.2. After v_1, v_2, and v_{12} make the same *fragReq* destined for v_{10}, v_4, v_5, and v_{10} will update their counter of the requested fragment to 3. This will trigger *DC* to add a new fragment cache.

Parameter θ has a great impact on the system performance as it affects the

frequency that the cached fragments are updated. We determine θ by estimating the average number of requests that each fragment cache will serve. If the number of actual requests exceeds the predefined value, then an extra fragment cache is necessary. The average number of requests is estimated as following:

$$\sum r_i = \textit{file request frequency}$$
$$\sum r_i = \textit{fragment request frequency for this file}$$
$$K = \textit{total \# of fragment copies of this file}$$

$$\frac{k \sum r_i}{K} = \frac{k \sum r_i}{\max\{n, \phi \cdot (\sum r_i)^{2/3}\}} \approx \frac{k}{\phi} \tag{5.1}$$

From Eq. 5.1, we then set θ to be k/ϕ.

5.3.2 How to coordinate the cache placement

In the previous example, when v_4, v_5 and v_{10} all reach the threshold defined by θ, only one of them should initiate its cache placement module. Since our objective is to minimize the distance from the file requestors to the fragment cache, v_4 seems to be the best candidate among the three. From the observation that the node closest to the file requestors reaches the threshold earlier than other candidate nodes (v_5 and v_{10}), v_4 can actively notify other candidates NOT to cache the fragment. This is achieved by piggybacking a flag in *fragReq* at v_4 to inform other nodes on the route, i.e., v_5 and v_{10}, to flush their request counters for this fragment. In this manner, only one new cache will be created, and it is placed closest to the file requestors.

5.3.3 How to select a cache agent

Although only v_4 will initiate its cache placement module, any node in its vicinity has a chance to be selected as the cache agent. v_4 coordinates with all its 1-hop neighbors and determines the best cache agent by comparing their *qualification scores*, defined in Eq. 5.2.

$$score(i) = I(i) \cdot \left\{ \alpha \cdot Pf_i + (1-\alpha) \cdot \frac{L_i - A_i}{L_i} \right\} \tag{5.2}$$

In Eq. 5.2, $I(i)$ is an indicator variable showing whether adding the new fragment cache will violate the security constraint on v_i, and α is a weight parameter in the range $(0, 1)$. We define the score in such a way to eliminate the nodes that may violate the security constraint, and give the nodes with lower failure probability or more buffer space higher score.

To be more specific on how the control messages are exchanged between v_4 and its neighbors, v_4 first broadcasts an exchange request (*exReq*) to its neighbors. Upon receiving the messages, nodes compute their qualification scores and reply with (*exRep*) messages. v_4 then compares those scores with its own score and sends a (*exCfm*) message to the node with the highest score (if the best node is not v_4 itself). The node selected to be the new cache agent will explicitly prefetch the corresponding fragment. The process corresponds to the *Step 5* in Figure 5.1.

5.4 Distributed Cache Replacement

Since caches may become inactive and the buffer may be fully occupied, a cache replacement policy is necessary to ensure the effectiveness of the caching algorithm. Similar to the centralized solution, the LFU algorithm is adopted to manage the

buffer. Specifically, we assign a reference number (*refs*) to each cached fragment, and the number is incremented by 1 whenever the fragment is accessed. If a cache agent's buffer is full, the cache replacement module will be activated and evict the fragment with the smallest *refs*. This process corresponds to *Step 6* in Figure 5.1.

5.5 Integrated Solution

Combining all the procedures described in this section, Algorithm 1 is the pseudocode illustrating our distributed caching framework.

Algorithm 1 Integrated Distributed Caching

```
1:  On arrival of a packet:
2:  if UDP packet && CDaemon port then
3:      sniff the payload, obtain message type
4:      if valid length of the type then
5:          wrap as a task, insert into task queue;
6:      end if
7:  else
8:      return;
9:  end if
10:
11: On processing of a task:
12: if taskType == fileReq then
13:     if first time to see it then
14:         check cachedId, re-broadcast
15:     end if
16: else if taskType == fileRep then
17:     if destined for itself then
18:         Process
19:     end if
20: else if taskType == fragReq then
21:     if destined for itself then
22:         Update refs
23:     else if optional then
24:         if reach the threshold then
25:             Piggyback, generate exReq
26:         end if
27:     end if
28: else if taskType == exReq then
29:     Compute score, generate exRep
30: else if taskType == exRep then
31:     Select agent, generate exCfm if necessary
32: else if taskType == exCfm then
33:     Retrieve the desired fragment
34:     ⋮
35: end if
```

6. SIMULATION RESULTS

Our main goal is to compare our proposed distributed caching (*DC*) framework with no-caching (*NC*), centralized caching (*CC*) and the ground truth ideal caching (*IC*) in terms of:

- Average energy consumption: 1 unit per fragment per hop for retrieving files (assuming uniform-size fragments).
- Average retrieval rate: the successful rate (percentage) of retrieving files among all the access requests.
- Average prefetching overhead: the same unit as energy consumption, but only accounting for overhead by prefetching caches to new cache agents.

Simulations were conducted using Matlab R2009a. We considered a mobile network, where nodes were randomly deployed and moved based on the Reference Point Group Mobility (RPGM) model [17]. Specifically, we used the 4-hour mobility traces generated in [5]. For the file access pattern, we assumed a Zipf's distribution with α set to 1. Table 6.1 presents the basic configurations for the experiments. We were particularly interested in evaluating the caching performance through the effect of the following parameters: 1) number of requests; 2) buffer size; 3) network size.

6.1 Effect of Requests Number

Figure 6.1 depicts the performance metrics of running *NC*, *CC*, *DC*, and *IC* algorithms with increasing number of requests. In general, all three caching algorithms introduce significant reduction (more than 50%) in energy consumption compared to *NC*, and the effect becomes more evident (more than 70%) as the number of

Table 6.1: Simulation Parameters and Basic Setting

Network Size	$400 \times 400m^2$
Communication Range	$120m$
File Encoding	$n = 7$, $k = 4$
Number of Requests	Varying from 300 to 1500 Reqs
Size of Buffer	Varying from 20 to 40 fragments per node
Number of Nodes	Varying from 14 to 26 nodes
Alg Running Interval	*10mins* for CC/IC

requests increases. With more caches residing in the network, the data availability (measured by retrieval rate) also increases (around 13%). This implies an improved energy efficiency since we are retrieving more files with less energy. On the other hand, relocating caches to new cache agents incurs overhead, the amount of which, however, is minor (less than 10%) with respect to the total energy consumption.

A closer observation on the caching algorithms shows that our proposed *DC* achieves better performance in comparison with the other two. Though the energy consumption in *CC* is smaller than that in *DC*, which is not surprising as *CC* uses the global knowledge for optimization, the difference between them is minor. This demonstrates that partial topology information and data access pattern are sufficient for *DC* to make a good caching decision. While looking at the huge prefetching overhead in *CC*, it may not be worthwhile to spend so much communication and computation cost to attain such small energy gains. In other words, the suboptimal solution given by *DC* is good enough for most applications. Table 6.2 shows their respective energy savings, where the gross saving is the difference between the actual energy consumption with caching from the one in *NC*, and the net saving is calculated by further deduction of the prefetching overhead. Apparently, *DC* is more efficient than *CC* when prefetching overhead is considered. Lastly, by investigating the ground truth *IC*, we see that more than 60% of the energy consumption is a

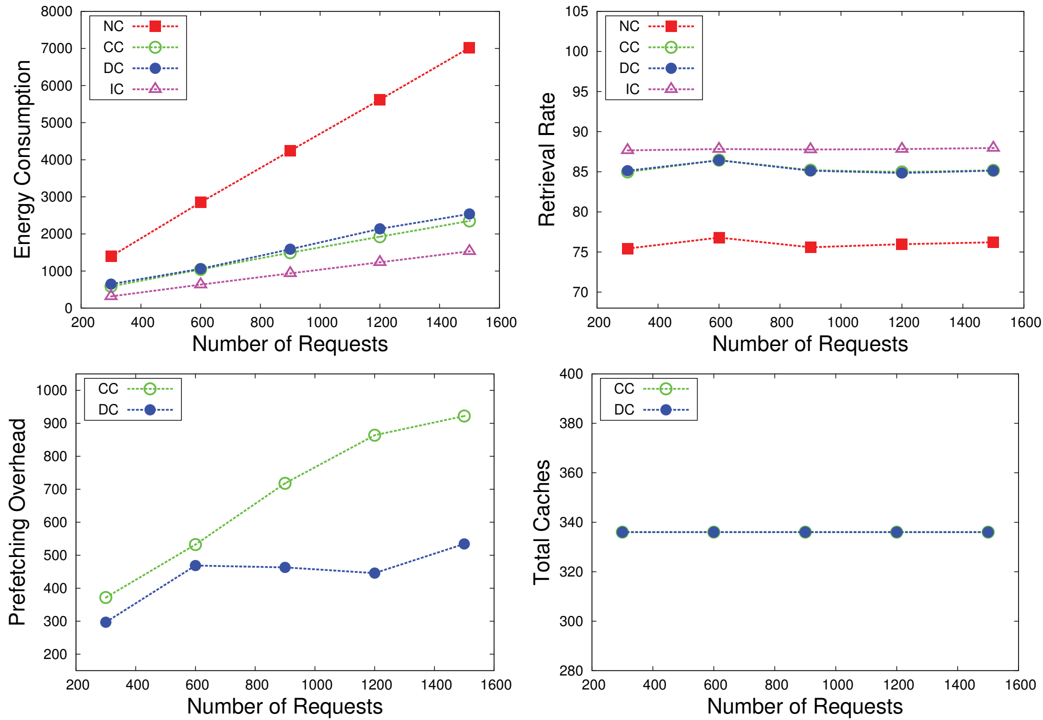

Figure 6.1: Effect of requests number on (a). Energy Consumption; (b). Retrieval Rate; (c). Prefetching Overhead; (d). Total Caches. The test scenario is based on 14 nodes, 12 files, and the buffer size is set to be holding up to 24 fragments.

must-pay price due to the buffer limit and security constraint, and the best retrieval rate under the given node failure and mobility model is close to *DC* (less than 3% better). As a result, given partial network topology and data access pattern information, the performance of *DC* can approach to optimal when the file access frequency is high.

6.2 Effect of Buffer Size

Figure 6.2 depicts the performance metrics of running *NC*, *CC*, *DC*, and *IC* algorithms with varying buffer sizes. Table 6.3 presents the gross and net energy savings for *CC* and *DC*. The most significant observation is that with more buffers, the performance of *CC* and *DC* are both improved, and their performance gaps

Table 6.2: Energy Savings under the Effect of Requests Number

Alg	Saving	300Req	600Req	900Req	1200Req	1500Req
CC	gross	809	1808	2748	3694	4666
	net	437	1275	2030	2830	3744
DC	gross	747	1790	2652	3481	4478
	net	450	1321	2189	3035	3943

decrease as the buffer size increases. The reason for the first result is straightforward: with larger buffers, more caches can be placed in the network, resulting in fewer hop counts when retrieving files. Such improvement ceases when reaching the maximum point exerted by security constraint. (Note that the maximum is 36 fragment caches per node for the given k value and file number.) This also explains for the change of the retrieval rate. As for the second result, it can be explained by the total number of caches. Starting from 28, the curve of total caches in *CC* falls below that in *DC*, indicating that *CC* does not make full use of the buffer. This is primarily because each round in *CC* finds the optimal placement for the current time slot independently. When the updated solution needs to merge with the previous solution, lots of conflicts may occur due to buffer capacity or security constraint, leading to a large number of evictions. On the other hand, *DC* avoids this effect by integrating those two factors into the score evaluation of cache candidates. This narrows the gap of energy consumption between *DC* and *CC*, and this also explains the huge prefetching overhead in *CC*. By comparison with *IC*, it is shown that the performance of *DC* approaches optimal when the buffer is relatively rich.

6.3 Effect of Nodes Number

Figure 6.3 depicts the performance metrics of running *NC*, *CC*, *DC* and *IC* algorithms with varying number of nodes in the network. Table 6.4 presents the gross and net energy savings for *CC* and *DC*. An immediate observation is that the perfor-

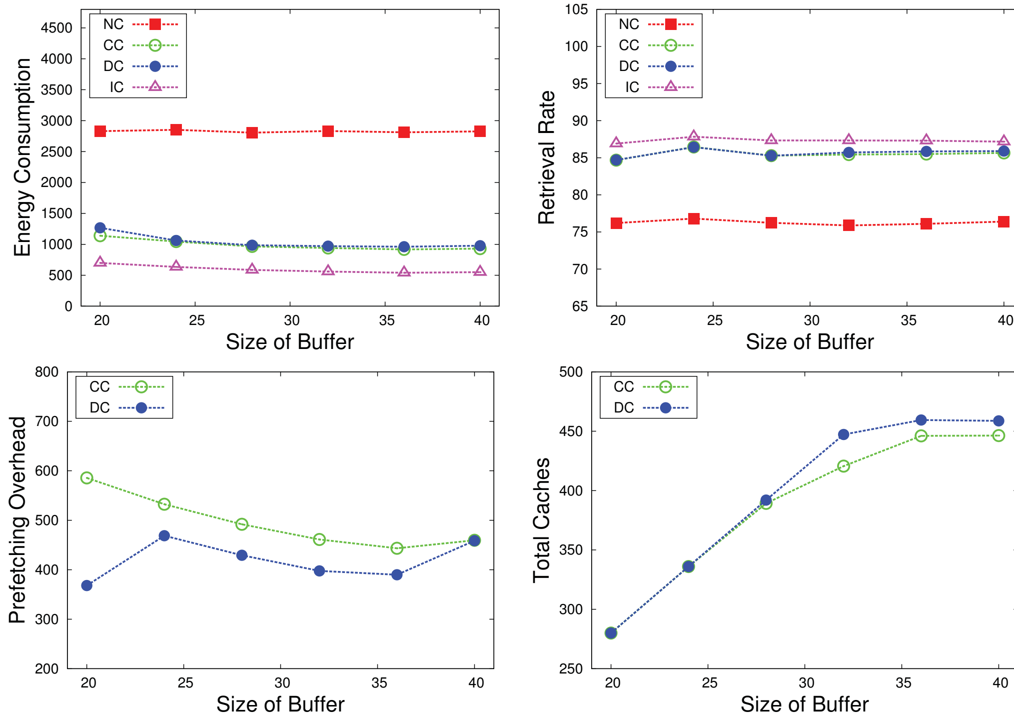

Figure 6.2: Effect of buffer size on (a). Energy Consumption; (b). Retrieval Rate; (c). Prefetching Overhead; (d). Total Caches. The test scenario is based on 14 nodes, 12 files, and the number of requests is fixed to 600.

mance of *NC* is highly subjective to the number of nodes and their movements. For energy consumption, more nodes usually implies more hops, therefore more energy consumption. When there are not enough nodes in the area and the network density is relatively low (below 22 in our simulation), adding more nodes to the network only forms paths with more hops, thus slightly increasing the energy consumption for retrieving the data. After the network density has reached a "saturated" point (22 in our simulation), the chance of nodes finding better or shorter paths to cache agents increases and thus the total energy consumption starts to decrease. As for the retrieval rate, higher number of nodes generally provides more candidate cache agents and thus improves the data availability. Another interesting observation is

Table 6.3: Energy Savings under the Effect of Buffer Size

Alg	Saving	20Buf	24Buf	28Buf	32Buf	36Buf	40Buf
CC	gross	1691	1808	1840	1891	1895	1897
	net	1105	1275	1348	1430	1451	1437
DC	gross	1563	1790	1820	1861	1851	1849
	net	1195	1321	1391	1464	1461	1451

Table 6.4: Energy Savings under the Effect of Nodes Number

Alg	Saving	14Node	18Node	22Node	26Node
CC	gross	1808	2013	2391	2386
	net	1275	1300	1600	1569
DC	gross	1790	2055	2334	2325
	net	1321	1491	1712	1673

that with caching enabled, the fluctuation of both energy and retrieval rate reduces because the file requests become more likely to be fulfilled by the nearby cache agents rather than the service centers farther away. As there are always cached fragments somewhere in the network, the failures of the service centers do not significantly bring down the performance of the system. After examining *CC*, *DC* and *IC*, it is clear that our proposed *DC* is much more effective under all circumstances.

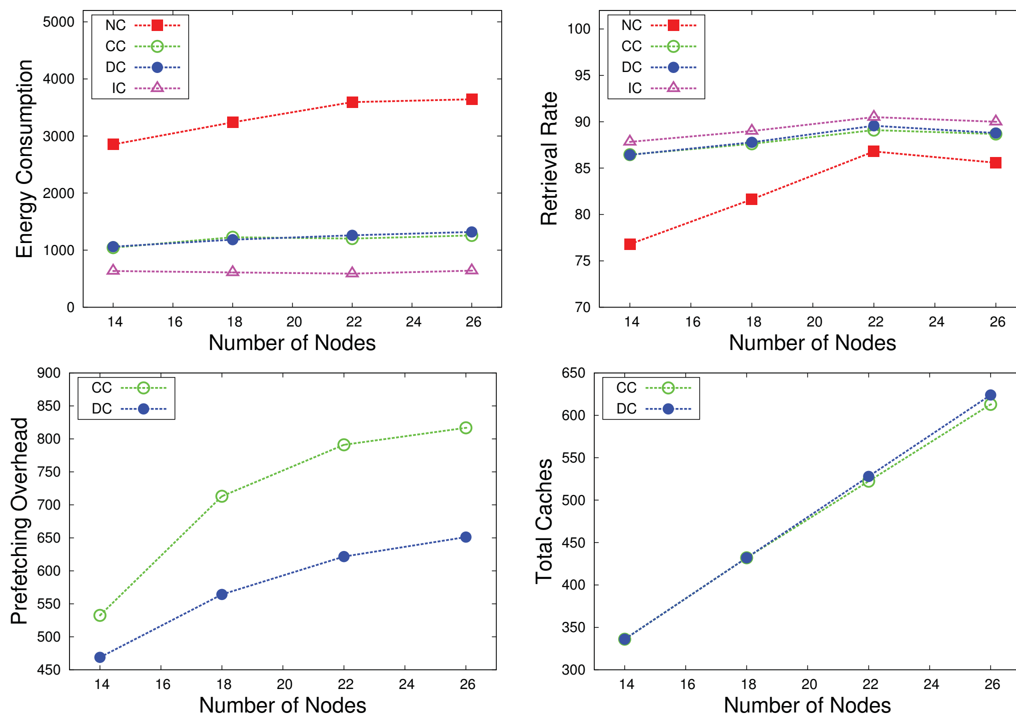

Figure 6.3: Effect of nodes number on (a). Energy Consumption; (b). Retrieval Rate; (c). Prefetching Overhead; (d). Total Caches. The test scenario is based on 12 files, 600 requests, and the buffer size is set to be holding up to 24 fragments.

7. SYSTEM IMPLEMENTATION AND EVALUATION

The hardware implementation was done on RouterBoard 433UAH, which had a 670MHz Atheros CPU, 128MB SDRAM, 512MB NAND storage, and was configured with three 10/100 Mbit/s Ethernet ports, and two 2.4/5GHz radio cards. We installed the OpenWrt operating system, an embedded system based on Linux kernel 2.6. The OpenWrt allowed us to customize the packages (applications/kernel modules) and build a user-configured image for flashing. Specifically, three packages we developed or modified were added into OpenWrt: 1) Kernel-AODV as a kernel module, for acting as the underlying routing protocol; 2) CDaemon as a kernel module, for sniffing packets and handling cache placement/replacement events; and 3) SimpleFS as an application, for emulating the mobile distributed file system (MDFS).

Our Kernel-AODV was developed based on [22], which provided an open source AODV routing protocol based on Linux kernel 2.4. We modified it so that it can adapt to the latest kernel version of the Openwrt, i.e., kernel 2.6. The radio cards were by default disabled, with no wireless interfaces configured, and the mode was set to *access point*. Furthermore, the firewall was by default configured to reject all packet forwarding events. To run Kernel-AODV on this board, we modified the files */etc/config/wireless* and *etc/config/firewall* to enable wireless transmission and packet forwarding in a wireless ad-hoc network. In addition, we manually configured a wireless interface for one of the radio cards, *wlan0*, and let it act as an *AODV* device, before inserting the Kernel-AODV module. With these, we were able to ping across nodes in a multi-hop manner.

For the CDaemon, we followed the logic of the algorithm described in Algorithm 1.

We used the kernel space socket interface for sending and receiving packets. To facilitate packet sniffing, we utilized Netfilter [21] mechanism. Its current architecture includes five hooks in the IP layer and $NF_IP_PRE_ROUTING$ is the first hook for all incoming packets. We registered a callback function for this hook and any packet that traveled through it would invoke the callback function. By reserving the first few bytes of the payload for message type, we could recognize the intercepted packets efficiently and deliver them to CDaemon if necessary. A vertical channel across multiple layers was created to facilitate cross-layer communication between our middleware and network layer.

Lastly, we implemented a SimpleFS that behaves like a simple mobile distributed file system (MDFS). Basically, SimpleFS accepted two different commands, *create* and *retrieve*. Command ***create fileId fragId*** created a fragment at the current node (service center) under the application-defined directory. At the time of inserting the CDaemon module, this particular directory would be scanned and used to initialize cache table (i.e., *cachedId*) maintained in CDaemon. Command ***retrieve fileId*** broadcasted a *fileReq* to the network and performed the whole-stack file retrieval procedure. We used *SimpleFS* to test and evaluate the performance of our CDaemon.

To demonstrate the effectiveness of *DC* on real hardware, we deployed 8 routers in our department, as shown in Figure 7.1. Because of the hardware and resource limitation, we considered only deployments in static network. We set transmission power to *15dBm* and ensured a multi-hop network topology, though some of the links might be unstable due to the interference or obstacle in between. 5 files were created and distributed by *SimpleFS* with $n = 5$ and $k = 4$. The service centers were determined at the time when the files are created. The buffer size at each node was set to be 6 (can cache up to 6 fragments). In a one-hour period, each node generated 60 requests (480 requests in total) according to files' popularity, which

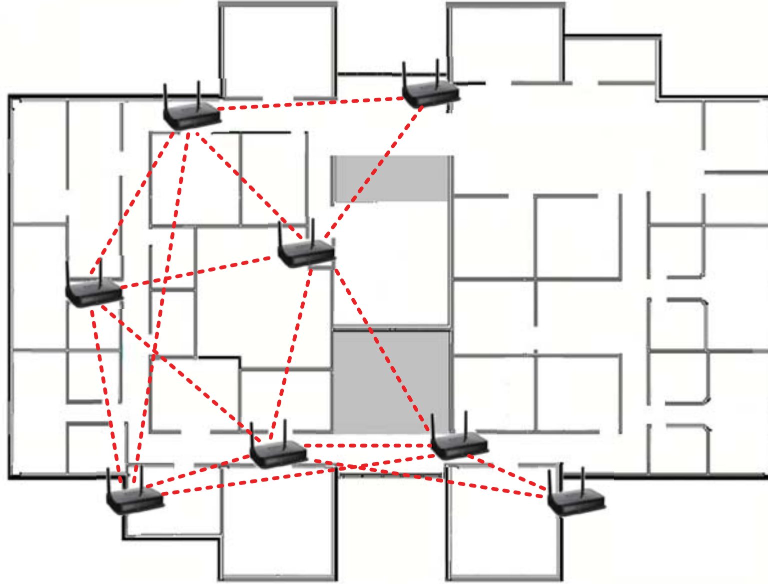

Figure 7.1: Router deployment and network topology

follows a Zipf's distribution.

Similar to the simulation in Section 6, we measured the energy consumption, retrieval rate, and prefetching overhead. Besides, communication overhead and routing overhead were also measured. Communication overhead included the control packets introduced by k-out-of-n framework (*flReq*, *flRep*, and *frReq*) and caching (*exReq*, *exRep*, and *exCfm*); routing overhead referred to the control packets for Kernel-AODV (*rreq* and *rrep*). We included the routing overhead to assess the performance of our caching framework under a reactive routing protocol.

Each row of Table 7.1 presents the performance of *NC* and *DC* under different popularity threshold θ. As expected, the energy consumption decreased when caching was enabled, and the effect of energy reduction was significant, especially under small θ (31% reduction when $\theta = 6$). (Alternatively, we could view *NC* as a special case of *DC*, where $\theta = \infty$.) However, the prefetching overhead (from 0 to 24.9 fragments)

and communication overhead (from 0 to 166.5 packets) introduced by caching was higher for smaller θ. The reason was that with small θ, the popularity threshold could be easily reached, leading to frequent cache agent selection, cache prefetch, and replacement activities. However, the frequent update indeed helped identify the most popular and active data fragments, which effectively reduced the energy consumption in future data access. As for the data retrieval rate, which was nearly 100% in all cases, the benefit from caching was not so obvious. This was majorly because we deployed the system in a small static network and the failure probability of nodes and links were low.

As for the communication overhead, it did not vary much with the change of θ. Counterintuitively, higher θ should generate more communication overhead as more nodes would reply to *freq*s and cause more *frep*s. However, at the same time when θ was higher, it was also more probable that the file requestors could go through fewer hops to retrieve the data. Therefore, the pros and cons brought by θ canceled out with each other and obscured the impacts from θ. A similar result was also observed in the change of routing overhead. It was also noticed that the overhead caused by routing protocol was not negligible (but acceptable) because Kernel-AODV was a reactive protocol, which discovered the routes on-demand by flooding control packets. We expected the routing overhead in a proactive routing protocol should be much smaller.

Based on these observations, there was an unavoidable tradeoff between the energy gain and overhead. Considering the fact that the control packets were much smaller (around 24 bytes in our case) than data fragments (which could be orders of megabytes), we might ignore their influence in our analyses. By adding up the energy consumption and the prefetching overhead, we found that the optimal θ was 8 in our network. This value conformed with the mathematical formulation derived

Table 7.1: Performance Metrics for Proof-of-Concept Evaluation

Alg	Energy Consumption	Retrieval Rate	Overhead for		
			Prefetch	Communication (Framework/Cache)	Route
$NC(\theta = \infty)$	292.5	95.83%	–	1926.5 / –	7930.5
$DC(\theta = 10)$	212.1	96.83%	10.3	1804.8 / 62.9	6610.4
$DC(\theta = 8)$	203.3	96.67%	16.0	1864.9 / 105.5	6972.1
$DC(\theta = 6)$	199.1	97.33%	24.9	2053.1 / 166.5	7240.1

in Eq. 5.1, which predicted 8.45 for the given parameters. This result proved that our formulation for estimating the optimal θ was correct.

8. CONCLUSIONS AND FUTURE WORK

This work investigates data caching in the k-out-of-n computing framework. A set of nodes are selected as cache agents for placing popular data fragments, such that the expected data retrieval energy can be minimized. Both the centralized and the distributed solutions are proposed and evaluated. The simulation results demonstrate that the transmission energy is reduced by up to 70%, and the data availability is improved by 13%, on base of no-caching framework. Comparing the Distributed Caching *DC* with the Centralized Caching *CC*, we show that while *DC* has no global information, its solution is close to the optimal one given by *CC*. If taking the prefetching overhead into account, *DC* may even outperforms *CC*. In the system evaluation, we observe a significant energy reduction (up to 31%) of *DC* over no-caching scenario even in a small network (8 nodes) with moderate request number (480 in total). The overhead introduced by data transmission and control packets in *DC* is less than 5%.

Continuing the work, we plan to evaluate the effect of routing protocols on *DC*. In particular, we are interested in experimenting *DC* under a proactive routing protocol such as OLSR. We envision that *DC* should perform even better under proactive protocols as the additional number of control packets generated by the routing protocol is much less. Although the evaluation has shown that our estimation of popularity threshold (θ) is correct, we are also working on an improved algorithm for determining θ in a hope to reduce the computation complexity. Lastly, we plan to integrate our distributed caching framework with MDFS into Hadoop architecture such that any MapReduce applications may be ported to mobile devices and benefit from the energy-efficient and reliable features of our framework.

REFERENCES

[1] Ivan Baev and Rajmohan Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *Proceedings of the 12th annual ACM-SIAM symposium on Discrete algorithms*, pages 661–670, 2001.

[2] Ivan Baev, Rajmohan Rajaraman, and Chaitanya Swamy. Approximation algorithms for data placement problems. *SIAM Journal on Computing*, 38(4):1411–1429, 2008.

[3] Jaroslaw Byrka. An optimal bifactor approximation algorithm for the metric uncapacitated facility location problem. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 29–43. Springer, Berlin, Germany, 2007.

[4] Chien-An Chen, Myounggyu Won, Radu Stoleru, and Geoffrey Xie. Energy-efficient fault-tolerant data storage & processing in dynamic networks. In *Proceedings of the 14th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 281–286, 2013.

[5] Chien-An Chen, Myounggyu Won, Radu Stoleru, and Geoffrey Xie. Resource allocation for energy efficient k-out-of-n system in mobile ad hoc networks. In *Proceedings of the 22nd International Conference on Computer Communications and Networks*, pages 1–9, 2013.

[6] Chi-Yin Chow, Hong Va Leong, and Alvin Chan. Group-based cooperative cache management for mobile clients in a mobile environment. In *Proceedings of the International Conference on Parallel Processing*, pages 83–90, 2004.

[7] Chi-Yin Chow, Hong Va Leong, and Alvin Chan. Peer-to-peer cooperative caching in mobile environments. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 528–533, 2004.

[8] David W Coit and Jia Chen Liu. System reliability optimization with k-out-of-n subsystems. *International Journal of Reliability, Quality and Safety Engineering*, 7(2):129–142, 2000.

[9] Alexandros G Dimakis, Vinod Prabhakaran, and Kannan Ramchandran. Decentralized erasure codes for distributed networked storage. *Networking, IEEE/ACM Transactions on*, 14(SI):2809–2816, 2006.

[10] Alexandros G Dimakis and Kannan Ramchandran. Network coding for distributed storage in wireless networks. In *Networked Sensing Information and Control*, pages 115–134. Springer, New York, NY, 2008.

[11] Alexandros G Dimakis, Kannan Ramchandran, Yunnan Wu, and Changho Suh. A survey on network codes for distributed storage. In *Proceedings of the IEEE*, volume 99, pages 476–489, 2011.

[12] Lawrence W Dowdy and Derrell V Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, 1982.

[13] Yu Du, Sandeep KS Gupta, and Georgios Varsamopoulos. Improving on-demand data access efficiency in manets with cooperative caching. *Ad Hoc Networks*, 7(3):579–598, 2009.

[14] Sandra G Dykes and Kay A Robbins. A viability analysis of cooperative proxy caching. In *Proceedings of the 20th Annual Joint Conference on Computer Communications*, volume 3, pages 1205–1214, 2001.

[15] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: A scalable wide-area web cache sharing protocol. *Networking, IEEE/ACM Transactions on*, 8(3):254–265, 1998.

[16] Takahiro Hara. Quantifying impact of mobility on data availability in mobile ad hoc networks. *Mobile Computing, IEEE Transactions on*, 9(2):241–258, 2010.

[17] Xiaoyan Hong, Mario Gerla, Guangyu Pei, and Ching-Chuan Chiang. A group mobility model for ad hoc wireless networks. In *Proceedings of the 2nd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 53–60, 1999.

[18] Scott Huchton, Geoffrey Xie, and Robert Beverly. Building and evaluating a k-resilient mobile distributed file system resistant to device compromise. In *Proceedings of the Military Communications Conference*, pages 1315–1320, 2011.

[19] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 213–222, 2002.

[20] Shudong Jin and Limin Wang. Content and service replication strategies in multi-hop wireless mesh networks. In *Proceedings of the 8th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 79–86, 2005.

[21] Jzsef Kadlecsik, Harald Welte, James Morris, Marc Boucher, and Rusty Russell. The netfilter/iptables project. `http://www.netfilter.org/`, 2004.

[22] Luke Klein-Berndt. Kernel aodv from national institute of standards and technology (NIST). `http://w3.antd.nist.gov/wctg/aodv_kernel/`, 2002.

[23] Yadi Ma, Thyaga Nandagopal, Krishna PN Puttaswamy, and Suman Banerjee. An ensemble of replication and erasure codes for cloud file systems. In *Proceedings of the IEEE International Conference on Computer Communications*, pages 1276–1284, 2013.

[24] Cherukuri Rajabhushanam and Ayyaswamy Kathirvel. Survey of wireless manet application in battlefield operations. *International Journal of Advanced Computer Science and Applications*, 2(1):50–58, 2011.

[25] George M Stephen, Zhou Wei, Chenji Harshavardhan, Myounggyu Won, Yong Oh Lee, Andria Pazarloglou, Radu Stoleru, and Prabir Barooah. Distressnet: a wireless adhoc and sensor network architecture for situation management in disaster response. *Communications Magazine, IEEE*, 48(3):1–9, 2010.

[26] Bin Tang, Himanshu Gupta, and Samir R Das. Benefit-based data caching in ad hoc networks. *Mobile Computing, IEEE Transactions on*, 7(3):289–304, 2008.

[27] Liangzhong Yin and Guohong Cao. Supporting cooperative caching in ad hoc networks. *Mobile Computing, IEEE Transactions on*, 5(1):77–89, 2006.