The PocketLocker Personal Cloud Storage System

Anandatirtha Nandugudi[1], Carl Nuessle[1], Geoffrey Challen[1], Emiliano Miluzzo[2], and Yih-Fahr Chen[2]

1: University at Buffalo, 338 Davis Hall, Buffalo, NY, 14260
2: AT&T Labs Research,1 AT&T Way Bedminster, NJ 07921

{ans25,carlnues,challen}@buffalo.edu,{miluzzo,chen}@research.att.com

Abstract. PocketLocker creates scalable, reliable, and performant personal storage clouds out of available space distributed across multiple personal devices. Designed to store rarely-changed files on both interactive devices with limited storage (such as smartphones) and non-interactive devices with large amounts of storage (such as storage appliances), PocketLocker differs from previous systems in not requiring that each device be able to store all available content or be configured to only view certain files. Instead, a storage orchestrator running as a cloud service distributes erasure-coded file chunks across all available devices to attempt to maximize performance and capacity and minimize energy usage at battery-powered clients while meeting configurable backup requirements. And unlike current cloud storage options, PocketLocker is free and will scale as users add devices.

We motivate PocketLocker's design by analyzing two months of file access traces taken from 100 smartphones, and evaluate its performance both using trace-based simulations to explore design parameters and measurements of a prototype Android implementation to establish real-world performance. By locating content close to where it will be accessed by mobile devices, PocketLocker provides low-latency access to large amounts of content. By exploiting mobility and charging habits, PocketLocker can meet backup requirements without draining the smartphone's battery.

1 Introduction

As smartphones become ubiquitous, it is natural to expect to have access to all of your personal content from these powerful devices—to view all of your photos and videos; play any track from your music collection; and browse through all of your previously sent chats, texts, and emails. All of these use cases require smartphones to efficiently access far more data than they can store locally, and yet both existing cloud storage solutions [6, 3] and recent research filesystem designs [10, 13] require each client store a complete replica. As users assemble clouds of personal devices—including smartphones, tablets, laptops, and desktops—that collectively contain large amounts of storage, their storage capacity should not be bottlenecked by the most storage-constrained device. Given

the cost and energy consumption of flash storage we do not anticipate mobile device storage capacity to keep pace with other types of devices.

To address this personal storage bottleneck we propose to allow users to apply the same techniques used to build reliable cloud storage to create *personal storage clouds*. By combining available space on existing personal devices, personal storage clouds can achieve a capacity far greater than offered by free cloud storage tiers. By utilizing nearby personal devices, personal storage clouds can provide better performance than cloud storage. And by applying modern approaches to reliability and availability, personal storage clouds can cope with failures and disconnections inherent to personal devices.

We present the design and implementation of PocketLocker, a system enabling scalable, reliable, and performant personal storage clouds (PSC). PocketLocker is designed to store rarely-changed files, such as photos, music, and videos, and to provide access to an entire personal storage cloud from any client device. PocketLocker exploits the locality of devices within the PSC to arrange rapid transfers over local-area networks when possible, and includes several energy-saving features to reduce battery drain on battery-powered mobile clients. While PocketLocker uses direct interaction between clients, it does not attempt to address the difficulties of building a true peer-to-peer distributed storage system. Instead, a cloud service called the *orchestrator* is used to maintain a consistent namespace and ensure that backup and availability requirements are met. Clients apply local data caching policies that reflect their usage patterns and their interaction with other clients. PocketLocker simplifies locating data within the personal storage cloud by utilizing Reed-Solomon erasure coding [15], allowing clients to reconstruct files as long as they can acquire any set of mutually-redundant chunks.

Our paper makes the following contributions. First, we examine one month of data from 100 users to better understand smartphone file access patterns. We conclude that today's users are generating and accessing far more content than can be stored directly on their mobile device, making file systems which require each client to store a complete replica unusable. However, a survey that we distributed to 47 people indicates that users do have free storage on other personal devices. These results motivate PocketLocker's design.

Second, we present the design of PocketLocker and describe how it uses multiple personal devices to build scalable, performant, and energy-efficient personal storage clouds to store rarely-changed files such as music, videos, and photos. PocketLocker uses erasure coding to divide files into multiple chunks which are distributed across all the users' participating devices—which can include smartphones, tablets, laptops, desktops, and dedicated storage appliances. PocketLockers orchestrator, which runs as a cloud service, distributes chunks across the users personal devices to maintain file availability, maximize performance, and meet configurable backup requirements.

Finally, we perform a detailed evaluation of PocketLocker that proceeds along two lines. First, we utilize our file access traces to examine the impact of several key PocketLocker design parameters and estimate the performance of file access

on PocketLocker. Second, we measure the energy consumption and performance of an Android prototype as it accesses files under a variety of real-world conditions. Our results confirm that by locating files intelligently, PocketLocker can provide mobile users with energy-efficient low-latency access to far more content than their mobile device can store locally.

Our paper is structured as follows. Section 2 presents several results that motivate and inform PocketLocker's design, which we present in detail in Section 3. Section 4 briefly describes the implementation of our current PocketLocker prototype for Android smartphones, which we evaluate in Section 5. Section 6 compares PocketLocker to similar systems before Section 7 discusses future work and concludes the paper.

2 Motivation

To better understand storage usage on smartphones and the potential to expand capacity by creating personal storage clouds, we performed several measurement studies. We were interested in answering the following questions: How much storage do users have available on their smartphones? How frequently are media files created, modified, and accessed? And how is available storage distributed across multiple personal devices?

2.1 Rate of Smartphone Storage Decline

To determine how rapidly users are creating content on their mobile devices, we ran an IRB-approved experiment on PhoneLab, a public smartphone testbed located at the University at Buffalo [11]. 288 students, faculty, and staff carry instrumented Android Samsung Galaxy Nexus smartphones and received subsidized service in return for willingness to participate in experiments. PhoneLab provides access to a representative group of smartphone users balanced between genders and drawn from a wide variety of age brackets, making our results representative of the broader smartphone user community.

We distributed a simple experiment that periodically logged the storage available on each smartphone which 105 PhoneLab joined for eight months, beginning shortly after PhoneLab users received new smartphones in August, 2013. Our log messages show users available storage declining by roughly 30 MB per day, approximately the size of 30 photos or a half-dozen music files. Aggregate capacity reflects both the rate at which users are creating content, but also the rate at which they may be moving content such as music on to their device. However, if this rate of decline continues it will only take the average PhoneLab participant three years to generate more content than they can fit onto their Samsung Galaxy Nexus [17] with its 32 GB of Flash. And this estimate shows users coping with the existing storage limitations of their personal devices. We expect that many already have far more photos, music, and video than will fit onto their mobile device.

2.2 Media Access Patterns

To obtain a more detailed picture of file access patterns for the media files that we expect users to want to access on many devices we distributed a second IRB-approved experiment on PhoneLab to collect more detailed file access patterns. By instrumenting the `bionic` C library used by Android applications, we were able to log every file open performed on all participating devices. We also logged the file size each time a file was opened. Our changes were distributed as a platform over-the-air update which PhoneLab participants downloaded in November, 2013. Participants indicated consent through a separate app. We collected one month of data from December, 2013, fi

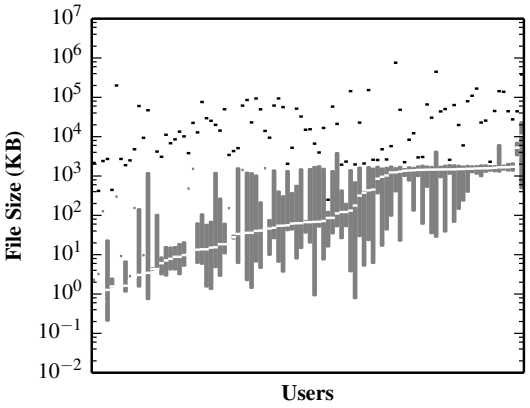


Fig. 1: **File Sizes.** Per-user distributions are shown for all media files accessed by users during the experiment. Most files are between 10 KB and 1 MB, but some are up to 100 MB.

We filtered the dataset by extension to only include media files which still left 1,780,617 opens of 147,756 distinct files by 100 users during the month. Figure 1 shows CDFs of the total number and size of the files opened by each PhoneLab user during one month, demonstrating the smartphone users access a large amount of media content from their mobile device.

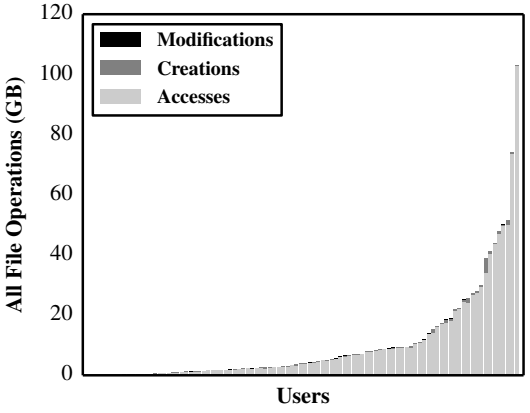


Fig. 2: **Media files are rarely modified.** Most file operations are accesses.

We marked 151,904 media files as created if they were empty when opened, and only 11,612 files as modified by comparing their sizes reported by successive open calls. This limited us to files that were opened multiple times during the trace, but we were still able to determine modifications for 89% of the file accesses we observed. Figure 2 shows modifications rates for photos, video, and audio files, demonstrating that these files are rarely modified on mobile devices. PocketLocker incorporates this assumption into its design.

Location	Min (GB)	Mean (GB)	Max (GB)
Home	8	308	3000
Work	7	97	600
Mobile	0.5	10	42
Total	15	415	3806

Table 1: **Storage space available at different locations.** Results from a survey of 47 people. Users have an order-of-magnitude less space available on mobile devices compared with their other personal devices.

2.3 Available Storage Distribution

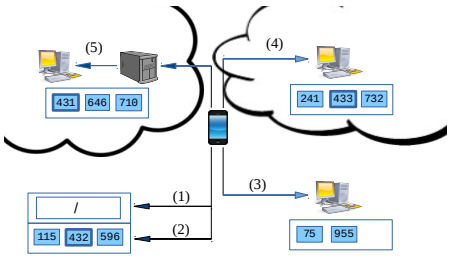
Finally, to investigate the potential to utilize other nearby personal devices as part of a personal storage cloud, we distributed an IRB-approved survey to undergraduates at our university. For each device they owned, respondents were asked to indicate how much storage capacity it had available and, for immobile devices, where they used it most: at work or school, or at home. Table 1 shows results collected from 47 volunteers. Results indicate that users have large storage capacity from other devices, such as laptops and desktops, available to them at multiple locations, while the free storage available on their smartphones was one order-of-magnitude smaller. By utilizing storage on other personal devices, PocketLocker can address the mobile storage bottleneck.

3 Design

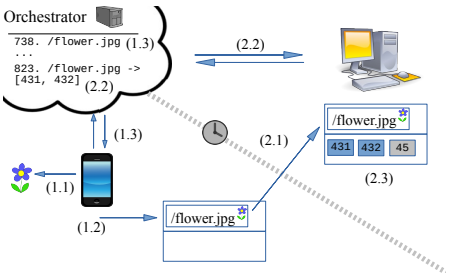
A PocketLocker personal storage cloud (PSC) consists of a set of clients—including smartphones, tablets, laptops, desktops, and dedicated storage appliances—and a cloud service called the *orchestrator*. Like most filesystems, PocketLocker uses a namespace to map paths to a set of n uniquely-identified *chunks* containing file data. Because chunks are the output of erasure coding, the number of distinct chunks required for reconstruction k is less than n, and k is stored in the namespace for each path.

The orchestrator maintains the authoritative namespace by using a monotonically-increasing counter to version all namespace modifications, including opens, renames, updates, and deletes. It also fixes the location of certain chunks to meet backup requirements and coordinates transfers between firewalled clients during open. While the orchestrator must track the location of some chunks to meet backup requirements, it does not maintain the location of all chunks. The orchestrator only requires a small amount of storage to facilitate transfers between firewalled PSC clients.

Clients contribute storage to the PSC which PocketLocker divides into a *file cache*, used to store reconstructed files, and a *chunk store*, used to store chunks. By applying updates from the orchestrator clients maintain a local cache of the namespace to efficiently perform path lookups. Clients also track what chunks for each path they have in their local chunk store. PocketLocker users configure several attributes when attaching clients:



(a) **Open.** The figure illustrates a case where the request is satisfied by locating $k = 3$ chunks: one in the client's local chunk store, and two on PSC devices on the WAN.



(b) **Creation.** This illustrates (1) path registration, performed immediately by a battery-powered client; and (2) erasure coding and chunk registration, performed later by a wall-powered client.

1. **Capacity.** The storage a client contributes to the PSC
2. **Backup.** Whether the client should be used to meet PSC backup requirements. If so, it's failure may lead to data loss. PocketLocker uses this attribute when determining where to backup chunks.
3. **Availability.** Whether files stored on the PSC can be unavailable when the client is unreachable. PocketLocker uses this attribute when determining where to store chunks so that files remain available.
4. **Interactivity.** Whether files are created or accessed on this client. PocketLocker uses this attribute when reclaiming storage.
5. **Power.** Whether the client is battery- or wall-powered. PocketLocker uses this attribute when acquiring chunks during open.

PocketLocker provides example sets of configuration options for common types of devices. A NAS appliance would be used for backup and availability, non-interactive and wall-powered. A laptop would be used for backup but not for availability, interactive and battery-powered. A smartphone would not be used for backup or availability, interactive and battery-powered. A desktop would be used for backup, interactive and wall-powered, and could or could not be used for availability depending on whether it was regularly shut off and whether the user cared if they were able to access their PSC when it was.

3.1 Creating, Modifying, and Deleting Files

To reduce energy usage on battery-powered clients, PocketLocker separates creation into two steps which can be performed on different clients: (1) path registration, a lightweight operation; and (2) erasure coding and chunk registration, a heavyweight operation. Figure 3b illustrates the process. When a file is created the creator moves the file into its file cache (1.1) and immediately registers the path with the orchestrator (1.2), which immediately publishes it to other clients to avoid path collisions (1.3). During the second part of creation, once the file is erasure coded (2.1) n new chunks will be created and registered with the in

setting up our experiments and providing information about the testbed. orchestrator, which assign them unique IDs and associates the set of IDs with the path (2.2). The client then moves the chunks into its chunk store (2.3). Battery-power clients will wait to transfer the file for a period of time configured as part of the backup process, described later in Section 3.4.

Modifications to existing PocketLocker files create a new version of a file. They require an additional round of erasure coding and distribution of new chunks. Because updating files is a heavyweight operation, PocketLocker is designed for files that are rarely or never changed, such as the media files in our traces. Renames simply alter the path associated with existing chunks, and deletes removes the path from the namespace.

Both updates and deletions invalidate chunks which clients add to a reclamation list, but chunks are not removed until storage is needed. Because clients do not coordinate chunk removal with the orchestrator, PocketLocker provides no guarantees about the existence of old version or deleted files. However, because the reclamation list is processed in FIFO order, modifications are generally removed first. Providing stronger semantics would require more client-orchestrator coordination which we have chosen to avoid.

3.2 Opening Files

To open a file, the opener first verifies that the path is valid. If the file is already in the file cache, the open completes immediately. Otherwise, the client maps the path to the n chunk IDs and locates k as follows:

1. **Local chunk store.** If the opener has k chunks of the file in its chunk store it reconstructs the file and adds it to its file cache.
2. **LAN transfer.** If the opener is on a LAN with other PSC clients it will broadcast a *chunk request* identifying the path and chunks it needs to its LAN PSC neighbors which will each report which requested chunks they store. PocketLocker clients discover neighbors using a simple UDP broadcast. Based on the replies the opener will acquire any needed chunks and add them to its chunk store. If it has k chunks, then the open continues as in Step 1. As an optimization, if an opener requests k chunks for a path and a PSC neighbor has the reconstructed file in its file cache, it will offer to transfer the file instead of chunks. This optimization is also applied in Steps 3 and 4.
3. **WAN transfer.** If the opener has not acquired k chunks after Step 2, it forwards the remaining request to other reachable PSC clients and processes replies as in Step 2.
4. **Orchestrated transfer.** If at the end of Step 3 the opener still does not have k chunks, it forwards the remaining request to the orchestrator. At this point the orchestrator may be able to facilitate transfers with clients not publicly reachable in Step 3, or the open may fail.

Figure 3a highlights the above process. The client needs chunks 431-433 to reconstruct a file. The client first checks its local cache (1) and is unable to locate

the file. It is, however, able to locate chunk 432 in its own chunk store (2). A search of LAN devices does not turn up any of the desired chunks (3), but the client finds chunk 433 on a WAN device (4). Finally, the Orchestrator is able to locate the last chunk, 431, on another WAN device (5).

PocketLocker reduces energy consumption at battery-powered clients in two ways. First, it allows them to delegate opens to a wall-powered client which receives a delegated chunk request and then proceeds as in Step 2: issuing any additional chunk requests on the battery-powered client's behalf and transferring all chunks to the opener when the open completes. Second, all clients will prefer to transfer chunks from wall-powered clients in Steps 2 and 3.

Depending on where required chunks are located, opening a file can take a variable amount of time. We allow apps to request a notification if an open may require more than a configurable amount of time, allowing them to notify the user or move themselves temporarily into the background until the required transfers complete.

Finally, PSC clients can request files as soon as they receive the path creation notification, meaning that this can occur before the file has been erasure coded and the chunks registered. In this case the open request only specifies the path, and clients reply if they have the file in their file cache. Normally the file creator will be the only PSC client with the file and required to transfer it to the opener. If the opener is wall-powered, it then performs the erasure coding and creation continues as described previously. We expect that in most cases when files are requested before they have been erasure coded the user has moved the creator onto the same LAN with the opener—such as when a user opens a smartphone photo on their laptop. If so, the time and energy required to transfer the file to the opener should be minimal.

3.3 Performance

PocketLocker attempts to use all available client storage to enable reliability, availability, and performance by intelligently locating chunks within the PSC. PocketLocker reduces the latency of file accesses in two ways. First, because many file accesses occur soon after the file is created, PSC clients opportunistically acquire k chunks of newly created files after receiving creation notifications from the orchestrator. On wall-powered clients this is done immediately; battery-powered clients wait until their next charging session. To evenly distribute chunks between clients to help meet later backup requirements, these chunk requests identify the path but not the chunk IDs, allowing the client receiving the request to provide distinct subsets of k chunks of the n available to different clients. Initial chunk requests also disable the optimization described previously to prevent the creator from transferring the reconstructed file rather than file chunks.

Second, PSC clients track local file access patterns to intelligently manage their local chunk store when reclaiming space. When under storage pressure, after emptying their reclamation list, clients can either (1) remove reconstructed files from their file cache or (2) remove chunks from their chunk store. Because

erasure decoding is more efficient than erasure encoding, clients first remove files in their file cache such that they have enough chunks in their chunk store to reconstruct.

At this point removing either files or chunks allows the client to make latency tradeoffs between different files. For example, keeping one chunk of many files reduces the access latency of all but provides low-latency access to none[1]; at the other extreme, keeping complete files—either in the file store or as k chunks—provides low-latency access to a smaller set of files but higher latency for the rest. Because file chunk size varies, removing one chunk of a large file can create more space than several chunks of smaller files, but removing chunks of more files increases the probability that a chunk will be required during open. We compare several algorithms for reclaiming storage in Section 5.1 evaluating their performance on our traces.

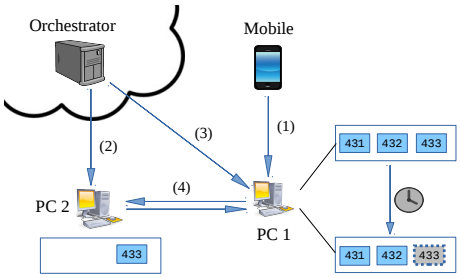


Fig. 4: **Backup.** A file is received and chunked by a powered device. Under the direction of the Orchestrator, pinned chunks are distributed to different devices.

Interactive and non-interactive clients reclaim storage differently. Interactive clients keep statistics on their own chunk access patterns and utilize them during reclamation, but do not track chunks transferred to other devices in response to chunk requests. Because non-interactive clients do not access files locally, they only keep statistics on chunks accessed to respond to chunk requests. As a result, interactive clients optimize for their own behavior, while non-interactive clients optimize for the clients they interact with.

3.4 Backup and Availability

The orchestrator both meets backup requirements and ensures availability by *pinning* chunks at clients to ensure that k chunks will always be available—as long as the client configured as available are reachable—and survive client failures. Pinning prevents clients from removing chunks during reclamation. PocketLocker allows users to configure their PSC to not lose any files older than a certain time interval (the backup window) if up to a certain number of clients fail (the backup threshold).

Figure 4 shows the most common steps in the backup process. A powered device receives and chunks a file originally created by a mobile device (3.1). All chunks are initially pinned by default. Next, the Orchestrator orders PC 2 to request chunk 433 from PC 1 (3.2), and PC 1 to unpin chunk 433 after the chunk

[1] Note that wall-powered PocketLocker clients can also repeat the erasure coding process to reconstruct missing chunks for files in their file store, but do to the overhead of erasure coding battery-powered clients will not.

has been copied to PC 2 (3.3). Next, PC 1 fetches and pins chunk 433 from PC 2, and PC 1 unpins chunk 433 (3.4).

Backup and availability requirements can reduce the usable size of the PSC depending on the distribution of storage contributed by clients and how they are configured. For example, a single small client can limit the size of the entire PSC if its storage must be used for backup. Or, a single large client may find its storage underutilized if it is not marked as available. PocketLocker estimates the capacity of the PSC at configuration time as the lesser of (1) the sum of all the capacity contributed by clients marked as available and (2) the sum of the storage contributed by the smallest backup clients required to meet the backup threshold. The tradeoff between client attributes and the PSC capacity is presented to the users when they configure clients and choose their backup threshold. Remaining PSC space is not unused: PocketLocker uses it to improve performance by caching chunks and reconstructed files, and to allow users to recover deleted files and old file versions.

When the orchestrator is unable to meet the backup or availability requirements the PSC is full and new files cannot be created. The user is warned when the PSC is nearing capacity and requested to add storage or remove files. To allow file access, interactive clients reserve a portion of their storage for the file cache; to allow chunk transfers, all clients reserve a portion of their chunk store for unpinned chunks.

The backup window allows PocketLocker to reduce energy usage on battery-powered clients by not forcing them to immediately transfer created files to other PSC clients or receive pinned chunks required for backup. When new files are created on battery-powered client, the client begins attempting to offload the file to a wall-powered client, which will perform the second part of the file creation process, including erasure coding and distributing chunks to other clients. Our current algorithm waits a configurable portion of the backup window for the device to be plugged in, and if that time window expires it transfers the file as soon as it reaches an energy-efficient network such as a wired or Wifi connection. When the backup window is about to expire, any available connection—including mobile data networks—is used as a last resort. Users are warned that short backup windows will produce high energy consumption when configuring their backup window.

Users can report client failures to PocketLocker manually or configure PocketLocker to consider a backup client as failed if the orchestrator cannot reach it for a period of time. Once a new client has been attached to the PSC after a failure, the orchestrator will immediately rerun the pinning algorithm described in Section 3.6 which will cause the new client to request chunks needed to meet the backup requirement. In certain cases erasure coding may need to be repeated for some files to recover the full set of n chunks, but this can proceed using any k chunks that are available.

3.5 Erasure Coding Parameters

The erasure coding parameters affect the design of the PocketLocker PSC in two ways. First, if n is smaller than the number of backup clients then the orchestrator may need to move a chunk from one client to another to rebalance storage usage while meeting backup requirements. Since this is undesirable, we choose n to be equal to the number of devices initially configured for backup.

Second, k determines both the chunk size—which is equal to the file size divided by k—and the overhead of erasure decoding, which increases with k. Using larger values of k and creating larger numbers of smaller chunks allows more even storage distribution over clients, and allows clients to make finer-granularity tradeoffs between storage and access latency by caching between 1 and k chunks of the file in their chunk store. However, due to PocketLocker's focus on supporting battery-powered clients, we set $k = 2$ to minimize the energy overhead of erasure decoding.

3.6 Chunk Pinning Algorithm

Periodically the orchestrator collects a list of chunks they are storing from all PSC clients and runs a *chunk pinning algorithm* to determines where to pin chunks to meet the user's backup and availability requirements. Our current placement algorithm uses a greedy approach that meets the backup requirements and availability requirements in separate passes. If the size of the PSC is constrained by the backup requirement, the availability pass proceeds first in order to avoid reducing capacity on clients needed for backup. If the size of the PSC is constrained by the availability requirement, the order is reversed.

In each pass, for each file the orchestrator begins with the client with the most capacity and pins chunks until the requirement is met. The backup pass stripes chunks across backup clients to meet the backup requirement, while the availability pass stacks chunks onto available devices to meet the availability requirement. The algorithm avoids transfers when possible by considering what chunks are already pinned or available in each client's store.

When clients receive a list of pinned chunks from the orchestrator, they retrieve any chunks they are missing using the chunk request process described previously. To ensure that chunks for newly-created files are not evicted before they can be pinned by the orchestrator, chunks for new files and file updates are initially pinned after creation at all clients. The next time the backup algorithm runs, many of these chunks will be unpinned.

3.7 Offline Operation

PocketLocker assumes clients are generally connected, but can supporti periods of disconnection. Disconnected clients can access any files in their file store or that they can reconstruct using chunks in their local chunk store. Any changes to the namespace, such as creations, are cached. When the client reconnects, it downloads namespace updates from the orchestrator and identifies any conflicts.

Because it is designed to store media files, PocketLocker does not attempt to merge conflicting versions. Instead, it asks users to choose between updates or to rename the file.

3.8 File Metadata

Finally, to support media files that may require metadata for browsing, such as photo thumbnails, PocketLocker allows metadata files up to a size limit to be associated with files stored in the PSC. Metadata files are stored in a separate part of each client's storage and retrieved during the initial chunk requests that follow file creation. Unlike chunks, however, metadata files are not reclaimed, since we assume their storage overhead is limited.

4 Implementation

We have implemented PocketLocker PSC as an Android background service on both interactive and fixed non-interactive devices. Galaxy Nexus and Nexus 5 smartphones constituted the mobile interactive devices, and Android x86 virtual machines [4] running on desktops served as the fixed non-interactive. The PocketLocker service runs in the background and exposes APIs to provide clients access to the files stored in the user's PSC. It also maintains chunk placement in the cache as directed by the orchestrator.

We chose to implement PocketLocker as a user application rather than integrating the service with the file system so that users are not required to have root privileges to install PocketLocker on their devices and PocketLocker can be distributed via the Android Play Store. [1] On both interactive and non-interactive devices, the PocketLocker service maintains the local file and chunk cache according to the placement directions calculated by the orchestrator. On fixed devices, PocketLocker additionally offers a pair of network services. The first, the discovery service, responds to chunk requests that are issued by interactive devices on the same local network. The second, the HTTP service, facilitates the transfer of newly created files and chunks among the user's PSC devices as per the chunk placement scheme.

PocketLocker exposes its APIs both to the orchestrator, to receive local cache maintenance directions, and to local client applications, to provide access to user files. PocketLocker clients interact with the PocketLocker service via the *binder* driver framework in Android. The binder facilitates thread safe inter process communication in Android. The orchestrator was implemented using the Tornado and Flask web frameworks. The orchestrator listens to status updates by the user's PSC devices and tracks and maintains the cache information at each of the devices in the user's PSC using an *SQLite* database. To push information to user's PSC device, the orchestrator uses the Google Cloud Messaging (GCM) framework to communicate information about new file creation and chunk placement with the user's PocketLocker devices.

5 Evaluation

We evaluate PocketLocker in two ways. First, we analyze the the file access traces we collected on PhoneLab to determine the impact of parameters important to PocketLocker's design. We also use the traces as inputs to a trace-based simulation to compare approaches to performing client storage reclamation. Second, we perform detailed measurements of our PocketLocker prototype engaging in the various types of file accesses described previously. Our results indicate how utilizing nearby clients can improve performance, and also how PocketLocker enables energy-efficient operation on battery-powered clients.

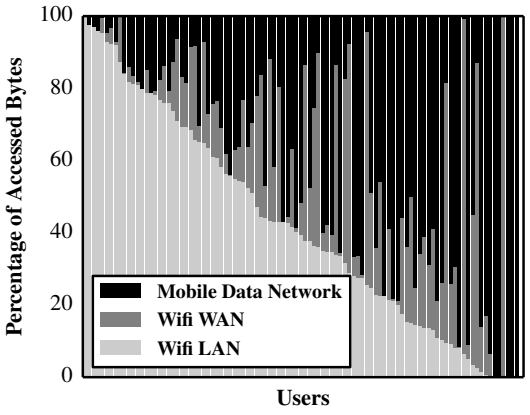


Fig. 5: **Connectivity During File Accesses.** Placing PSC clients on each user's two most frequently-used Wifi networks could absorb a large portion of their file access activity.

5.1 Trace Analysis

PocketLocker relies on nearby clients to improve performance of file accesses. In the best case, other PSC clients are located on the same LAN. To determine whether nearby clients can assist with file accesses, we performed further analysis of the traces described in Section 2.

Because PhoneLab only provides visibility into participant smartphones, we have to infer where users would have other PSC clients nearby. To do so, we simulated the presence of PocketLocker PSC clients on the two Wifi networks that each user spent the most time connected to, which could represent home and work networks. We then divided file accesses into three categories: (1) ones that occur on the same LAN with a simulated PSC client, (2) those that do not occur on a PSC LAN but still occur while the user is connected to a high-speed and energy-efficient Wifi network, and (3) those that occur when the user is connected to a mobile data network[2]. Figure 5 shows the results; for around half of the users, even without a local cache half of the file accesses could be served by two clients placed at their most used Wifi networks.

We were also interested in how many file creations could be offloaded to powered clients by delaying transfer until the user plugged their smartphone in to charge. Figure 7 shows per-user distributions of the of time between file creations

[2] We found no file accesses that occurred more than five minutes from log messages indicating the presence of a mobile data network, a reflection of the always-connected nature of smartphones.

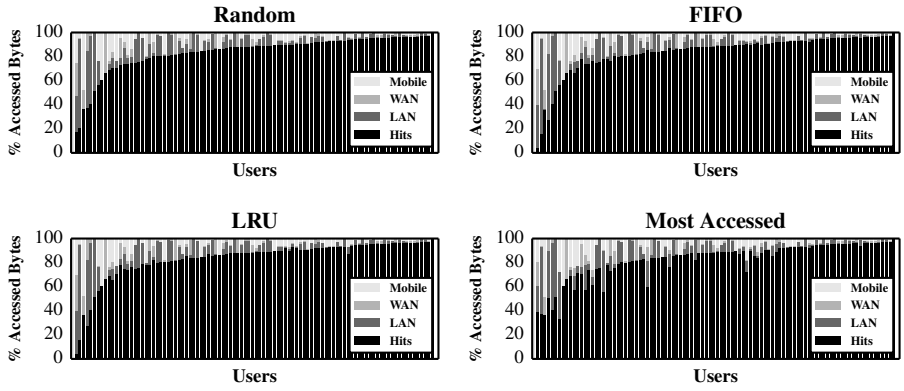


Fig. 6: **Comparison of Reclamation Algorithms.**

and the next charging session. For all users, the median is under 10 hours with worst-case maximums approaching a day. Overall, the results suggest that by delaying the initial file transfer required during creation for a portion of the user's backup window, PocketLocker can enable energy-neutral transfers and reduce overhead on battery-powered clients.

Finally, we built a simple trace-based simulator to experiment with different policies for managing the mobile client chunk store. We configured each PSC client smartphone with 1 GB of storage, considerably less than the amount of file accesses we observed during our one-month experiment, and managed the chunk store using four different algorithms: random eviction, first-in-first-out (FIFO), least-recently-used (LRU), and least-accessed first (Access). Figure 6 compares the results. When file accesses missed the chunk store, we classified the access as described previously based on the smartphone's connectivity at that moment. Surprisingly, we did not observe any large performance differences between these algorithms, although they were able to manage the local chunk store to absorb a large number of file accesses. A great deal of inter-user variation is also visible, and we are continuing to study how to better

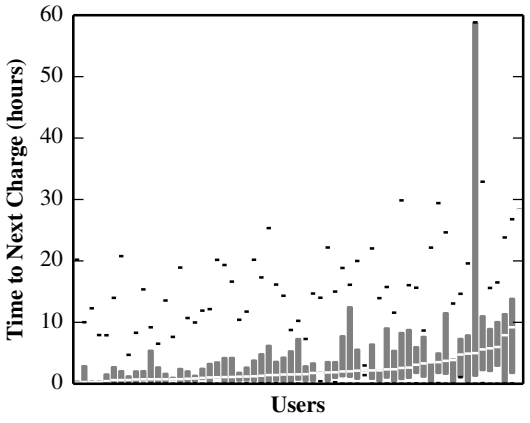


Fig. 7: **Time Until Next Charge After File Creation.** Separating the process of creating files into two steps allows PocketLocker to reduce energy consumption by performing transfers during the next charging cycle.

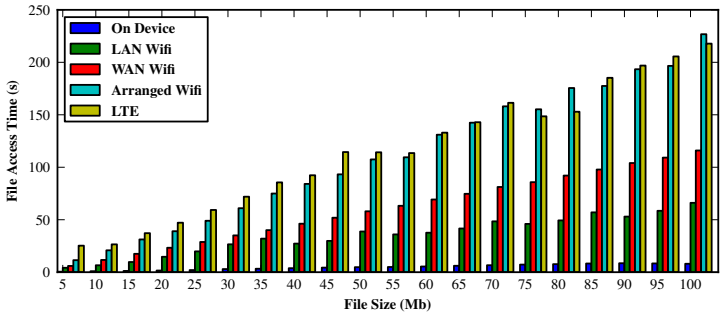


Fig. 9: **PocketLocker file access times.** Figure illustrates the times required to access files of various sizes by PSC in different types of connectivity.

adapt PocketLocker's reclamation algorithms to the specifics of each users file access patterns.

5.2 Prototype Performance Evaluation

We evaluated the prototype of the implementation described in Section 4 in two ways. First, we measured the time required to access files of various sizes with devices connected to different networks types. Secondly, we measured the energy consumption to access files. In our experiments we chose $k = 2$ as the number of chunks required for reconstruction. We used Samsung Galaxy S4 and Nexus 5 smartphones as interactive devices, and utilized Android VMs running PocketLocker as fixed nearby devices.

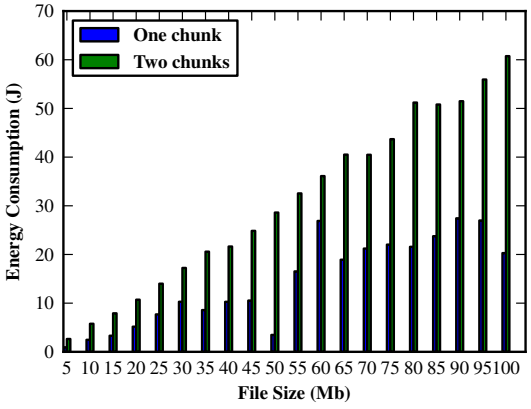


Fig. 8: **PocketLocker energy savings.** The figure illustrates the savings in energy when an interactive device downloads one chunk compared to downloading two chunks to access the file.

File access: Figure 9 illustrates the time required to download files of different sizes with clients having to download k chunks to reconstruct the file when connected to different networks. The *On Device* scenario denotes the time required only to reconstruct the chunks locally to reconstruct the original. This is the best scenario as there is no download of chunks involved. In *LAN Wifi*, we have a fixed device present on the same LAN as the interactive device. The device downloads both chunks from the fixed device and is the fastest compared to any other connection type. *WAN Wifi* has fixed devices that are publicly accessible

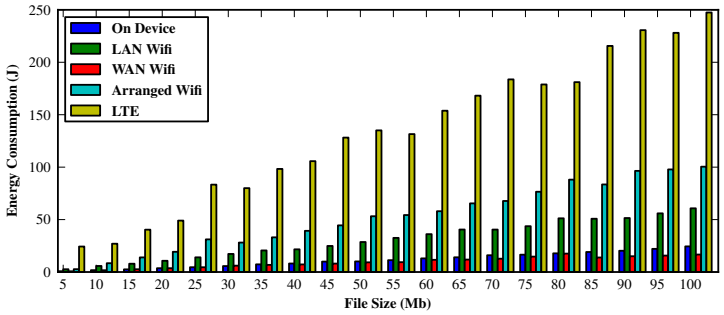


Fig. 10: **PocketLocker energy consumption.**Figure illustrates the energy consumption on interactive device to access files of different sizes from fixed devices in various types of network.

over the Internet to interactive devices. WAN Wifi is analogues to downloading files from the cloud today. *Arranged Wifi* presents the scenario where the fixed devices are not publicly accessible and data transfers are done via a relay. Here, the access times to open a file when the chunks are downloaded on the LAN Wifi are almost 50% faster when compared to WAN Wifi scenario. This result is encouraging, as we envision most chunk transfers happening over LAN Wifi.

Energy Consumption: We used the Monsoon power monitor [2] to measure the energy consumption for file access time for the scenarios described in Section 5.2. Figure 10 illustrates the energy consumption on the interactive device. As expected, data transfers over the cellular network consumes the most amount of energy. We also noticed that the energy consumed for the WAN Wifi was lesser when compared to LAN Wifi. We believe this occurred because the smartphone was connected to an open access point for WAN transfers whereas the interactive device was connected to an access point with WPA security for LAN Wifi. Figure 8 compares the energy consumption of file access when downloading two chunks with the energy required to access the file by downloading one chunk. The energy consumption of PocketLocker when accessing the file by downloading one chunk is less than the consumption to access the file by downloading both chunks. This is a positive result for PocketLocker as it stores only some of the required chunks instead of all chunks under storage pressure.

6 Related Work

Mobile devices, being relatively new, did not contribute to the design of prototype distributed file systems. Early systems such as Coda [9] and Ficus [8] were concerned with addressing the base problem of file caching and replication. The limitations of mobile devices, particularly constrained storage and energy and intermittent connectivity, were not relevant. Standard network file systems such as NFS [12] did not provide direct offline access or redundancy.

By contrast, there are robust commercial solutions such as TimeMachine [5] that furnish redundant storage from any device. These cloud solutions are also typically limited in space and use third party storage.

The approach taken by EnsemBlue [13] focuses on replicating files among mobile devices. Users can specify file groups that are automatically replicated. Cimbiosys [14] narrows this approach, implementing data filters such as file type to determine replication policy. Files that do not match the filter are not replicated. These approaches limit access to files that can fit on a particular user's device. Additionally, since a file will not always be replicated, there is no specific attempt to provide file backup. Since offline edits are allowed, conflicts occur and must be resolved. PRACTI [7] focuses on maximizing the tradeoffs of the general goals of consistency, replication and independence. This necessarily unfocuses the specific needs of mobile storage.

PocketLocker aims to make all files in the PSC available. Which files are maintained locally are determined by usage patterns and network conditions. Those that are not are still available with a possible delay. The size of the PSC can thus greatly exceed the local storage of a particular device. The chunk distribution system of PocketLocker minimizes the impact of device failure and ensures file redundancy.

The Eyo system [16] provides a distributed unified namespace. While file metadata is automatically replicated, replication of file data is left to rules specified by client programs. Thus, files may not be replicated against failure. If a user wants to access a nonlocal file, the system can furnish its current location but does not automatically retrieve it. Editing a file offline can result in a conflict that must be resolved. The system addresses storage pressure by pruning file version history without respect to possible loss of redundancy.

The concept of separating the distribution of file metadata from data underpins another system, Ori [10]. Accessing remote file data depends on being able to access that device directly. Otherwise, the call fails. Ori permits users to move versioned file histories among devices—permitting offline editing but incurring storage overhead and producing conflicts. File backup focuses on versioning. Whether a file is replicated depends upon whether the user has mounted a remote system. Implementation of deliberate redundancy, in the form of multiple copies on multiple devices, remains a function of user choices.

PocketLocker handles replication of both file metadata and data directly. The system, having a bird's eye view of all storage devices, can ensure that files are always chunked and replicated to disparate devices to guard against failure. Distribution of the chunks is tuned to the differing storage capacities of different devices. Storage reclamation policy follows file history and usage patterns in order to maximize backup potential. The centralized design of PocketLocker also allows it to handle potential remote access issues. If a file or chunks are not directly reachable from a client device due to firewall issues, the Orchestrator can often mediate an indirect relay transfer rather than simply failing.

7 Conclusion

PocketLocker addresses an emerging need of mobile systems by crafting a personal storage cloud from multiple personal devices. It targets storing rarely changing files. An intelligent orchestrator arranges storage to maximize usage of devices of different sizes and minimize network costs. PocketLocker is free and uses no additional devices. System storage and backup policies are based upon data gleaned from an extended testing using 100 smartphones.

References

1. Google Play. `https://play.google.com/`, 2014.
2. Monsoon Solutions Inc. Power Monitor. `http://www.msoon.com/LabEquipment/PowerMonitor/`, 2014.
3. One account. All of Google. `https://drive.google.com`, 2014.
4. Run Android on Your PC. `http://www.android-x86.org/`, 2014.
5. Time Machine. `www.apple.com/support/timemachine`, 2014.
6. Your Stuff, Anywhere. `http://www.dropbox.com/`, 2014.
7. N. M. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In NSDI, volume 6, pages 5–5, 2006.
8. R. G. Guy, J. S. Heidemann, W.-K. Mak, T. W. Page Jr, G. J. Popek, D. Rothmeier, et al. Implementation of the ficus replicated file system. In USENIX Summer, pages 63–72, 1990.
9. J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. ACM Transactions on Computer Systems (TOCS), 10(1):3–25, 1992.
10. A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières. Replication, history, and grafting in the ori file system. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 151–166. ACM, 2013.
11. A. Nandugudi, A. Maiti, T. Ki, F. Bulut, M. Demirbas, T. Kosar, C. Qiao, S. Y. Ko, and G. Challen. Phonelab: A large programmable smartphone testbed. In Proceedings of First International Workshop on Sensing and Big Data Mining, pages 1–6. ACM, 2013.
12. B. Nowicki. Nfs: Network file system protocol specification. 1989.
13. D. Peek and J. Flinn. Ensemblue: Integrating distributed storage and consumer electronics. In Proceedings of the 7th symposium on Operating systems design and implementation, pages 219–232. USENIX Association, 2006.
14. V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In Proceedings of the 6th USENIX symposium on Networked systems design and implementation, pages 261–276, 2009.
15. I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. Journal of the Society for Industrial & Applied Mathematics, 8(2):300–304, 1960.
16. J. Strauss, C. Lesniewski-Laas, J. M. Paluska, B. Ford, R. Morris, and F. Kaashoek. Device transparency: a new model for mobile storage. ACM SIGOPS Operating Systems Review, 44(1):5–9, 2010.
17. Wikipedia. Galaxy Nexus. `http://en.wikipedia.org/wiki/Galaxy_Nexus`.