Context-Aware File Discovery System for Distributed Mobile-Cloud Apps

Nafize R. Paiker, Xiaoning Ding, Reza Curtmola, Cristian Borcea
Department of Computer Science, *New Jersey Institute of Technology* Newark, NJ 07102, USA
Email: {nrp48, xiaoning.ding, reza.curtmola, borcea}@njit.edu

***Abstract*—Recent research has proposed middleware to enable efficient distributed apps over mobile-cloud platforms. This paper presents a Context-Aware File Discovery Service (CAFDS) that allows distributed mobile-cloud applications 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. CAFDS provides low-latency through a cloud-based metadata server, which uses a decision tree to locate the nearest files that satisfy the context and content features requested by applications. We implemented CAFDS in Android and Linux. Experimental results show CAFDS achieves substantially lower latency than peer-to-peer solutions that cannot leverage context information.**

***Index Terms*—Mobile-cloud computing, File discovery service, Computation offloading, Decision tree**

I. Introduction

According to a Cisco forecast [1], 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. 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 to the cloud [2]–[6]. 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. Non-participating users may have the files that the app needs and may be willing to share them, but the app cannot locate these files. This significantly reduces the number of files available to the app and lowers the quality of user experience.

Second, each DMC app has to search and examine files independently. However, 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 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, e.g., search engines [7], [8] and file searching functionalities in storage systems and peer-to-peer systems [9]–[15] do not solve well the three issues for DMC apps. Search engines focus on searching file content with keywords instead of more general context and content features 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 a large number of independent users. Peer-to-peer systems offer distributed file searching functionalities. However, they introduce large latency due to their multi-hop networking nature.

This paper 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.

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, 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 contains 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. 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 [16]. Instead of searching through files one by one, CAFDS locates a few groups where the required files are likely to be found, and then searches in those groups.

The contributions of this paper are summarized as follows. (a) We designed CAFDS 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 [17] and SPOON [14].

The rest of the paper is organized as follows. Section II provides a brief introduction to distributed mobile-cloud apps and platforms. Section III overviews the main concepts of the CAFDS system. Section IV presents the detailed design and implementation of CAFDS. Section V evaluates its performance. Section VI discusses the related work, and Section VII concludes the paper.

II. 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 multiple users 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*) that enhances photos and creates 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 [4], [5].

Although the concept of CAFDS is generic and can be implemented on any mobile-cloud platform, we have implemented it over our Avatar [6], [18] platform. 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 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.

III. CAFDS Overview

A. The Problem

A challenging issue for DMC apps is how to quickly locate the required files and access the files with low overhead. 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*.

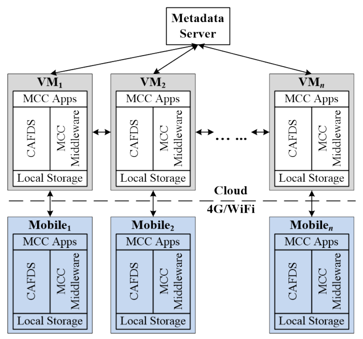


Fig. 1: Architecture of CAFDS Ecosystem.

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.

B. CAFDS Functionality and Main Components

Our key idea 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, and answers file search requests from apps. All DMC apps use this service, as they do not need to implement their own file search code. We name this layer Context-Aware File Discovery Service (CAFDS).

CAFDS implements file search by running an instance of the service within the mobile-cloud computing (MCC) middleware on mobiles and VMs, as shown in Figure 1. These instances accept requests from DMC apps to share and search for files of interest. The metadata server is the core component of CAFDS. It is responsible for managing the feature information of the files and responding to search requests. The metadata server does not store file contents, which remain at their original locations, e.g., mobile phones and cloud storage. The metadata server saves a File ID for each file, which is the SHA-256 hash value of the file contents.

The middleware at each entity serves as the interface between app instances and the metadata server. For file search, it translates API calls into a set of operations and interacts with the metadata server to finish the search. File search allows apps to search based on file features (context, content, and traditional file metadata), and it potentially returns multiple files. Apps can also search for a specific file using the hash value of the file. In this case, the middleware monitors the file operations (e.g., *open* and *read* calls) of the app instances, and it contacts the metadata sever for the cloud location of a file with the same content. For the files that mobile users want to share, the middleware marks them as searchable and collects the information needed for file discovery.

C. 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, an app can call CAFDS API to specify what matadata (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.

When an app defines a new feature, it must register the feature at the metadata server. The metadata server periodically updates the file features for classifying files based on registered new features, to improve the quality of classification.

D. 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.

IV. CAFDS Design and Implementation

A. CAFDS Instances and API

As shown in Figure 1, an instance of CAFDS runs on the mobile and VM of each user who wants to use the CAFDS service. A CAFDS instance consists of two layers. The upper

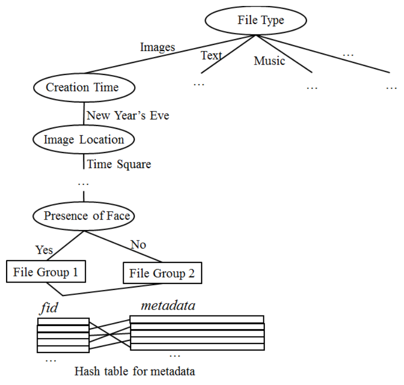


Fig. 2: **Example of a decision tree for a group of files in CAFDS.**

layer is embedded in each app through either compilation (for lower overhead) or AspectJ [19] (for increased compatibility). It exposes the CAFDS API to the app that requires CAFDS file search support and intercepts file I/O operations to generate individual file search requests. The lower layer is an independent system service. It 1) forwards requests to the metadata server, 2) extracts features from local files for easier search, 3) sends files to other instances to satisfy their requests, and 4) receives files and forwards them to apps to satisfy app requests.

As summarized in the Table I, CAFDS exposes a set of API functions to support the query and management of file features and to perform file search. The API follows an event-driven and callback-based asynchronous design. The first eight API functions are for creating and managing file features; *createMetadata()* is to create the file context; and the last function is to search based on the file context.

B. Metadata Server Design

The metadata server is a cloud component for processing file requests submitted by apps. The core of the metadata server is the data structure used to manage metadata and to support searches. The metadata server uses a hash table to manage file metadata with each entry being the metadata of a file and the key being the file ID (e.g., the hash value of the file content).

The metadata server classifies files into groups 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.

There are two key concerns with this method. The first concern is how to classify the 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 3) searching criteria are highly diverse. The second concern is how to organize and search file groups efficiently. This is important since the number of file groups can be large.

TABLE I: CAFDS 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*.
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*
locateAndRetrieveFile(FileMetadata metadata)	Sends a file search request to the metadata server to locate a file and waits for the requested file. The search is performed using file metadata, *metadata* If the *fid* is not included in the *metadata*, CAFDS performs a search based on features defined in *fileContext*. Otherwise, a search for a specific file is performed.

We address the above concerns using a decision tree, as illustrated in Figure 2. 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 contain pointers to the file entries in the hash table, such that the information of the files can be located.

The selection of file features used to construct the decision tree has a great impact on the efficiency and quality of the search. CAFDS uses the ID3 algorithm [16] to select features and construct the decision tree. The ID3 algorithm employs a top-down, greedy method to search through the space of possible features with no backtracking using *Entropy* as a metric. *Entropy* measures the homogeneity of the file groups represented by the leaf nodes. A lower Entropy means that the files in each group are more homogeneous. The ID3 algorithm constructs the decision tree in a way that minimizes Entropy.

After the decision tree is constructed, CAFDS uses it to locate file groups when answering search requests. However, the file contexts used in searches may change over time, and thus the Entropy of the file groups on the decision tree may also change. CAFDS needs to monitor the Entropy and reconstruct the decision tree when it becomes too high. Since calculating the Entropy is costly, we use a simpler method to detect when the decision tree should be reconstructed. CAFDS monitors the success rate of recent requests and reconstruct the tree when the rate drops below a threshold, T_h. The success rate is the ratio between the number of files found and the total number of files in the file groups located during the searches.

V. Performance Evaluation

We have implemented a prototype of CAFDS in Android and Linux, and compared its performance against Chord [17] and SPOON [14]. Although Chord is not new, we use it to compare the performance of CAFDS against the class of DHT-based P2P schemes. SPOON is a newer P2P system focused on mobile users, and its lookup scheme has some similarity to CAFDS. We also compared CAFDS with OFS [20], an overlay file system for mobile-cloud computing. OFS gives us a baseline performance to measure improvements.

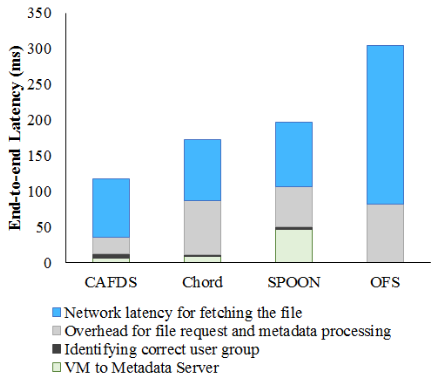


Fig. 3: Decomposition of average end-to-end latency.

A. 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. CAFDS instances are 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. The traces include 50,000 file requests from 18 different apps for 50 different users within 24 hours. To support the requests, we populated the metadata server with the information of 2,500 files of 4 file types (text, image, video, and audio files). The types of files used by the aforementioned apps are generated based on the related information from BIOtrace [21]. 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, and user tags for text files. We generated the files with random content and distributed them randomly on the VMs.

B. Experimental Results

Figure 3 shows the decomposition of the end-to-end latency for CAFDS, Chord, SPOON and OFS. From the figure it is clear that the end-to-end latency for searching a file in

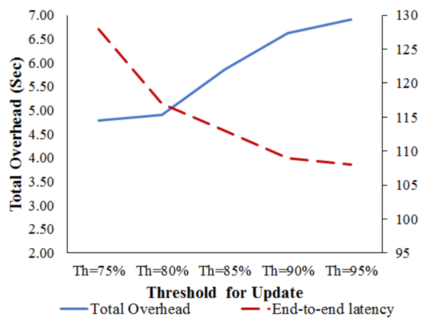
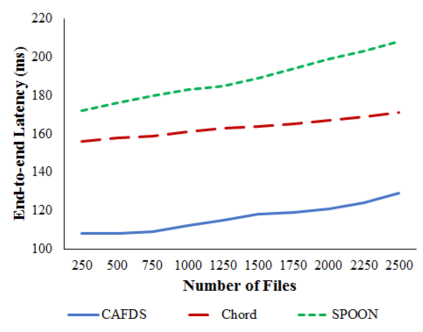
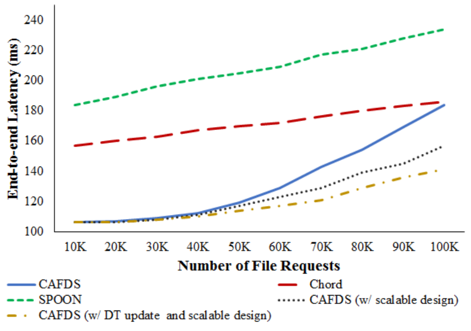


Fig. 4: **Effect of update algorithm with different thresholds on end-to-end latency and total overhead**



(a) Increasing number of files



(b) Increasing number of file requests

Fig. 5: **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.**

CAFDS is substantially lower than that of other systems. Chord, SPOON and OFS have 1.6x, 1.9x and 2.9x higher end-to-end latency than CAFDS. 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. OFS fetches files directly from mobile device. Therefore, its latency is the highest, and is dominated by network communication (i.e., WiFi latency is higher than latency within the cloud data center).

The metadata server reconstructs the tree when the success rate of recent request drops below a threshold T_h. The results in Figure 4 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 11% because the classification with the updated tree fits the searching criteria better. However, it causes an increase of 27% in overhead, because the tree needs to be reconstructed more frequently.

Instead of reconstructing the tree completely, an alternative approach is to incrementally update it in the background when new searching criteria are observed. We compared this approach with the reconstruction approach. On average, with the update approach, the end-to-end delay is slightly lower (9%), and the overhead is 20% higher compared the reconstruction approach due to the extra overhead incurred by updates.

Finally, 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 5.

Figure 5(a) shows that, with the increase in the number of files, the average end-to-end latency increases for all systems. The latency increases by 8%, 5% and 11% on average for CAFDS, Chord, and SPOON, respectively. 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 overhead 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. Thus, the increase of average latency in CAFDS is lower than that of SPOON.

With an increasing number of file requests in each experiment, the end-to-end latency is increased by 27%, 11% and 16% on average for CAFDS, Chord, and SPOON, respectively, as shown in Figure 5(b). 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, each VM and the metadata server need to process a larger number of requests. This causes it to increase the average latency.

To increase the scalability of CAFDS, we split the decision tree into multiple chunks, with each chunk responsible for a number of requests. These chunks are placed onto different machines for faster processing. Also, if more than one VM stores the same file, we implemented a round-robin method to select a VM to respond the request. This scalable design is implemented with both the decision tree reconstruction and the aforementioned decision tree update approach. As shown in Figure 5(b), the scalable design of CAFDS improves the average latency by 19% and 13% respectively.

VI. Related Work

This paper addresses the issue of file discovery in distributed mobile-cloud (DMC) computing based on a variety of file features that include file context and file content. Traditional file search engines like Google Files Go [7] and Apple Spotlight [8] or web search engines cannot be used because they locate files 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, which run on multiple mobile devices and VMs.

Systems [9]–[13] proposed to search files in distributed and large scale file systems are not optimized for distributed processing on mobile-cloud platforms. Propeller [10] creates file indexes based on access sequences, and use them for search. VSFS [11] uses namespace-based queries to locate appropriate files. Glance [9] 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 our requirements.

Systems such as Spyglass [13] and SmartStore [12] 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.

Many P2P file systems, where file search is possible, can also be applied to our scenario. Earlier P2P systems [17], [22] were usually implemented on a single structure like DHTs [17] or structuring points in d-dimensional space [22]. Compared to CAFDS, their lookup schemes involve multiple network hops which cause an increase in the file access latency. To address this issue, newer systems [14], [15] 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.

VII. Conclusion

This paper proposed the 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. CAFDS is implemented as a service within a mobile-cloud middleware that enables apps to perform seamless file searching. A prototype of CAFDS was implemented and validated in Android and Linux. By using a simple machine learning technique, i.e., a modified decision tree, CAFDS provides lower latency file access than traditional peer-to-peer techniques. Therefore, CAFDS is expected to support novel, data-intensive DMC apps, with low-latency requirements.

Acknowledgment

This research was supported by the NSF under Grants No. CNS 1409523, CNS 1801430, SHF 1617749, and DGE 1565478.

References

[1] "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].

[2] 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 *MobiSys '10*, 2010, pp. 49–62.

[3] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in *EuroSys 2011*, 2011, pp. 301–314.

[4] 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.

[5] H. Debnath, G. Gezzi, A. Corradi, N. Gehani, X. Ding, R. Curtmola, and C. Borcea, "Collaborative offloading for distributed mobile-cloud apps," in *IEEE MobileCloud '18*, 2018, pp. 87–94.

[6] 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 *MobileCloud '16*, 2016.

[7] Google, "Files go," https://filesgo.google.com/, [Online; accessed 14-Jul-2018].

[8] A. Inc., "Introduction to spotlight," https://developer.apple.com/library/content/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.html, [Online; accessed 14-Jul-2018].

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

[10] L. Xu, H. Jiang, L. Tian, and Z. Huang, "Propeller: A scalable real-time file-search service in distributed systems," in *ICDCS'14*, 2014.

[11] L. Xu, Z. Huang, H. Jiang, L. Tian, and D. Swanson, "VSFS: A searchable distributed file system," in *PDSW'14*, Nov 2014, pp. 25–30.

[12] 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 *SC '09*, 2009, pp. 10:1–10:12.

[13] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *FAST'09*, 2009, pp. 153–166.

[14] 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, Feb 2014.

[15] 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 Comput. Surv.*, vol. 48, no. 3, pp. 36:1–36:37, 2015.

[16] "ID3 algorithm," https://en.wikipedia.org/wiki/ID3_algorithm, [Online; accessed 14-Jul-2018].

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. of the ACM SIGCOMM*, 2001, pp. 149–160.

[18] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. Khan, and H. Debnath, "Avatar: Mobile distributed computing in the cloud," in *MobileCloud '15*, 2015, pp. 151–156.

[19] "AspectJ," https://eclipse.org/aspectj/, [Online; accessed 14-Jul-2018].

[20] 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 Tran. on Cloud Computing*, 2017.

[21] D. Zhou, W. Pan, W. Wang, and T. Xie, "I/O characteristics of smartphone applications and their implications for eMMC design," in *IEEE Int. Symp. on Workload Characterization*, Oct 2015, pp. 12–21.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 161–172, 2001.