ABSTRACT

STORAGE SYSTEMS FOR MOBILE-CLOUD APPLICATIONS

by
Nafize R. Paiker

Mobile devices have become the major computing platform in todays world. However, some applications on mobile devices still suffer from insufficient computing and energy resources. A key solution is to offload resource-demanding computing tasks from mobile devices to the cloud. This leads to a scenario where computing tasks in the same application run concurrently on both the mobile device and the cloud.

This dissertation aims to ensure that the tasks in a mobile application that employs offloading can access and share files concurrently on the mobile and the cloud in a manner that is efficient, consistent, and transparent to locations. Existing distributed file systems and network file systems do not satisfy these requirements. Furthermore, current offloading platforms either do not support efficient file access for offloaded tasks or do not offload tasks with file accesses.

The first part of the dissertation addresses this issue by designing and implementing an application-level file system named *Overlay File System* (OFS). OFS assumes a cloud surrogate is paired with each mobile device for task and storage offloading. To achieve high efficiency, OFS maintains and buffers local copies of data sets on both the surrogate and the mobile device. OFS ensures consistency and guarantees that all the reads get the latest data. To effectively reduce the network traffic and the execution delay, OFS uses a delayed-update mechanism, which combines write-invalidate and write-update policies. To guarantee location transparency, OFS creates a unified view of file data.

The research tests OFS on Android OS with a real mobile application and real mobile user traces. Extensive experiments show that OFS can effectively support consistent file accesses from computation tasks, no matter where they run. In addition, OFS can effectively reduce both file access latency and network traffic incurred by file accesses.

While OFS allows offloaded tasks to access the required files in a consistent and transparent manner, file accesses by offloaded tasks can be further improved. Instead of

retrieving the required files from its associated mobile device, a surrogate can discover and retrieve identical or similar file(s) from the surrogates belonging to other users to meet its needs. This is based on two observations: 1) multiple users have the same or similar files, e.g., shared files or images/videos of same object; 2) the need for a certain file content in mobile apps can usually be described by context features of the content, e.g., location, objects in an image, etc.; thus, any file with the required context features can be used to satisfy the need. Since files may be retrieved from surrogates, this solution improves latency and saves wireless bandwidth and power on mobile devices.

The second part of the dissertation proposes and develops a *Context-Aware File Discovery Service* (CAFDS) that implements the idea described above. CAFDS uses a self-organizing map and k-means clustering to classify files into file groups based on file contexts. It then uses an enhanced decision tree to locate and retrieve files based on the file contexts defined by apps. To support diverse file discovery demands from various mobile apps, CAFDS allows apps to add new file contexts and to update existing file contexts dynamically, without affecting the discovery process.

To evaluate the effectiveness of CAFDS, the research has implemented a prototype on Android and Linux. The performance of CAFDS was tested against Chord, a DHT based lookup scheme, and SPOON, a P2P file sharing system. The experiments show that CAFDS provides lower end-to-end latency for file search than Chord and SPOON, while providing similar scalability to Chord.

STORAGE SYSTEMS FOR MOBILE-CLOUD APPLICATIONS

by
Nafize R. Paiker

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

December 2018

APPROVAL PAGE

STORAGE SYSTEMS FOR MOBILE-CLOUD APPLICATIONS

Nafize R. Paiker

Cristian Borcea, PhD, Dissertation Co-Advisor — Date
Professor, Computer Science, New Jersey Institute of Technology

---

Xiaoning Ding, PhD, Dissertation Co-Advisor — Date
Associate Professor, Computer Science, New Jersey Institute of Technology

---

Narain Gehani, PhD, Committee Member — Date
Professor Emeritus, Computer Science, New Jersey Institute of Technology

---

Reza Curtmola, PhD, Committee Member — Date
Associate Professor, Computer Science, New Jersey Institute of Technology

---

Yifan Zhang , PhD, Committee Member — Date
Assistant Professor, Computer Science, Binghamton University

BIOGRAPHICAL SKETCH

Author: Nafize R. Paiker

Degree: Doctor of Philosophy

Date: December 2018

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2018
- Bachelor of Science in Computer Science and Engineering,
  Military Institute of Science and Technology, Dhaka, Bangladesh, 2008

Major: Computer Science

Presentations and Publications:

N. R. Paiker, X. Ding, R. Curtmola, and C. Borcea, "CAFDS: context-aware file discovery system for distributed mobile-cloud environments", [in preparation]

D. Scotece, N. R. Paiker, L. Foschini, P. Bellavista, X. Ding, and C. Borcea, "MEFS: mobile edge file system for edge-assisted mobile apps", in *IEEE International Conference on Pervasive Computing and Communications (PerCom)* [Under submission], 2019

N. R. Paiker, X. Ding, R. Curtmola, and C. Borcea, "Context-aware file discovery system for distributed mobile-cloud apps", in *10th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2018

H. Debnath, M. A. Khan, N. R. Paiker, N. Gehani, X. Ding, R. Curtmola, and C. Borcea, "The moitree middleware for distributed mobile cloud computing: design, implementation, and evaluation", *Journal of Systems and Software* [under submission]

N. R. Paiker, J. Shan, C. Borcea, N. Gehani, R. Curtmola, and X. Ding, "Design and implementation of an overlay file system for cloud-assisted mobile apps", *IEEE Transactions on Cloud Computing (Early Access), doi: 10.1109/TCC.2017.2763158*

J. Shan, N. R. Paiker, X. Ding, N. Gehani, R. Curtmola, and C. Borcea, "An overlay file system for cloud-assisted mobile applications", in *32nd International Conference on Massive Storage Systems and Technology (MSST)*, 2016

P. Neog, H. Debnath, J. Shan, N. Paiker, N. Gehani, R. Curtmola, X. Ding, and C. Borcea, "FaceDate: a mobile cloud computing app for people matching", in *7th International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications (Mobilware)*, 2016

M. A. Khan, H. Debnath, N. R. Paiker, N. Gehani, X. Ding, R. Curtmola, and C. Borcea, "Moitree: A middleware for cloud-assisted mobile distributed apps", in *4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, 2016

N. C. Sheel, M. Ameer Ali, and N. R. Paiker, "Robust customer satisfaction model using QFD", *International Journal of Product and Quality Management*, vol. 6, no. 1, pp. 112-136, 2009

Z. Karim, N. R. Paiker, M. A. Ali, G. Sorwar, M. M. Islam, "Pattern based object segmentation using split and merge", in *IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, 2009

A. B. M. Faruquzzaman, N. R. Paiker, J. Arafat, M. A. Ali, and Golam Sorwar, "Robust object segmentation using split-and-merge", *International Journal of Signal and Imaging Systems Engineering*, vol. 2, no. 1-2, pp. 70-80, 2009

A. B. M. Faruquzzaman, N. R. Paiker, J. Arafat, Z. Karim and M. A. Ali, "Object segmentation based on split and merge algorithm", in *TENCON 2008 - IEEE Region 10 Conference*, 2008

A. B. M. Faruquzzaman, N. R. Paiker, J. Arafat, and M. A. Ali, "A survey report on image segmentation based on split and merge algorithm", in *IETECH Journal of Advanced Computations*, vol. 2, no. 2,pp. 86-101, 2008

A. B. M. Faruquzzaman, N. R. Paiker, J. Arafat, M. A. Ali and G. Sorwar, "Literature on image segmentation based on splitandmerge techniques", in *5th International Conference on Information Technology and Applications (ICITA)*, 2008

I dedicate this thesis to my parents: Golam Rabbani Paiker and Syeda Nasim Paiker, my brother: Nayem Paiker, and both of my grandparents: Late Abdul Jobber Paiker and Rawshanara Begum Paiker, Late Syed Badir Uddin Ahmed and Late Anowara Begum who have always kept faith in my abilities and helped me to dream bigger.

ACKNOWLEDGMENT

I would like to convey my wholehearted gratitude to my dissertation co-advisor, Dr. Cristian Borcea and my dissertation co-advisor Dr. Xiaoning Ding for their immense support throughout my Ph.D. study and research. It would be impossible to describe the patience they have shown to improve my scientific thinking and technical writing. The inspirational discussion with them always motivated me during my difficult and depressing period of research. Without their persistent guidance and support, it would not be possible to complete this dissertation. I consider myself truly fortunate to have them as my advisors.

I am also indebted to Dr. Reza Curtmola, Dr. Narain Gehani for helping me with valuable suggestions and feedback throughout my research. I cannot thank them enough for the patience they have shown to improve the writing and presentation of my research papers. I would like to thank Dr. Yifan Zhang from Binghamton University for being a part of my dissertation committee.

I would like to thank my fellow lab-mates Hillol Debnath, Jianchen Shan, Mohammad Ashraf Khan, Nora Almalki, Pradyumna Neog, Giacomo Gezzi and Domenico Scotece for their support during my research. The constructive discussions with them have always helped me to clarify confusion and improve my ideas. I thank them also for sharing the frustration with me whenever I had a paper rejected.

Thanks are also due to the (NSF) National Science Foundation (under grants CNS 1409523, CNS 1054754, SHF 1617749, DGE 1565478 and DUE 1241976), the National Security Agency (NSA) (under Grant H98230-15-1-0274), the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) (under Contract No. A8650-15-C-7521) for their financial support that I otherwise would not have been able to develop my scientific discoveries.

Above all, I could not possibly arrive this stage of my life without the dedication, love, and support my parents and my brother has provided me. I am forever indebted to my father, Golam Rabbani Paiker, my mother, Syeda Nasim Paiker and my brother,

Nayem Paiker for all the sacrifices they have made to provide me the best education and support. I would like to thank my grandparents Late Abdul Jobber Paiker and Rawshanara Begum Paiker, Late Syed Badir Uddin Ahmed and Late Anowara Begum. Their faith inspired me during many frustrating and hopeless situations.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Background and Problem Statement

Rapid proliferation of smartphones has profound effects on people's daily life. It is predicted that in United States alone the number of smartphone users will reach 270 million by 2022 [1]. Mobile devices, such as smart phones and tablets, have become major personal computing devices. These devices are gradually replacing the more traditional sources of computation, like desktop. In 2015, globally the total number of users of smartphones were more than 1.8 billion which is approximately 200 million more than the desktop users [2]. As a result, people are spending more and more time on mobile devices.

Despite their popularity, mobile devices have limited computing resources (e.g., CPU power, energy supply, memory space, network bandwidth etc) due to their compact size and mobility. To get desired performance and energy conservation, various systems have been designed to allow mobile apps to use cloud resources (e.g., public cloud, personal cloud, or cloudlet). This is done by offloading their resource-demanding tasks to the cloud in the form of threads, objects, or procedures [3–9]. For example, a mobile app may record video clips on a mobile device, analyze and augment them in the cloud, and then play back the video clips on the mobile device. This whole process requires decomposing the tasks into units of computation (e.g., methods, threads, and objects) and distributing the related memory states to the cloud when the tasks are offloaded to the cloud or migrated back to the mobile. This leads to a scenario where the computation tasks in the same mobile app can be offloaded to the cloud and/or run concurrently on both the mobile device and the cloud. These tasks work collaboratively and may need to save, read, and overwrite files simultaneously on both the mobile device and the cloud.

The decomposition and distribution of tasks and their memory states have been studied extensively, and a few programming models, along with the supporting

middleware and system infrastructure, have been developed, e.g., Avatar [8,10], MAUI [3], ThinkAir [4], CloneCloud [5], Sapphire [7], and COMET [6]. However, supporting efficient file access, especially file sharing between the tasks in the same mobile app running on both the mobile device and the cloud remains a challenging issue and has received little attention. Due to this issue, systems such as MAUI and COMET cannot offload tasks in mobile apps if the tasks need to access files.

With the maturity of task offloading technology and systems, millions of mobile devices may offload computation to surrogates running in the cloud. Apps on these mobile devices may access and share data in diverse and complex manners. For example, some apps run on the devices of the same user or collaborating users. They may need to share and access same/similar files on different devices. Some personal files may have been exchanged between family members and friends. When an offloaded task needs to access a file, instead of retrieving the required file content from its associated mobile device, a surrogate can discover and retrieve same/similar content from other surrogates in the cloud to meet its need. This improves latency and saves wireless bandwidth and power on mobile devices. However, different apps can search files using different criteria, and same set of files may fit in different categories. Also, the search procedure must be efficient to benefit from such system. It is a challenging task to locate and retrieve files in a fast and efficient manner such that offloaded tasks can be executed more efficiently.

The dissertation addresses these problems by designing and developing an overlay file system targeting the cloud assisted mobile apps on each individual mobile device (Section 1.2) and a context-aware file discovery service for efficiently search and retrieve the required files (Section 1.3).

1.2 An Overlay File System for Cloud-Assisted Mobile Apps

As explained earlier, majority of the offloading frameworks cannot offload tasks with file access to the cloud. Existing file systems are not effective in handling remote file access for the offloaded tasks of mobile apps. Thus, they seriously limit the capability of mobile systems to freely offload tasks to the cloud. Network file systems and distributed file

systems, such as NFS [11] and cloud storage, like Dropbox [12], only support remote file access from the platforms where their client software is properly set up and configured. However, setting up and configuring the client software (in a network/distributed file systems) usually requires root privilege, which the mobile user may not have. It also needs the credentials of the user to access the file server, which the user may not be willing to release to the cloud. Moreover, if a task is accessing an open file saved in a network/distributed file system, it must reopen the file after the task is offloaded in order to continue accessing the file. This requires that mobile apps must be aware of task offloading, which makes programming cumbersome and error-prone.

Another issue with existing network file systems and distributed file systems is that they cannot satisfy the consistency requirements of cloud-assisted mobile apps at low overhead. To guarantee correct execution, tasks concurrently running on the cloud and the mobile device often require strong consistency (i.e., no stale data returned to the tasks). Most network/distributed file systems, especially those designed for mobile devices (e.g., Coda [13, 14]), cannot guarantee such consistency. Some systems even rely on users to manually resolve inconsistencies. The inconsistencies caused by such systems will lead to incorrect results or application crashes. Some other file systems (e.g., NFS) support strong consistency but at high costs of network traffic and energy on the mobiles, and thus are not practical for mobile apps.

Moreover, with majority of the conventional distributed file systems, file system clients maintain the states of opened files and file operations at the system level. These states include not only the data structures for managing the files and file operations but also the file contents buffered in memory (e.g., new data generated by a task). When a task is rescheduled to another device, the related states must also be moved with the task. However, it is challenging, if not impossible, to separate such states for individual tasks on one device and merge the states on the new location. These states the limit of the mobility of tasks.

To address these problems, we propose an application-level file system named *Overlay File System* (OFS). OFS supports remote file access by providing the tasks on

the mobile device and the cloud with an efficient, consistent, and transparent view of data that is accessible as local storage. It supports task offloading in the form of threads, objects, or procedures. OFS manages file access and file sharing in a mobile app. It effectively hides the boundary between the mobile device and the cloud, and provides a unified environment for the tasks in the mobile app, such that the tasks can migrate freely between the mobile device and the cloud. OFS ensures that all tasks whether on the mobile or offloaded to the cloud read the latest data in the file. OFS uses an adaptive method named delayed-update, which combines the conventional write-invalidate and write-update policies, to reduce file access latency and network traffic overhead, while ensuring strong consistency. To guarantee location transparency, OFS creates a unified view of the data that is independent of location and is accessible as local storage.

1.3 A Context-Aware File Discovery Services for Distributed Mobile-Cloud Apps

To fully exploit mobile data in distributed mobile apps, two problems must be solved. One is how to quickly locate and obtain the required data. The other is how to efficiently process the data. The latter has been effectively addressed with the recent advancements in mobile-cloud computing, which allow distributed mobile apps to offload costly computation and networking to the cloud in order to reduce response time and energy consumption on mobile devices [3, 5–7, 10, 15–24]. However, the former remains largely unsolved due to three issues that impact the effectiveness of the distributed mobile-cloud (DMC) apps and the efficiency of their executions.

First, to find the required files, a DMC app can only examine the files on the devices of the participating users (i.e., the devices that it runs on). Non-participating users may have the files that the app needs and may be willing to share them, but the app cannot locate or access these files. This significantly reduces the number of files available to the app, and in turn lowers the quality of the results and/or user experience.

Second, each DMC app must search and examine files independently. Different apps may search for files using similar criteria, and the same set of files may fit the needs of

different apps. For example, many apps (e.g., those handling disaster situations or law enforcement) need photos taken at the location and within the time window of specific events. It is inefficient to implement the searching code in each app and run the code repeatedly for different apps.

Third, DMC apps cannot locate the files with low latency and low overhead. A file may have multiple copies distributed at different locations with different access latency and overhead. For example, a photo is copied among the mobile devices of a group of friends, and one of them uploads it to the cloud. Retrieving the photo from the cloud incurs lower latency than getting it from mobile devices.

Conventional solutions, such as search engines [25, 26] and file searching functionalities provided in storage systems and peer-to-peer systems [27–38] do not solve well these three issues for DMC apps. Search engines mainly focus on searching file content with keywords instead of more general context and content features (e.g., location and time of file generation, image files containing faces) as required by DMC apps. File searching functionalities in storage systems are usually tightly coupled with the system design and rely on a global file system space. DMC apps, on the other hand, need to access data from many independent users. Peer-to-peer systems offer distributed file searching functionalities. However, they introduce large latency due to their multi-hop networking nature.

This dissertation presents Context-Aware File Discovery Service (CAFDS) to fundamentally address these issues. CAFDS is implemented as a middleware that runs on participating mobile devices and in the cloud. Its main component is a metadata server that runs in the cloud and indexes the files shared by users based on three types of searching criteria: file context, file content, and traditional file metadata. CAFDS provides several benefits to DMC apps: 1) It reduces the programming effort to write file searching code in different apps. 2) It can increase the searching scope and provide the apps with more data. 3) When multiple files with the same content are available, it returns the file with the lowest access latency.

1.4 Contributions of Dissertation

1.4.1 An Overlay File System for Cloud-Assisted Mobile Apps

In order to support offloading of the tasks containing file I/O to the cloud, an overlay file-system, OFS has been proposed. Different mobile apps require different level of consistency support. OFS implements *delayed-update* consistency policy to support strong consistency required by different apps. For the cases where strong consistency is not strictly necessary, a relaxation feature has also been introduced. The programmers can use this feature to support various degrees of relaxed consistency. To increase compatibility with different offloading systems, OFS is decoupled from the offloading frameworks and has a narrow interface with them. The purpose of this design is to keep the interaction between OFS and the offloading frameworks minimal. As most mobile devices are not rooted, and applications do not have root privilege, OFS is implemented on the userspace. This research studies several implementation techniques and builds an OFS prototype on Android OS based on these studies.

The dissertation has also implemented a real app, named *photo enhancement app*. This app and real mobile user traces have been used to test the functionalities and performance of OFS. The experimental results show that the delayed-update policy used in OFS can effectively reduce file access latency by up to 21% relative to commonly used write-update and write-invalidate consistency policies while reducing the network traffic incurred by file accesses by up to 67% than that with the write-update policy.

1.4.2 A Context-Aware File Discovery Services for Distributed Mobile-Cloud Apps

Context-Aware File Discovery Service (CAFDS) allows distributed mobile-cloud applications (DMCs) to find and access files of interest shared by collaborating users. CAFDS enables programmers to search for files defined by context and content features, such as location, creation time, or the presence of certain object types within an image file. The contributions of this paper are summarized as follows. (a) We designed CAFDS as a system solution to effectively address the file discovery problem for distributed

mobile-cloud apps; (b) we employed a modified decision tree for fast and accurate file discovery; and (c) we implemented CAFDS in Android and Linux, and tested its performance by replaying mobile file traces. Our experiments show that CAFDS outperforms peer-to-peer file systems such as Chord [39] and SPOON [34]

1.5 Contributors to This Dissertation

OFS was designed collaboratively with my colleague Jianchen Shan. My contribution is the design and implementation of OFS, photo-enhancement app and an app for testing the performance of OFS on real user traces and photo-enhancement app using actual implementation. Other than collaborating to the design of OFS, Jianchen also emulated both OFS and NFS to compare their performance. To understand my contribution, the whole platform is presented in this dissertation, including Jianchen's part.

1.6 Structure of Dissertation

The rest of the dissertation is organized as follows. Chapter 2 discusses about related works. Chapter 3 presents design and implementation of OFS, the overlay file system for single-user offloading platform. Chapter 4 discusses Context-Aware File Discovery Service (CAFDS) for searching and retrieving files using various file features and contexts. Finally, the dissertation concludes in Chapter 5.

CHAPTER 2

RELATED WORK

2.1 File I/O in Existing Cloud Offloading Systems

This section first discusses the offloading frameworks that offloads computation from the mobile device of a single user to the cloud and their inability to provide efficient file access support for the offloaded tasks. Then, it explores the offloading platforms that enable multiple users to offload computation to the cloud and how they differ from the systems that only allows a single user to offload computation. Finally, it discusses the lack of efficient support file sharing and file access among offloaded tasks by different users in the multi-user settings. In order to distinguish between the offloading platforms of these two categories, we will refer to the systems that allow offloading from a single user to the cloud as *single-user* offloading platforms and those allow multiple users to offload computation to the cloud as *multi-user* offloading platforms.

2.1.1 Single-User Offloading Systems

A few systems that offload computation from the mobile of a single user to the cloud have been developed [3–7, 9, 40, 41]. However, none of them is able to handle the file I/O of offloaded tasks efficiently, if at all. Some of them, such as MAUI [3], and ThinkAir [4] assume that the to-be-accessed files are already available in the cloud when tasks are migrated. They do not have mechanisms to support consistent remote file accesses. On the other hand, some systems like, offloading tool for Android applications based on autonomous method selection proposed by Zanni *et al.* [41], and ULOOF [9] do not offload methods with file I/O. It should be noted that all of these offloading systems, like OFS, work on user-level.

CloneCloud [5] migrates threads in application-level VMs. It supports access to local files. CloneCloud punches through the abstract machine to the process system call interface in order to access native resources. CloneCloud places all methods that share

the same native state in either mobile device or the VM. In other words, if more than one method accesses the same file, either all of them have to be offloaded or none of them can be offloaded. But accessing and updating the same file from both the mobile device and the cloud simultaneously is not supported.

COMET [6] provides distributed shared memory support for migrating threads between mobile devices and cloud. However, it does not support offloading threads that perform file operations.

Sapphire [7] is a distributed programming platform for developing and deploying apps spanning mobile devices and clouds. Tasks are distributed using Sapphire Objects (SO) that encapsulate both data and code. Sapphire SOs may access remote files with a simple RPC-based mechanism. But the design lacks transparency and efficiency. For example, SOs accessing files cannot move, and all the file accesses must go through network.

Just-in-time (JIT) provisioning in cloudlets [40] uses a synthesis server to help prepare virtual disks for the tasks offloaded to cloudlets. Since the files to be accessed by the tasks are included in the virtual disks, JIT provisioning and cloudlets can satisfy file I/O requests of offloaded tasks. This design is for VM-based task offloading, which usually incurs a high overhead. OFS targets offloading tasks in the context of threads, objects, or procedures.

2.1.2 Multi-User Offloading Systems

The previous subsection discusses several frameworks for offloading complex and resource consuming computation for a single user. A few systems [15–20,42,43] have been proposed that can offload computation in multi-user scenario where the users can share cloud resources. The single-user and multi-user offloading platforms differ from each other based on how the offloading decision is made and how the tasks are executed. In the first case, the offloading decision is solely dependent on the optimizing computation and resource usage of the mobile device of a single user. As there is only one user, there is no need to schedule the upload of the offloaded task to the cloud or schedule its execution in the cloud. In the

multi-user scenario, the offloading decision relies on factors like priority of computation based on estimated execution time, input size and bytes to be transferred, optimal use of shared communication channel and cloud resources. The offloading frameworks of this category often optimize one or more of these factors.

Some of the multi-user offloading frameworks [17, 18, 20] do not consider the tasks with file I/O for offloading, while others do not support efficient file access for offloaded tasks [42]. For example, the offloading service in Avatar platform [42], which supports offloading of tasks from a group of users, allows the users to offload a task containing file I/O. However, the files need to be available on the cloud before the task is offloaded. While Avatar platform supports file transfer from mobile to the cloud, it does not support offloading of tasks where the files are accessed simultaneously on both mobile and the cloud. It also does not provide a solution regarding how the common files between its users can be shared efficiently for supporting the offloaded tasks.

The partitioning framework proposed by Yang *et al.* [15] presents an application partitioning model where the applications can be portioned into smaller tasks each processing a portion of a data stream. In this model, several users can share a task or a piece of data. This data is usually collected by the sensors of the mobile devices. This model is suited for tasks where the data itself is not updated. MuSIC [16], on the other hand, proposes a model that executes location-time-workflow based on available cloud services (like music stream) and user services (like decoders, music players) in a 2-tier cloud (private and public cloud). Unlike the offloading platform proposed in [15], this system, however, does not address how to offload the tasks containing file I/O.

2.2 Distributed and Network File Systems

Various distributed and network file systems were developed for different purposes [11, 13, 14, 44–48]. Most distributed and network file systems (e.g., NFS [11], AFS [49], Coda [13, 14], and BlueFS [48]) are for users accessing their files from different devices or sharing files. Some of them (e.g., Coda and BlueFS) target mobile users and take into consideration the characteristics of mobile devices (e.g., limited resources and network

connection). OFS and CAOFS are designed mainly to support the file accesses and file sharing for the tasks offloaded to the cloud from mobile devices.

OFS and CAOFS differs from existing distributed and network file systems from the following perspectives. First, conventional distributed and network file systems usually require that the client software be installed and configured before they can access files, making them cumbersome to use in task-offloading scenarios. Both of the proposed systems, however, work at the application level and can be established on demand when a task is offloaded to the cloud. Second, unlike OFS and CAOFS, conventional distributed and network file systems do not provide support for tasks that have opened files at the time of offloading. Last but not least, the proposed systems support efficient and consistent file sharing between different devices. Unlike the traditional network/distributed file systems, CAOFS can support efficient file sharing among different users during task-offloading despite the lack of information regarding common files.

2.3 Consistency Policies

Different policies are adopted in distributed and network file systems to enforce consistency. For example, Coda [13, 14] supports disconnected operations, which allow users to update files when network is disconnected. This leads to consistency issues that need to be solved by users. BlueFS [48] cannot avoid conflicts either, and it requires users to manually resolve the conflicts. This is not practical for mobile apps that offload tasks to the cloud – any benefits in performance will be lost if the users are asked to help solve consistency issues through conflict resolution.

NFS [11] supports close-to-open consistency. To guarantee file consistency, applications need to use either file locks or shared reservations to avoid interleaving file sessions. This model does not fit task-offloading scenarios, where tasks running in parallel at the mobile and the cloud may need to update/read a file concurrently.

Mobile File System (MFS) [47] is a cache manager for adapting data accesses in collaborative applications to network variability when they access a distributed file system. MFS supports consistent accesses to shared files. But the consistency scheme is designed

to target network bandwidth variation and network latency is not a major concern. The scheme may cause high file I/O latency, which is not desirable in task-offloading scenarios.

Raindrop File System (RFS) [44] aims at mobile devices accessing files saved in cloud. It implements a client-centric management scheme, in which clients decide synchronization points to manage consistency. However, how to select appropriate synchronization points is a challenging and unsolved problem. When used in task-offloading scenarios, RFS increases the difficulty of programming and cannot guarantee the required file consistency.

Simba [45, 46] provides a reliable and consistent synchronization service for mobile devices. With Simba, mobile apps can always see a consistent view of their data, and the data can be stored locally on the mobile device, in the cloud, and/or on other mobile devices. In addition to calling Simba API to access/update data, it is also the app's responsibility to call Simba API to register data, synchronize updates, and resolve conflicts. OFS, on the other hand, does not require apps to handle these operations, and can be used when apps do not have offloading logic.

Data consistency has been intensively studied. On top of the consistency method-s/policies discussed above, a large number of other solutions have been proposed for various specific parallel and distributed system scenarios [50–55].

Earlier studies [50, 52] presented several distributed shared memory (DSM) implementations that implements that either implements sequential consistency or release consistency. These systems uses either write-update or write-invalidate mechanism for coherence. Munin DSM [51] introduces software-release consistency with multiple consistency protocols for the user to choose the best one. This model requires memory to be consistent at specific synchronization points and guarantee correctness within a request/release or lock/unlock pair. Because of the random nature of data access pattern by mobile users and the factors involved in task offloading (like energy cost, computational latency or privacy), individually these models are not perfect for all scenarios.

HCCM consistency mechanism [53] presents a two-layer consistency protocol for WAN. In this protocol, each LAN/domain maintains a domain server and there is a

central/main server. All read/write locking has to be approved by both servers for a host to write/read. This model can be adapted for our case where cloudlets can act as domain servers and a cloud can act as main server. The presence of domain server reduces the network overhead. However, all read/write locking requests have to approved by both domain server and main server. This can increase the cost of offloading and/or I/O latency which is not suitable for task offloading in mobile devices. The consistency protocol proposed by Bzoch *et al.* [54] is an adaptation of close-to-open consistency policy of NFS that optimizes the time for validation check performed the beginning of the close-to-open session. Another similar approach is presented in DCIM [55]. It is an invalidation based approach. In this approach, the client checks all cache items after a certain amount of time for validation and sends an update request to the server. Both approaches are invalidation based consistency model and very similar to NFS in terms of design. Due to the optimization of time to perform validation check, this model will have improved the hit ratio. However, as demonstrated section 3.5, both approaches will incur higher average I/O overhead compared to OFS.

The file systems proposed by this thesis target the scenario in which concurrent and collaborative tasks run both on the mobile device and in the cloud, and may access the same file(s) concurrently. We have not found other work providing a consistency solution similar to that provided by OFS.

2.4 File Discovery Services

Context-Aware File Discovery Service (CAFDS) addresses the issue of file discovery in distributed mobile-cloud (DMC) computing based on file features. This system tries to classify files into different group based on these features. In this setup, the files are generated and available in multiple mobile devices and VMs.

2.4.1 Classifying Mobile App Data

Classifying mobile app data is not a new idea [56–60]. Wang *et. al.* [56] and Donato *et. al.* [58] applies various statistical features extracted from data traffic and applies them

into various machine learning model to identify app that generated the traffic. System proposed by Mongkolluksamee *et. al.* [59] and METCS [60] on the other hand uses packet size distribution combined with communication pattern and application layer payload pattern to identify correct app. These works are mainly focused on identifying correct mobile app using various features derived from data traffic. While these systems can be adapted to identify files used by different apps, they do not consider any features that are defined by the app itself. As a result, these systems are unsuitable for our current context.

2.4.2 Traditional File Search Engines

Traditional file search engines like Google Files Go [25] and Apple Spotlight [26] or web search engines cannot be used because they usually locate file using simple features such as file name, keywords or tags. Also, they are not designed to serve as a discovery and retrieval service for DMC apps.

While systems [27–33] have been proposed to search files in distributed and large scale file systems, none of them are optimized for distributed processing on mobile-cloud platforms. Propeller [28] creates file indexes based on access sequences, and use them for search. VSFS [29] uses namespace-based queries to locate appropriate files. Glance [27] uses approximate processing of aggregation and top k-queries on a small file sample for file search. As file generation is highly dependent on user behavior and file features may be defined/modified by different apps, these systems are too restrictive for the requirements of our scenario.

2.4.3 File Searching Using Metadata

Systems such as Spyglass [31], CEFLS [33], and SmartStore [30] use metadata search for locating appropriate files. In our scenario, the definition of the metadata may be updated frequently by different apps. Therefore, it can take a considerable amount of time to update the existing metadata. Also, many of these systems are dependent on existing

file directories to optimize the file search, which may be difficult to implement in our distributed mobile-cloud environment.

2.4.4 File Searching in P2P File Systems

Many P2P file systems, where file search is possible, can also be applied to our scenario. Earlier P2P systems [39, 61–63] were usually implemented on a single structure like DHTs [39, 61] or structuring points in d-dimensional space [62]. Compared to CAFDS, their lookup schemes involve multiple network hops which cause an increase of the overall file access latency. To address this issue, newer systems [34–38] employ multi-layer P2P overlays. These systems divide users into interest groups, which can later be used for searching the files. While these systems support interest-based groups, they do not support complex app-defined features for search, and thus cannot be easily used in our scenario.

2.4.5 File Searching in Content-centric Networks

Various systems [64–68] uses interest/features driven from the content of the data for routing and in-network caching. In general, these systems create interest tables to store the interest of a piece of data [64–66]. This interest are often hash of the content [68], social relationship [69] or some other information like publisher, scope, user-defined label, etc [66]. These interests are then used for caching the data [65, 67, 68] or routing [64–66]. While routing implemented by these systems are often efficient and allow search a piece of data using some interest value, interests defined by these systems are simple in nature and do not allow complex context with multiple features. Also, these systems do not support dynamic changes in the interest.

2.5 Chapter Summary

In this chapter, we have discussed about existing single-user and multi-user offloading frameworks and their incapability of offload tasks with file I/O to the cloud. We also discussed how they are unable to support efficient file sharing among different users and

different devices. Next, we have discussed existing network and distributed file systems, and consistency policies. We also analyzed why they cannot be used appropriately with single-user offloading frameworks. Finally, explored various file discovery services like traditional file search engines, metadata search, file searching in P2P file systems and content-centric networks.

CHAPTER 3

AN OVERLAY FILE SYSTEM FOR CLOUD-ASSISTED MOBILE APPS

Compared to conventional network/distributed file systems, Overlay File System (OFS) has several advantages for running cloud-assisted mobile apps. First, the strong consistency model ensures the correct execution of computation tasks distributed across the mobile device and the cloud. Second, tasks accessing files can be moved freely across different devices. This is because the states of files and file operations are in the app's user space, and thus can be duplicated and moved with the tasks to new locations. Third, at the application-level, it simplifies application development and system management. For example, with OFS, root privilege is not required to set up the system and there is no need to save the to-be-accessed files into a network/distributed file system before the app runs, and special attention for handling different path names in the programs incurred by different mounting points on different devices is not required either. Programmers do not have to worry about whether a task is running on the mobile or has been offloaded to the cloud.

The special features of mobile systems and the requirement to run OFS at the application level present a few implementation challenges. For example, most mobile devices are not rooted, and applications do not have root privilege. In addition, mobile OSs (e.g., Android) may kill processes and reclaim their memory spaces, making it challenging to maintain OFS system states at the application level. Focusing on these challenges, the chapter has studied several implementation techniques and built an OFS prototype on Android OS. The prototype uses a set of "sticky" application services to implement major OFS functionalities. An app uses the code injected by OFS with AspectJ [70] to get OFS services.

The dissertation has also implemented a real app, named *photo enhancement app*, and has used this app and real mobile user traces to test the functionalities and performance of OFS. Our case study with the photo enhancement app shows that OFS

can effectively support consistent file accesses from computation tasks, no matter whether they run on a mobile device or has been offloaded to the cloud, and that existing cloud storage systems, including DropBox and Google Drive, cannot provide such support.

We used a Nexus 6 phone as mobile device and an Android 6 x86 VM running on OpenStack as cloud component in our experiments. We compared the performance of OFS with two well-known consistency policies: write-invalidate and write-update in similar setup. Our experiments on photo enhancement app shows, OFS can improve average I/O latency by approximately 8% and 12% compared to write-invalidate and write-update policies. The experiments on user traces achieves 14% and 21% I/O latency improvement on average compared to write-invalidate and write-update policies. It also achieves 67% improvement on network overhead compared to write-update policy. OFS incurs 8% higher network overhead on average compared to write-invalidate which is the lower limit for network overhead.

To the best of our knowledge, this is the first work that provides a system solution to support efficient and transparent file access in cloud-assisted mobile apps. We make the following contributions. First, we determine the requirements for a file system to effectively support offloading tasks to the cloud. Second, we design and implement OFS as a solution to meet these requirements. Third, we use a real app and user traces to show that OFS can effectively support task offloading and efficient execution of offloaded tasks by significantly decreasing both file access latency and network traffic incurred by file accesses.

The rest of the chapter is organized as follows. Section 3.1 outlines the background and motivation for designing OFS. Sections 3.2 and 3.3 present the OFS design and implementation details. A case study with a real app is presented in Section 3.4. The evaluation of OFS is presented in Section 3.5. The chapter summary is presented in Section 3.6.

3.1 Background and Motivation

This section introduces first several approaches to offload tasks in mobile apps to the cloud. Then, it presents a few example apps to illustrate the demand for consistent and transparent file access and sharing. Finally, it summarizes the requirements on file systems for cloud-assisted mobile apps, which underpin the design of OFS.

3.1.1 Approaches to Offload Computation to the Cloud

To effectively leverage cloud-assistance, a system needs to support task migration between the mobiles and the cloud. To simplify programming, the tasks should not require modifications, and the program itself does not need to implement the offloading logic. Instead, the system software dynamically schedules and runs unmodified computation tasks of an app on the mobile device and the cloud.

To make scheduling decisions, the system uses a certain cost function, which balances the cost and the benefit of offloading a task to the cloud based on factors such as the workload of the task, dependencies on software and hardware resources, the state of the resources on the mobile device, network performance, and the overhead of transferring the task. To support the execution of unmodified tasks in the cloud, the system should recreate the execution contexts required by the tasks in the cloud, such as system support, supporting libraries, code, and all the required data sets. While system support, library, and sometimes application code can be pre-deployed, the data sets are usually transferred dynamically with the tasks or based on demand for a few reasons. For example, some data sets are generated/updated dynamically, and apps may use different data sets in different executions.

A few different methods can migrate tasks, including their code and the required in-memory data sets. Some systems (e.g., Sapphire and Avatar) encapsulate and transfer the code and memory state of a task (e.g., data in heaps) in an object. Other systems (e.g., COMET) offload tasks in the form of threads. They use distributed shared memory (DSM) and transfer the memory state on-demand when it is accessed remotely by the threads. A computation task may also be offloaded by making remote procedure calls

(RPC) to the cloud. Migrating threads offers a few advantages over RPC, especially when distributed shared memory support is provided [6]. For example, a thread may be migrated at any time during the execution of the app, while with RPC only whole procedures can be offloaded.

Cloud-assistance can also be implemented with a VM-based approach (e.g., Cloudlet [40]). Since a VM is a complete running environment for an app, from memory state to storage, offloading tasks to the cloud can be achieved by migrating the VM containing the tasks. However, compared to moving a thread/object/procedure, migrating a VM inevitably incurs much higher overhead and sacrifices flexibility, since a VM has much more information (e.g., OS kernel state, buffered data, etc.) than individual tasks and all the tasks in a VM must be moved together.

In this chapter, we target the approaches that offload computation tasks in the form of objects, threads, or procedures. The cost function used by the system to balance the overhead and the benefit of task offloading is beyond the scope of this chapter. At the current stage, we assume that there is a cost function that comprehensively considers the overhead of both transferring in-memory data and accessing files remotely for making task offloading decisions.

The collaborative tasks in an app run concurrently at the cloud and the mobile device, and they often need to access their data sets saved in files. The needs cannot be satisfied by transferring the files to be accessed by a task before offloading the task to the cloud. It is not easy to identify all these files, especially in cases when a task may need to access new files that are generated after it starts. Thus, not all the files can be transferred a priori. More importantly, tasks on the mobile device and the cloud may update and read the same set of files concurrently. This method cannot guarantee the consistency of the shared files, and inconsistency may lead to incorrect results or application crashes. For these reasons, systems supporting task offloading (such as COMET and MAUI) usually cannot migrate tasks if they need to access files. [1]

[1]The DSM model implemented in COMET can be extended to help accessing memory-mapped files. However, the files must be opened and memory-mapped on the mobile device before tasks are offloaded to the cloud. Opening a file and establishing memory mapping in the cloud require

This problem can be mitigated by using networked/distributed file systems (e.g., NFS) or cloud storage platforms (e.g., DropBox). However, existing networked/distributed file systems and cloud storage systems are not designed for collaborative tasks on mobile devices. They are designed for scenarios, in which a file is opened, modified, and closed on one device, and then is opened and accessed somewhere else. Concurrent reads and writes on different devices to the same file are not designed or implemented [71]. Thus, their implementations cannot support consistent file access and file sharing with low overhead.

3.1.2 Motivating Examples

With the growth in the number of mobile devices, the amount of data (e.g., multimedia data) generated and operated by mobile apps also increases. Many of these operations (e.g., image/video recognition and augmentation) are too resource consuming to run on mobile devices and require the help of the cloud for optimized performance [72]. Meanwhile, most apps interact with users. Their interactive tasks must run on mobile devices for desirable user experience and reduced overhead. Some apps rely on the hardware resources (e.g., sensors) on mobile devices, and the related tasks must also be executed on mobile devices. This leads to scenarios in which an app has tasks on the mobile device and tasks in the cloud working collaboratively.

For example, enhanced camera apps can take photos or video clips, use the cloud to analyze (e.g., recognizing the people and landmarks in the files and tagging them properly) and improve them (e.g., removing red eyes and reducing blurring), and play back the improved photos or video clips on mobile devices. In such an app, a thread taking the photos/videos needs to save them. A processing thread may be migrated to the cloud when it is about to process some photos/videos and the system estimates that the benefit of offloading the tasks (e.g., better user experience with lower response time) exceeds the overhead (e.g., the cost to transfer the thread and the photos/videos). The system may migrate the thread back when the thread needs to process some other photos

---

additional system support beyond the DSM mechanism. The DSM model cannot facilitate file accessing through a standard file I/O interface.

and it is not cost-effective to transfer these photos to the cloud. Thus, the thread may read the saved photos/videos from the cloud or the mobile device, and it generates improved photos/videos where it runs. The generated photos/videos are then read out by a thread on the mobile device for playback. At the same time, the processing thread and other threads in the app may form a pipeline and run concurrently. For example, the processing thread running on the cloud first sends back an improved photo/video segment. When the thread on mobile device plays back this photo/video segment, the processing thread may improve another photo/video segment in the cloud concurrently. Thus, the photos/videos must be well-managed to satisfy the concurrent accesses from both the mobile device and the cloud.

In another example, a video surveillance app may keep recording videos, which are analyzed in the cloud in real time to promptly detect, recognize, and tag moving objects. Other interactive apps (e.g., doodle clipboard apps and games) need to recognize and understand (in the cloud) complex user inputs collected on mobile device (e.g., doodles drawn by the users, gesture and eye movements of the users), and react to these inputs. In all these apps, a file system that supports the tasks running on the mobile device and the cloud to access and share the photos/videos/doodles and other data saved in files is critical to effectively leverage the computing power of the cloud.

3.1.3 Requirements on File System Design

To support remote file access and file sharing among the distributed tasks of cloud-assisted mobile apps, a file system should be able to locate and transfer data, and to manage data sharing. To accommodate features of mobile apps and hardware characteristics of mobile devices, a file system must satisfy the following requirements:

- **Location transparency**: The file system should be able to provide an app with access to remote files as though they were local, and should be able to maintain file sessions during the location changes of a task (i.e., task migrations) such that a task does not need to close all its files before migration. In this chapter, a file session is defined as the set of file operations between opening and closing a file and the set of states

that are managed by the file system to correctly handle the operations. Existing file systems cannot provide enough transparency. For example, a task can only access the files opened on its current device and must re-open the files after it moves to another device.

- **Consistency**: Reading stale data may lead to incorrect results or crash an app. Thus, the file system must guarantee strong consistency by default so that a task always reads the latest updates. However, in the case where an app can tolerate relaxed consistency, the file system should be able to take the opportunity to relax consistency and improve performance.

- **Performance**: Mobile devices have limited resources in terms of energy and network bandwidth. Thus, cloud-assisted apps often need to pay for the network traffic through cellular networks. It is important for the file system to satisfy file access requests with low latency (for higher performance and power efficiency) and little network traffic (for lower monetary cost and energy consumption). Existing networked/distributed file systems are not optimized for cloud-assisted apps.

- **Easy deployment**: To freely offload tasks, a design that can simplify the deployment of the file system and data is highly desirable. Since a mobile user may have limited privileges on the cloud platform accepting offloaded tasks, the deployment of the file system should require minimal privileges in addition to those needed to run the task. At the same time, the file system should have minimal requirements on data deployment. Conventional networked/distributed file systems usually require that files be deployed under specific directories to enable remote access. However, it is challenging, if not impossible, to identify all the files to be accessed remotely by mobile apps and organize them accordingly, since the files to be accessed by mobile apps may be determined by user requests. At the same time, most networked/distributed file systems require root privilege to deploy and to run, which is missing on most mobile devices.

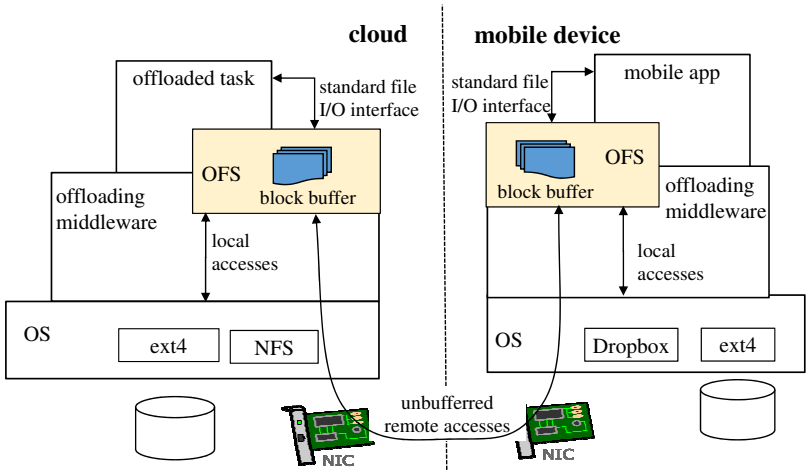


Figure 3.1 Overall architecture of offloading ecosystem.

3.2 OFS Design

3.2.1 Overall System Architecture

OFS is a component of the system that offloads and manages computation tasks. Figure 3.1 illustrates the position of OFS on the mobile device and the cloud platforms, and explains how OFS interacts with other components in the platforms. Unlike conventional file systems, which are part of the operating system, OFS functions at the application level. Its code is executed in user mode, and its data structures (e.g., information about the files, file accesses, and the buffer caching file data) are maintained in user space. However, OFS relies on the native file systems in the OS to actually read data from the storage or write data into the storage.

There are several reasons for this application-level design. First, OFS is solely designed to provide file accesses for the correct and efficient execution of mobile apps. It does not provide system-wide management, e.g., user access control, or a tree of files and directories presented to the user. It does not manage storage space either. Second, building OFS at the application level makes it an *overlay* file system that sits above all the native file systems, thus allowing it to work with any native file systems through the standard system call interface. Third, keeping all the functionality and data structures within virtual memory spaces at the application level simplifies deployment. For example, there is no need to acquire root privilege to set up the file system. Finally, this design

helps to improve efficiency since accessing the data structures and file data cache in virtual memory space does not incur costly kernel-application context switches.

The objective of OFS is to provide efficient, transparent, and consistent file accesses and file sharing for the tasks in a cloud-assisted mobile app. For this purpose, OFS intercepts and monitors the file access requests from the tasks in the app. File access requests can be intercepted without modifying existing apps using techniques such as code injection and byte code manipulation. How this is achieved in our OFS prototype will be described in Section 3.3.2. OFS fulfills the file access requests for accessing local files by passing them to the OS and then to the corresponding native file systems holding the files. For the requests accessing remote files, OFS maintains a buffer named *block buffer* to cache the blocks read from remote files through the network. To fulfill the requests, OFS looks up the *block buffer* and serves the requests if the desired file blocks are cached there. Otherwise, it redirects the unsatisfied requests to the platform storing the files. Note that a file may be stored on the mobile and requested by a task from the cloud or vice versa.

OFS maintains consistency between the blocks in the block buffer and their counterparts saved in remote files, such that a task can always access the latest updates no matter where it runs. In addition to file accesses, OFS must also handle other file related requests, such as opening/closing files, creating/removing files, etc. OFS handles these requests by forwarding them to the platform storing the files and by updating the related metadata maintained on both the platform that opens the file and on the platform, that stores the file.

3.2.2 OFS Architecture and Design

As shown in Figure 3.2, OFS consists of four major components. The *native/OFS switch* intercepts the file I/O requests before they reach the OS and decides for each request whether it should be handled by a native file system or by OFS. Generally, OFS handles all the requests to be files that are currently accessed by offloaded tasks, and forwards other requests to native file systems. Thus, in the cloud, all the requests made by offloaded

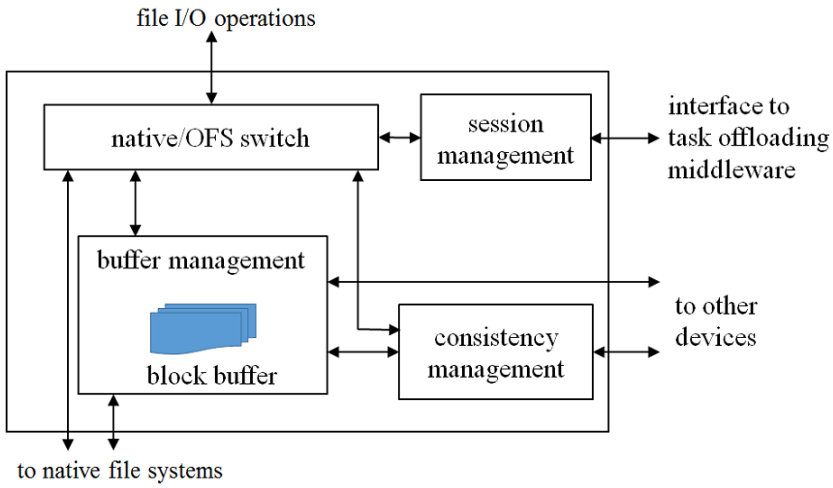


Figure 3.2 Overlay File System (OFS) architecture.

tasks are handled by OFS. On the mobile device, if a file is not currently accessed by offloaded tasks, the accesses to the file should be forwarded to the corresponding native file system; otherwise, they are handled by OFS. To improve performance, read-only files (e.g., libraries) can be distributed on both sides and accessed locally without the intervention of OFS.

The *native/OFS switch* needs to notify the *consistency manager* about all the accesses before it passes the requests to either a local file system or the buffer management component. When handling a write request, it only proceeds after the consistency manager confirms that the write will not cause inconsistency. When handling a read request, it just notifies the consistency manager, since the access information is needed there to detect access patterns.

The *buffer management* is in charge of managing the block buffer. To look up the buffer, we maintain a mapping table for each file and save the mapping table in the data structure of the file. We also maintain the status of the blocks in the mapping table. Thus, when the file is accessed, OFS can quickly locate the mapping table, from which it determines whether the requested block is buffered, and, if it is, whether the buffered block is up-to-date.

We use an LRU-like algorithm to evict blocks to keep the buffer size within a pre-set limit, which is selected by the user during installation based on the memory capacity of

the devices. Due to the high network overhead, it is not cost-effective to offload tasks accessing a large amount of data. Thus, a small size limit (e.g., 1/32 of memory capacity as the default limit) should work well for most of the workloads.

The LRU-like algorithm organizes all the buffered blocks into a linked list. When a file is closed, the algorithm moves all the blocks of the file to the LRU end of the list. When the content of a block becomes stale, the block is also moved to the LRU end. When a block is accessed, the algorithm moves it to the MRU (most-recently-used) end of the list. When space is needed, the algorithm selects and evicts the blocks at the LRU end.

We create the *block buffer* in the virtual address space. This is not only for fast access and ease of deployment, but also to simplify the system design, since the management of the physical space of the buffer (e.g., space allocation/deallocation and swapping) can be done with by the memory management of the operating system. At the same time, it puts the physical memory space occupied by the block buffer under unified management with other system components and apps. This helps the operating system balance system memory usage for the overall benefit of system performance. For space efficiency, the block buffer only caches the content of remote files. It does not buffer the content in local files to avoid double buffering in both the block buffer and the OS buffer cache.

The *session management* component maintains file sessions and prevents them from being interrupted by task migrations. Specifically, when a task is migrated, the session management component is notified. On the destination platform, the session management component must correctly set up the state required by the unfinished file sessions in the task. For example, it must copy file states, such as the current offset in each file and the opening mode of the file, from the source platform.

Though buffering data improves efficiency, it incurs consistency issues. The *consistency management* component provides the consistency guarantee that is required by concurrent programs. For this purpose, it monitors all the accesses to the shared files, as well as the blocks cached in the block buffer. Enforcing consistency usually incurs a large amount of network traffic (e.g., when *write-update* policy is used) or increased read

access latency due to increased misses in the buffer (e.g., when *write-invalidate* policy is used). Both long access latency and increased network traffic are not desirable for task offloading in mobile apps. Thus, we use an adaptive algorithm named *delayed-update* combining write-invalidate and write-update (Section 3.2.3) to reduce both latency and network traffic.

3.2.3 Consistency Management in OFS

Consistency Management Design Objectives The main goal of OFS is to provide an environment in which the tasks of a mobile app can access and share their files concurrently from both the mobile device and the cloud in the same way as they do when they run on the same device, where they share the OS buffer cache and can always see the latest updates. This will not only guarantee the correct execution of mobile apps, but will also simplify app development, because programmers will not be concerned with getting stale data in apps. Therefore, the first design objective is to ensure strong consistency.

Enforcing strong consistency may incur high overhead. There are two common policies for keeping consistency. *Write-invalidate policy* invalidates all the duplicates of a file block before writing the block locally. *Write-update policy* ensures that a write operation does not complete until all the duplicates are updated. The write-invalidate policy minimizes the amount of data transferred over the network (i.e., network overhead) but increases the latency for read operations because invalidating duplicates reduces the number of local accesses. The write-update policy helps to keep the duplicates valid and, thus, read access latency low, but incurs a large amount of network traffic for broadcasting all updates and high overhead for write accesses. Therefore, the second design objective is to reduce the network traffic incurred by enforcing strong consistency and, at the same time, keep the access latency low.

Strong consistency may not be always desirable. There are situations in which enforcing strong consistency is not necessary or the overhead incurred by enforcing strong consistency is too high. Thus, the third design objective is to satisfy consistency demands

other than strong consistency. For example, a health monitoring app collects wellness data of a user every second using the sensors on a mobile device and analyzes the data in the cloud. While the latest data is preferred by the analysis in the cloud, using the data collected a few seconds ago still generates sensible results. If the mobile device is short of resources (e.g., low power level), updating the data lazily is a better choice than enforcing strong consistency.

Delayed-Update Algorithm To achieve the strong consistency, we design a hybrid approach named *delayed-update*, which combines the write-invalidate and write-update policies. This new policy gives better file latency and reduces network traffic. On a write operation, delayed-update invalidates duplicates first to ensure consistency. Then, instead of waiting for a read operation to trigger an update of a duplicate, it predicts when a duplicate is about to be read and it updates this duplicate just before the read. The delayed-update approach reduces network traffic because it does not transfer the updates that have been overwritten before a read. It keeps the access latency low because duplicates are validated and updated before reads. A challenging issue with delayed-update is to predict when the duplicates should be validated and updated. We address this issue by monitoring the file access patterns of of mobile apps, as described later in this section.

In some scenarios, accessing the latest data is not required. For example, in a health-monitoring app, health related data, such as body temperatures and heart rates, is collected and saved periodically. The values of the data may not change rapidly over time. Thus, it may not cause problems if the health-monitoring app uses the data collected recently, e.g., 5 seconds ago. For such scenarios, OFS provides a relaxation mechanism that allows an app to use recent but not the latest copies of file data. The mechanism extends the delayed-update approach with a knob named relaxation to relax the requirement on enforcing consistency. Using the same health monitoring app as an example, if the app can use the data generated 5 seconds ago, the relaxation is set to 5. A suitable relaxation value is application-dependent and data-dependent. By default,

OFS sets relaxation to 0 in order to enforce strong consistency. In the cases where relaxation can be applied, OFS relies on application developers and users to decide suitable relaxation values and adjusts the values through an API provided by OFS. With a large relaxation value, delayed-update can update duplicates even less frequently to reduce resource consumption.

The delayed-update algorithm keeps information to reflect the current status of a block. The following information is kept on both the mobile device and the cloud, for each block of data in the block buffer or in local storage that has been accessed by the app.

- A *shared* flag indicates if there are duplicates of the block cached in block buffers or saved in storage.
- A *valid* flag indicates if the block content is up-to-date.
- For each valid block, we also attach an *expiration time* to implement the relaxation feature. A valid block with a non-zero expiration time indicates that the block content is not up-to-date, but can still be used by the app until the expiration time. The block is invalidated when the expiration time is reached.
- The *location* of the latest update.
- An *overwritten threshold* indicates when remote duplicates should be updated.
- An *overwritten counter* counts how many times a block has been overwritten.

When a block is being read, its content is returned immediately if the block is valid; otherwise, the latest update is fetched remotely, and the status of the block is updated to valid and shared.

When a block is being written, the block is updated immediately if it is not shared; otherwise, a message is sent to invalidate the duplicates before the block is updated and the "shared" flag is reset. When such an invalidation message is received on either the mobile device or the cloud, the corresponding block is invalidated (when the relaxation is

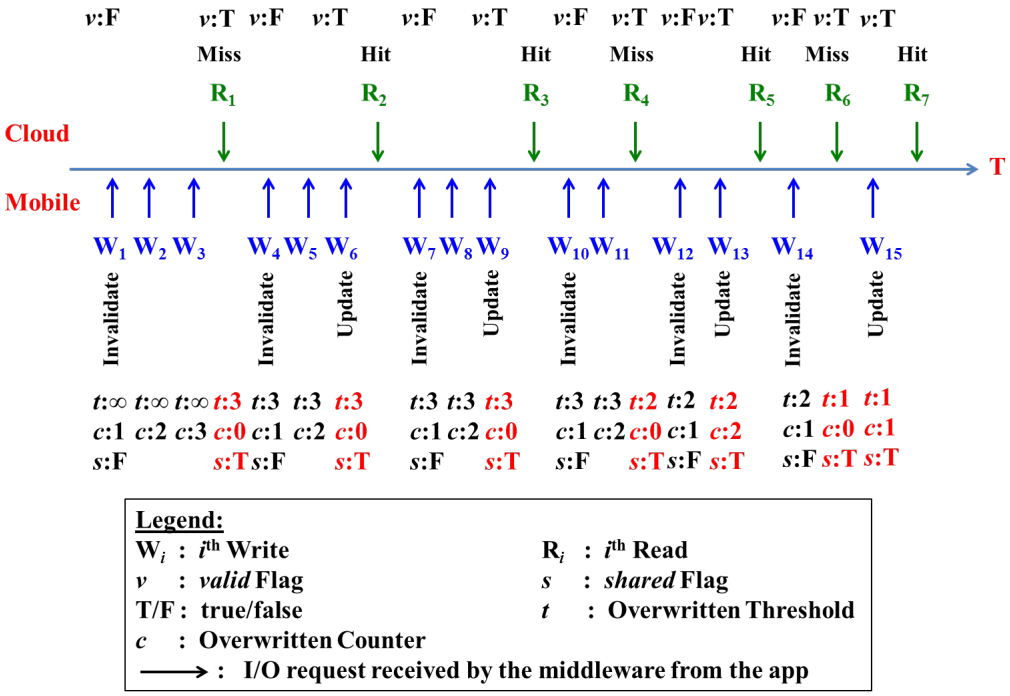


Figure 3.3 Workflow of Delayed-Update algorithm.

zero) or marked with an expiration time (when the relaxation is greater than zero); at the same time, the location of the latest update is recorded in the mapping table maintained by the buffer management component.

The delayed-update algorithm tries to update remote duplicates when they are about to be read. To achieve this goal, the algorithm updates and uses the overwritten threshold as an indicator. When the number of block overwrites reaches this threshold, the remote duplicates are updated. The threshold is dynamically updated based on the history of accesses. Specifically, every time a block is overwritten, the overwritten counter is incremented. When the content updated in the block is accessed somewhere else (i.e., the platform other than the one generating the content), the overwritten threshold is updated with the value of the overwritten counter, and the overwritten counter is reset. Thus, the threshold reflects how many times a block is overwritten before the content is used, and can be used to predict when remote duplicates should be updated.

In order to describe the basic idea of the delayed-update algorithm, we use an illustrative example with a series of reads ($R_1 \sim R_7$) and writes ($W_1 \sim W_{15}$) on the same

block, as shown in Figure 3.3. Writes are on mobile device, and reads are in the cloud. The states of the valid flag, shared flag, overwritten threshold, and overwritten counter used in the algorithm are marked with v, s, t, and c in the figure.

When the block is being written for the first time on the mobile device (W_1 in Figure 3.3), the shared flag shows that it has a duplicated copy, thus an invalidation message is sent to the cloud to invalidate the copy. On receiving the message, OFS in the cloud sets the valid flag to $false$ and acknowledges the message. On receiving the acknowledgement, OFS in mobile device sets the shared flag to $false$. Subsequent updates to the block, W_2 and W_3, can be performed directly since there is no duplicated copy. When the cloud tries to read the block, it checks the valid flag first. If the block is invalid (e.g., R_1 in Figure 3.3), a miss is occurs and the block is propagated. Thus, the shared flag on the mobile device is changed to $true$, and further updates (W_4) will result in an invalidation message. Until now, the algorithm performs exactly as a write-invalidation algorithm, except that the algorithm maintains an overwritten counter and an overwritten threshold for the block on each side (c and t in the figure for the mobile device). The counter is reset every time the block is propagated (e.g., R_1), and incremented every time the block is overwritten. The value of the overwritten counter is saved into the overwritten threshold before it is reset (e.g., the change of the t value corresponding to R_1). With more updates performed on the block (W_5 and W_6), the overwritten counter keeps increasing. When the value of the overwritten counter reaches the value of the overwritten threshold (3 when W_6 is performed), the mobile device propagates the new content in the block to the cloud before a read is issued in the cloud (R_2). This reduces the latency. This part shares a similar idea with the write-update algorithm. However, it only performs updates when it predicts that the updates are necessary. The prediction relies on the program maintaining a regular access pattern (e.g., the time period from W_1 to R_3). Misprediction occurs when the program changes its access pattern (e.g., W_{10}, W_{11}, and R_4). But the algorithm can quickly adapt and adjust the prediction based on the new pattern, as it does for W_{12}, W_{13}, and R_5.

In case if an app tries to write a block simultaneously from two different threads or objects running on both mobile and cloud, the write that reaches OFS first will be able to complete successfully. And the second one will be completed after the first write is completed. This might lead to inconsistency if the writes arrive to OFS in wrong order. Such scenarios are highly application dependent. Therefore, OFS depends on programmers to implement appropriate locking mechanism for their apps.

3.3 OFS Implementation

OFS sits between mobile apps and the offloading middleware and it is implemented at the application level rather than the OS level. This presents several challenges to the implementation, including the lack of root privilege and state loss when application is killed due to the short of resource. This section introduces the implementation details of OFS, particularly how these challenges are addressed.

3.3.1 Implementation Details

We have implemented an OFS prototype with Java on Android OS and using an Android-x86 VM to run offloaded tasks. Though the implementation is Android-based, the techniques used in the implementation are generic and can be adapted to implement OFS on other mobile OSs.

At the application level, OFS can be implemented in two ways: as a library that is dynamically linked into each app, or as a set services, which are independent threads running in the background without interaction with users. With the library implementation, the OFS code, system states, and block buffer are in the memory space of each app. Thus, the app can directly access OFS functionalities and data with high efficiency. However, this implementation incurs consistency issues, since mobile OSs, such as Android and iOS, may kill an app and reclaim its memory space when it is switched to the background. Inconsistency is caused if there are unsynchronized OFS system states or file data in the memory space, which are lost when the memory space is reclaimed.

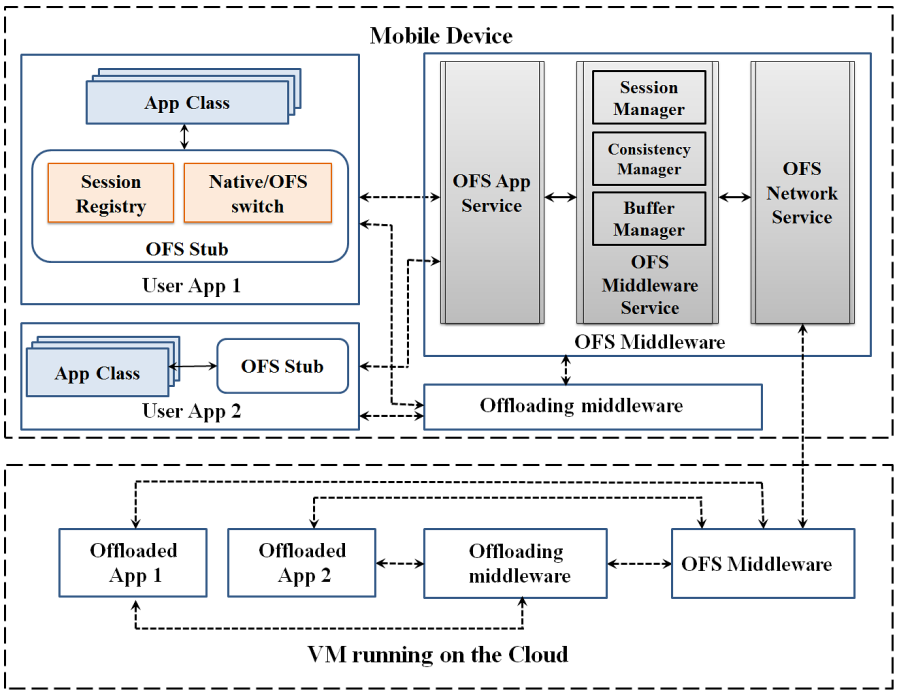


Figure 3.4 Architecture of OFS implementation.

The other issue is that the library implementation does not support file sharing between apps.

We choose to implement OFS into a set of application services (named *OFS middleware*) and keep application-specific states inside each app. The services start automatically when the system is on. They are marked as "sticky" services, so that they are less likely to be killed by the OS than normal application threads and other services. In some rare cases when these "sticky" services are killed by the OS, they will be restarted later automatically by the OS before other services and apps. OFS has a simple check-pointing mechanism implemented to back-up OFS system states into storage before the services are killed. The check-pointing mechanism can also be used to handle network disconnection problems of offloaded tasks. If an offloaded task was disconnected due to network issues, OFS can roll back to the states before the task was offloaded.

To facilitate the accesses to OFS services, we provide a component, named *OFS stub*, which is linked into each app process as the interface between the app process and OFS.

Figure 3.4 shows the architecture of the implementation, which has two layers. The upper layer, named OFS stub, is mainly in charge of intercepting file I/O requests, maintaining app-specific information, and interacting with OFS middleware to satisfy file I/O requests. It consists of two major components, the native/OFS switch and the session registry. The native/OFS switch is as introduced in Section 3.2. The session registry, as a part of session management in Figure 3.2, is in charge of maintaining file sessions by managing and updating the data structures used by the app for accessing open files, such as status of the file (location, open mode, etc), current offset, length and so on.

For the tasks offloaded into the cloud, the session registry provides the data structures required for accessing files. For the tasks running on the mobile device, the session registry mainly serves as a registration list of all the files that are currently accessed by offloaded tasks. The list is used by the native/OFS switch on the mobile device to filter requests[2]. The session registries in the cloud and on the mobile device are updated consistently when a file is accessed by an offloaded task and the information of the file cannot be found in the session registries. Specifically, before a task is offloaded, the session registry on the mobile device is empty, and thus all the file accesses of the app are handled by native file systems on the mobile device. Later, when a task is offloaded into the cloud and the task starts to access a file, the session registry in the cloud is searched. Since the required data structure for accessing the file cannot be found there, the access cannot proceed before the data structures are set up and registered as a file session. To set up and register the data structures, the OFS stub in the cloud generates a reopening request, which is forwarded to the OFS stub in the mobile device. On receiving the re-opening request, the OFS stub in the mobile device registers the file in its session registry. In this way, the file is marked as being accessed by an offloaded task, and later accesses to the file are forwarded to OFS by the native/OFS switch. Then, the OFS stub in the mobile device sends back the information required for accessing the file (e.g., file offset and open mode) to the OFS stub in the cloud, which then uses the information to update

the session registry in the cloud. With the information, later accesses can be handled by OFS.

Using Filesystem in Userspace (FUSE) [73] may simplify the implementation. However, FUSE requires root permission and rooted systems. Android needs to be recompiled in order to implement OFS in the existing FUSE daemon. Thus, rather than using FUSE, we implemented the OFS middleware using a few app services, which run on both mobile device and the cloud. The main service, named *middleware service*, implements the other three major components of OFS described in Section 3.2.2. Two supporting services assist the main service to interact with other system components. Specifically, the *app service* interacts with apps to receive requests and deliver responses; and the *network service* is responsible for maintaining the interaction between the OFS instances running in the cloud component and the mobile device.

We build OFS in an event-driven architecture. Apps and OFS services interact with each other with events and messages. The OFS middleware is an event-driven middleware. In our implementation, we differentiate between the events handled by the OFS middleware by whether they are directly related to maintaining consistency. Events are mainly used to convey control information, such as creating and destroying file I/O sessions; and messages are mainly used to transfer file data and metadata. Since control information usually has higher urgency than other data, we organize events and messages separately and give higher priority to processing events.

In OFS, a large amount of data may be exchanged over network or locally across different OFS components, and some messages (e.g., events and updates on staled data) must be processed promptly. Thus, OFS must handle data communication with high efficiency. For network communication, we adopted a NIO-based TCP library named Kryonet [74], which is usually used by online games for high network throughput and low latency. For local communication, we used Android's Binder IPC mechanism. Though Android provides another IPC mechanism called BroadcastReceiver, it is for exchanging

[2]The native/OFS switch in the cloud determines that all the file accesses should be handled by OFS, except for the accesses to the files pre-configured to be accessed locally (e.g., read-only files).

small amounts of information. Based on our experiments, using BroadcastReceiver can reduce the communication throughput between the OFS stub and the OFS middleware by up to 3x.

We used the offloading service of the Avatar platform [42] as the offloading platform for our implementation. Avatar is a distributed mobile-cloud platform where each mobile device has a surrogate in the cloud. It also supports offloading Plain Old Java Object (POJO) to the cloud. The POJO is a software engineering term used to describe a Java object not bound by any special restriction or external class path. As the Avatar platform supports multi-threading programming, offloading an object only blocks the relevant threads in the mobile device instead of all of them. Unlike a regular offloading platform, offloading in Avatar aims to improve battery consumption, network bandwidth and latency for a group of users. It uses annotations to intercept the targeted code segment and uses a profiler to decide whether to offload based on a QoS defined by the targeted user group. For the experiments conducted in Section 3.5, we hardcoded which operations are being offloaded to the cloud in order to ensure the intended task is always offloaded.

3.3.2 Implementation Issues

To implement OFS in userspace, we had to solve several issues. One issue with a user-level implementation is how OFS can interact with different apps to intercept their file I/O requests and satisfy them. To address this issue, our implementation intercepts library calls, instead of system calls. The interception of library calls does not require a system-level privilege and can be implemented with various approaches, e.g., manipulating symbol tables or binary weaving [70, 75]. Our current prototype uses AspectJ [70] in the OFS stub to automatically link the required code to interact with an app with the existing code of the app without additional effort from the app programmer. In this way, an app can be automatically enhanced with OFS support; and the app developers do not need to be aware of task offloading or implement the code that handles file I/O issues for offloaded tasks.

Specifically, in our implementation, the method interception mechanism in AspectJ is used to capture the calls related with file I/O requests. Then, the code to analyze the requests and to call the methods in OFS stub is injected into the app with the weaving mechanism. While the capturing of file I/O requests and the injection of OFS code can be performed when the app is compiled or after the app is compiled (e.g., when the app is being loaded), the current prototype finishes this process at compile time to minimize runtime overhead. [3]

Another issue with a user-level implementation on Android is how to manage the accesses to app-private files. In Android, an app can have two types of files. Private files are saved in the internal (private) storage space, and are only accessible from the apps that created the files. Public files are saved in the external (public) storage space, and can be accessed by any apps. Since OFS middleware runs as application services and cannot access private files of any app, if an offloaded task needs to access a private file saved on the mobile device, the accesses to the private file are forwarded back from OFS middleware to the OFS stub in the corresponding app and performed by the OFS stub. OFS does not buffer the data in private files. This rarely degrades performance, since private files are usually small files, such as settings, configurations, and cached data, and are infrequently accessed.

3.3.3 Interface with Task Offloading Systems

OFS must work synergistically with task offloading systems. However, it is challenging for an OFS implementation to be compatible to different offloading systems. While these systems are designed to offload computation tasks dynamically, they may be implemented in fundamentally different ways at different system levels. As explained in Section 3.1, computation tasks may be offloaded in the form of objects, threads, or procedures. These different methods correspond to different ways of system implementation. If OFS is built

[3] An alternate approach that does not need recompilation is to interpose the library function call paths. This can be done by instrumenting the binaries of the app with tools such as PIN [76] or ProbeDroid [77].

inside an offloading system as a component, different OFS implementations are needed for different task offloading systems.

To increase compatibility and reduce development efforts, we decouple our OFS implementation from any specific task offloading systems and keep a narrow interface between OFS and task offloading systems. With our implementation, the middleware service and the OFS stubs in apps do not interact directly with task offloading systems. A simple utility, named *offloading service*, is developed to accept notifications from task offloading middlewares. The offloading service is notified when a cloud-assisted app is launched, when there is a task newly offloaded to the cloud, or when an offloaded task is about to be migrated back to the mobile device. Based on the notification, the offloading service instructs the OFS middleware to update system states and the related app threads to update application-specific states.

For example, when an object is migrated into the cloud by the Avatar offloading platform, the OFS offloading service in the cloud is notified about the migration with information, such as the ID of the offloaded object. The offloading service contacts the application thread responsible for the offloaded object in the cloud, such that the injected OFS code in the thread can re-establish existing file sessions by re-opening files and moving file pointers. Then, it notifies the OFS middleware about the offloaded object, such that subsequent file I/O requests from the offloaded thread can be serviced by the OFS middleware. Such interactions induce a one-time overhead which is included in the performance results presented in Section 3.5.

3.4 Case Study with a Real App

We implemented a photo enhancement app as a case study. It illustrates the demand for transparently supporting file accesses of cloud-assisted apps, and demonstrates the advantages of OFS. With the app, we also explain in detail how our OFS implementation efficiently supports the file accesses of the tasks distributed across the mobile device and the cloud.

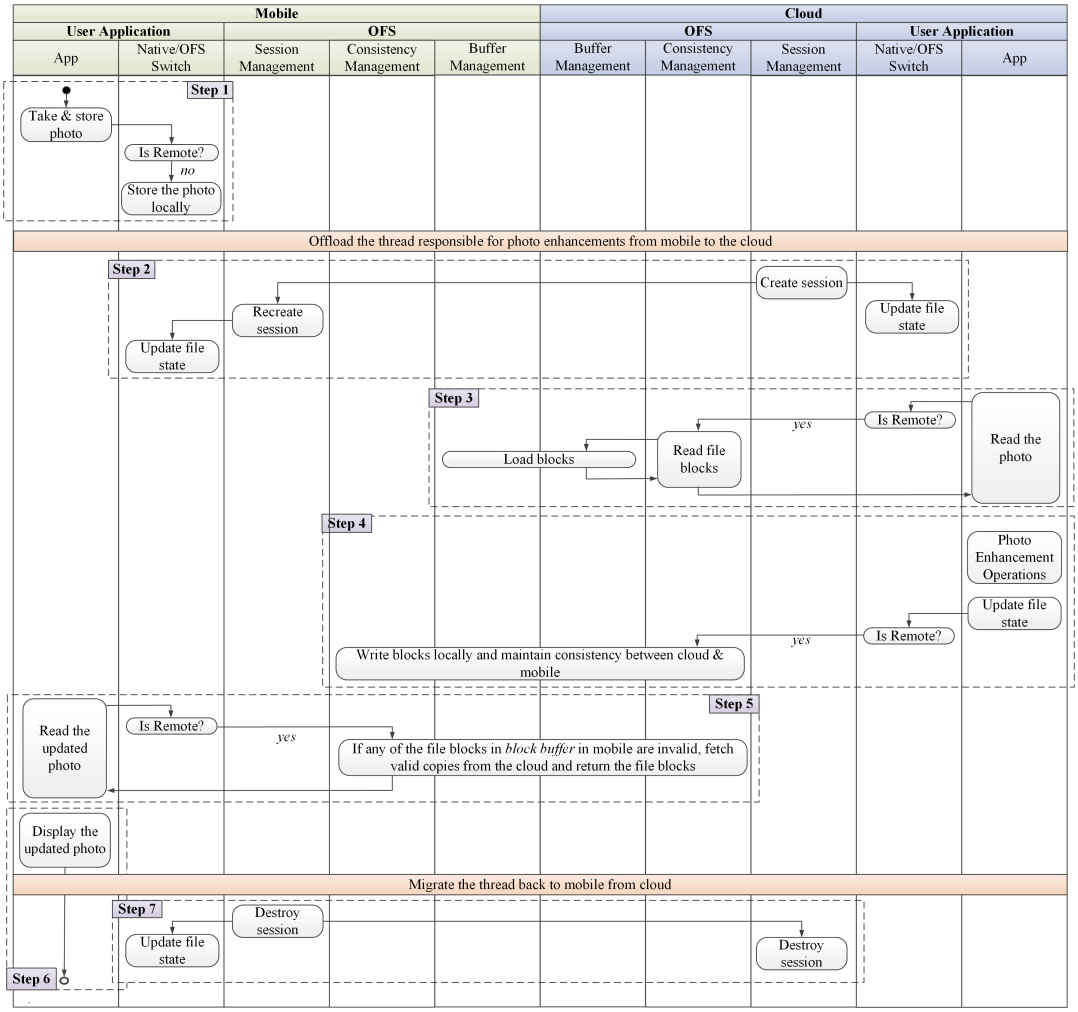


Figure 3.5 The workflow of OFS in an enhanced camera app.

The photo enhancement app processes the photos selected by the user. For each photo, it performs a few photo enhancement operations, including applying color reduction, adding salt noise, applying sharpening filters, and adding a watermark. The app displays the photo to the user after the operations. We implemented each photo enhancement operation in a Java class using OpenCV [78].

To explain the interaction between various components of OFS and the photo enhancement app, detailed workflow is presented in Figure 3.5. As shown in Step 1 in this figure, a user starts the app, takes a photo, and attempts to store it on the mobile. The operation is intercepted by *native/OFS switch*, which determines the operation to be local. Then, the app launches a thread to process the photo. Due to the heavy workload in

the thread, it is offloaded to the cloud. As shown in Step 2 in the figure, in the cloud, the *session manager* sets up the states required by the thread to access the photo and updates the configuration of the session registry of OFS stub and mark the file as remote. Thus, upcoming accesses to the photo from the thread will be determined by the *native/OFS switch* to be remote accesses and will be forwarded to the *buffer manager*. In Step 3, when the thread processes the photo for the first time in the cloud (e.g., to detect faces), the photo is loaded into the block buffer residing in buffer manager. Then, the for each enhancement operation on the photo (e.g., color reduction, adding salt noise, applying sharpening filters, and adding a watermark) Step 4 is repeated multiple times. The photo enhancement operations can access the data cached in the block buffer. Once the enhancement operation is finished, the *consistency manager* in the cloud sends messages to its counterpart on the mobile device to invalidate and/or then update the changed blocks of the photo saved on the mobile device based on the consistency policy enforced by the delayed-update algorithm. The details about how consistency is maintained during the whole process are discussed in the Section 3.2.3. Later, the user interface thread of the app displays the photo on mobile device. The thread will display the newly modified photo (Steps 5 and 6). When the processing thread is migrated back, the remote file access sessions are destroyed (Step 7).

Based on the same source code implementation, we have built three versions of the app: 1) a conventional mobile app named *PE-Mobile* that executes all the operations locally on the mobile device, including the enhancement operations, 2) a cloud-assisted app named *PE-Offload* that can offload photo enhancement operations to the cloud and access the photos using a cloud storage system, and 3) a cloud-assisted app named *PE-OFS* that can offload photo enhancement operations to the cloud and access the photos using OFS.

For fair comparison, in the app, we hard-coded the task offloading part, such that all the photo enhancement operations can be offloaded to the VM. We did not link *PE-Offload* with OFS stub, in order to test whether a cloud storage system (e.g., DropBox or Google Drive) can be used to support the file accesses of the app. The last version, *PE-OFS*,

was built with OFS support. Compared to *PE-Offload*, the enhancement with the OFS support in *PE-OFS* only requires the linking of OFS supporting library with the app, and does not incur additional efforts on programming or annotation.

Before running the app, we first deployed the OFS middleware on a mobile device and an Android-x86 virtual machine (detailed configuration in Section 3.5). Though OFS can be distributed and deployed through app stores, such as Google Play Store, currently the middleware is packed in Android application packages. Thus, we copied the packages (the apk files) into the mobile device and the virtual machine, and side-loaded the packages. Root privilege was not requested during the installation. However, we performed some simple configuration before *PE-OFS* could run: pair the mobile device and the VM, and allow access to library files.

We first run *PE-Mobile* to process a set of photos with different sizes to verify the functionalities of the app. Then, we run PE-Offload to process the same set of photos. We want to justify the necessity to design a system to transparently support the file accesses of a cloud-assisted application. To make the photos accessible to the tasks offloaded to the VM, we installed the DropBox client app on both the mobile device and the VM. Before the execution, we must first upload the photos into a Dropbox directory on the mobile device and mark them available for offline accesses in order to download them into the device. Only after the downloading is finished, can we launch *PE-Offload*. Even though the photos were accessible from the VM, we found that the photo enhancement tasks offloaded to the VM crashed during execution. This is because these tasks access each photo using the file handle created on the mobile device when the photo file is opened before any enhancement operations start, and the file handle is invalid in the VM. To solve the problem, we have to change the source code of the app, such that a photo must be re-opened before each enhancement operation and closed after the operation. With this improvement, the enhancement operations can be finished on the VM without crashing. But we find that the app may display may display the old versions of the photos on the mobiles, even though newer versions with enhancements exist in the cloud. This is caused by DropBox being unable to promptly update the copies on the mobile device. Thus,

we have to re-examine the photos after both the DropBox instance on the VM and the instance on the mobile device finish the synchronization with DropBox server. We have also tested PE-Offload by saving the photos into a Google Drive and experienced similar problems.

Despite the increased management and programming efforts, with existing cloud storage systems, a cloud-assisted program still may not be able to deliver correct results. This clearly shows that existing cloud storage systems cannot meet the requirements of cloud-assisted apps and a system must be designed to support the file accesses of these apps transparently and consistently.

We have also tested *PE-OFS* with the same set of photos. We run *PE-OFS* for two times. We first run *PE-OFS* completely on the mobile device without offloading any enhancement operations. Then we run it with the enhancement operations offloaded to the VM. With *PE-OFS*, the photos can be enhanced and correctly displayed after the enhancements no matter whether the enhancement operations are offloaded to the cloud or not. When the enhancement operations are performed on the mobile device, *PE-OFS* shows similar performance as *PE-Mobile*. The end-to-end latency for the enhancement operations on each photo is less than 0.6% higher than *PE-Mobile*. When the enhancement operations are offloaded to the cloud, compared to *PE-Mobile*, the end-to-end latency is reduced by 43% on average with *PE-OFS*, and the combined energy consumption of both the app and OFS middleware running on the mobile device is reduced by on average 3%.

The above experiments show that OFS has low overhead and can effectively support the seamless execution of cloud-assisted apps on the mobile device and in the cloud. We will present the detailed performance results in Section 3.5. In this section, we focus on explaining how OFS transparently supports the consistent file accesses of *PE-OFS* on both mobile device and in the cloud.

3.5 Performance Evaluation

This section assesses the performance of OFS and evaluates its delayed-update consistency policy by comparing the performance with other consistency policies. We use the following

metrics: 1) *Execution time* and *average file I/O latency* measure the performance of OFS and comparison solutions. 2) *Network overhead* quantifies the network overhead introduced by each solution. It practically represents the cost of achieving lower I/O latency. 3) *Number of overwrites per data transfer* measures how many overwrites are performed on a file block until it is transferred. Practically, it helps us estimate the benefits of delayed-update policy. The higher the values of this metric, the more reduction in network overhead. 4) *Power consumption* quantifies the power consumed by both the OFS middleware and the app that uses OFS.

We conducted two sets of experiments. With the first set of experiments, we used the photo enhancement app with OFS. We measured the end-to-end delay for the photo enhancement operations in order to show that offloading tasks to the cloud can effectively reduce the active time of mobile devices, leading to faster app execution. With the second set of experiments, we analyze how the delayed-update method can effectively reduce both network overhead and I/O latency.

3.5.1 Experiment Setup

The experiments were conducted on a Nexus 6 mobile phone running Android 7 and a x86 VM running Android 6. The VM was hosted on an OpenStack-based cloud. It has 2 virtual CPUs, 3GB memory, and 16GB storage. The physical machine hosting the VM has an Intel Xeon (E5-2630) processor, 78GB memory, and 2TB storage. We installed the OFS middleware on both the mobile phone and the VM. In the middleware, in addition to the delayed-update policy, we also implemented the write-invalidate and the write-update policies, which can replace the default delayed-update policy through OFS configuration. For our experiments, we set the block buffer size to be 64MB. The replacement algorithm is run when block buffer is full and new data needs to be saved. With this size, hit ratios are above 95% for all workloads.

We tested our implementation by running the aforementioned app and an app that replays the file I/O traces collected from real mobile users. For the experiments with the enhanced camera app, we used the app to enhance three sets of photos with different

resolutions. Each set contains 15 photos. The resolutions of these three sets of photos are 2.1 megapixel, 5.0 megapixel and 9.7 megapixel, respectively. For each photo, the app first displays the original photo. Then, it enhances the photo on the VM. When the enhancements finish, it displays the enhanced photo on the phone immediately.

To replay traces, we first developed an app. The app creates some threads on the phone and some other threads on the VM. These threads read the records of file I/O operations in a trace and perform the corresponding operations on the corresponding files. To support the execution, we created a set of files based on the file names and paths in the traces. The contents in the files are randomly generated, since no computation is carried out on the contents.

In order to compare the performance of OFS with an existing network file system, NFS [11], a trace-driven emulation is used. As stock Android does not implement NFS, we used emulation instead of actual implementation. In the emulator, a mobile device is connected to a VM (an Amazon EC2 instance in US-East region) through a cellular network (LTE) with a latency of 35 milliseconds and a bandwidth of 5Mbps. NFS implements a close-to-open policy: when an NFS client closes a file, it flushes all the modified data back to the server; later, when another NFS client opens the file, the client can read the latest data from the server. For consistency, clients need to use file locks or shared reservations to avoid concurrent file sessions. This reduces the flexibility of accessing files concurrently.

The traces were collected on the PhoneLab testbed [79] from six real mobile users, one trace for each user. Specifically, file I/O system calls were captured on Android phones using boinic [80] when the users were actively using these phones for different amounts of time and executing different apps with different I/O patterns. To imitate the concurrent execution of the tasks offloaded to the cloud and the tasks on the phone, we divided the file operations in each trace into two sets, and re-played one set of operations with the threads on the phone and the other set with the threads on the VM.

We divided the operations in two ways to imitate two different task offloading schemes:

- *Thread offloading*: The traces have the IDs of the threads performing I/O operations. We sorted the threads based on the number of I/O operations performed by them, then divided the threads into two sets, each set of threads having approximately 50% of the I/O operations. We replayed the I/O operations of one set of threads in the VM and the rest of the file operations one the phone.

- *Procedure offloading*: for each thread, we first replayed 30% of its file operations on the phone, then replayed 50% of its file operations in the cloud, and finally replayed the rest (20%) of its file operations on the phone again. The 50% file operations replayed in the VM imitate those caused by procedure offloading.

Thus, we obtained 12 workloads: one set of six traces for thread offloading and one set of six traces for procedure offloading.

3.5.2 Results with Photo Enhancement App

This subsection evaluates the performance of the photo enhancement app *PE-OFS* we built for the case study (Section 3.4) to understand the advantages and overhead of OFS.

For each set of photos with different sizes, we first run *PE-Mobile* on the phone; then we run *PE-OFS* under four different scenarios: 1) *PE-OFS (mobile only)*: only on the phone without task offloading, 2) *PE-OFS (write-invalidate)*: on the phone with photo enhancement operations offloaded to the VM and the write-invalidate policy used to maintain consistency, 3) *PE-OFS (write-update)*: on the phone with photo enhancement operations offloaded to the VM and the write-update policy used to maintain consistency, and 4) *PE-OFS (delayed update)*: on the phone with photo enhancement operations offloaded to the VM and the delayed-update policy used to maintain consistency operations on corresponding files. When *PE-OFS* is launched, the photos are saved in the VM in the second scenario, and are saved on the mobile device in other scenarios. Under each of the above scenarios for *PE-Mobile* and *PE-OFS* executions, we collect the average end-to-end processing time for the photos in each set.

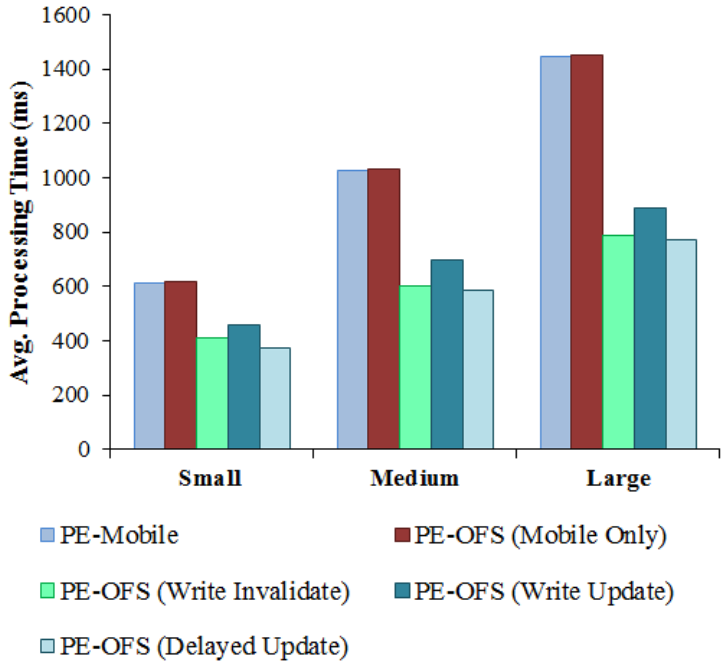


Figure 3.6 Average processing time of the photo enhancement app under six settings.

Figure 3.6 compares the end-to-end processing time for the above scenarios. First, it shows that, when running on the phone without task offloading, *PE-OFS* has similar performance with *PE-Mobile*, indicating the low overhead of OFS. On the VM, photo processing can be finished much faster than on the phone. Based on our measurement, the processing time is reduced by 86% on the VM on average for all the photos than on the phone. Therefore, despite that large network latency can offset the benefits of task offloading, when *PE-OFS* run on the phone with photo processing tasks offloaded to the VM, the average processing time with *PE-OFS* is still shorter than that with *PE-Mobile* by at least 31%. As shown in the figure, the performance advantage of *PE-OFS* is more prominent with larger photos.

In Figure 3.6, the last three bars in each group compare the performance of *PE-OFS* when the OFS middleware uses three different consistency policies: write-invalidate, write-update, and delayed-update. The average processing time is the longest with the write-update policy, and is the shortest with the delayed-update policy. Compared to the delayed update policy, the average processing times with write-invalidate and write-update policies are 5% and 20% higher than that with delayed-update.

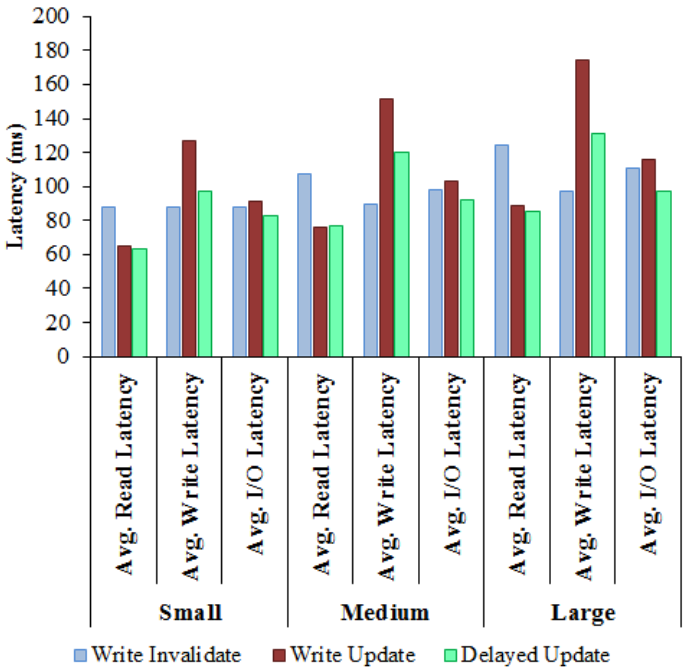


Figure 3.7 Average read latency, average write latency and average I/O for photo enhancement app.

To better understand the performance difference, we measure latency of file read operations and the latency of file write operations, and show the average latency of reads, writes, and all the file operations in Figure 3.7. Generally, average latency are higher with bigger photos than with smaller photos, because the app reads/writes a whole photo in one I/O operation. As shown in the figure, among three policies, the write-invalidate policy incurs the highest average read latency due to the long latency caused by reading invalidated duplicates; and the write-update policy incurs the highest write latency, since it must update all the duplicates on each write. The delayed-update policy updates duplicates only when they are predicted to be read soon. Compared to write-update, the average write latency with delayed-update is 23% lower, since it does not need to update duplicate on every write with delayed-update. Compared to write-invalidate, the average read latency is 29% lower with delayed-update, which may have validate duplicates before they are read. On average, the average latency of file I/O operations are 8% and 12% lower with delayed-update policy than with write-invalidate and write-update policies, respectively.

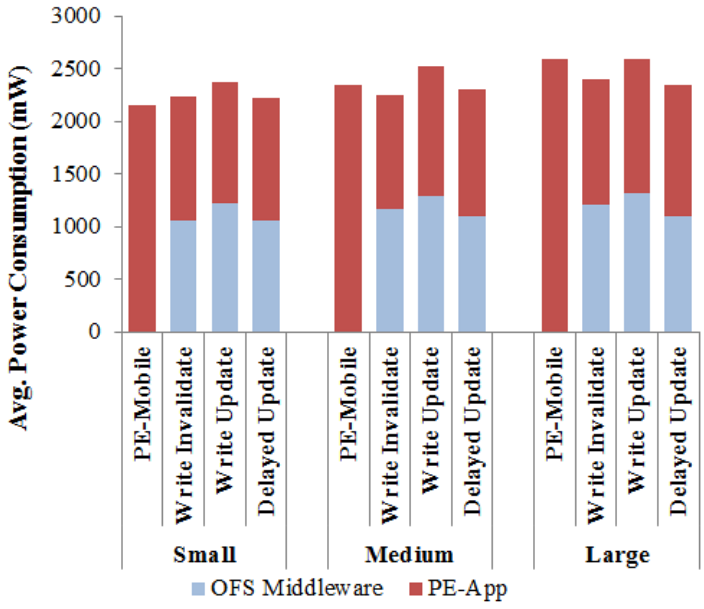
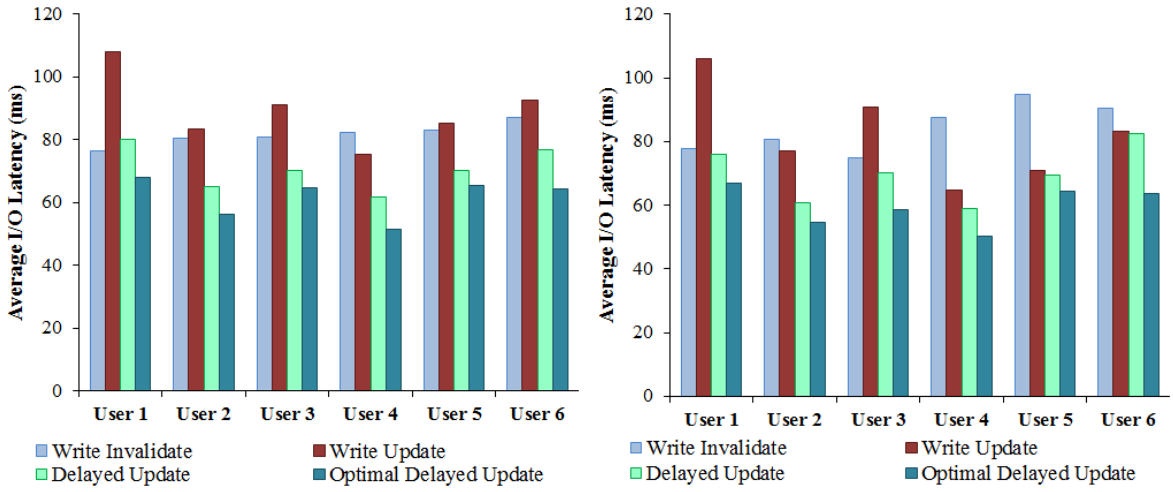


Figure 3.8 Average power consumption for the photo enhancement app.

The last experiment measures the power consumption (i.e., energy consumed every second) incurred by the app using Trepn Power Profiler [81]. For *PE-OFS*, the power consumption consists of two parts: the power consumption of the app itself and the power consumption of OFS middleware. Figure 3.8 shows the average power consumption during photo processing. From the figure, it is clear that with the increase of image size, the average power consumption increases. For small photos, the average power consumption with *PE-Mobile* is lower than that with *PE-OFS*. For medium and large photos, the average power consumption with *PE-Mobile* is higher than that with *PE-OFS* if write-invalidate or delayed update policy is used. When write-update is used, due to the large amount of energy consumed for updating duplicates frequently, the power consumption with *PE-OFS* is higher than *PE-Mobile*. With similar or lower power consumption to *PE-Mobile*, the reduced processing time with *PE-OFS* is also translated into reduced energy consumption, particularly when the consistency policy used in OFS is properly chosen. Since photo sizes keep increasing, this advantage will be more prominent.



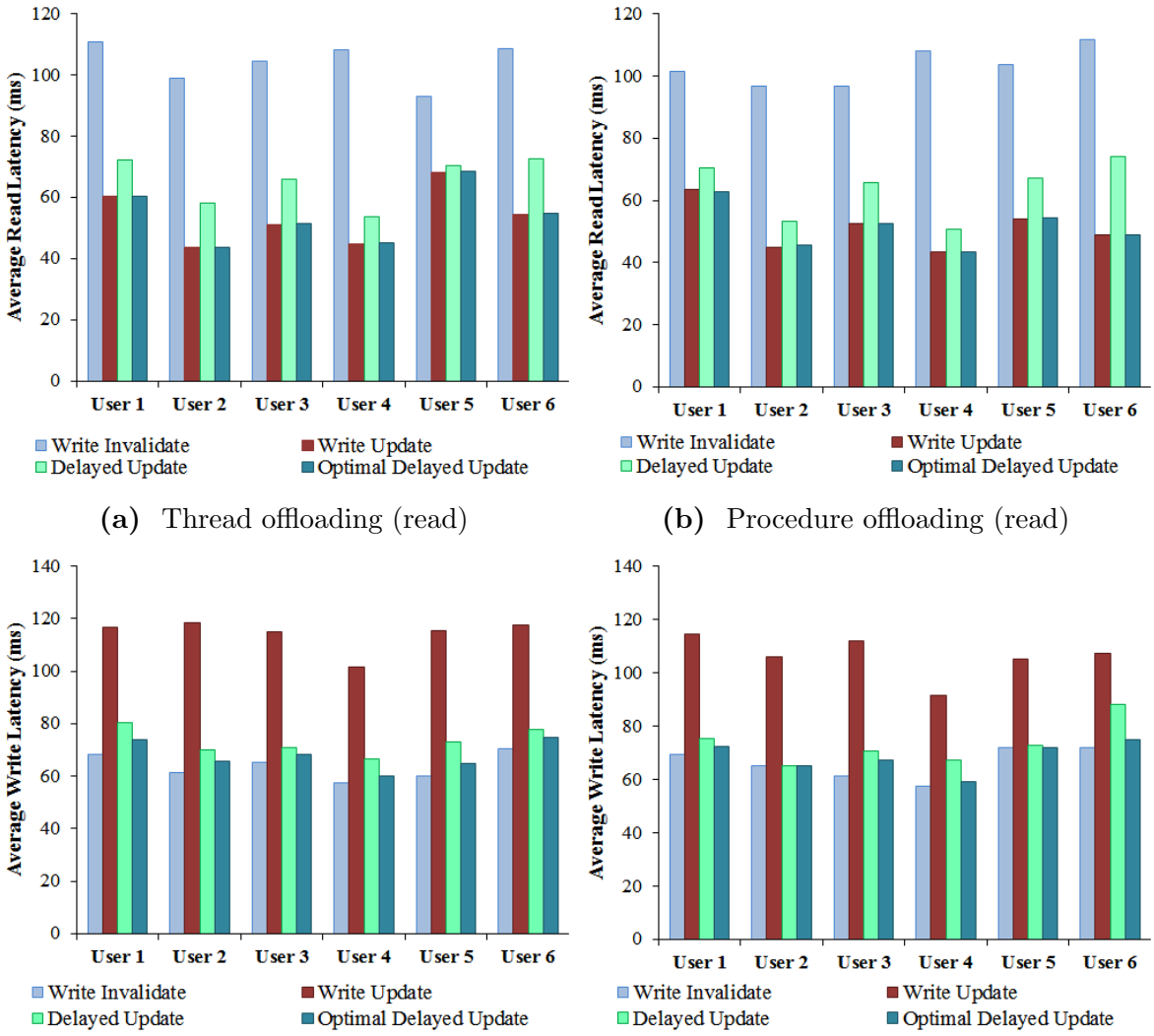
(a) Thread offloading **(b)** Procedure offloading

Figure 3.9 Average I/O latency for six mobile users.

3.5.3 Results with the Real Mobile User Traces

This section tests the performance of OFS using the file operations in the traces. We compare the performance of delayed-update policy with three alternative consistency policies: write invalidate, write update, and optimal delayed-update. The original delayed-update policy relies on the prediction of future accesses to make decision on whether remote duplicates should be updated. The optimal delayed-update policy can be considered as an improved delayed-update policy, in which the prediction is 100% correct, with the knowledge on the file accesses issued in the future. Though it is not possible to make 100% correct prediction in practice, by comparing the performance between the delayed update policy and the optimal delayed-update policy, we can estimate the potential to further improve the delayed update policy. We implemented the optimal delayed-update policy by modifying the OFS middleware to accept the hints passed from the trace-replaying app.

Figure 3.9 compares the average latency of file I/O operations with these policies for thread offloading and procedure offloading. The I/O latency mainly consists of network latency, the time to access the local storage, and the time spent on IPC between the user app and the OFS middleware. As shown in this figure, the average I/O latency



(a) Thread offloading (read)

(b) Procedure offloading (read)

(c) Thread offloading (write)

(d) Procedure offloading (write)

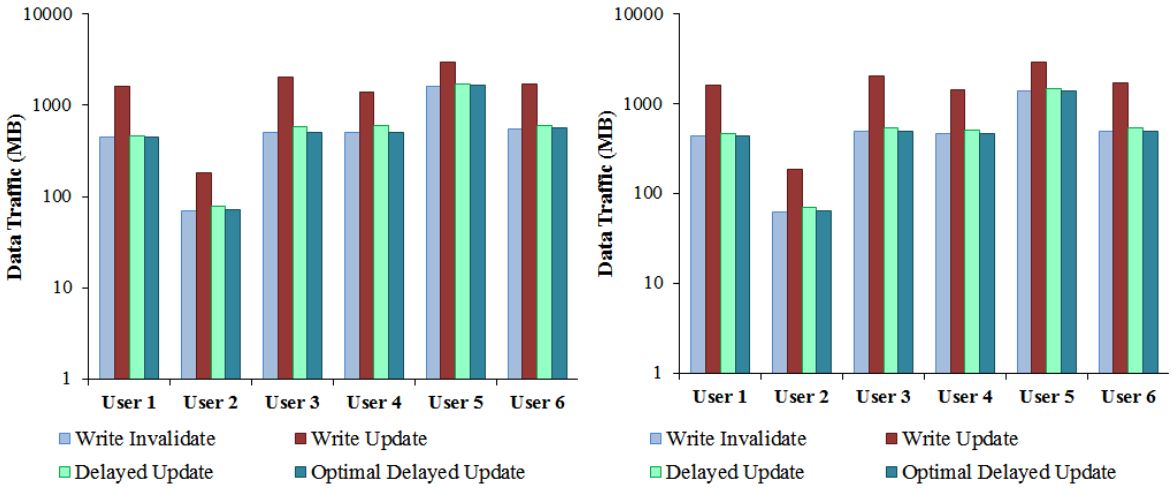
Figure 3.10 Average latency of read operations and write operations.

with delayed-update policy is lower than that with the write-update policy across all the workloads. Compared to the write-update policy, with the delayed-update policy, OFS can reduce I/O latency by 3% $\sim$ 28% for different workloads (21% on average). The delayed-update policy also incurs lower I/O latency than the write-invalidate policy for these workloads (6% $\sim$ 33% lower), except for the trace of user 1 in the thread offloading scenario (4% higher). Compared to the delayed-update policy, with the optimal delayed-update policy, the average I/O latency can be reduced by 7% $\sim$ 24% for different workloads. This shows that the performance of delayed-update policy can be further improved if sophisticated algorithms can be used to improve prediction accuracy.

The results also show that, in general, procedure offloading benefits more from OFS than thread offloading. This is because in procedure offloading, a bulk of I/O operations are migrated to the cloud together, where in thread offloading, different threads can run on mobile device and cloud simultaneously while accessing same file. This causes thread offloading to be more expensive in order to maintain consistency. Therefore, we conclude that offloading systems should implement procedure offloading in order to take full advantage of OFS.

To gain further insights into the behavior of OFS, Figure 3.10 shows the average latency for read operations and write operations for the two sets of workloads. As expected, write-update achieves the lowest read latency and the highest write latency due to its design of updating blocks for every write, while write-invalidate achieves the lowest write latency and the highest read latency due to its design of updating blocks upon read operations. The optimal delayed-update policy combines the advantage of write-update on low read latency and the advantage of write-invalidate on low write latency, with read latency and write latency close to those of write-update and write-invalidate respectively. Though the delayed-update policy cannot achieve such low latency limited by its prediction accuracy, it balances read latency and write latency well to improve overall performance. For these workloads, compared to write-update, on average it reduces write latency by 34%, at the cost of 18% higher read latency; compared to write-invalidate, on average it reduces read latency by 38%, at the cost of slightly increased write latency (11% higher). Relative to the optimal delayed-update policy, the write latency with the current delay-update policy is 7% higher and the read latency is 19% higher, indicating that the delayed-update policy tends to over-predict the arrival time of read operations.

We also measured the amount of network traffic with the two sets of workloads. Figure 3.11 shows that, as expected, the write-update policy incurs the most network traffic, while the write-invalidate policy incurs the least. Generally, the network traffic incurred by OFS (the delayed-update policy) is much less than that of write-update (67% less on average), and only slightly higher (3% $\sim$ 14%) than that of write-invalidate. Note that, with write-invalidate, updates are transferred only when they must be propagated to



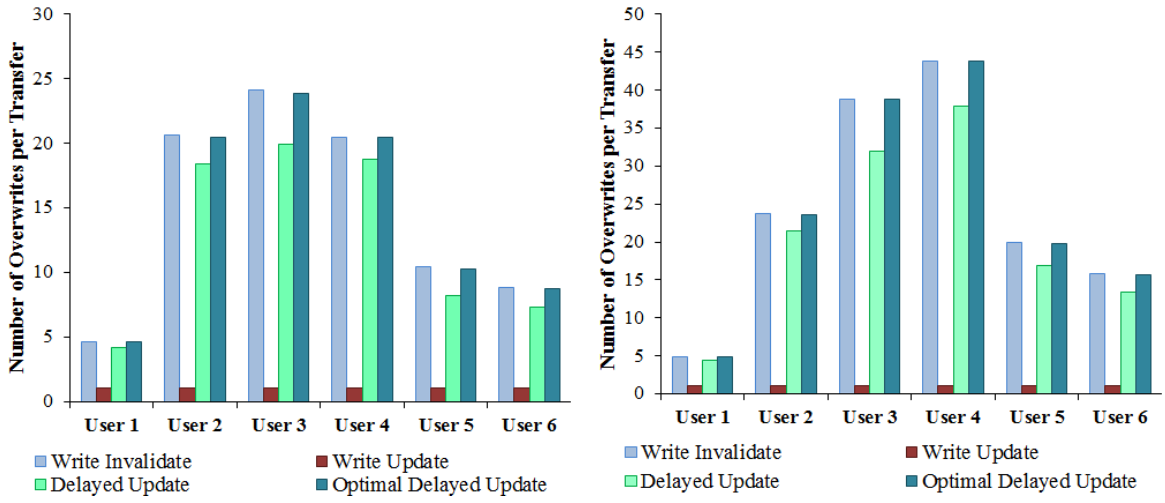
(a) Thread offloading **(b)** Procedure offloading

Figure 3.11 The amount of network overhead incurred by the workloads. The Y-axis is in log scale.

satisfy the requests for data. Thus, the network overhead can hardly be further reduced. This is mirrored by the network traffic incurred by the optimal delayed-update policy, which is also slightly higher than that with the write-invalidate policy by 1.2%. Let us also note that similar to the results for file I/O latency, procedure offloading leads to lower network overhead.

The results with file I/O latency and network traffic clearly demonstrate the advantages of OFS. It reduces the file I/O latency substantially compared to the write-invalidate policy, while maintaining a similar network overhead. The write-update policy performs poorly in terms of both average file I/O latency and network overhead.

To gain insights into the factors that lead to the OFS benefits, we collected the number of overwrites per transferred data block. As shown in Figure 3.12, a block may be overwritten multiple times before it is transferred. This is the reason why the policies except for write-update can effectively reduce the latency of write operations (Figure 3.10) and network overhead (Figure 3.11). This figure also shows that with OFS the average number of overwrites per transfer ($4 \sim 37$ times across different users) is only slightly lower that with the write-invalidate policy and the optimal delayed-update ($5 \sim 43$ times across different users). This explains why the latency and network traffic of write operations with



(a) Thread offloading **(b)** Procedure offloading

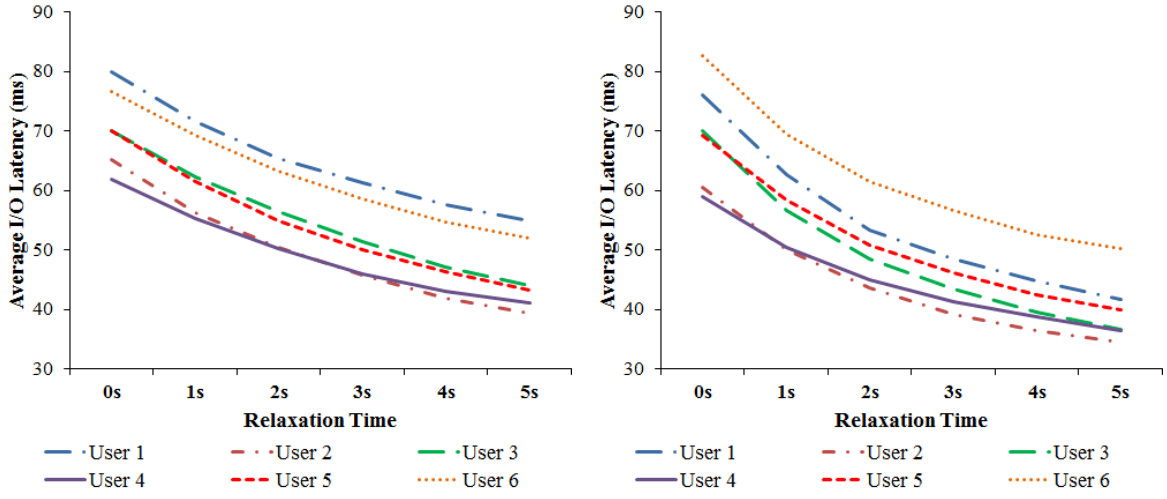
Figure 3.12 The average number of overwrites per data transfer.

OFS is slightly higher to that with the write-invalidate and the optimal delayed-update policy (Figures 3.10 and 3.11).

Figure 3.12 also shows that the number of overwrites is significantly higher for procedure offloading than for thread offloading. This result explains why procedure offloading performs better than thread offloading in terms of write latency and network overhead.

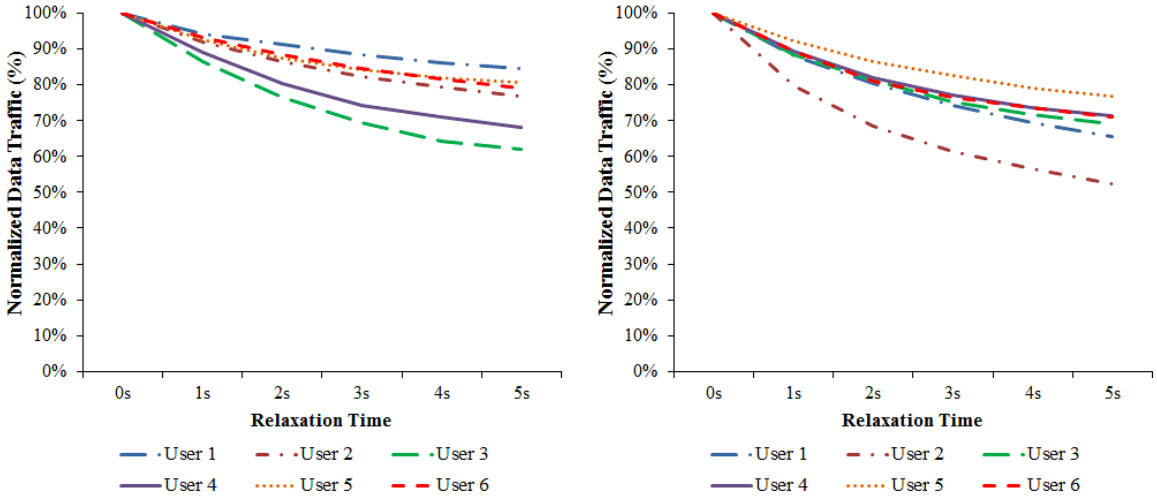
OFS supports weak consistency via *relaxation* (Section 3.2.3). Relaxation defines a period for each file block after its latest update, during which its content is considered to be valid no matter whether there are updates on other devices to the block in the middle of the period. To understand the impact of relaxation time on performance, we changed the length of relaxation time from 0 seconds (i.e., regular delayed-update with no relaxation) to 5 seconds, and measured the average I/O latency and total network overhead.

Figures 3.13 and 3.14 show the decrease of I/O latency and network overhead with relaxation time for different workloads. Since the network overhead varies significantly access different users, we normalized the overhead to that without relaxation for each user, and show in Figure 3.14 the normalized network overhead. When the relaxation time is increased to 5 seconds, the average I/O latency is reduced by 36% on average for the traces



(a) Thread offloading **(b)** Procedure offloading

Figure 3.13 Average I/O latency when the value of relaxation time is increased from 0 to 5 sec for delayed-update(OFS) consistency policy.



(a) Thread offloading **(b)** Procedure offloading

Figure 3.14 Normalized network overhead incurred when the value of relaxation time is increased from 0 sec to 5 sec for delayed-update(OFS) consistency policy.

of the six users with thread offloading and 43% on average with procedure offloading; the amount of network overhead is reduced by 25% on average with thread offloading and 32% on average with procedure offloading. The average I/O latency is reduced because the overhead associated with the latest update to each block across the internet is amortized by a larger number of read operations before the relaxation time expires. The amount of network overhead is reduced because multiple updates to the same file block on a

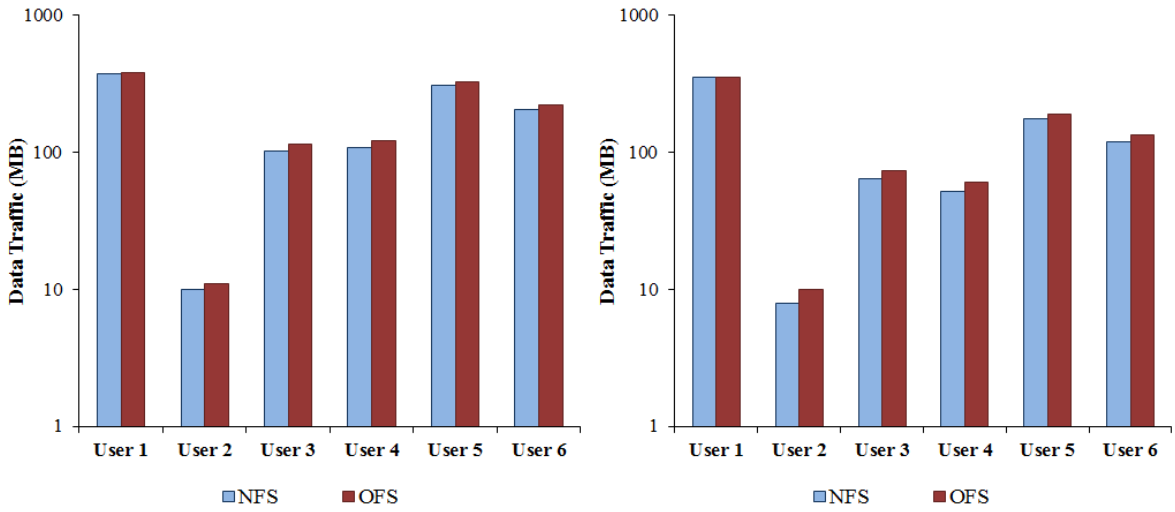
device can be consolidated and propagated together with one network transfer when the relaxation time expires.

The figures also show that, with the increase of relaxation time, though average I/O latency and the amount of network overhead keep getting reduced, the reduction becomes less prominent. The reasons are as follows. With the increase of relaxation time, the cost of propagating an update is amortized by an increasingly larger number of read operations, and thus the benefit from amortization is diminishing. At the same time, there are a limited number of updates to the same file block in a period; thus the number of updates that can be consolidated before the relaxation time expires may not keep increasing.

3.5.4 Comparison with NFS

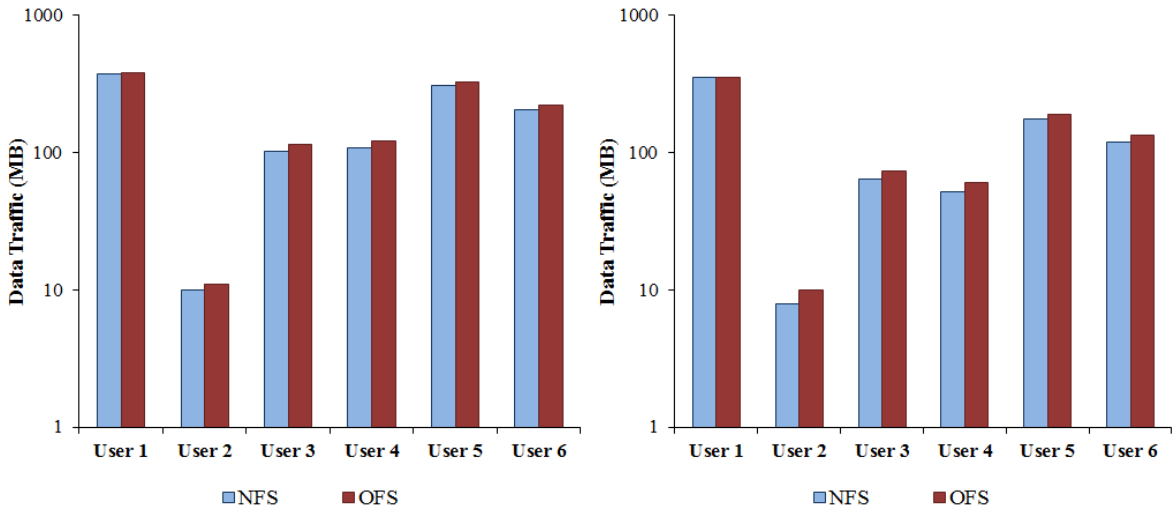
Unlike OFS, NFS uses close-to-open consistency. For this experiment, the emulator implements both delayed-update consistency policy of OFS and close-to-open consistency policy of NFS. The I/O latency induced by OFS during the emulation is lower than that of real implementation presented in Section 3.5.3. This phenomenon is caused by several factors: (a) unlike real implementation, the emulation does not take into account the delay induced by the IPC between the OFS middleware and the app which offloading task belongs to, and (b) in the emulation all network communication induces a fixed delay of 35ms, whereas in real implementation this value changes with the amount of the data transferred. Similarly, the total data traffic during emulation is slightly lower than real implementation. This is due to the fact that during the emulation, only data transferred by the I/O operations are considered, whereas in real implementation, all data transfer including state transfer during offloading of the task to the cloud and migration of the task back to the mobile, metadata transfer are considered.

Figure 3.15 shows average latency for read, write, and I/O operations for different workloads when NFS and OFS are used during offloading. Figure 3.16, on the other hand, shows total network overhead in similar condition. Form the figure it is clear that compared to NFS, OFS incurs on average 90% lower read latency and 29% lower I/O



(a) Thread offloading **(b)** Procedure offloading

Figure 3.15 Average read latency, average write latency and average I/O latency for six mobile users. Two consistency policies are considered: close-to-open consistency (NFS) and delayed-update (OFS).



(a) Thread offloading **(b)** Procedure offloading

Figure 3.16 Network overhead for six mobile users. Two consistency policies are considered: close-to-open consistency (NFS) and delayed-update (OFS). Y-axis is in log-scale.

latency across all workloads. This lower read latency is achieved at the cost of higher write latency (2 times higher). This is due to the fact that during read operations NFS reads the latest data from the server, thus resulting higher read latency. For the write latency, however, file lock is required to avoid conflict. During this period, all the write operations are essentially local to the client writing the file, thus lowering the average

write latency. Upon closing the file, all the updates are propagated to the server. This mechanism is essentially similar to write-invalidate which, as described earlier, induces lowest network overhead. As shown in the Figure 3.16, OFS incurs slightly higher (6% higher) network overhead compared to NFS. Unlike NFS, OFS transfers the data block right before they are about to be used at the remote end, thus the increase in the total network overhead.

3.6 Chapter Summary

Research described in this section has been driven by the demand for offloading mobile app tasks to the cloud. The chapter has identified one major obstacle to satisfying this demand, namely the lack of effective support to allow the offloaded tasks to access and share files with the rest of the app on the mobile device. To remove this obstacle, we have presented and implemented an overlay file system (OFS), which provides efficient, consistent, and location transparent access to files in a mobile cloud environment where app tasks could be executed at either platform. The experimental results based on real app and real mobile user traces have demonstrated that OFS can effectively support consistent file accesses from both the mobile device and the cloud and achieve substantially lower file access latency than competing methods. Furthermore, OFS is able to reduce the response time and energy consumption of mobile apps by speeding up the app execution through offloading support. As a result, the battery life of the mobile devices can be extended. Finally, we have learned that OFS works best for read-intensive apps, with few writes, and for systems that implement procedure offloading.

CHAPTER 4

A CONTEXT-AWARE FILE DISCOVERY SERVICES FOR DISTRIBUTED MOBILE-CLOUD APPS

According to a Cisco forecast [82], the amount of global mobile data traffic per month will reach 49 exabytes by 2021. Driven by the increasing demand for sharing and exploiting mobile data, mobile distributed apps enabling direct collaboration among users proliferate rapidly. For example, crowdsourcing using smart phone photos shared by users has played an important role in applications such as handling natural disasters [83, 84] and law enforcement [85]. This collaboration among the users allows distributed mobile applications (DMCs) to use the shared data files for their own computation. This chapter presents a Context-Aware File Discovery Service (CAFDS) that allows such apps to find and access files of interest shared by collaborating users.

The design of CAFDS addresses two major challenges. One is how to determine whether a file meets the feature requirements of a DMC app. CAFDS labels and then searches files based on implicit and explicit file features. A feature of a file can be the hash value of its content, its type, its size, user-generated tags, location and time for file creation, objects identified in an image file, keywords extracted from its text, etc. CAFDS starts with a set of predefined features (e.g., file size, type, location, etc.), and it allows apps the flexibility to add app-defined features (e.g., an image file contain faces). The other challenge is how to quickly locate the required files. The searching process is intended to serve the computation of a DMC app, and many such apps have low latency requirements (e.g., apps for disaster situations or interactive apps). To keep the search latency low, the metadata server organizes files into groups based on their feature similarity and structures the groups using an enhanced decision tree model [86]. Instead of searching through files one by one, CADFS locates a few groups where the required files are likely to be found, and then searches in those groups.

The rest of the chapter is organized as follows. Sections 4.1 and 4.2 discuss the distributed mobile-cloud apps and platforms and overview of CAFDS. Section 4.3 outlines the the API exposed by CAFDS. The detail design and implementation is discussed in Sections 4.4 and 4.5, respectively. The performance evaluation of CAFDS is presented in Sections 4.6. Finally, the chapter summary is presented in Section 4.7.

4.1 Distributed Mobile-Cloud Apps and Platforms

Distributed mobile apps leverage data from collaborating users to provide new and rich functionality for enhanced user experience. Consider a scenario where people take photos in Time Square in New York City on New Year's Eve. Some people use an app (referred to as *3D model creation app* for brevity) to enhance photos and create a 3D model of Time Square from the photos. Other people use a face recognition app (referred to as *Person-finding app*) to find people of interest based on the same set of photos. Both apps need to process a large number of images. The more photos they can access and process, the better results they can deliver.

Processing a large number of photos requires high computing power and consumes much energy, which mobile devices may not have. Thus, techniques are developed to offload intensive data processing workloads to the cloud. A number of mobile-cloud platforms implement such techniques for distributed mobile-cloud (DMC) apps [17–22].

Although the concept and design of CAFDS are generic and can be implemented on any mobile-cloud platform, we have implemented CAFDS to work on top of our Avatar [23] platform and its Moitree [10] middleware. In Avatar, each user has a virtual machine (called avatar) in the cloud working as the surrogate of her mobile device, which assists the execution and communication of the user's DMC apps. Specifically, a DMC app is executed on the set of mobile devices and avatars belonging to the group of users collaborating within the app. App components can be offloaded from mobiles to their avatars to speed up execution and save battery power.

4.2 CAFDS Overview

4.2.1 The Problem

A challenging yet unaddressed issue for DMC apps is how to quickly locate the required files and access the files with low overhead. This issue can be expanded into two separate problems for files known to the app and unknown files. For a known file that is available in the mobile of the requester, there could be a low-overhead copy of the file available at the VMs of other users in the cloud. However, it is difficult for the DMC app to determine whether such a copy exists or which collaborative user has the file. In the case of unknown files, the challenge is to find files that match the context requirements of the DMC apps at a potentially large number of collaborative users. A potential solution has to overcome three challenges.

- **Limited searching scope**: For example, the *3D model creation app* can only search the files of the users who installed the app on their mobile devices. However, there are other mobile users who have taken photos of Time Square and shared them through the *Person finding app*.

- **Redundant coding and searching efforts**: The searching code and searching are done in each app redundantly, even though the apps need to find the same set of files (e.g., the photos taken in Time Square on New Year's Eve for the two aforementioned apps). It would be better to implement this code as a system service used by all apps.

- **Potentially higher access latency**: For instance, a user takes a photo and then shares it with her friends, who upload it to their clouds. When an app, such as the *Person finding app*, needs to access the photo, since the computation is offloaded to the cloud, accessing a copy from the same cloud incurs lower latency than reading it from any mobile device or other clouds. However, the app is not aware of all the existing copies of the photos, even if the users are willing to share them.

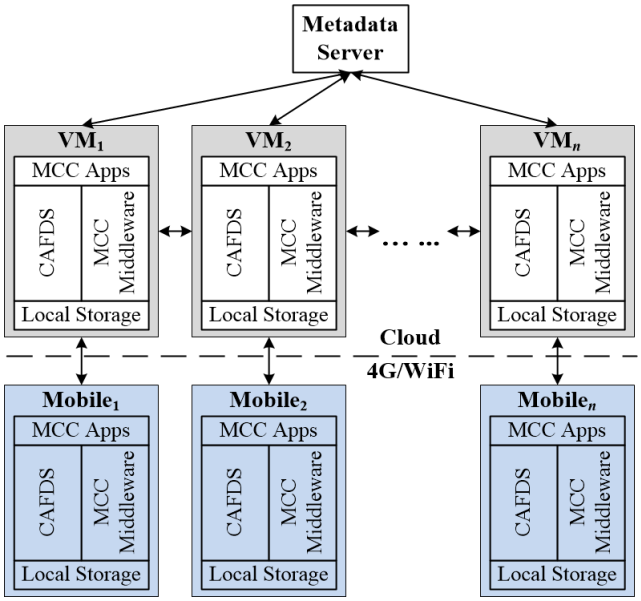


Figure 4.1 Architecture of CAFDS ecosystem.

4.2.2 CAFDS Functionality and Main Components

The key idea of CAFDS is to use a common service layer running on the devices and VMs of all the users who want to share their files with others. This layer indexes the files to be shared and their locations, handles file search requests from different apps, and responds with the files meeting the search criteria. CAFDS provides two types of support to satisfy the needs of apps for files with low overhead.

- **API support**: CAFDS provides API to apps to initiate requests for file search and retrieval.

- **Transparent support**: When an app in the cloud tries to access remotely a file that is currently saved in a mobile device, CAFDS involves tranparently to retrieve the file. CAFDS first predicts whether it is likely to find a copy of the file, which can be retrieved with a lower cost than retrieving it from the mobile device. If there is likely such a copy, CAFDS searches for and tries to retrieve the copy. The search is done based on the File ID, which is the SHA-256 hash value of the file content. Only when CAFDS determines that such a copy does not likely exists or cannot find such a copy, does it retrieve the file from the mobile device.

With the transparent support, the app itself does not need to call any CAFDS API methods in its program; CAFDS needs to monitor the file I/O operations of the app and automatically generate API calls required for file search and retrieval.

CAFDS runs an instance of middleware service in the mobile-cloud computing (MCC) middleware on mobiles and VMs, as shown in Figure 4.1. These middleware instances accept requests to share and search for files of interest. The requests are made by DMC apps through API calls. For a file search request, the middleware instance translates the API call into a set of operations and interacts with the metadata server to finish the search. For a request to share a file, it marks the file as *searchable* and collects the metadata information needed for indexing the file.

As the core component of CAFDS, the metadata server indexes files and actually handles search requests. It does not store file contents, which remain at their original locations, e.g., mobile phones and cloud storage. Instead, it saves a File ID for each file and other metadata information collected by CAFDS middleware instances.

4.2.3 File Features and File Contexts

CAFDS uses *file features* as a key concept. Apps use features to describe what files they need, and the metadata server uses features to organize file information. Features are based on facts about the files. Examples of valid features based on the location where files were created are: "the location is Time Square" and "the location is not Europe". When an app performs a file search, it needs to first provide a set of features. For simplicity, we refer to a set of file features as a *file context*. For example, in the *Person finding app*, the requested files need to have the following features: 1) file type is image, 2) creation location is Time Square, 3) creation time is New Year Eve, and 4) the file contains faces.

Apps can use pre-defined file features and may also define new features. In CAFDS, some file features are pre-defined based on the file metadata (e.g., size is larger than 1MB, type is JPEG, files created in July 2018, etc.). However, apps might be interested in additional features. Thus, CAFDS allows apps to define their own features.

CAFDS provides two methods for apps to define new features. If a new feature is based on file metadata saved at the metadata server, an app can call CAFDS API to specify what metadata (e.g., file size, type, etc.) should be examined, and the criteria for selecting a file (e.g., feature greater than a threshold or being of a specific type). If a new feature is based on file content (e.g., whether a photo contains faces), an app must provide the code to run on participants' mobile devices in order to extract the required file features from the file content. To prevent the execution of such code from violating user-privacy, the code must be developed based on template classes, that have been proven or tested to be safe, and must only call the functions from libraries that have been thoroughly tested to not contain malicious code.

4.2.4 Execution Flow of File Search

A file search request from an app is forwarded along with the file context through the middleware to the metadata server. The metadata server then identifies a group of files and their locations. The metadata server uses a set of file features to classify files into groups in order to speed up the search. Files in the same group have similar features. When the files and their locations have been found, the metadata server forwards the requests to the middleware instances located at the mobiles or VMs that have the requested files. The middleware instances are in charge of sending the files to the requester.

Finally, let us note that access control is an orthogonal problem to file search. We believe standard access control mechanisms in distributed systems can be applied to our scenario.

4.3 CAFDS API

CAFDS exposes a small set of API for file context management, metadata management and file search. CAFDS API is exposed to a mobile cloud app as an application stub. This stub is responsible for forwarding API calls from the application to the CAFDS middleware and handling their responses. The API follows an event-driven and callback-based asynchronous design. An API call sends a request to the CAFDS middleware

Table 4.1 CAFDS API: File Context Management API

Method	Description
getFeatureTemplateList( )	Returns a list of all possible feature templates.
getAllDefinedFeatures( )	Returns a list of all defined features based on feature templates from the metadata manager.
getFeatureTemplate(FeatureID fid)	Returns an object containing feature template from the metadata manager.
getFileFeature(FeatureID fid)	Returns an object containing feature definition from the metadata manager.
onRegisterFileFeature(AppID aid, List<FeatureID> featureList)	Method for registering features from a list, *featureList* to app with id, *aid*.
createFileFeature(FeatureID fid, Feature feature)	Method for creating new file feature with ID *fid* and feature values *values*. The definition of the feature, *definition* is registered at the metadata manager.
updateFileFeature(FeatureID fid, Feature updatedFeature)	Method for updating an existing file features with ID *fid*. It replaces the current feature with an updated feature *updatedFeature*
removeFileFeature(FeatureID fid)	Removes a file feature with ID *fid*.

which is either processed in the middleware locally or forwarded to the metadata server for further response. As shown in Tables 4.1, 4.2 and 4.3, the API is mainly designed for managing file features and file metadata and performing file/content search.

4.3.1 File Context Management API

The metadata sever uses a set of file features to manage file metadata, such that it can respond to the requests for files with these features. The features are pre-defined by CAFDS or created dynamically by apps based on their needs. CAFDS provides API calls (Table 4.1) to allow an app to examine these features, add/update/remove a feature, and claim the features to be used in their search requests.

Before an app uses CAFDS to search for files, it must use the method *onRegister-FileFeature* to claim a number of file features to be used in its searches. CAFDS ensures that the system is ready to respond to the search requests with these features, and is also adapted to serve all search requests with low latency. For example, if the features are new, CAFDS needs to collect and process required file metadata and organize the metadata information in a way to accelerate the searches. CAFDS maintains the organization before the app stops using the features and calls *removeFileFeature* to remove the features.

Table 4.2 CAFDS API: Metadata Management API

createMetadata(FileID fileId, Map<FeatureID, FeatureValue> fileContext)	Creates and returns metadata of a file with Id *fileId*, which is hash of the content, and file context, i.e., a collection of file features, and their respected values presented in *fileContext*
getUpdatedMetadata(FileID fileId)	Returns currently available metadata of file with id *fileId* from metadata server.
markFileDirectory(String fileDirectory)	Mark a file directory, *fileDirectory* searchable so that all files under the directory can be searched.
markFiles(Map<FeatureID, FeatureValue> fileContext)	Mark all files that have the features with certain feature values mentioned in *fileContext*.

To claims the file features to be used, an app can examine and reuse the features that are already being used by CAFDS. It may use method *getAllDefinedFeatures* to get all the features in the system and use *getFileFeature* to get the definition of a particular feature. It may also update an existing feature using *UpdateFileFeature*.

To support the creation of new features, CAFDS provides some feature templates in metadata server, which prescribe the definitions and semantics of new features. An app must select a feature template to define a feature. It may use method *getFeatureTemplateList* to list all available file templates from metadata server and method *getFeatureTemplate* to get a particular template.

4.3.2 Metadata Management API

Upon a search request, CAFDS searches the file metadata it manages for the files that can satisfy the request. File metadata is collected by CAFDS middleware instances on each mobile device, which provides an interface for user to select "sharable" files and collects the metadata information of these files. This can be achieved by calling method *markFileDirectory*, which marks all the files under a directory as "sharable", or by calling method *markFiles*, which marks a list of files "sharable". File metadata may also be reported to the metadata server by the apps generating files. This can be achieved by calling *createMetadata*. The method *getUpdatedMetadata* is for obtaining updated metadata. List of methods that fall under this category can be found in Table 4.2.

Table 4.3 CAFDS API: Search Related API

locateAndRetrieveFile(FileMetadata metadata)	Sends a file search request to the metadata server to locates a file and waits for the requested a file with FileMetadata, *metadata*. *metadata* contains id *fid* with file contexts *fileContexts*. If the *fid* filed is null, it searches for all the files that matches the contexts provided in *fileContexts*.

4.3.3 Search Related API

Method *locateAndRetrieveFile* (Table 4.3) is for generating a file request and sending it to the metadata server. The call blocks the calling thread, waiting for the files to be received. Searching criteria should be specified in parameter *metadata*. The parameter can contain only a file ID when the caller needs a file with specific file content (a file ID is the hash of the content). It may also contain some file features when the caller needs some files with the required features and the exact contents are unknown. The method uses a time out. Once a request is failed, a duplicate request is generated and the metadata server initiates a more general search, as long as the time-out value has not been reached.

4.3.4 Example Code

In order to show how CAFDS uses file context for file and content search, this section presents code snippets describing the process.

Code 4.1 shows the code executed by the app for presence of a face in an image file as a file feature and registering it with the metadata server. Line 3 shows how an app can request a feature template from the metadata server using feature ID. The app can get a list of all feature templates, thus their IDs using *getFeatureTemplateList( )* method (Table 4.1). Lines 4–9 show how file type, feature type and values can be set using *FeatureTemplate*. Finally, line 11 –27 shows how a new feature can be defined. The *match* function presented from 13–25 is used for how to extract the face and set feature value depending on the presence of the face. This function is used by OFS middleware during feature extraction and set feature value.

```
1 String aid = getApplicationContext().getPackageName();
2 ...
```

```
3  FeatureTemplate featureTemplate = CafdsStub.
       getFeatureTemplate(ImageObject);
4  String featureValueType = featureTemplate.getfeatureValueType
       ();
5    if(featureValueType.equals("Boolean")) {
6      List<FeatureValue> featureValues = new ArrayList<>();
7      featureValues.add(new FeatureValue("Boolean",true));
8      featureValues.add(new FeatureValue("Boolean",false));
9      featureTemplate.setFileType("Image");
10     ...
11     Feature feature = CafdsStub.createFileFeature(
           featureTemplate, featureValues) {
12     @Override
13     public boolean match()  {
14       boolean facePresent = false;
15       /* Run face detection on the file
16       content and set facePresent if a
17       face is present. */
18       if(facePresent) {
19         this.matchedFeature = featureTemplate.getValue(0);
20         return true;
21       } else  {
22         this.matchedFeature = featureTemplate.getValue(1);
23         return true;
24       }
25     }
26   }
27 };
28 ...
```

Code 4.1 Defining presence of face in image as new feature with CAFDS API.

Once the required features are defined, CAFDS can register list of features or a file context to an app for faster execution. Then it need to define how the app want to mark its shareable files. The process is described in Code 4.2. In the code snippet, Line 4 shows how to register a list of file features to an app. Lines 6–7 shows how an can use a specific app directory to mark the shareable files. Alternatively, it can use *markFiles( )* to mark the required files using certain features and certain values.

```
1 ...
2 List<Feature> featureList = new List<>();
3 ...
4 CafdsStub.onRegisterFileFeature(aid, featureList);
5 ...
6 String appDirectory = ...
7 MarkFileDirectory(appDirectory);
```

Code 4.2 Register features to an app and mark shareable files.

Code 4.3 shows how to use *locateAndRetrieveFile( )* to search for files based on predefined file features. The file search requires a metadata describing the context or list of required features of the search. Lines 4–10 shows how the app can define a metadata using app ID and a map containing file features and their values for the search. If a file Id is included in the metadata definition (Line 7), it uses the context defined in the metadata to search for same/similar files. The actual search process is described in Line 11.

```
1  ...
2  feature = ... // Assume feature is defined as presence of a
       face
3  ...
4  Map<FeatureID, FeatureValue> fileContext = new Map<>();
5  fileContext.add(feature.getFeatureId(),feature.getValue(0));

6  ...
7  FileMetadata metadata = new FileMetadata.Builder();
8                  .appId(aid);
9                  .fileContext(fileContext);
10                 .build();
11 File file = CafdsStub.locateAndRettrieveFile(metadata);
```

Code 4.3 File search and retrieve in CAFDS.

4.4 CAFDS Design

Figure 4.2 shows the internal modules in CAFDS middleware and CAFDS metadata server, as well as the interaction between these modules. The figure only shows one instance of CAFDS middleware on a VM running in the cloud. There are other instances running on mobile devices and other VMs, as shown in Figure 4.1. Since these instances have similar modules and interact with the apps and the metadata server in the same way as the instance included in Figure 4.2, they are not shown to save space.

4.4.1 CAFDS Middleware Design

As shown in Figure 4.2, a CAFDS instance consists of two layers. The upper layer is implemented as an **application stub** and embedded in each app through either compilation (for lower overhead) or AspectJ [70] (for increased compatibility). It exposes CAFDS API to the app for make file search requests. It also intercepts file I/O operations

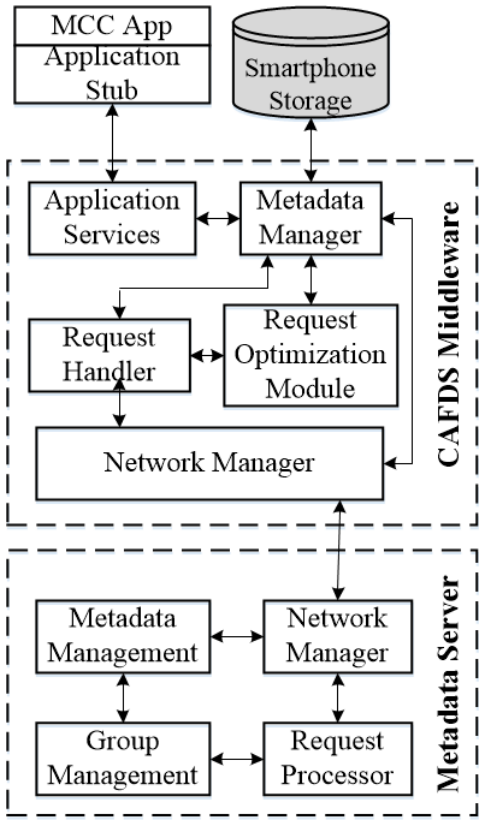


Figure 4.2 Context-Aware File Discovery Service (CAFDS) design.

made by the app to remotely access files located on mobile devices, generates requests to find other copies of the files based on file IDs, and retrieves the file copies with lower overhead.

The lower layer of CAFDS instance is implemented as a middleware service, called **CAFDS middleware**. It 1) forwards requests to the metadata server, 2) extracts features from local files for easier search, 3) matches the features from a file request to those from the sharable files available locally and sends appropriate files to other instances to satisfy their file needs, and 4) receives files and forwards them to apps to satisfy app requests.

The CAFDS middleware consists of five major modules, which are introduced as follows. The **application service** is the interface of CAFDS. It communicates with application stubs to receive file search requests and send responses to the application stubs when they are processed. It also interacts with the MCC middleware executing the app to receives notifications about the status changes of DMC apps (e.g., task migrated to/from

the cloud). The **network manager** handles communications between the middleware instances on different devices and the communications between a middleware and the metadata server.

The **request optimization module** performs key optimizations for transparent CAFDS support. In the cloud, when an app tries to access a file that is currently saved in a mobile device, the request optimization module predicts whether it is likely to find a copy of the file, which can be retrieved with a lower cost than retrieving it from the mobile device. If such a copy is likely to be found, the CAFDS middleware generates a file search request to retrieve the file. Also, in order to respond to a search request, CAFDS middleware needs to extract features from the users' shareable files. This module predicts which files are likely to be requested and extract features from them in advance to minimize the search latency. The search requires the ID of the file, which is the hash value computed based on the content of the file saved in the mobile device. Computing the hash value of a file on-demand on the mobile device causes significant delay, making retrieving a low-overhead copy of the file less efficient than retrieving the copy on the mobile device. In CAFDS, the hash values are pre-computed before the app is launched. The request optimization module on a mobile device is responsible for predicting which app is to be launched next and which files they will access, so that the IDs of these files can be pre-computed.

The request optimization module uses Viterbi algorithm [87] to make predictions, which is a dynamic programming algorithm used for finding the most probable hidden sequence for an outcome based on historical data. On a mobile device, the request optimization module uses the historical data of the latest 100 app launches to predict which app is to be launched next; and uses the pathnames of the files accessed by up to last 100 executions of an app to predict which files an app may access in its next execution.

In the cloud, for each app, the request optimization module monitors the latest 100 predictions to establish a relationship between the prediction failure rate and the average overhead of file retrievals. A prediction fails when a low-overhead copy is predicted to be

likely existent but cannot be found. If this module is too optimistic and almost always predicts that low-overhead copies exist, both the failure rate and the average overhead are high. If it is too pessimistic and mostly predicts that low-overhead copies do not exist, the failure rate is close to 0; however the average overhead is also high due to the high overhead of retrieving the files from mobile devices. The request optimization module predicts in a way to maintain the prediction failure at a level that can minimize the average overhead of file retrievals. This module also uses the last 100 search requests to predict which files are likely to be searched and extract features from them in case there are any new and updated feature(s) available for these files.

The **metadata manager** is responsible for collecting file metadata including file IDs on each mobile device or cloud entity. It maintains a local copy of the metadata, such that it can report only metadata changes to the metadata server. It collects file metadata when users mark some files to be "sharable" or when the request optimization module predicts that some files are to be accessed. To reduce costs, it updates the metadata only when the files have been modified since last time it collects the metadata on these files.

Finally, **request handler** handles various requests generated in CAFDS, such as requests for registering/updating file features and searching for files, and deliver the responses. Specifically, it forwards to metadata server the requests that are generated locally, and delivers the responses to the application service. It also responds the requests from the metadata server by forwarding them to local metadata manager and forwards the responses to either the metadata server or the corresponding CAFDS middleware that initiated the search requests.

4.4.2 Metadta Server Design

The metadata server is a central component for processing requests made by different apps aon different devices. The reason behind the central design is to improve efficiency by decreasing the number of network hops in processing a file request. As shown in Figure 4.2, the metadata server has four major components: (a) network manager, (b) metadata management, (c) group management, and (d) request processor. The **network**

manager is responsible for the network communication with the CAFDS middleware instances in the cloud. The **metadata management** serves as a central storage for the metadata. It uses a hash table to manage file metadata with each entry being the metadata of a file and the key being the file ID[1]. Each entry includes traditional file metadata (e.g., file size, file type, the time of creation, etc), various file features, and file locations. The content of an entry changes over time when new file features are added to the systems or stale file features are removed. The **request processor** manages the requests (e.g., file search requests and requests for updating file features and metadata) with a request queue and forwards them to metadata management or group management, where they are actually processed.

The **group management** classifies and organizes the files into file group based on their features. The classification is done to significantly reduce search complexity, since the number of files can be huge and it is not realistic to go through all the files one by one. After classification, the files in the same group are roughly homogeneous when evaluated with various searching criteria (i.e., file contexts). Thus, instead of checking all the files one by one, searching is done much more efficiently by first locating file groups and then examining the files in the groups. An alternative approach would be grouping users based on the similarity of the files they own, as some P2P file sharing systems do. However, our experiments shows grouping the files based on their similarity yields to more efficient search.

The group management design addresses two key issues to support efficient file search. The first issues is how to classify files. A few facts make it challenging: 1) there are various types of features; 2) the value sets of some features (e.g., file sizes, creation time, and location of origin) have huge cardinalities and some feature can be added or updated dynamically; and 3) searching criteria are highly diverse. The second issue is

[1]Though a file ID is the hash value of the file content, it is not updated every time when the file is modified. Instead, it is updated lazily when a CAFDS app is predicted to access the file. Also, most files managed by CAFDS are read-only. Thus, frequent changes to a small number of files will not significantly increase management overhead.

how to organize and search file groups efficiently. This is important since the number of file groups can be large.

To address the first issue, group management uses a two stage clustering method, which is similar to the one used in [88] and combine two unsupervised clustering algorithms, Self-Organizing Network (SOM) and k-means clustering. SOM is mainly used to handle noise and outliers and to determine the number of clusters k, which is later used in k-means clustering to classify files again into k groups. Combining these two methods leads to more robust results.

For efficient file group organization and search, CAFDS implements and enhances an ID3 decision tree in group management, and use it to organize and search file groups. ID3 is a supervised learning method that partitions a set of instances (e.g., files in CAFDS) into homogeneous subsets (e.g., file groups in CAFDS), i.e., subsets that contain instances with similar values. An example of a group organization based on decision tree is shown in Figure 4.3. The root of the decision tree represents the whole set of files. Each child node represents a subset of the files of the parent node, classified using a certain file feature. For example, in the figure, the first child of the root represents the files classified using the file feature "file type is image". The leaf nodes of the decision tree represent file groups and contain pointers to the file entries in the hash table in metadata management, such that the information of the files can be located.

Like the traditional ID3 decision tree, we used *Information Gain* [86] as a measure of the homogeneity of the subsets. The idea behind the decision tree is to use a series of features to identify a group that yields to minimal uncertainty based on a training dataset. It selects the features that can provide highest information gain, i.e., can categorize the highest number of files correctly first and then remove the feature from available feature list. This process is continued until no feature can be selected.

In this example, the file type of the request is identified as the feature with highest information gain so, it is chose first. Then for images, it the process until no feature can be selected. The resulting groups represents groups containing similar files.

Once correct group(s) is identified, the request is forwarded to all VMs that contains files from that file group, who responds to the request (Section 4.4.1). If no clear group can be identified, we generalize the search by removing a feature and identify all possible groups by going up one or more level in the decision tree. To increase the possibility of finding a file, we continue the process until we select the nodes in top one-third of tree length before it is decided that the requested file is not present.

After each search, it collects the information regarding the success of the search from the requester VMs which is used for calculating the success rate for serving the request correctly. We use a threshold, Th which sets a minimum boundary for success rate. Once this threshold is reached, it triggers the group reconstruction to reclassify the file groups based on updated metadata values collected during the file search. This, in turns, leads to the reconstruction of the decision tree.

We use decision tree as a supervised learning method for few reasons. Firstly, classifying an unknown sample based on their features using decision tree is very fast. This allows us to reduce the overhead required for file search significantly. Secondly, the dynamic nature of how the features are added and updated prevents us from using more sophisticated methods like support vector machine (SVM) or deep learning, as they require a fixed number of features to be present. Furthermore, methods like deep learning requires a large number of features present for using it successfully, which might not always be the case under current circumstance. While we can limit the maximum number of features our system, thus having a fixed length feature vector for search, this would limit the number of the features allowed in a system. In order to achieve more versatile app design, we choose not to limit the number of features. Also, setting this number very high might lead to a large number of feature vector empty thus increasing both request size and overhead for the search.

Another alternative option is to use random decision forest or simply random forest [89] for processing the file search instead of a decision tree. As the number of features is not fixed, there is a possibility that at some point the number features becomes less than the size of the random sample in order to implement the random forest. Under

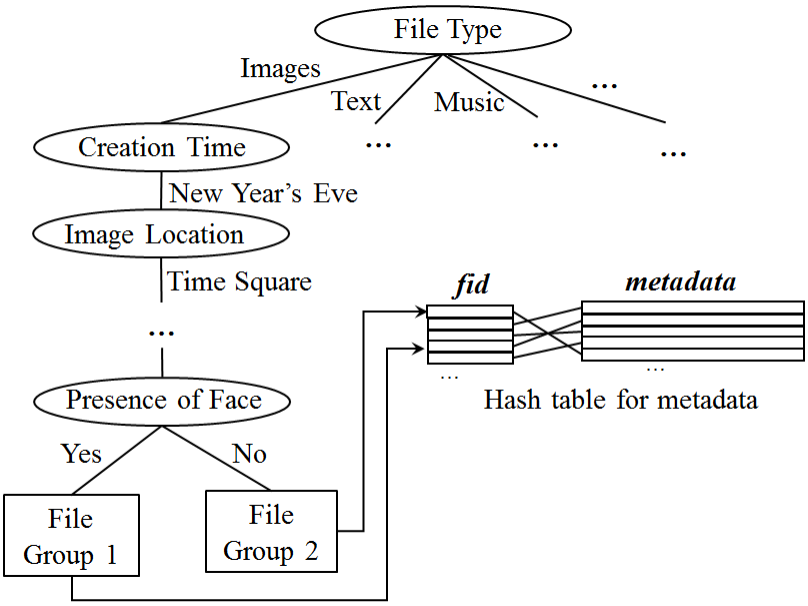


Figure 4.3 Example of a group organization in CAFDS.

such circumstances, a decision tree performs better than the random forest. Thirdly, even if all the features of a file is not present during a search request, a decision tree can identify all possible file groups more effectively.

In order to increase the probability of identifying the correct group, we made some changes to the traditional decision tree. Unlike the traditional decision tree, we keep all the branches attached to the node if more than one feature has same information gain. As the file request might not contain all the file features, keeping all the branches allows a higher margin of success. Also under such circumstance, it is possible that no clear file group can be identified. The traditional ID3 algorithm returns a leaf node (file group) with highest information gain, i.e., the lowest amount of uncertainty in this case. However, we forward the search request to all the file groups and add the file to all possible groups. After the sufficient data is collected, the file is categorized into the correct group. With these minor changes, the accuracy of the decision tree for classifying the file group is improved by approximately 6%.

Normally, other than post-pruning, no updates are made to the decision tree. As new features can be added or a feature can be updated or removed by apps at any given time, we proposed an update scheme to the decision tree to incorporate them. Traditionally in

Algorithm 1: Update decision tree

```
1  updateDecisionTree(commandType, feature)
2      if (commandType = REMOVE_FEATURE)
3          for (All leaf nodes)
4              if (isIncludedInDT(feature) = TRUE)
5                  removeNode(feature);
6      else if (commandType = MODIFY_FEATURE)
7          if (isNewFeature(feature) = FALSE and
             isIncludedInDT(feature) = TRUE)
8              node = findNode(feature);
9              constructDT(node);
10         else
11             gain = infoGain(feature);
12             if (gain ≥ informationGain(root))
13                 groupReconstruction();
14                 constructDT(root);
15             else
16                 for (All nodes starting with leaf nodes)
17                     if (gain ≥ infoGain(node) and
                         gain ≤ infoGain(parent(node)))
18                         insert(feature, node);
```

such cases, the entire decision tree is reconstructed. In our current case, adding a feature to one type of file (e.g., image file) might not affect other types of file (e.g., text file). Also, if the update does not contain a feature that is on a root-leaf path or a feature near the leaf on a root-leaf path, the change in the tree might be minor. Considering these issues, we propose a heuristic to update decision tree, presented in Algorithm 1.

The main idea behind the update is to modify only the affected part of the tree. Lines 2–5 indicates that in case of feature remove request, if the feature is in root-to-leaf

path of the decision tree, then remove the feature and reconstruct the tree using function *removeNode*. Otherwise the remove request is ignored by the decision tree. In the case of modifying a feature (for both add a new feature and update an existing feature), if the feature is in a root-leaf path, then recalculate the decision tree starting for a subtree with the node representing modified feature as the root (Lines 6–9). If the file feature is a new feature, and it has highest information gain then initiate group reconstruction to reclassify the collected data and recreate the decision tree (Lines 12–14). Otherwise, the node can be inserted in the middle of the tree. Find a node such that information gain of the feature is higher than the that of the node but lower than that of the parent of the node(Lines 16–17). Then insert the node and construct the subtree (Line 18).

4.4.3 Scalability of CAFDS

As CAFDS responds the file requests to a large number of nodes, scalablity is a plays a huge role in the design. To make the CAFDS more scalable, we focus on two factors: (a) fast processing of the requests in metadata server, and (b) how make the response to files faster by the nodes who have the file.

Metadata server is a central element, therefore it can be a bottleneck for the design. The decision tree handles requests one-by-one. Therefore, to improve the performance of the metadata server, we divide the decision tree in group management vertically and divide the hash table managed by the metadata management accordingly. Multiple subtrees of the the decision tree and relevant hash tables can be placed in same server or different server based on their loads. This enables the decision tree to server multiple requests at the same time.

Let's assume, AR_{DT} and PT_{DT} is the arrival rate of the file search requests to the metadata server and the processing rate for each request in the decision tree in a time period T respectively. pt_{node} is the processing time of a file request in a node in a time period T and n_{node} is the number of requests processed by a node. Also, len_{qP} is the percentage of the queue that is filled with requests, N_T is total number file search requests arrived in metadata server in a time period T and H is the height of the decision

Algorithm 2: Decision tree division algorithm

```
1 divideDecisionTree(root)
2     if (PT_DT/AR_DT ≥ 1 and len_qP ≥ T)
3         for (i = 3 to 3 * H/4)
4             for (each node, n at level i)
5                 if (n_node ≥ n_parent(node) and n_node ≤ max(n_child(node)))
6                     Do nothing;
7                 else
8                     divide(n);
```

tree. Each node in the decision tree maintains updated pt_{node} and n_{node}. To figure out when to divide the decision tree we use Algorithm 2. For the decision tree, Check whether PT_{DT}/AR_{DT} is greater than 1 and len_{qP} is higher than a threshold, T (Line 2). If yes, then check whether the the number of requests processed by the node during time period T (n_{node}) is higher than that of its parent and less than that of child with highest number of request processed (Lines 3–5). If a node if failed to pass the test, then divide the tree using method *divide* (Lines 6–8).

Logically, the decision tree could be divided multiple times. However, we cannot divide the decision tree vertically more than once to minimize potential delay for communication between different subtrees of the decision tree. This might lead to a scenario when one subtree handles more request than others. To handle this situation, we replicates the subtrees of the decision trees as required. For any subtree, dt, if PT_{dt}/AR_{dt} is higher than 1, we create another replica of the subtree dt. Then based on feature value in the file request for the feature at the root of the tree, we decide whether the original subtree or the replica will handle the request. When a feature is updated or modified, we apply the update algorithm at the leaf of the associated decision tree and leaf node associated with DT_{root} if necessary.

To maintain fairness and avoid a user from processing a large number of requests compared to other users, metadata server issues ticket a file to a user if more than one user

contains the file. If a user holds the ticket for a file and a file search request matches the file, then the user responds to the request. For fairness, the metadata server categorizes the users into three groups with different priorities to respond to a request: low priority, medium priority and high priority. The group is divided based on number of file requested and number of request responded by the user. If a user responds to a low number of requests but requests a high number of files, it is placed into high category group. If the number of requests responded and number of file request made by the group is similar, it is placed into medium category. If the number of request responded by the user is significantly higher than the number of request made, then the user is placed into low category. When assigning a ticket, the metadata server uses priority-based scheduling. If more than one user have same priority, then the ticket is issued to the user who holds lower number of ticket.

4.5 Implementation Details

CAFDS middleware sits between the DMC app and offloading middleware and runs on both mobile of the user and its surrogate running on the cloud. It collects the file requests from the DMC apps and forwards the file request to metadata server running on the cloud or the mobile of the user depending on the request. Both CAFDS and metadata server is implemented using Java on Android OS and Linux OS respectively. CAFDS uses a NIO-based TCP library called Kryonet [74] for network communication and Android's binder mechanism for communicating between DMC app and CAFDS middleware. Although the implementation of CAFDS middleware is based on Android, implementation techniques are generic and can be implemented on other OSs.

CAFDS is an event-driven system that allows two types of events: CafdsEvent and CafdsMessage where CafdsEvent represents events related to CAFDS maintanance and internal data flow and CafdsMessage contains request form an DMC app and their response from CAFDS middleware.

The metadata server is implemented as a collection of threads. The metadata management uses radis [90]for storing the metadata. Redis is a in memory store that

provides persistent storage and can be configured for both single node or a cluster. The group management uses multiple threads to implement unsupervised classifiers (SOM and k-means clustering) and different parts of the decision tree. This design allows group management to handle multiple request in parallel. These threads communicates with both metadata management and request handler to process the data which runs on different thread. The request handler implements a request queue that stores incoming events from different VMs. Depending on the load of the server, group management and metadata management can run on multiple servers. This allows CAFDS to maintain some degree of availability and load balancing.

CAFDS middleware handles file search requests rather than updating the file, it is implemented on application level. Because of this, it can only extract file features and share files from files available on external (public) storage rather than the files stored on internal (private) storage of an app. If an app wants to share a private file, it needs to make the file public.

We implemented CAFDS middleware as a set of application services. We marked these services as "sticky" to ensure that they are restarted automatically in case they are killed. The metadata of the files and historical data such as search history and application history are backed up periodically to prevent the loss of application data. The CAFDS API is exposed to the mobile apps via an application stub. The application stub can also intercept the I/O requests made by the DMC apps using AspectJ [70] to identify a file request. This allows apps to search for specific files without any change in the application code.

4.6 Performance Evaluation

We have implemented a prototype of CAFDS in Android and Linux, and compared its performance against Chord [39], a DHT-based distributed lookup scheme, and SPOON [34], a P2P-based file sharing system. SPOON's lookup scheme has some similarity to CAFDS. When looking up a file, it first selects a group of users who may own files similar to the desired file and then looks for the file within the group. We also

compared CAFDS with OFS [91, 92], an overlay file system for mobile-cloud computing. OFS gives us a baseline performance to measure improvements.

4.6.1 Experimental Settings

The experiments were conducted on a Nexus 6 smartphone running Android 7 and Android x86 VMs running Android 6. The phone was used as the mobile device where apps are first launched. The Android x86 VMs are hosted in an OpenStack-based cloud, and are used to accept the computation offloaded from the phone. Each VM has 2 virtual CPUs, 3GB memory and 16GB storage. These VMs are hosted on 8 physical machines each of which has an Intel Xeon (E5-2630) CPU, 78GB memory, and 2TB storage.

CAFDS middleware is installed on both the mobile device and the VMs. The metadata server runs directly on the Linux OS of a physical machine. To drive CAFDS, we generate synthetic traces, play the traces on the VMs, and measure the end-to-end latency of file searches. This traces are replayed by a separate app that runs on the VM and sends the file requests to the CAFDS middleware running on VM.

4.6.2 Trace Generation

The synthetic traces include upto 100K file requests from 18 different apps for 50 different users within 24 hours. The applications and file types used by these applications and their I/O pattern from BIOtrace [93] to generate the required I/O traces. This information was used together with I/O operation distribution from the PhoneLab trace [79] to generate two 24 hour trace for I/O calls from different apps for 50 different users. Each of these traces has I/O operations from all 18 applications where we assume the apps are executed at least once in a random sequence. To emulate task offloading scenario, we assume that first 30% I/O operations are executed on mobile device, next 50% on the cloud and remaining 20% on mobile. Based on the I/O operations running on the cloud, we derive the trace for file request for each user.

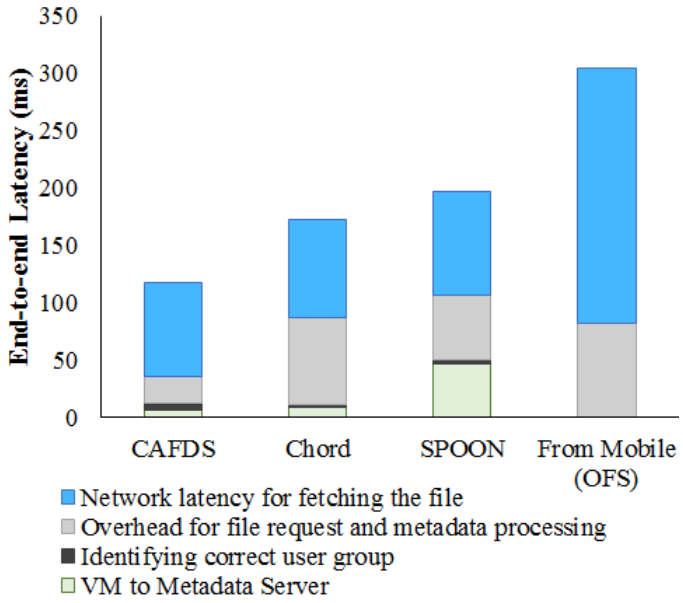


Figure 4.4 Decomposition of average end-to-end latency for CAFDS, Chord, SPOON and fetching the file directly from mobile (using OFS).

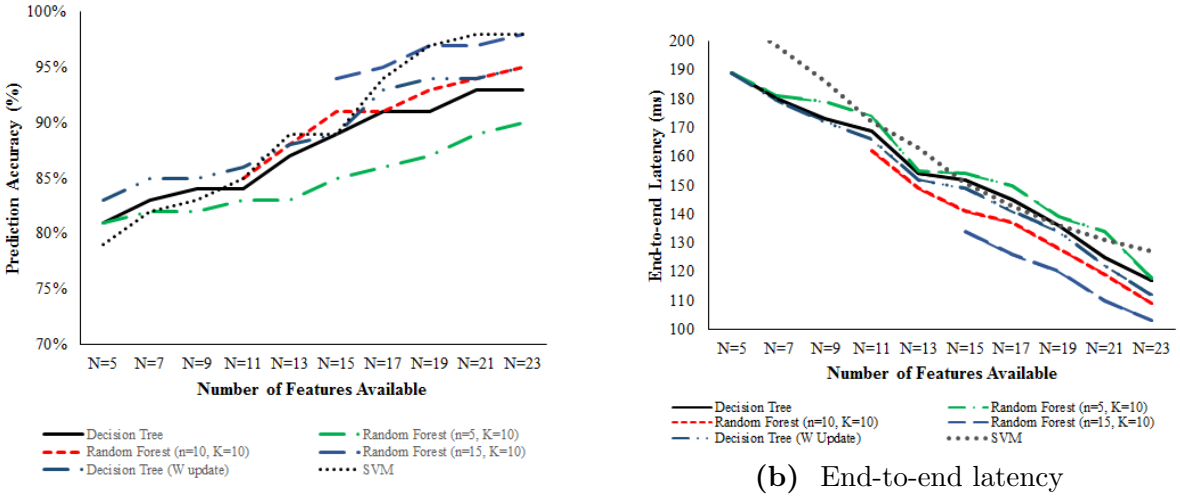
These traces has at least 15% of files that are used only by a single user him/herself, 30% files that was used by at least two users, 25% files that was used by at least three users, 20% by at least four users and 10% by five or more users.

To support the file requests, we populated the metadata server with the information of 2,500 files of 4 file types (text files, images, video files, and audio files, with 625 files in each type). In addition to 7 conventional file features, such as file name, size, location and time of file creation, etc., we also included 16 different types of file information, which include image context, quality, photo tag, histogram, presence of an object/face for images and video, speaker, lyrics and keywords, key points for music, and keywords and user tags for text files. We generated the files with random content and distributed them randomly on the VMs. For search requests that need to look into file content (e.g., searches for the photos containing a certain object), we included the features required by the searches in the metadata of the files. Thus, CAFDS can answer the requests in the experiments without actually checking the file content.

4.6.3 Experimental Results

To verify the effectiveness of CAFDS, we first compare the average end-to-end latency of CAFDS with that of Chord, SPOON. We also compared their performance against fetching the required directly from the mobile using OFS. Figure 4.4 shows the decomposition of the end-to-end latency for each of the systems. From the figure it is clear that the end-to-end latency for searching a file in CAFDS is substantially lower than that of other systems. CAFDS decreases end-to-end latency by 31% and 39% compared to Chord and CAFDS respectively. The figure also shows that the time spent on network communication contributes a major part in the total latency, which consists of three major components: latency to send the file request from VM to metadata server (CAFDS) or VM to first node in the lookup scheme (Chord) or community leader (SPOON) (labeled as "VM to Metadata Server" in the figure), latency for the file request and metadata processing (labeled as "Overhead for file request and metadata processing" in the figure), and latency for fetching the file from its current source (labeled as "Network latency for fetching the file" in the figure). As expected the DHT-based lookup scheme Chord spends the least amount of time on computation (labeled as "Identifying correct user group" in the figure) but the highest amount of time on communication since it needs multiple network hops to locate the required files. The amount of time spend by SPOON on computation is lower than CAFDS because of its relatively simple lookup scheme. However, due to its complex paths for forwarding search requests and retrieving files, the time spent on processing requests and the time spent on fetching files are both higher than those with CAFDS. Our experiment also shows that all three of these systems have lower end-to-end latency compared to when files are fetched from the mobile directly. The latency of fetching the file from the mobile is the highest, and it is dominated by network communication (i.e., WiFi latency is higher than latency within the cloud data center).

An alternative design of metadata server is to use a random decision forest and SVM for classification, instead of a decision tree. These alternatives increase design complexity and incurs higher overhead (e.g., space overhead and the cost spent on updating the classfier), compared to using a single decision tree. In the experiments, we implement a



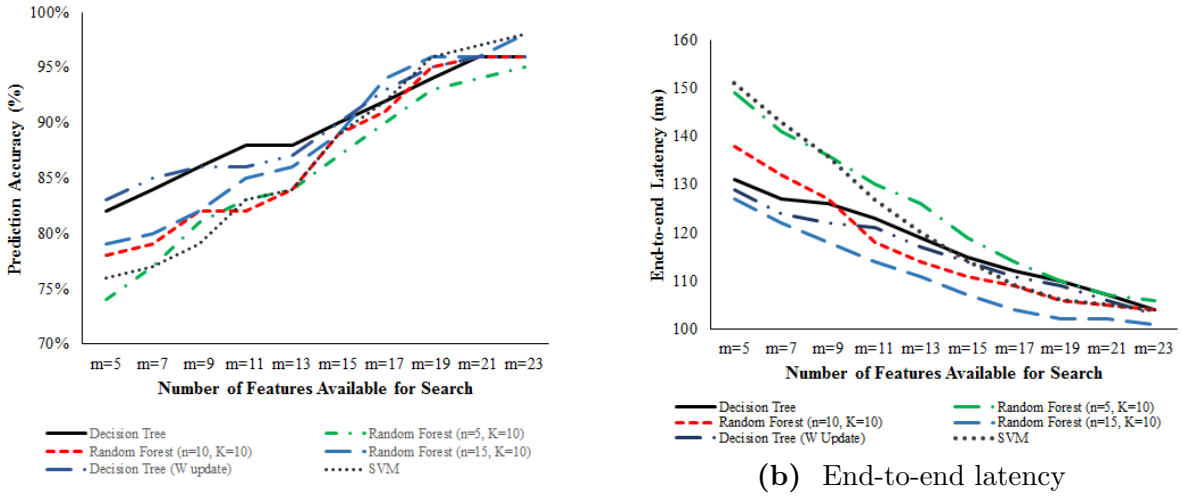
(a) Prediction accuracy

(b) End-to-end latency

Figure 4.5 Effect of increasing number of file features during the construction of decision tree, random forest and SVM: (a) prediction accuracy, (b) average end-to-end latency.

CAFDS variant with a random forest consisting of 10 trees ($K = 10$). Then, we compare the CAFDS performance with one decision tree against that with a random forest for different file feature counts. We gradually increase the number of file features from 5 to 23. We also repeat the above experiments for different configurations of the random forest by changing the file features for each tree in the forest (denoted with n) from 5 to 15. Finally, we compare the performance of our design with support vector machine (SVM).

To evaluate the effect of increasing number of features (N), we first increase number of features used for constructing the classifier (e.g., decision tree, random forest or SVM) (Fig. 4.5). As shown in the figure, the performance with one decision tree is similar to that with a random forest and SVM. Random forests with $n = 10$ and $n = 15$ show 2% and 5% better accuracy and 6% and 14% improvement in end-to-end latency compared to decision tree when the number of features are increased. However, when the total number of features, N is less than n, no random forest is generated due to the fact that number of required features is higher than the available features. Also, the experiments shows that decision tree achieves lower end-to-end latency (on average by 5%) than SVM even though SVM has higher prediction accuracy (on average 4% higher). The reason is that, while SVM in general has higher rate of prediction accuracy, it cannot always predict correct



(a) Prediction accuracy

(b) End-to-end latency

Figure 4.6 Effect of increasing number of file features in search requests when decision tree, random forest and SVM are used: (a) prediction accuracy, (b) average end-to-end latency.

file group if number of features used in the request is lower than the number of features used to train SVM. Under such circumstances, we often need to fetch the file from mobile device rather than trying to widen the search in the VMs. Based on the experiment, the performance of decision tree with update algorithm (Section 4.4.2) decreased end-to-end latency up to 5%, compared to the decision tree without the update algorithm. We also notice that increasing the number of features in each tree and increasing the number of features used in the whole system can both improve performance. Thus, it is always viable to include more features in a decision tree instead of using random forest to improve performance.

In order to investigate the effect of using different number of features for file request, we varied number of features used for the file request (m) while keeping the value of N to 23. The results are shown in Figure 4.6. Based on the figure it is clear that while for higher values of m, the performance of different classifiers are similar. When the value of m is lower than 13, the change of performance is noticeable. The prediction accuracy for SVM and random forest with $n = 5$ and $n = 10$ degrades 6%, 7% and 4% respectively compared to decision tree without update algorithm. Under similar conditions average end-to-end latency degrades 9%, 10% and 6%. The performance variation of decision tree

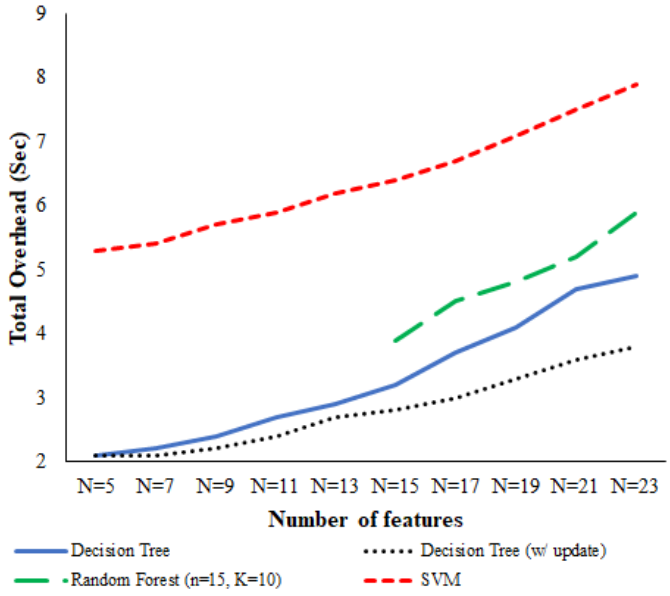


Figure 4.7 Effect of increasing number of file features in decision tree, random forest and SVM on total overhead

with and without update and random forest with $n = 15$ are very similar (varies less than 2.5%) under such circumstances. The reason behind such performance is when correct file group cannot be identified, CAFDS employs a generalize search to identify all possible file groups based on its decision trees (Section 4.4.2). However, if generalize search fails, the required files are fetched from the mobile instead.

Adding new features, and updating or removing an existing features results dynamically triggers to updating or reconstructing the classifier in metadata server. To investigate the effect of different number of features during the construction of the classifier on the total overhead for constructing and maintaining the classifier, we compared total overhead of decision tree without update with decision tree with update, random forest with $n = 15$ and SVM (Figure 4.7). The result clearly shows the advantage of decision tree with update algorithm. The total overhead of random forest and SVM is 1.2x and 2x higher than that of decision tree without update algorithm. The reason behind this performance is if there is a change in the features, the classifier needs to be reconstructed. The decision tree without update algorithm incurs 1.3x times higher overhead compared to that of one with update for same reason.

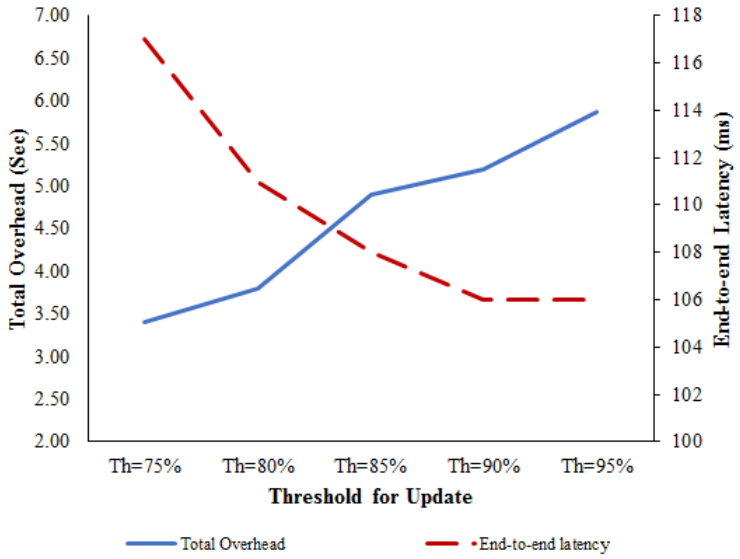
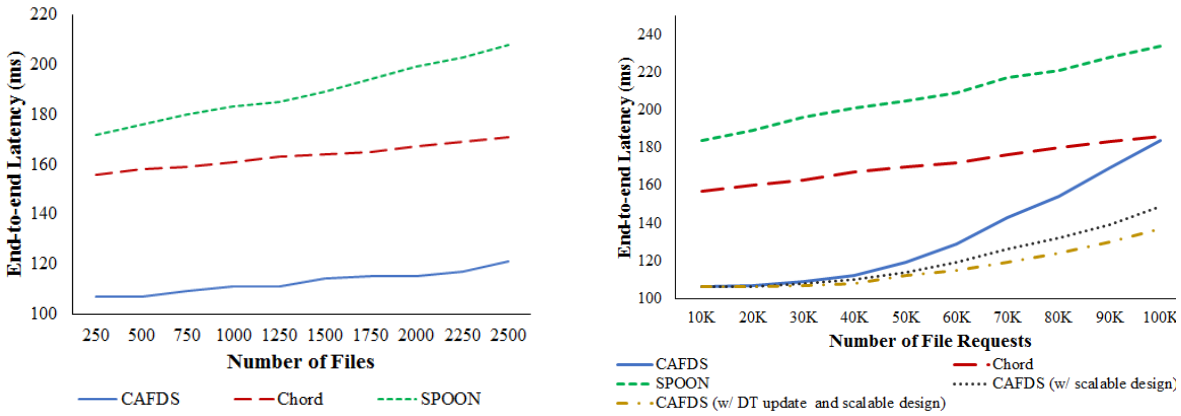


Figure 4.8 Effect of update algorithm with different thresholds on end-to-end latency and total overhead.

Based on the results, it is clear than while random forest with large number of features available during construction of its construction and file search, the performance of the decision tree is similar and it can yield to results when random forest is unable to. Thus, decision tree is a clear choice for classifying file groups in metadata server. However, the lower total overhead for managing the decision tree indicates the advantage of update algorithm.

As mentioned earlier, the decision tree in the metadata server needs to be reconstructed or updated to better serve requests with new searching criteria. The metadata server reconstructs or updates the tree when the success rate of recent request drops below a threshold. We have investigated how the threshold value T_h affects performance and the total overhead for updating the decision tree. The results in Figure 4.8 show that increasing the threshold reduces end-to-end latency, but also increases total overhead. For example, increasing the threshold value from 75% to 95%, decreases end-to-end latency by 8% because the classification with the updated tree fits the searching criteria better. However, it causes an increase of 41% in overhead, because the tree needs to be reconstructed more frequently.



(a) Increasing number of files

(b) Increasing number of file requests

Figure 4.9 Effect of different number of files and file requests on average end-to-end latency: (a) increasing number of files, (b) increasing number of file requests.

To test the scalability of the design, we perform two tests. First, we increased the number of files present in the system while keeping the number of users to 36 and the number of file requests to approximately 46,000. Then we increase the total number of search requests over the course of each experiment (3.5 hours) while keeping the number of users and files to 36 and 2,500, respectively. For each experiment, we show the average end-to-end delay for CAFDS, Chord and SPOON in Figure 4.9.

Figure 4.9(a) shows that, with the increase in the number of files, the average end-to-end latency increases for all systems. When the number of files is increased from 250 to 2500, the end-to-end latency increases by 18%, 10% and 21% for CAFDS, Chord, and SPOON, respectively. As the number of requests remains the same, the apps are requesting for a higher number of files. As the files are distributed over a large number of nodes, Chord requires higher network overhead to locate the files. However, due to its DHT-based lookup scheme, it has the lowest amount of increase. In SPOON, users are organized in super-peers. With the increasing number of files, the number of files to be fetched from other super-peers is increased. This results in an increase in average latency. CAFDS organizes the files into groups based on their similarity. Therefore, even though the number of files is increased, the number of file groups does not increase as much. As a result, each user has to process a higher number of requests which requires additional

Table 4.4 Statistics regarding number of requests processed by each VM

Num. of requests		CAFDS with update				CAFDS with update and scalable			
Total	**Average**	**Max**	**Min**	**Std. Dev.**	**Median**	**Max**	**Min**	**Std. Dev.**	**Median**
10K	278	638	49	46.2	324.28	379	169	30.9	297.23
20K	556	882	269	42.7	612.96	667	449	29.8	571.84
30K	834	1127	598	40.3	891.45	926	741	28.9	842.51
40K	1112	1369	931	36.9	1261.39	1198	1013	26.4	1094.6
50K	1389	1587	1181	33.7	1491.74	1467	1311	24.4	1404.74
60K	1667	1803	1526	32.1	1741.02	1729	1619	22.7	1677.38
70K	1945	2208	1812	27.4	2027.36	1991	1923	19.3	1954.72
80K	2223	2417	2096	24.8	2281.47	2259	2183	17.5	2219.49
90K	2500	2841	2339	20.2	2582.19	2553	2476	14.2	2510.44
100K	2778	3096	2592	16.5	2819.73	2804	2751	12.1	2774.24

computation. However, due to its simple architecture, the increase of average latency in CAFDS is lower than that of SPOON.

With a 10x increase in the number of file requests, the end-to-end latency is increased by 71%, 21% and 35% for CAFDS, Chord, and SPOON, respectively, as shown in Figure 4.9(b). As the number of requests increases, requests arrive at higher rates, and each VM has to process a larger number of requests in a time period. Chord distributes the requests over a large number of users which causes it to have the lowest increase of average latency. In SPOON, the local super-peer handles a large number of additional requests. Due to requests to other super-peers, the average latency is higher in this case. In CAFDS, in addition to a larger number of requests processed by each VM, the metadata server also needs to process a large number of requests. This causes it to increase the average latency.

To analyze the scalability of CAFDS, the scalability mechanism (Section 4.4.3) is implemented with both the decision tree reconstruction and the decision tree update approach (Section 4.4.2). As shown in Figure 4.9(b), the scalable design of CAFDS decreases the end-to-end latency by 7% and 10%, respectively compared to basic CAFDS.

Finally, to evaluate the load on each individual user in the cloud (VMs) we calculated the maximum and minimum number of requests, standard deviation and median of the number of requests processed by individual user in CAFDS with update algorithm applied

on decision tree for both with and without scalable design. The result is shown in Table 4.4. Based on the result it is clear that scalable design decreases the difference between maximum and minimum number of requests processed by each individual users. The difference of standard deviation and median between different version of CAFDS also shows similar trend. Based on this result, we can conclude that the scalable design of CAFDS increases the fairness in terms of number of request processed by each user.We have implemented a prototype of CAFDS in Android and Linux, and compared its performance against Chord [39], a DHT-based distributed lookup scheme, and SPOON [34], a P2P-based file sharing system. SPOON's lookup scheme has some similarity to CAFDS. When looking up a file, it first selects a group of users who may own files similar to the desired file and then looks for the file within the group. We also compared CAFDS with OFS [91, 92], an overlay file system for mobile-cloud computing. OFS gives us a baseline performance to measure improvements.

4.7 Chapter Summary

This chapter has proposed a Context-Aware File Discovery System (CAFDS) for distributed mobile-cloud (DMC) apps. CAFDS expands the file searching scope beyond a single app to the mobile devices and VMs of all users willing to share files. It allows distributed mobile-cloud apps to dynamically define and modify their custom features for searching files. CAFDS is implemented as a service within a mobile-cloud middleware that enables apps to perform seamless file searching. A prototype of CADFS was implemented and validated in Android and Linux. By using simple machine learning techniques, like self-organizing maps, k-means clustering and a modified decision tree or random forests, CAFDS provides lower latency file access than traditional DHT-based and peer-to-peer techniques. Therefore, CADFS is expected to support novel, data-intensive DMC apps, with low-latency requirements.

CHAPTER 5

CONCLUSION AND FUTURE WORKS

5.1 Conclusion

Increasing popularity of mobile devices such as smartphones and tablets, and recent improvements in cloud technologies have recently opened door to a new class of mobile apps which can delegate heavy data processing computations to the cloud. This allows mobile devices to perform computationally expensive tasks on these data faster while saving energy and resources on users mobile.

Despite mobile devices being a large source of data in recent years, existing offloading platforms have limited file access capabilities. For example, they do not provide concurrent and consistent support for accessing files from both mobile and cloud during task offloading to leverage the data originated from mobile devices. They also do not provide efficient file search for the files that already available in the cloud to improve overall performance. These limitations inspired us to investigate the feasibility of providing such file search and file system support without modifying the existing mobile apps.

In this dissertation, we proposed *Overlay File System* (OFS) for providing efficient support for task offloading. OFS allow the mobile apps to offload computations to a surrogate running in the cloud with consistent, concurrent and efficient file access. This overlay file systems is easily deployable and have versatile designs such that they can work with various offloading platform very easily.

The case study of overlay file system has shown that it can allow the apps to offload data intensive tasks to the cloud using various offloading platforms with minimal effort. It also provides a unified view of the data. This helps the programmers focus on application logic without concern about the location of the files or how to provide consistency. The performance evaluation of this platform also shows that it can support I/O operations with low I/O and network overhead.

To improve the scope of file access, this dissertation also introduces a file search mechanism for distributed mobile-cloud apps called *Context-Aware File Discovery System* (CAFDS). This system enables mobile-cloud apps to locate a file using its content or search for similar file(s) or files with specific characteristics using app-defined file contexts, and retrieve the file once it is located. CAFDS uses simple machine learning techniques like self-organizing maps, k-means clustering or modified decision tree to improve latency for file access.

CAFDs also allows apps to dynamically define and modify file contexts so that the programmers can create new apps or improve existing apps without concentrating how to improve file search during a computation. The performance evaluation of CAFDS proves that it provides more efficient and versatile file search than existing file DHT-based or P2P file systems. While this system is proposed for task offloading in a distributed mobile-cloud platform, it can easily be adopted in any system where computational tasks need to search for files dynamically during their execution.

We believe that both proposed platforms will inspire the developers to use complex apps that can effectively leverage the cloud assistance for their apps. As these platforms are easily deployable and do not require root privilege, it will allow the users to take advantage of a large amount of data available in their mobile devices or in the cloud.

5.2 Future Works

While mobile-cloud solutions allow faster processing of data using cloud resources, they often exhibit the drawback of increased latency due to mobile-to-cloud communication to transfer data files to/from the cloud. To reduce latency, one promising research direction is to leverage edge computing for offloading tasks in a mobile-edge-cloud architecture. Despite the potential advantages of the employing edge to decrease network overhead, computation offloading cannot be employed on mobile-edge-cloud paradigm because they need concurrent accesses to files from at least two hosting environments. To address this issue, *Overlay File System* (OFS) presented in Chapter 3 can be extended to allow transparent file access for the tasks running on the edge nodes. The benefit of such

systems is that it can retain low latency advantage of edge nodes while continue to use cloud as a controller used for fault tolerance.

The mobile-edge-cloud paradigm may minimize the overhead of transferring files from mobile to cloud. This paradigm is especially useful for the devices with very limited computational resources and lower energy capacity such as various smart wearable devices and IoT devices. These devices can employ such solutions to perform computationally expensive tasks on large amount of data while saving their storage [94,95]. *Context-Aware File Discovery Service* (CAFDS) presented in Chapter 4 can be enhanced to enable such devices to search files from other available sources from the cloud for more efficient file access.

This chapter presents MEFS as an extension of Overlay File System (OFS), describes its basic architecture, and presents brief performance evaluation. It also describes the potential future works for extending Context-Aware File Discovery Service (CAFDS).

5.2.1 Future Work for Overlay File System (OFS)

The purpose of designing OFS is to provide transparent and efficient file access to mobile-cloud apps. While OFS accomplishes its goal, the overall efficiency of OFS can be enhanced by employing edge computing [96–99] in the mix. The edge computing paradigm enables the deployment of middleboxes for supporting and enhancing service provisioning at the locations of mobile users. This allows improved scalability and reactivity in the interaction with mobile nodes, any time local control decisions are applicable. More specifically, Multi-access Edge Computing (MEC) [96] is an architectural model and specification proposal by the European Telecommunications Standards Institute (ETSI). MEC aims at evolving the traditional two-layer cloud-device integration model, where mobile nodes directly communicate with a global cloud through the Internet, with the introduction of a third intermediate middleware layer that is executed at the network edge.

Although a few solutions [98–100] have been proposed to contribute to the field of MEC by addressing challenges in computation offloading, there is no ready-to-use solution

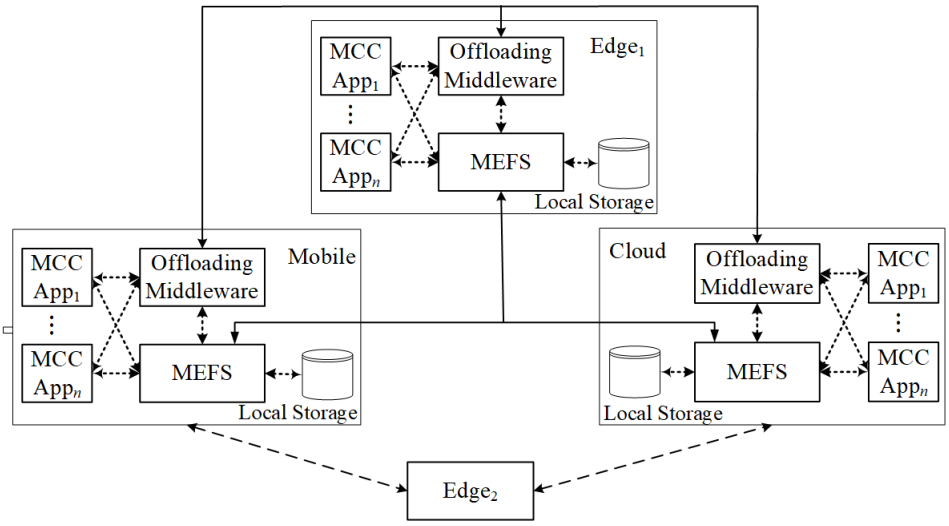


Figure 5.1 Overall architecture of MEFS for MEC environment.

to satisfy the requirements of supporting file system access for edge-assisted apps. As a consequence, we present *Mobile Edge File System (MEFS)*, an expansion of OFS to make computational offloading practical in the MEC environment. Compared to existing file system designs, MEFS takes into consideration the mobility of the users and the failures of MEC servers.

Figure 5.1 depicts the general architecture of MEFS and the offloading middleware deployed on mobiles, edges nodes, and the cloud; in this scenario, a mobile app can offload its computation to a nearby edge node. The cloud is used as a controller that helps with fault-tolerance, but is not generally involved in app computation.

MEFS leverages OFS to manage efficient and transparent remote file access and file sharing among the distributed components of edge-assisted mobile apps. Furthermore, it provides support for application portability and resilience. MEFS can portably transfer apps between MEC servers. When the user moves from one edge node to another (e.g., from $Edge_1$ to $Edge_2$ in the figure), MEFS is able to seamlessly perform handoff in order to maintain communication locality and low latency. Once the handoff between two edge nodes is started, MEFS transfers the file system state and associated metadata, while the offloading middleware transfers the app state (i.e., app variables). To protect against node or communication failures, MEFS leverages the cloud, as a controller entity, to provide

fault-tolerance. If a MEC node fails or if the user moves away from the current MEC node and there is no other MEC node available in his/her proximity. Under such circumstances, the cloud is in charge of restoring the affected app either in the cloud or at a new MEC node.. To this end, MEFS provides a transparent mechanism that synchronizes the file system state and associated metadata between edge nodes and the cloud.

5.2.2 Future Work for Context-Aware File Discovery Service (CAFDS)

CAFDS is a file directory service that can locate and fetch required files from other users for task running in mobile cloud distributed computing. While this system can search files for the apps running on offloading platforms like Moitree [10], CASINO [22] and a few other offloading frameworks [15–17], it must be extended to serve apps in the cloud-edge-mobile paradigm and cloud of things [94, 95, 101].

First, CAFDS must be extended to deal with mobility. The computation in an app and the data needed by the computation may move dynamically to the edge server closest to the mobile device. An additional layer must be added to deal with the location changes of files. It can also implement a prediction module to determine based on the trajectories of mobile devices which files are likely to have lower access overhead. This would allow the system to determine the most optimal source of the file and how to fetch it.

While fetching a file from the VMs of other users can reduce the overall time, malicious users can use this method to obtain files that does not belong to them from others. Several research [102–106] have been conducted to introduce proof of ownership to prevent such scenarios. The majority of these schemes require high I/O and computational overhead. However, spot-checking based techniques [103–105] have relatively lower overhead. In CAFDS, such proofs can only be adapted once the VMs with correct files are identified and the requester VM chooses one of them as a potential source of the file.

BIBLIOGRAPHY

[1] "Number of smartphone users in the United States from 2010 to 2022 (in millions)," https://www.statista.com/statistics/201182/forecast-of-smartphone-users-in-the-us/, [Online; accessed 12-Aug-2017].

[2] D. Chaffey, "Mobile marketing statistics compilation," http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/, July 2018, [Online; accessed 12-Aug-2017].

[3] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys)*, June 2010, pp. 49–62.

[4] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proceedings of the 31st Annual IEEE International Conference on Computer Communications (IEEE INFOCOM)*, March 2012, pp. 945–953.

[5] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in *Proceedings of the 6th EuroSys Conference (EuroSys)*, April 2011, pp. 301–314.

[6] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 93–106.

[7] I. Zhang, A. Szekeres, D. Van Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy, "Customizable and extensible deployment for mobile/cloud applications," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Berkeley, CA, USA: USENIX Association, 2014, pp. 97–112.

[8] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath, "Avatar: Mobile Distributed Computing in the Cloud," in *Proceedings of the 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, March 2015.

[9] J. L. D. Neto, S.-Y. Yu, D. F. Macedo, J. M. S. Nogueira, R. Langar, and S. Secci, "ULOOF: a user level online offloading framework for mobile edge computing," November 2018.

[10] M. A. Khan, H. Debnath, N. R. Paiker, N. Gehani, X. Ding, R. Curtmola, and C. Borcea, "Moitree: A middleware for cloud-assisted mobile distributed apps," in *Proceedings of the 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, March 2016, pp. 21–30.

[11] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, "The NFS version 4 protocol," in *Proceedings of the 2nd International SANE Conference*, 2000.

[12] "Dropbox," https://www.dropbox.com/, [Online; accessed 12-Feb-2018].

[13] J. J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 3–25, February 1992.

[14] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan, "Exploiting Weak Connectivity for Mobile File Access," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 143–155, December 1995.

[15] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, April 2013.

[16] M. R. Rahimi, N. Venkatasubramanian, and A. V. Vasilakos, "MuSIC: Mobility-aware optimal service allocation in mobile cloud computing," in *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing (Cloud)*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 75–82.

[17] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, October 2016.

[18] K. Liu, J. Peng, H. Li, X. Zhang, and W. Liu, "Multi-device task offloading with time-constraints for energy efficiency in mobile cloud computing," *Future Generation Computer Systems*, vol. 64, no. C, pp. 1–14, November 2016.

[19] E. Meskar, T. D. Todd, D. Zhao, and G. Karakostas, "Energy aware offloading for competing users on a shared communication channel," *IEEE Transactions on Mobile Computing*, vol. 16, no. 1, pp. 87–96, January 2017.

[20] X. Lyu, H. Tian, C. Sengul, and P. Zhang, "Multiuser joint task offloading and resource optimization in proximate clouds," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 4, pp. 3435–3447, April 2017.

[21] W. Chen, D. Wang, and K. Li, "Multi-user multi-task computation offloading in green mobile edge cloud computing," *IEEE Transactions on Services Computing*, pp. 1–1, 2018.

[22] H. Debnath, G. Gezzi, A. Corradi, N. Gehani, X. Ding, R. Curtmola, and C. Borcea, "Collaborative offloading for distributed mobile-cloud apps," in *Proceedings of the 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, March 2018, pp. 87–94.

[23] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. Khan, and H. Debnath, "Avatar: Mobile distributed computing in the cloud," in *Proceedings of 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, March 2015, pp. 151–156.

[24] M. Shiraz, A. Gani, R. H. Khokhar, and R. Buyya, "A review on distributed application processing frameworks in smart mobile devices for mobile cloud computing," *IEEE Communications Surveys Tutorials*, vol. 15, no. 3, pp. 1294–1313, March 2013.

[25] Google, "Files go," https://filesgo.google.com/, [Online; accessed 02-Mar-2018].

[26] A. Inc., "Introduction to spotlight," https://developer.apple.com/library/content/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.html, [Online; accessed 02-Mar-2018].

[27] H. H. Huang, N. Zhang, W. Wang, G. Das, and A. S. Szalay, "Just-in-time analytics on large file systems," *IEEE Transactions on Computers*, vol. 61, no. 11, pp. 1651–1664, November 2012.

[28] L. Xu, H. Jiang, L. Tian, and Z. Huang, "Propeller: A scalable real-time file-search service in distributed systems," in *Proceedings of the 34th International Conference on Distributed Computing Systems*, June 2014, pp. 378–388.

[29] L. Xu, Z. Huang, H. Jiang, L. Tian, and D. Swanson, "VSFS: A searchable distributed file system," in *Proccedings of the 9th Parallel Data Storage Workshop*, November 2014, pp. 25–30.

[30] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian, "Smartstore: A new metadata organization paradigm with semantic-awareness for next-generation file systems," in *Proccedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*. New York, NY, USA: ACM, 2009, pp. 10:1–10:12.

[31] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *Proccedings of the 7th Conference on File and Storage Technologies (FAST)*. Berkeley, CA, USA: USENIX Association, 2009, pp. 153–166.

[32] H. Sim, Y. Kim, S. S. Vazhkudai, G. R. Vallée, S.-H. Lim, and A. R. Butt, "Tagit: An integrated indexing and search service for file systems," in *Proccedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. New York, NY, USA: ACM, 2017, pp. 5:1–5:12.

[33] X. Li, B. Dong, L. Xiao, L. Ruan, and D. Liu, "CEFLS: A cost-effective file lookup service in a distributed metadata file system," in *Proccedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2012, pp. 25–32.

[34] K. Chen, H. Shen, and H. Zhang, "Leveraging social networks for P2P content-based file sharing in disconnected MANETs," *IEEE Transactions on Mobile Computing*, vol. 13, no. 2, pp. 235–249, Febuary 2014.

[35] H. Shen, G. Liu, and L. Ward, "A proximity-aware interest-clustered P2P file sharing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. June, pp. 1509–1523, June 2015.

[36] Z. Li and H. Shen, "Social-P2P: Social network-based P2P file sharing system," in *Proccedings of the 20th IEEE International Conference on Network Protocols (ICNP)*, October 2012.

[37] T. Paolo, "A twolayer model for improving the energy efficiency of file sharing peertopeer networks," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 13, pp. 3166–3183, 2018.

[38] S. Brienza, S. E. Cebeci, S. S. Masoumzadeh, H. Hlavacs, O. Özkasap, and G. Anastasi, "A survey on energy efficiency in P2P systems: File distribution, content streaming, and epidemics," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 36:1–36:37, December 2015.

[39] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160.

[40] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, "Just-in-time provisioning for cyber foraging," in *Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services (Mobisys)*. New York, NY, USA: ACM, 2013, pp. 153–166.

[41] A. Zanni, S. young Yu, S. Secci, R. Langer, P. Bellavista, and D. Macedo, "Automated offloading of android applications for computation/energy optimizations," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, May 2017, pp. 990–991.

[42] G. Gezzi, "Smart execution of distributed application by balancing resources in mobile devices and cloud-based avatars," Master's thesis, University of Bologna, Bologna, Italy, 2016.

[43] S. Barbarossa, S. Sardellitti, and P. D. Lorenzo, "Joint allocation of computation and communication resources in multiuser mobile cloud computing," in *Proceedings of the 14th IEEE International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, June 2013, pp. 26–30.

[44] Y. Dong, H. Zhu, J. Peng, F. Wang, M. P. Mesnier, D. Wang, and S. C. Chan, "RFS: A network file system for mobile devices and the cloud," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, pp. 101–111, February 2011.

[45] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu, "Reliable, consistent, and efficient data sync for mobile apps," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015, pp. 359–372.

[46] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu, "Simba: Tunable End-to-end Data Consistency for Mobile Apps," in *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, 2015, pp. 7:1–7:16.

[47] B. Atkin and K. P. Birman, "MFS: an adaptive distributed file system for mobile hosts," http://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/mfs.pdf, Tech. Rep., 2003, [Online; accessed 12-Nov-2018].

[48] E. B. Nightingale and J. Flinn, "Energy-efficiency and storage flexibility in the blue file system," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI)*. Berkeley, CA, USA: USENIX Association, 2004, pp. 25–25.

[49] R. Többicke, "Distributed file systems: Focus on andrew file system/distributed file service (AFS/DFS)," in *MSST'94*, 1994, pp. 23–26.

[50] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: concepts and systems," *IEEE Parallel and Distributed Technology: Systems and Applications*, vol. 4, no. 2, pp. 63–71, Summer 1996.

[51] J. B. Carter, "Distributed shared memory: concepts and systems," *IEEE Parallel and Distributed Technology: Systems and Applications*, vol. 29, no. 2, pp. 219–227, September 1995.

[52] L. I. Kontothanassis, M. L. Scott, and R. Bianchini, "Lazy release consistency for hardware-coherent multiprocessors," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (Supercomputing)*. New York, NY, USA: ACM, 1995.

[53] L. Guangchun, Z. Jun, L. Xianliang, and L. Jun, "HCCM: A novel cache consistence mechanism," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 2, pp. 25–36, April 2003.

[54] P. Bzoch and J. afark, "Maintaining cache consistency for mobile clients in distributed file system," in *Proceedings of the 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC)*, August 2013, pp. 55–62.

[55] K. Fawaz and H. Artail, "DCIM: Distributed cache invalidation method for maintaining cache consistency in wireless mobile networks," *IEEE Transactions on Mobile Computing*, vol. 12, no. 4, pp. 680–693, April 2013.

[56] R. Wang, Z. Liu, and Z. Yang, "Benchmark data for mobile app traffic research," in *15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous)*, 2018.

[57] H. Yao, G. Ranjan, A. Tongaonkar, Y. Liao, and Z. M. Mao, "SAMPLES: Self adaptive mining of persistent lexical snippets for classifying mobile application traffic," in *Proccedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2015, pp. 439–451.

[58] W. D. Donato, A. Pescape, and A. Dainotti, "Traffic identification engine: an open platform for traffic classification," *IEEE Network*, vol. 28, no. 2, pp. 56–64, March 2014.

[59] S. Mongkolluksamee, V. Visoottiviseth, and K. Fukuda, "Enhancing the performance of mobile traffic identification with communication patterns," in *Proccedings of the 39th Annual Computer Software and Applications Conference*, vol. 2, July 2015, pp. 336–345.

[60] X. Han, Y. Zhou, L. Huang, L. Han, J. Hu, and J. Shi, "Maximum entropy based IP-traffic classification in mobile communication networks," in *Proccedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, April 2012, pp. 2140–2145.

[61] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proccedings of the ACM/IFIP/USENIX 12th International Middleware Conference (Middleware)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 329–350.

[62] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 161–172, August 2001.

[63] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell, "A survey of peer-to-peer storage techniques for distributed file systems," in *Proccedings of the International Conference on Information Technology: Coding and Computing (ITCC)*, vol. 2. Washington, DC, USA: IEEE Computer Society, 2005, pp. 205–213.

[64] J. Choi, J. Han, E. Cho, T. Kwon, and Y. Choi, "A survey on content-oriented networking for efficient content delivery," *IEEE Communications Magazine*, vol. 49, no. 3, pp. 121–127, March 2011.

[65] D. Perino and M. Varvello, "A reality check for content centric networking," in *Proccedings of the ACM SIGCOMM Workshop on Information-centric Networking (ICN)*. New York, NY, USA: ACM, 2011, pp. 44–49.

[66] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies CoNEXT*. New York, NY, USA: ACM, 2009, pp. 1–12.

[67] S. Wang, J. Bi, J. Wu, and A. V. Vasilakos, "CPHR: In-network caching for information-centric networking with partitioning and hash-routing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2742–2755, October 2016.

[68] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A data-oriented (and beyond) network architecture," in *Proccedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. New York, NY, USA: ACM, 2007, pp. 181–192.

[69] J. Iqbal and P. Giaccone, "Interest-based cooperative caching in multi-hop wireless networks," in *IEEE Globecom Workshops (GC Wkshps)*, December 2013, pp. 617–622.

[70] "AspectJ," https://eclipse.org/aspectj/, [Online; accessed 12-Jan-2017].

[71] O. Kirch, "Why NFS sucks," *Linux Symposium*, vol. 2, pp. 51–64, 2006.

[72] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Networks and Applications*, vol. 18, no. 1, pp. 129–140, feb 2013.

[73] "Filesystem in userspace (FUSE)," https://github.com/libfuse/libfuse, [Online; accessed 12-Jun-2017].

[74] "Kryonet," https://github.com/EsotericSoftware/kryonet/, [Online; accessed 12-Jan-2017].

[75] W. R. Dieter and J. E. Lumpp Jr, "User-level checkpointing for linuxthreads programs." in *USENIX ATC*, 2001, pp. 81–92.

[76] "Pin - A Dynamic Binary Instrumentation Tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, June 2012, [Online; accessed 14-Jun-2017].

[77] "ProbeDroid: A dynamic binary instrumentation kit for android app analysis," http://www.zsshen.org/program-analysis-and-binary-instrument, [Online; accessed 12-Jun-2017].

[78] "OpenCV: Open source computer vision," http://opencv.org/, [Online; accessed 12-Jan-2017].

[79] "PhoneLab: A smartphone platform testbed," https://www.phone-lab.org/, [Online; accessed 02-Jan-2017].

[80] "Bionic sources (official repository)," https://android.googlesource.com/platform/bionic/, [Online; accessed 5-Mar-2016].

[81] "Trepn power profiler," https://developer.qualcomm.com/software/trepn-power-profiler, [Online; accessed 12-Jan-2017].

[82] "Cisco visual networking index: Global mobile data traffic forecast update, 20162021 white paper," https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html, [Online; accessed 02-Mar-2018].

[83] D. Butler, "Crowdsourcing goes mainstream in typhoon haiyan response," https://www.nature.com/news/crowdsourcing-goes-mainstream-in-typhoon-response-1.14186, November 2013.

[84] D. MacKenzie, "Social media helps aid efforts after typhoon haiyan," https://www.newscientist.com/article/dn24565-social-media-helps-aid-efforts-after-typhoon-haiyan/, November 2013, [Online; accessed 24-Nov-2018].

[85] "Boston's legacy: Can crowdsourcing really fight crime?" https://www.nbcnews.com/tech/internet/bostons-legacy-can-crowdsourcing-really-fight-crime-n74831, 2014, [Online; accessed 24-Nov-2018].

[86] "ID3 algorithm," https://en.wikipedia.org/wiki/ID3_algorithm, [Online; accessed 02-Mar-2018].

[87] S. M. Ross, "Predicting the states," in *Introduction to Probability Models*, tenth edition ed., S. M. Ross, Ed. Boston, USA: Academic Press, 2010, pp. 191–290.

[88] K. Van Laerhoven, "Combining the self-organizing map and k-means clustering for on-line classification of sensor data," in *International Conference on Artificial Neural Networks (ICANN)*. Springer, 2001, pp. 464–469.

[89] "Random forest," https://en.wikipedia.org/wiki/Random_forest, [Online; accessed 02-Mar-2018].

[90] "Redis," http://redis.io/, [Online; accessed 02-Jan-2017].

[91] J. Shan, N. R. Paiker, X. Ding, N. Gehani, R. Curtmola, and C. Borcea, "An overlay file system for cloud-assisted mobile applications," in *Proceedings of the 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, May 2016, pp. 1–14.

[92] N. R. Paiker, J. Shan, C. Borcea, N. Gehani, R. Curtmola, and X. Ding, "Design and implementation of an overlay file system for cloud-assisted mobile apps," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2017.

[93] D. Zhou, W. Pan, W. Wang, and T. Xie, "I/O characteristics of smartphone applications and their implications for eMMC design," in *2015 IEEE International Symposium on Workload Characterization (IISWC)*, October 2015, pp. 12–21.

[94] J. Al-Jaroodi, N. Mohamed, I. Jawhar, and S. Mahmoud, "CoTWare: A cloud of things middleware," in *Proccedings of the 37th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW)*, June 2017, pp. 214–219.

[95] S. Walraven, E. Truyen, and W. Joosen, "A middleware layer for flexible and cost-efficient multi-tenant applications," in *ACM/IFIP/USENIX 12th International Middleware Conference (Middleware)*, F. Kon and A.-M. Kermarrec, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 370–389.

[96] "New white paper: ETSIs mobile edge computing initiative explained," https://www.etsi.org/news-events/news/1009-2015-09-news-new-white-paper-etsi-s-mobile-edge-computing-initiative-explained, [Online; accessed 12-Nov-2018].

[97] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, October 2009.

[98] G. Orsini, D. Bade, and W. Lamersdorf, "Computing at the mobile edge: Designing elastic android applications for computation offloading," in *Proceedings of the 8th IFIP Wireless and Mobile Networking Conference (WMNC)*, October 2015, pp. 112–119.

[99] C. You and K. Huang, "Multiuser resource allocation for mobile-edge computation offloading," in *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*, December 2016, pp. 1–6.

[100] X. Wei, S. Wang, A. Zhou, J. Xu, S. Su, S. Kumar, and F. Yang, "MVR: An architecture for computation offloading in mobile edge computing," in *Proceedings of the IEEE International Conference on Edge Computing (EDGE)*, June 2017, pp. 232–235.

[101] S. H. Kim and D. Kim, "Enabling multi-tenancy via middleware-level virtualization with organization management in the cloud of things," *IEEE Transactions on Services Computing*, vol. 8, no. 6, pp. 971–984, November 2015.

[102] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *Proceedings of the 18th ACM conference on Computer and communications security (CCS)*. New York, New York, USA: ACM Press, 2011, p. 491.

[103] R. Di Pietro and A. Sorniotti, "Boosting efficiency and security in proof of ownership for deduplication," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. New York, New York, USA: ACM Press, 2012.

[104] J. Blasco, R. D. Pietro, A. Orfila, and A. Sorniotti, "A tunable proof of ownership scheme for deduplication using bloom filters," in *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, October 2014, pp. 481–489.

[105] C.-m. Yu, C.-y. Chen, and H.-c. Chao, "Proof of ownership in deduplicated cloud storage with mobile device efficiency," *IEEE Network*, vol. 29, no. 2, pp. 51–55, mar 2015.

[106] J. Hur, D. Koo, Y. Shin, and K. Kang, "Secure Data Deduplication with Dynamic Ownership Management in Cloud Storage," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 11, pp. 3113–3125, nov 2016.