University of Tennessee, Knoxville

TRACE: Tennessee Research and Creative Exchange

Masters Theses Graduate School

12-2013

A Privacy-Aware Distributed Storage and Replication Middleware for Heterogeneous Computing Platform

Jilong Liao
University of Tennessee - Knoxville, jliao2@utk.edu

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes

Part of the Computer and Systems Architecture Commons, and the Data Storage Systems Commons

Recommended Citation

Liao, Jilong, "A Privacy-Aware Distributed Storage and Replication Middleware for Heterogeneous Computing Platform. " Master's Thesis, University of Tennessee, 2013.
https://trace.tennessee.edu/utk_gradthes/2619

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Jilong Liao entitled "A Privacy-Aware Distributed Storage and Replication Middleware for Heterogeneous Computing Platform." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Qing Cao, Major Professor

We have read this thesis and recommend its acceptance:

Hairong Qi, Wei Gao

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

A Privacy-Aware Distributed Storage and Replication Middleware for Heterogeneous Computing Platform

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Jilong Liao

December 2013

© by Jilong Liao, 2013
All Rights Reserved.

This thesis is dedicated to my LORD who changes my heart.

Acknowledgements

I would like truly thankful to all the people for their dedicated efforts on helping me throughout the past three years. I am especially thankful to Dr. Qing (Charles) Cao, my major advisor, who put a great effort to guide me on my research, support my research ideas and academic decisions throughout my career. I appreciate him for listening my life situation, encouraging me to make my best decision for my life and helping me get through tough days.

I am also grateful to my committee members Dr. Hairong Qi and Dr. Wei Gao for their supervision, discussion and advice on both my graduate education and research. I will be always thankful for Dr. Qi's precise and constructive suggestions on our collaborated research projects. And I appreciate Dr. Gao's great comments and literature collects in the Mobile Network System Design course which illuminates a new area for me.

I would like to appreciate my colleagues, Kefa Lu, Zhibo Wang, Yanjun Yao, Lipeng Wan, Sisi Xiong and Zheng Lu in the Laboratory for Autonomous Interconnected, and Embedded Systems (Lanterns) leaded by Dr. Cao, and Yue Tong and Yifan Wang, with whom I spent a lot of time discussing and validating my research ideas and decisions.

Finally, I am grateful to my wife Yanjin Li, my parents and my parents in law. Without their help, care and support, this work would not have been possible.

Abstract

Cloud computing is an emerging research area that has drawn considerable interest in recent years. However, the current infrastructure raises significant concerns about how to protect users' privacy, in part due to that users are storing their data in the cloud vendors' servers. In this paper, we address this challenge by proposing and implementing a novel middleware, called *Uno*, which separates the storage of physical data and their associated metadata. In our design, users' physical data are stored locally on those devices under a user's full control, while their metadata can be uploaded to the commercial cloud. To ensure the reliability of users' data, we develop a novel fine-grained file replication algorithm that exploits both data access patterns and device state patterns. Based on a quantitative analysis of the data set from Rice University [Shepard et al., 2011], this algorithm replicates data intelligently in different time slots, so that it can not only significantly improve data availability, but also achieve a satisfactory performance on load balancing and storage diversification. We implement the Uno system on a heterogeneous testbed composed of both host servers and mobile devices, and demonstrate the programmability of Uno through implementation and evaluation of two sample applications, *Uno@Home* and *Uno@Sense*.

Table of Contents

1 Introduction **1**

2 Related Work **5**

3 Design Principle of Uno **7**

3.1 The Uno System Core 7

3.2 The Replication Subsystem 8

3.3 The Simplified API Design 9

3.3.1 list() 10

3.3.2 publish() 10

3.3.3 backup() 10

3.3.4 search() 10

4 Implementation of Uno Operations **12**

4.1 Core Architecture 12

4.2 Object accesses 14

4.3 Replication Algorithm 15

4.4 Local object management 17

5 Evaluation of Uno Replication Subsystem **20**

5.1 Availability Rate 21

5.2 Number of Replicas 22

5.3 Load and Storage Balance 23

5.4 Tradeoff of Time Slot Length 24

6 Uno-based Application Case Studies **25**

6.1 Case Study 1: Uno@Home for File Sharing 25

6.2 Case Study 2: Uno@Sense for Sensor Sharing 28

7 Conclusions **31**

Bibliography **32**

Vita **36**

List of Tables

1 Notation table 15
2 Object changes notification implementation. 19
3 The major metrics at different time slot. 24
4 The Android runtime footprint of Uno@Home and Uno@Sense 28
5 Typical Evaluation Benchmark Specs 29

List of Figures

1 The diagram of Uno platform. 3

2 The online rate of different devices (by hour) 9

3 The data usage histogram and kernel density 9

4 The master server diagram of Uno system 13

5 The client diagram of Uno system (using an Android device as an example) 14

6 Monte Carlo Test for Greedy Algorithm 18

7 The available rate by devices (sorted) 21

8 Each device's average online rate (sorted) 21

9 The replicas needed by each user across all time slot 22

10 The load balance of each device (sorted) 23

11 The storage load on each device (sorted) 23

12 Balance change with storage budget 24

13 The Sample App View 26

14 Upload Performance of Uno@Home 27

15 Download Performance of Uno@Home 27

16 Battery Level of Uno@Home 28

17 Dynamic Power of Uno@Home 28

18 Battery Level of Uno@Sense 30

19 Dynamic Power of Uno@Sense 30

20 Battery Level of Sense-on-request Policy 30
21 Power Consumption of Sense-on-request Policy 30

Chapter 1

Introduction

With the recent advances in cloud computing, one critical challenge faced by commercial clouds, such as Google, Amazon S3, Evernote, and Dropbox, is data privacy. Although the benefits brought by those public clouds are undeniable, such as guaranteed data availability, and access from anywhere in any device, the users' concerns about data privacy are never eliminated. In contrast, customers such as small business companies have been reluctant to store sensitive data in these cloud-based storage services for privacy concerns [Feng et al., 2011].

The fundamental dilemma is that whenever users' data are stored in the cloud vendors' machines, they have little control over the data, which may be leaked due to hacker intrusions or unexpected mistakes, or used by the vendors for business purpose like advertising. Even after significant efforts have been invested into assuring privacy through approaches such as better isolation and protection mechanisms [Takabi et al., 2010], few companies are willing to use Google Drive or Dropbox to synchronize highly sensitive information, such as internal financial reports, because a data leak could cause significant consequences, especially in today's society that malicious cyber-attacks are becoming increasingly more outrageous [Cashell and of Congress. Congressional Research Service, 2004].

Besides, the pricing strategy is still not attractive if people want to use more than the free tier. We are hesitate because we are doubt the value of the current cloud storage can give us, let alone we have paid for the storage for our personal devices such as smartphones, tablets and computers. Those personal devices can give us impressive storage freedom near our hands.

In this thesis, we address these challenges on privacy by separating data into content (physical data) and metadata, and only keeping metadata with cloud vendors. In other words, we still rely on the public cloud's infrastructure for data synchronization, but only to the extent that we trust them with storing the metadata of critical files, i.e., the file names, their creation and modification dates, and so on. This way, a user can easily browse and access another file in another device, as if those files were synchronized with a common cloud storage service, but without any concerns on leaking the contents of the files themselves. Consequently, we are interested in whether we can achieve the same level of reliability with this approach compared to storing all data in cloud vendors' machines, as the primary challenge is that individual users' devices are much less reliable. To this end, we develop a novel replication algorithm to increase the availability of files. This algorithm aims to maximize the likelihood that critical files will continue to be made available even when their original storing devices have been turned off, and its design is based on a quantitative analysis of real file and device usage traces made available by Rice University [Shepard et al., 2011].

Based on the replication algorithm, we develop *Uno*, a unified object-oriented storage and backup system that seamlessly ties cloud vendors' storage services with privacy-sensitive user needs. Its architecture is shown in Fig. 1, where heterogeneous devices exploit the existing cloud infrastructure for storing metadata, but exchange physical data directly between themselves for improved user access and data availability. Note that data sharing is not limited to conventional data, but also includes real-time sensor data streams when applicable, e.g., a user may access the current readings of sensors on their smartphone remotely. A practical, economical

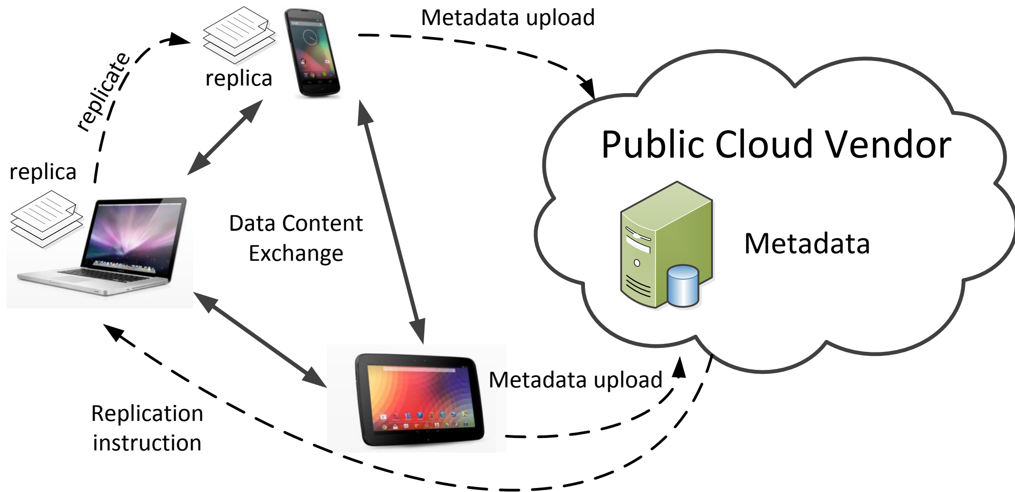


Figure 1: The diagram of Uno platform.

advantage of Uno is that because it only stores metadata on cloud service providers, its sharing and synchronization is not limited by their quota policies and pricing plans. Finally, Uno is also more convenient than remote desktop applications in that it allows scalable inter-operation between heterogenous devices: instead of developing remote desktop interface between any two devices, Uno only requires each new device to support a common set of APIs to join the existing "private device cloud".

The key contributions of Uno are as follows. First, to our best knowledge, Uno is the first distributed storage system that explicitly addresses the challenges to store privacy-sensitive data of users. Second, to specifically handle those devices that have a lower availability than cloud vendors' servers, Uno provides an adaptive replication algorithm that dynamically evaluates the value of files, and replicates them across devices to maximize data availability. Finally, Uno provides APIs to application developers, whose effectiveness are demonstrated through two case studies, *Uno@Home* and *Uno@Sense*. We also present evaluation results to demonstrate that both applications have reasonably acceptable performance at runtime.

The rest of this theis is organized as follows. In Chapter 2, we describe the most relevant related work. Chapter 3 illustrates the design of Uno. Chapter 4 describes the implementation details of Uno, and its replication mechanism. Chapter 5 evaluates the performance of Uno through replication mechanism validation and Chapter 6 presents two case studies. This theis ends with conclusions in Chapter 7.

Chapter 2

Related Work

Cloud based storage has drawn considerable interest in both industry and academia in recent years [Armbrust et al., 2009]. One major challenge that concerns enterprise and personal users to use commercial vendors' storage services is privacy. Although recent works [Jia et al., 2011], [Pearson, 2009] and [Chen et al., 2011] have been trying to solve this issue through advanced encryption techniques and anonymous P2P sharing, our research effort with Uno explores the possibility to prevent the privacy issues in the first place, by only storing physical data locally as opposed to in the cloud. In this sense, Uno works in a similar manner as classic networked file systems such as Andrew File System (AFS) [Howard et al., 1988, Kistler and Satyanarayanan, 1992] and Google File System (GFS) [Ghemawat et al., 2003]. Specifically, Uno adopts AFS's idea to store the owners' data locally rather than collecting them into a central server. Uno also uses heartbeat messages for liveness detection, similar to GFS.

The design of Uno faces similar challenges on replication and availability as previous distributed storage systems. For example, Ivy [Muthitacharoen et al., 2002] is a read/write peer-to-peer based file system allowing users to store data in a distributed environment, in which cross-platform replication, inconsistency, conflicts and flexibility are significant issues. To address the replication problem, [Veeraraghavan et al., 2009] proposes Polyjuz, a fidelity-aware mobile platform

replication system; to solve the inconsistency problem, [Parker Jr et al., 1983] considers a mutual detection method, and [Petersen et al., 1997] proposes flexible update methods for weakly consistent replication; to address conflicts, [Kumar and Satyanarayanan, 1995] and [Reiher et al., 1994] use file type recognition and content semantics to detect update orders.

Uno's replication model is remarkably different from these previous work in that Uno takes into account the unique file access and device availability patterns in personal devices, as well as develops efficient algorithms that make decisions based on a variety of factors such as load-balancing, energy efficiency, and bandwidth availability, to maximize the likelihood that the files will be available upon users' needs. Another work, Eyo [Kaashoek et al., 2010], presents a device transparent personal data synchronization platform, and puts special focus on version control.

Finally, two industry commercial products share the similar idea with Uno by only using local personal devices as the storage base. One is BitTorrent Sync [BitTorrent, 2013] which allows users to synchronize their documents and files among their personal devices in a P2P fashion. The other is aeroFS [Air Computing, 2013] which allows the user to synchronize their documents and files as well, but in a file system fashion. However, Uno is quite different from the two products that the basic concept is different. Uno treats all personal devices as a whole, not as individual devices, so the user does not need to synchronize manually. The Uno replication system is capable of handling synchronization and replication issues, and the user, therefore, only need to request to access the data. In addition, Uno uses object oriented abstraction which can handle not only documents and files, but apps and sensors as well.

Chapter 3

Design Principle of Uno

3.1 The Uno System Core

The design principles of Uno stem from its adoption of the principle of *separation of physical data and metadata*. The physical data refer to the actual file contents, while the metadata refer to the meta information of these files, such as their names, access control lists, and last modification dates. Uno stores physical data in devices under the owner's control, while the metadata can be synchronized and accessed from any device with the help of commercial cloud computing services. This way, the cloud vendor's servers can be considered as *metadata keepers* because they reliably store all metadata of documents.

To manage data and resources (e.g., sensors on smartphones) under this data model, Uno proposes an object-oriented resource abstraction, where heterogeneous resources and entities, such as sensors or documents, are unified into objects with their own operations. For example, although a document object may be updated, a sensor object may be read-only. In fact, the object concept in Uno is extremely flexible: the storage devices themselves, such as the smartphones or users' computers, are also mapped into objects, so that a unified access control model is applicable.

Based on this model, each object is represented by both metadata and (optionally) physical data. All metadata are uploaded and synchronized into the cloud service back-end periodically, so that all devices have access to these metadata. For example, upon obtaining the name, size, and version of a document that is stored on another device through its shared metadata, a user can quickly decide whether this object is the desired one or not. On the other hand, metadata can also be used to control users' access rights: if a smartphone is put into the offline mode, its metadata will be updated to reflect this change, and any future accesses from other devices will be notified of this change. Finally, a third use of metadata is in the replication phase. Given the current metadata of different devices, such as their spare memory and battery level, Uno can decide where to replicate files according to users' needs under timing and energy constraints.

One natural use of metadata is to create a resource sharing graph that enables Uno to keep track of legitimate user accesses. The graph is bipartite with the vertices on one side as devices, and the vertices on the other side as resources. To construct the sharing graph, each computing device reads the metadata as input, and generates the sharing graph as well as historical changes as output. The sharing graph is periodically updated so that all devices will have a consistent view on these updates over time.

3.2 The Replication Subsystem

Since Uno is running on personal devices, we need a mechanism to prevent from data loss or data unavailability. Different from those data replication services in data centers [Mohd. Zin et al., 2012], our design stems from observations on the unique characteristics and usage patterns of users' mobile devices. These observations are drawn based on the Livelab dataset [Shepard et al., 2011]. First, we find that these user devices are highly mobile, and exhibit relatively diversified online or offline usage patterns. The average online ratio of a device is only about 30% (Fig. 2), meaning

that we can hardly guarantee its availability over long periods of time. Second, these devices are usually battery powered, and their storage and bandwidth are significantly limited. Third, the data usage of these devices are highly personalized and unique according to the users' needs. For example, we illustrate three different apps' usage patterns in Fig. 3. Note that we treat the app usage traces and data access traces as the same, because each app only accesses its dedicated data on the mobile devices. As shown here, each device's availability and data access is highly divergent from each other: some may have very high availability while the others can be very low. Considering the availability and access variances, we seek to maximize the availability of most frequent data objects for users' accesses. We present the algorithm design in Chapter 4.3.

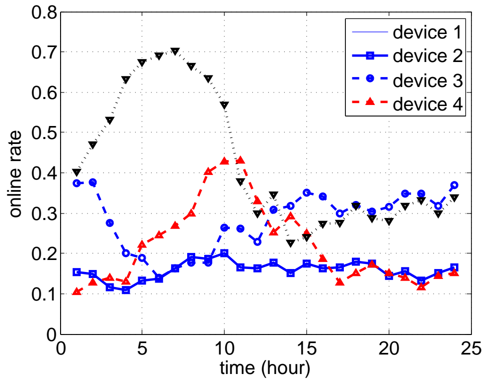


Figure 2: The online rate of different devices (by hour)

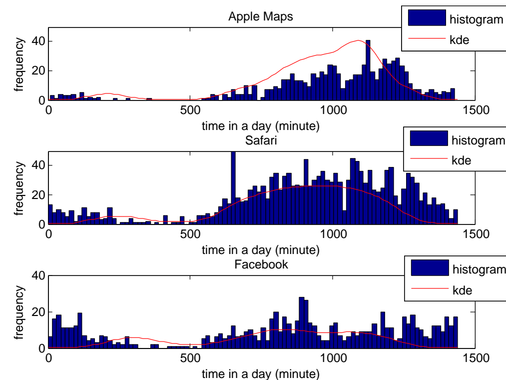


Figure 3: The data usage histogram and kernel density

3.3 The Simplified API Design

Uno is developed not only to directly interact with the end user, but also support APIs to third-party developers to construct additional applications. Currently, Uno supports the following APIs: *list()*, *publish()*, *backup()*, and *search()*.

3.3.1 list()

This interface provides a simple way to browse the available objects from a given device. It allows users to retrieve object lists in different levels, such as active devices, documents in a particular device, or sensors across the entire Uno collection. The sharing graph will first check accessibility information to ensure that the user has enough privilege to query the corresponding objects before they can obtain more fine-grained information.

3.3.2 publish()

This interface publishes certain objects as available for other users/devices, by essentially uploading its metadata into the backend cloud server. Note that the physical data remains in the owner's machine, but the user may not explicitly publish such information into the cloud. After publishing the metadata, the sharing graph is updated accordingly, where a new graph is constructed and replaces the old copy. This approach is lightweight: publishing metadata is much faster compared to uploading physical data in practice, and all changes are made visible within a sufficiently short time scale.

3.3.3 backup()

The API *backup()* provides a simple interface such that a user can back up their physical data from one device to one or more other devices. This API is also called extensively in the replication phase (discussed in Chapter 4.3), but it can also be initiated by the third-party developer.

3.3.4 search()

This API is necessary in two folds: one is to initiate a keyword based search in the sharing graph for any matches based on metadata, and the second is to start a

distributed search on individual devices to locate any matches in the physical data. Note that the latter operation is considerably more expensive compared to the former because it requires the participation of multiple devices. In practice, such a search operation is allowed only if the user has sufficient privilege to access the remote objects.

Chapter 4

Implementation of Uno Operations

Following the design principles of Uno, we discuss in details how we turn those principles into implementation.

4.1 Core Architecture

The core architecture of Uno contains two processes on each participating device, which we call the master process and the client process. Note that the master and the client are running on the same device, whether it is a PC, a laptop, or a smartphone. Indeed, there is no separate, dedicated *server* node, since all devices form a peer-to-peer relationship to share data. The advantage of this approach is that all devices can make decisions autonomously without being affected by any single point of failure.

The master node interacts with the metadata stored by the cloud vendors directly. Given that it is extremely rare for the public cloud vendor's service to be unavailable, the master process on each device can run very reliably without being affected by the downtime of its peer devices. Each master process contains four major components, shown in Fig. 4: the object metadata cache, the front service, the query processor, and the replication subsystem. The object metadata cache maintains all metadata information from all devices, and stores them locally on the current device. Whenever one device needs to access objects remotely, its master process will invoke the front

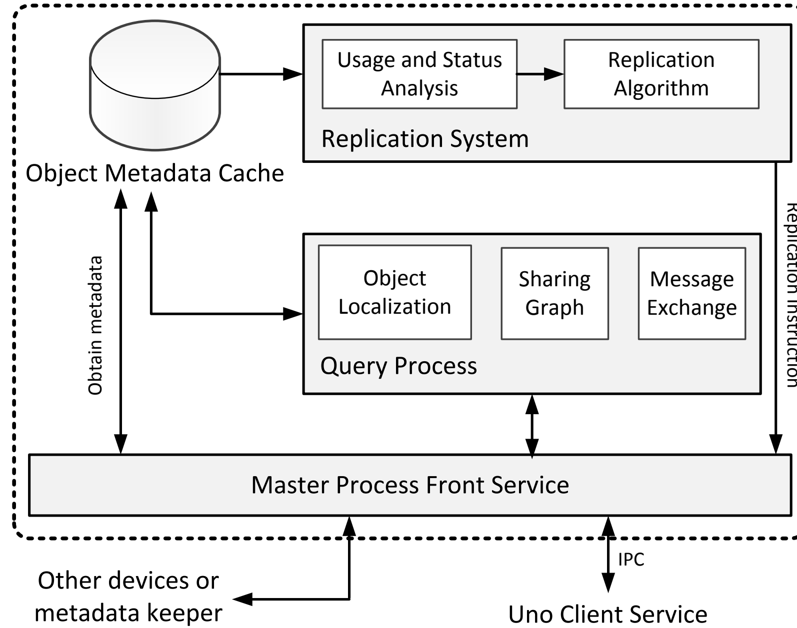


Figure 4: The master server diagram of Uno system

service to initiate a request. Such a request is sent directly to the target device, which will be handled by the corresponding query processor. Specifically, the query processor will check the latest local sharing graph (which is always synchronized through the cloud service) to decide whether the access is legitimate. If it is, the query processor will send a response containing the requested object data back. Finally, to ensure proper load balancing and improve availability of data, the replication subsystem periodically checks the current status of the cloud through the metadata updates, and issues replication requests to appropriate devices.

On the other hand, the client process on each Uno device plays the role of managing the local physical data, extracting their metadata, and sending updates to the master process periodically (via IPC). Additionally, the client process keeps track of the usage of every single remote access, and monitors status change of the device. Its architecture is shown in Fig. 5.

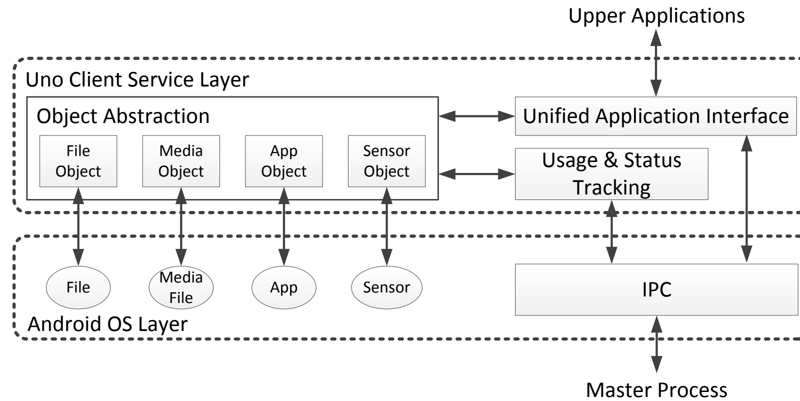


Figure 5: The client diagram of Uno system (using an Android device as an example)

4.2 Object accesses

As each device relies on the use of access rights to decide whether an access will be legitimate, there are three types of access rights: *public*, *group*, and *off-line*. The reason to maintain these three privileges is to accommodate different users' needs when publishing objects. The users can also mark objects as completely offline by stopping updating metadata. Once the metadata staleness is detected by other devices, it will be removed. As part of the object attributes, the access right information of an object can either be updated through periodic heartbeats, or by explicitly invoking the *publish()* API.

Finally, if a device is offline, its metadata will be temporarily marked as offline in that case. Of course, a user can always completely remove an object by stopping reporting its metadata, so that all other devices will quickly detect that such metadata is stale, and will need to be removed. As part of the object attributes, the access right information of an object can either be updated through periodic heartbeats, or by explicitly invoking the *publish()* API to update the metadata pool maintained on other devices.

The detailed procedure for object access works as follows. Imagine the master process on a device now initiates a request to another object on a remote device. By

looking up the metadata stored locally, the master process determines the *(name, location)* pair of the target device, if permitted by the access rights; otherwise, the master process will abort this request because it knows the remote device will not accept this request. Next, the master process needs to check if the destination object is available or not, by sending a *heartbeat* message to the remote device. If the destination object is available, the master process will establish a TCP connection for transmitting physical data. If not, the master device will abort its action. If the two master processes on the sender and receiver devices successfully establish a TCP connection, the object's physical data can be securely exchanged. Note that, in practice, there may be multiple target devices that keep replicas of the same object. Whenever a request is made, the master process chooses target devices in a load-balanced manner, so that no single device will become the bottleneck of the entire system.

4.3 Replication Algorithm

Before we discuss the replication algorithm, we introduce the notations which will be used in this section in Table 1.

Table 1: Notation table

$\mathbf{R}^{(t)} = [R_{ij}^{(t)}]$	the feasible replication match at slot t
$\mathbf{p}^{(t)} = [p_i^{(t)}]$	the device availability vector at slot t
$\mathbf{A}^{(t)} = [A_i^{(t)}]$	the object availability vector at slot t
$\mathbf{f}^{(t)} = [f_i^{(t)}]$	the object access frequency vector at slot t
$\boldsymbol{\gamma}^{(t)} = [\gamma_i^{(t)}]$	the replication factor vector at slot t
$\mathbf{b}^{(t)} = [b_i^{(t)}]$	the device storage budget vector at slot t
$p_{max}^{(t)}$	the maximum availability among all devices
D	the total number of devices
K	the total number of objects

We first define the device availability $p_i^{(t)}$ within the time slot t as the fraction of the device's online time among the total time of the slot (Eq. 4.1). If we replicate object $\gamma_i^{(t)}$ times across all available devices, the object's availability $A_i^{(t)}$ is defined in Eq. 4.2, where the replication assignment matrix $R_{ij}^{(t)}$ is defined in Eq. 4.3. Besides, the number of replicas necessary for object i should be related to the object's access frequency $f_i^{(t)}$, because the mobile devices are storage and energy constrained. Therefore, we do not need to place extra replicas somewhere if the object is seldom accessed. Therefore, we use a *sigmoid* function to control the replication factor $\gamma_i^{(t)}$ in Eq. 4.4. Observe that by its design, Eq. 4.4 can restrict the replication factor in a scale between 10% and 90% of the total number of devices, so that the replication can neither decrease the availability too much at the low access level, nor can overwhelm the devices to make too many replication copies.

$$p_i^{(t)} = \frac{T_{online}^{(t)}}{T_{total}^{(t)}} \tag{4.1}$$

$$A_i^{(t)} = 1 - \prod_{i=1}^{D}(1 - p_i^{(t)} R_{ij}^{(t)}) \tag{4.2}$$

$$R_{ij}^{(t)} = \begin{cases} 1 & \text{replicate object } i \text{ to device } j, \\ 0 & \text{otherwise.} \end{cases} \tag{4.3}$$

$$\gamma_i^{(t)} = \lceil \frac{D}{1 + e^{-5(f_i^{(t)} - 0.5)}} \rceil \tag{4.4}$$

Our goal of the replication algorithm is to maximize the availability among all objects under the constraints of the replication factor and device storage budget. Each object is associated with an access frequency $f_i^{(t)}$ such that $f_i^{(t)} A_i^{(t)}$ means the effective availability when an object is accessed. Therefore, we can define the optimization problem as:

$$\begin{aligned} \underset{R_{ki}^{(t)}}{\text{maximize}} \quad & \sum_{k=1}^{K} f_k^{(t)} (1 - \prod_{i=1}^{D}(1 - p_i^{(t)} R_{ki}^{(t)})), \\ \text{subject to} \quad & \sum_{k=1}^{K} R_{ki}^{(t)} \leq b_i^{(t)}, \\ & \sum_{i=1}^{D} R_{ki}^{(t)} = \gamma_k^{(t)}. \end{aligned} \tag{4.5}$$

In other words, the maximization problem is a replication assignment problem that maximizes the total effective availability under the limited storage constraints. The problem is similar to the replication placement problem in P2P networks [Ye and Chiu, 2007], where it is proved to be NP-complete. Therefore, we solve this problem through a greedy approach in Algorithm 1. The sorting of $\mathbf{f}^{(t)}$ and $\mathbf{p}^{(t)}$ costs $O(Klog^K + Dlog^D)$, then the algorithm iterates all K objects, where each of which needs to probe at most D devices for replication. Overall it runs in $O(K \cdot D)$ time. Thus, the total complexity is $O(K \cdot D)$.

To better understand the performance of the greedy algorithm, we want to evaluate its approximation ratio. To this end, by using Monte Carlo methods, we can empirically evaluate its performance, and found that the algorithm is at least a 1.2-approximation of the optimal solution. More specifically, since the problem itself is NP-Complete, we cannot get the optimal solution OPT by exhaustive search in polynomial time. Instead, we derive a *super-OPT* solution z^* in Eq. 4.6 by setting the device availability vector $\mathbf{p}^{(t)}$ to $[p_{max}^{(t)}, ..., p_{max}^{(t)}]$. That is, z^* is defined as follows:

$$z^* = \sum_{k=1}^{K} f_k^{(t)} (1 - \prod_{i=1}^{D} (1 - p_{max}^{(t)} R_{ki}^{(t)})) \tag{4.6}$$

It is obvious that $z^* \geq OPT$. Next, we evaluate the performance of our greedy algorithm by running the Monte Carlo test on the greedy algorithm and compare it with z^*. Our results (Fig. 6) show the greedy algorithm can achieve more than 83% of *super-OPT* results, which means the algorithm is at least 1.2-approximation.

4.4 Local object management

One essential issue for the client process on each device is that it needs to detect if an object has been recently changed. For example, it needs to detect modified documents as soon as such documents are saved. A brute-force way is to keep a list of all the objects, and periodically scan the local file system. Unfortunately, this approach will

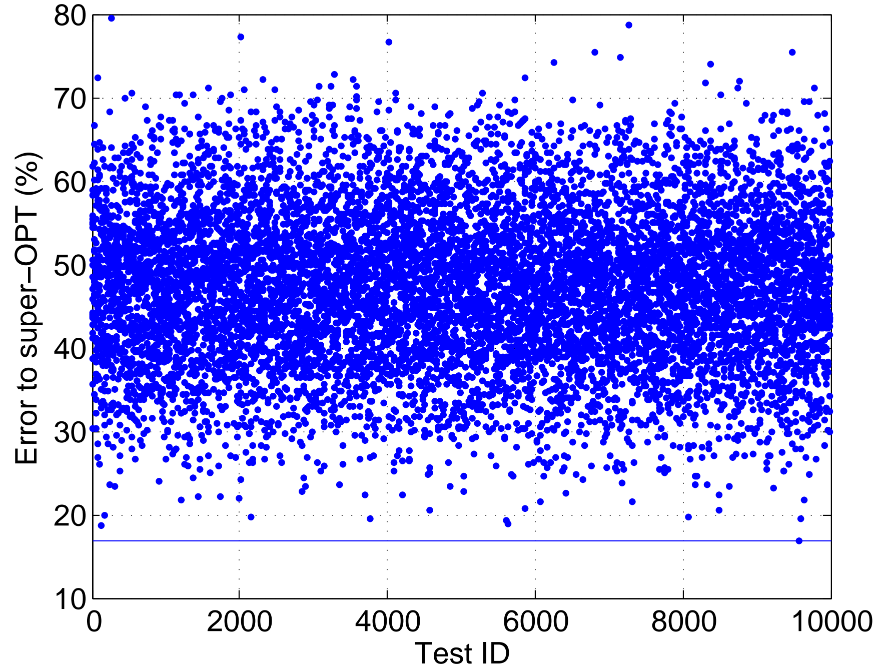


Figure 6: Monte Carlo Test for Greedy Algorithm

Algorithm 1 Greedy-Search

Input: $\mathbf{p}^{(t)}, \mathbf{f}^{(t)}, \mathbf{b}^{(t)}, \boldsymbol{\gamma}^{(t)}$
Output: $\mathbf{R}^{(t)}$

```
1:  init R^(t) ← 0
2:  sort descendly f^(t) and p^(t)
3:  for k = 1 to K do
4:      replica = 0
5:      for d = 1 to D do
6:          if b_d^(t) > 0 then
7:              reduce b_d^(t) by 1
8:              mark R_kd^(t) as 1
9:              increase replica by 1
10:         end if
11:         if replica ≥ γ_k^(t) then
12:             break
13:         end if
14:     end for
15: end for
```

reduce both the operating system's performance and the user experience considerably. Instead, Uno resolves this issue by listening to the file system's low level events, and if there is a change (e.g. deletion), the client process captures this signal, extracts any metadata change, and initiates an update through the cloud front service. Multiple implementation techniques that we adopt are summarized in Table 2, where multiple OS systems are supported using different OS-specific APIs.

Table 2: Object changes notification implementation.

Operating System	Implementation Approach
Linux/Unix	*inotify* [Kerrisk, 2010]
Windows	NTFS *Change Journal Records* [Microsoft, 2010]
Android	*FileObserver* [Google, 2010]
iOS/OS X	*File System Events* [Apple, 2010]

Chapter 5

Evaluation of Uno Replication Subsystem

In this chapter, we systematically present the evaluation results for Uno's replication subsystem. We verify our replication policy by demonstrating its performance with the data set from Rice University, which covers 24 students' iPhone usage for more than one year. It tracks each user's app usage, device status and power, and other parameters. We use the app usage and device status to evaluate our replication system. For our experiment scenario, we choose a set of 21 users' data, whose data are complete, among all data available. In total, $1,125,786$ data object requests from these dataset traces are replayed in our simulations. In particular, we split each user's app usage trace half by half; the first half is used for training while the second for testing.

During the training phase, we perform the data analysis in time slots. Each time slot is a 4-hour period, so we have 6 slots in any day. For each time slot, we analyze each app's usage pattern to derive its data access pattern, as well as each device's online rate. Next, we perform data replication based on our replication algorithm. Each device has the storage budget of 150 units. Finally, in the testing phase, we evaluate whether an app can access its data from any one of multiple replicas

successfully. In addition, since both app usage patterns and device status patterns are dynamically changing over time, we periodically update those information and collect real-time statistics from our simulations.

5.1 Availability Rate

The measured availability rate, defined as $\frac{N_{available}}{N_{total}}$, is the number of successful accesses among the total number of accesses for an arbitrary data object. When the replication algorithm is applied, the average rate is improved to 90.89%, in contrast to the online rate of individual devices (Fig. 8), which is measured as just 34.46% on average. The results are plotted in Fig. 7, where 15 out of 21 users' trace obtain an availability rate greater than 90%, while only two devices cannot reach 80%. Overall, the improvement is huge (about 200%) compared to the device online rate. In addition, we also investigated the reason of the lower availability rates in this experiment. It turns out that a couple of requests are made by those devices in the 5th and 6th time slots, which are the lowest in the availability rate among all devices. This explains the low availability rate as observed.

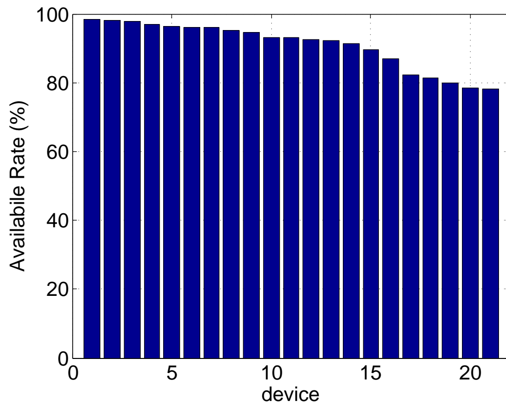


Figure 7: The available rate by devices (sorted)

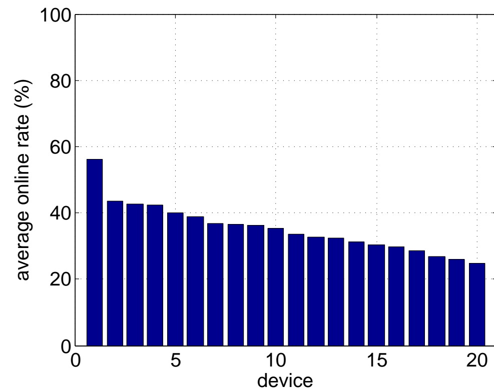


Figure 8: Each device's average online rate (sorted)

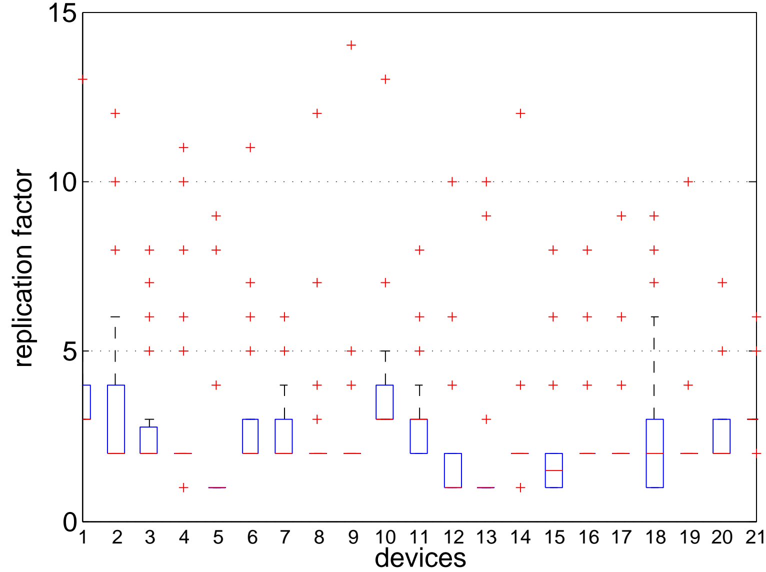


Figure 9: The replicas needed by each user across all time slot

5.2 Number of Replicas

In our replication policy, the value $\gamma_k^{(t)}$ is related to the object frequency by Eq. 4.4, which is supposed to achieve both high data availability and low cost. In this experiment, we collect all the 21 devices' average $\gamma_k^{(t)}$ values across all apps at all time slots. The average $\gamma_k^{(t)}$ values across all devices is 1.15. The $\gamma_k^{(t)}$ is used to decide the replication at each time slot, but the total number of replicas for a resource is still unknown. Assume we have 6 time slots, the total number of replicas can be up to $1.15 \times 6 = 6.9$. Fortunately, our evaluation (Fig. 9) shows we have much fewer replicas that the average number is about 3.36, which reduces around 50% of the storage. One possible reason is that the device usually sustains its high online rate to the next time slot where the same replication in the next slot has no cost (the replica has already existed).

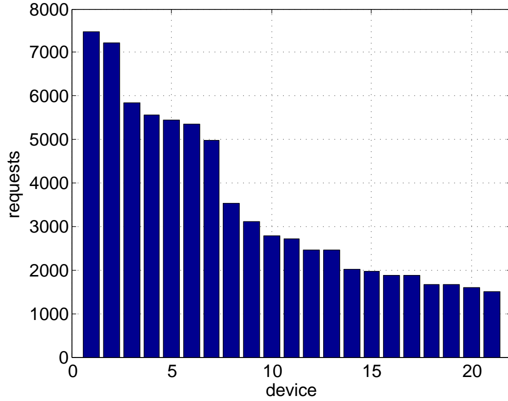


Figure 10: The load balance of each device (sorted)

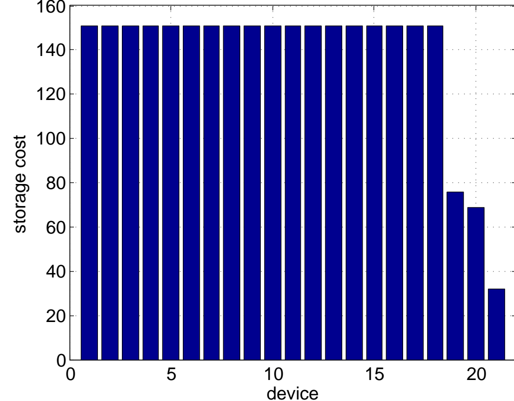


Figure 11: The storage load on each device (sorted)

5.3 Load and Storage Balance

Throughout the experiment, we counted each device's number of received requests. As is shown in Fig. 10, the top 2 devices has about 25% more load than the next 5 devices. We observe that each of the device has a very high availability compared to other devices. These two hotspots become the most popular replicators so that they increase the chance of being selected as the service device when accessing an object. Besides them, the next 5 devices are similar to each other, which indicates they are relatively well balanced, but have lower service rate than the top devices. In addition, the remaining devices' service rates form a long-tail distribution because the lower the availability rate, the smaller the chance it will be selected. We also investigate storage balance among all devices, where Fig. 11 shows that the storage load is quite balanced. In this figure, 18 devices have reached their budget limits but there is still available space in the whole system. Besides, we find the storage budget affects the balance a lot. Shown in Fig. 12, we can see that the increase of budget (although can potentially increase the availability) brings more balance problem because both load and storage variance will increase. Thus, we would not recommend to set a huge budget even if the device allows.

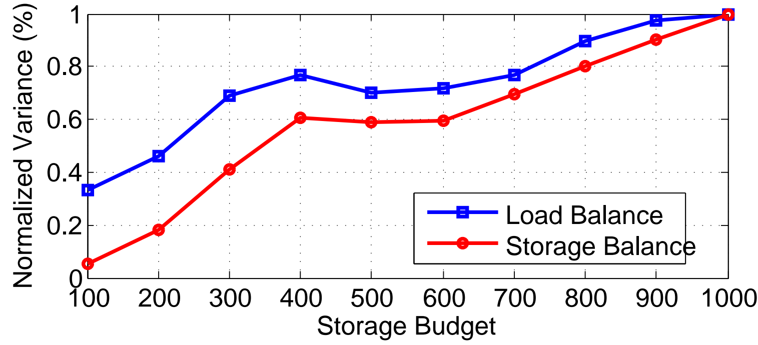


Figure 12: Balance change with storage budget

5.4 Tradeoff of Time Slot Length

Recall that we have set the time slot length as 4 hours. This section gives an empirical evaluation of this decision. Table 3 displays a series of experiment results by tweaking the time slots from 1 to 12 hours. According to the table, we have found a good tradeoff to be located between 3-hour or 4-hour time slots, for the following reasons: 1) only those time slots that are not higher than 4-hour can provide good availability rate performance; 2) compared to 3-hour or 2-hour time slots, the 4-hour selection achieves a good combined performance on the number of replicas, storage overhead, and load balancing. Therefore, we conclude that the 4-hour time slot selection is a good tradeoff.

Table 3: The major metrics at different time slot.

	Availability		Replicas		Storage		Load Balance	
Slot	*mean*	*var.*	*mean*	*var.*	*mean*	*var.*	*mean*	*var.*
1h	94.5	3.1	10.5	0.7	455.1	619.8	11836	7791
2h	93.4	4.0	9.1	0.9	315.9	523.1	11686	9286
3h	90.9	6.9	8.1	1.8	237.4	487.3	11296	9799
4h	**91.3**	**4.7**	**7.1**	**0.7**	**259.2**	**491.4**	**11388**	**10368**
6h	89.1	6.7	6.8	0.9	208.3	465.7	11077	10155
8h	87.0	6.6	5.8	0.6	173.3	405.8	10823	11131
12h	85.0	6.2	5.0	0.4	199.5	465.8	10596	12490

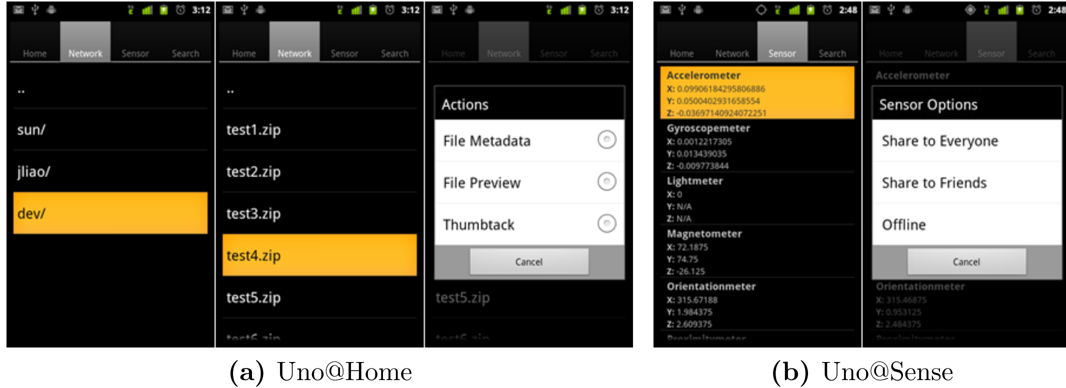
Chapter 6

Uno-based Application Case Studies

In this chapter, we demonstrate the flexibility of Uno's system APIs by designing and implementing two applications, which are deployed on a testbed that consists of heterogeneous types of devices, including one PC and several Android smartphones. These two applications are *Uno@Home* and *Uno@Sense*, where Uno@Home is a file sharing application across smartphones, much like Dropbox, and Uno@Sense is a sensor sharing service that allows different users to view each other's sensor readings remotely.

6.1 Case Study 1: Uno@Home for File Sharing

To evaluate Uno@Home, we develop it on the Android 2.3.4 operating system and deploy it over three Google Nexus S smartphones and a desktop on the University's wireless network, and evaluate its performance for over 24 hours. During this period of time, the battery on a smartphone drops from roughly 90% to 20%. A screenshot of this Android app is shown in Figure 13a.



(a) Uno@Home **(b)** Uno@Sense

Figure 13: The Sample App View

In this experiment, we generated files in sizes of 2.5MB, 5MB, 10MB, 20MB, 50MB and 100MB, then tried to access those files from smartphones remotely. Those files were transmitted over the wireless network in a peer-to-peer fashion, and we compare the obtained statistics with Dropbox and Google Docs for the same file sizes. Fig. 14 shows the comparison of uploading time. For files of relatively small size (up to 20MB), we observe that all three approaches achieve similar performances and the total elapsed time increases linearly with the size of files. However, Google Docs does not support uploading large files from smartphones, so we cannot measure the performance of large file transfers in 50MB and 100MB cases. For the remaining two approaches, Uno@Home performs better at 50MB file size, but gets surpassed by Dropbox at 100MB file size. The possible reason is that our implementation is not as optimized for larger files as Dropbox, which exploits bandwidth more effectively.

Next, we evaluate the downloading performance of these three approaches, and the results (Fig. 15) are similar: Uno@Home performs comparably well to Dropbox and Google Docs at file sizes up to 50MB, but its performance becomes worse sharply at 100MB, which probably can be attributed to the less optimized network stack of Uno compared to Dropbox.

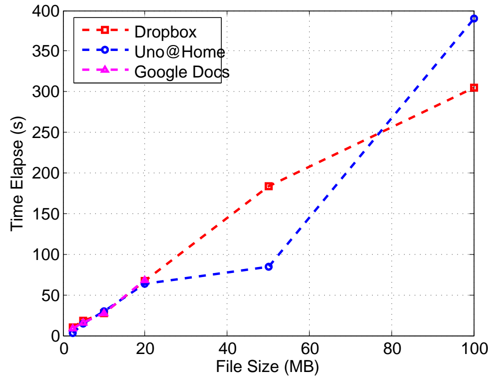


Figure 14: Upload Performance of Uno@Home

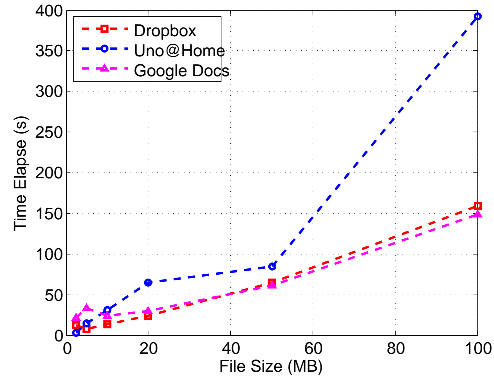


Figure 15: Download Performance of Uno@Home

Energy consumption is another significant issue we need to evaluate. In Fig. 16, we plot our evaluation results on real-time battery levels and dynamic power consumption. To compare between normal usage and Uno@Home, we set up a standardized benchmark to control the operation of the smartphone (Table 4). In this benchmark, we let Uno@Home perform back-and-forth file transfers. This leads to a steeper decrease in remaining energy compared to the case when Uno@Home is turned off. Fig. 16 and 17 illustrate the battery level and dynamic power consumption of Uno@Home compared to normal usage when running this benchmark, respectively. As expected, Uno@Home consumes additional energy compared to when it is turned off. In total, about 600MB of data were replicated and the battery life was shortened by about 300 minutes, which indicates Uno@Home costs 0.5 minutes of battery life in order to replicate 1MB data.

Finally, we also evaluate the memory footprint of Uno@Home, by measuring its application size and runtime memory usages. Table 4 compares application size and runtime memory consumption of Uno@Home to other commonly used applications in Android smartphone. As shown, both application size and memory consumption are relatively lightweight compared to other applications.

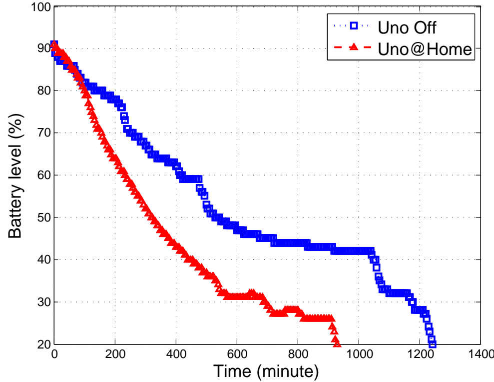


Figure 16: Battery Level of Uno@Home

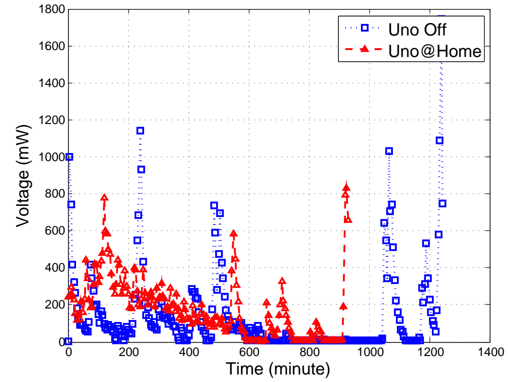


Figure 17: Dynamic Power of Uno@Home

Table 4: The Android runtime footprint of Uno@Home and Uno@Sense

Application name	Runtime memory	App size	CPU usage
Gmail	29 MB	9,850 KB	$< 1\%$
Google Docs	29 MB	4,660 KB	$< 1\%$
Dropbox	24 MB	3,550 KB	$< 1\%$
Google Services	23 MB	2,820 KB	2%
Android Keyboard	21 MB	930 KB	$< 1\%$
Google Maps	21 MB	292 KB	$< 1\%$
Google +	20 MB	23,500 KB	$< 1\%$
Uno@Home	19 MB	192 KB	$< 1\%$
Uno@Sense	19 MB	204 KB	$< 1\%$
Google Search	16 MB	44 KB	$< 1\%$

6.2 Case Study 2: Uno@Sense for Sensor Sharing

In the second case study, we implement a sensor sharing cloud on the Uno platform, which we call Uno@Sense. This study allows users to share their sensor readings (such as accelerometer and location data) between different devices, and access remote sensor readings directly, which is beneficial for applications such as crowd sensing [Philipp et al., 2011]. A screenshot of this application is shown in Figure 13b. Specifically, we carry out the experiment as follows: we deploy a total of four smartphones to users, which are divided into two groups: one group of users followed normal usage as shown in the Table 5, with a one-hour idle time between application

Table 5: Typical Evaluation Benchmark Specs

Normal Usage	Test Case	Duration (hours)
Gmail	Gmail	0.5h
-	Uno@Home/Uno@Sense	1h
Google Reader	Google Reader	0.5h
-	Uno@Home/Uno@Sense	1h
Facebook	Facebook	0.5h
-	Uno@Home/Uno@Sense	1h
Google Music	Google Music	0.5h
-	Uno@Home/Uno@Sense	1h
Google Maps	Google Maps	0.5h
-	Uno@Home/Uno@Sense	1h
Youtube	Youtube	0.5h

transitions. The second group, on the other hand, used the idle time to run the Uno@Sense application, where the user either remotely retrieved a sensor's instant readings (in the first two idle periods), or activated sensor logging for the remaining idle periods.

Fig. 18 and 19 show the energy comparison results, including battery level and power consumption measurements. As expected, Uno@Sense performs worse than normal application usage. Specifically, the average battery lifetime is reduced by 375 minutes, in exchange for $9,161,518$ sensor readings. This translates into approximately 0.00245 seconds of battery life for each sensor reading. Although this decrease in battery lifetime may appear significant, it is in fact an overestimate because in this experiment, we turned on all sensors to record as fast as possible. In practice, fewer sensor samplings will be made, leading to less energy consumption.

To demonstrate this point, we also evaluate the sense-on-request policy in Fig. 20 and 21. Specifically, the sensor object has a method *read* that permits retrieving its instant readings. However, this method will cost considerable energy if we turn on the sensor all the time to wait for the immediate readings request. To solve this issue, Uno turns off the sensors until the reading request arrives, which enables a corresponding sensor, and takes a single sample before the sensor is turned off again. Observe that with this policy turned on, the sensing tasks become much more energy

efficient. Finally, we also measured the application size and memory cost, where the results are similar to that of Uno@Home, as shown in Table 4.

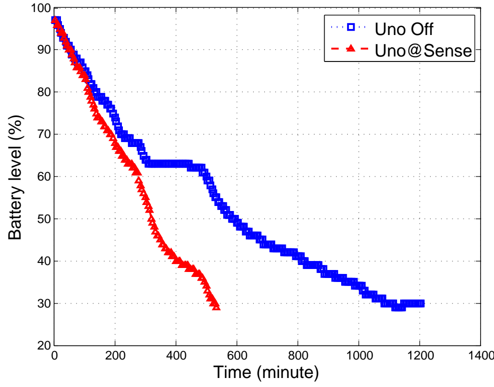


Figure 18: Battery Level of Uno@Sense

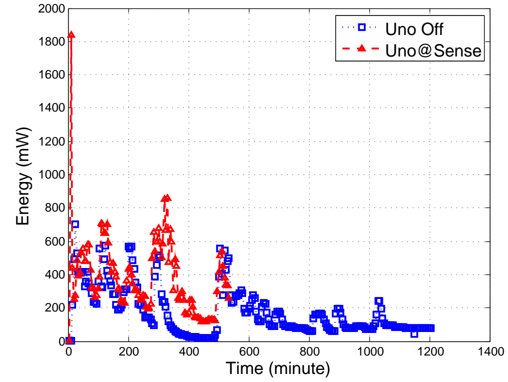


Figure 19: Dynamic Power of Uno@Sense

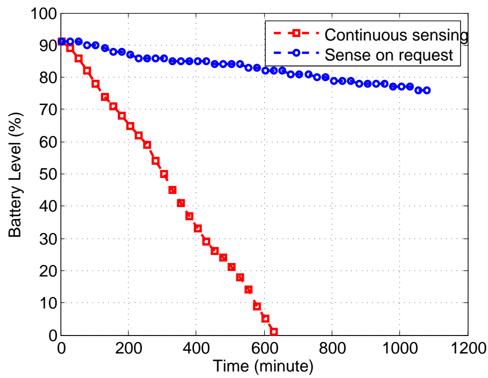


Figure 20: Battery Level of Sense-on-request Policy

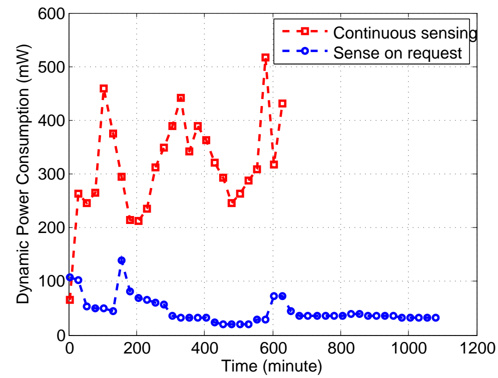


Figure 21: Power Consumption of Sense-on-request Policy

Chapter 7

Conclusions

In this thesis, we present Uno, which presents a novel object-oriented architecture for sharing heterogeneous computing, storage, and sensing resources across multiple platforms in a privacy-aware way. The key contribution is to store the data content locally across multiple personal devices instead of uploading them to the cloud servers and to provide the simplified programming environment to developers. We also proposed a fine-grained statistical replication system to guarantee the availability of data contents. Through the data analysis from Rice University and two sample applications, and the two case studies, we systematically demonstrated the effectiveness of Uno. Therefore, we believe that Uno is a good alternative to preserve a user's data privacy from commercial cloud vendors.

Bibliography

[Air Computing, 2013] Air Computing, I. (2013). aerofs. 6

[Apple, 2010] Apple (2010). Using the file system events api. 19

[Armbrust et al., 2009] Armbrust, M. et al. (2009). Above the clouds: A berkeley view of cloud computing. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*. 5

[BitTorrent, 2013] BitTorrent (2013). Bittorrent sync. 6

[Cashell and of Congress. Congressional Research Service, 2004] Cashell, B. and of Congress. Congressional Research Service, L. (2004). The economic impact of cyber-attacks. Congressional Research Service, Library of Congress. 1

[Chen et al., 2011] Chen, K., Shen, H., and Zhang, H. (2011). Leveraging social networks for p2p content-based file sharing in mobile ad hoc networks. In *IEEE MASS*, pages 112–121. IEEE. 5

[Feng et al., 2011] Feng, J., Chen, Y., and Summerville, D. (2011). A fair multi-party non-repudiation scheme for storage clouds. In *Collaboration Technologies and Systems (CTS), 2011 International Conference on*, pages 457–465. IEEE. 1

[Ghemawat et al., 2003] Ghemawat, S., Gobioff, H., and Leung, S. (2003). The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM. 5

[Google, 2010] Google (2010). File observer. 19

[Howard et al., 1988] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. (1988). Scale and performance in a distributed file system. *ACM TOCS*, 6(1):51–81. 5

[Jia et al., 2011] Jia, W. et al. (2011). Sdsm: A secure data service mechanism in mobile cloud computing. In *IEEE INFOCOM Workshops*, pages 1060–1065. IEEE. 5

[Kaashoek et al., 2010] Kaashoek, M., Morris, R., Strauss, J., et al. (2010). *Device-transparent personal storage*. PhD thesis, Massachusetts Institute of Technology. 6

[Kerrisk, 2010] Kerrisk, M. (2010). *The Linux programming interface*. No Starch Press. 19

[Kistler and Satyanarayanan, 1992] Kistler, J. and Satyanarayanan, M. (1992). Disconnected operation in the coda file system. *ACM TOCS*, 10(1):3–25. 5

[Kumar and Satyanarayanan, 1995] Kumar, P. and Satyanarayanan, M. (1995). Flexible and safe resolution of file conflicts. In *USENIX ATC*, pages 8–8. USENIX Association. 6

[Microsoft, 2010] Microsoft (2010). Change journals. 19

[Mohd. Zin et al., 2012] Mohd. Zin, N. et al. (2012). Replication techniques in data grid environments. *Intelligent Information and Database Systems*, pages 549–559. 8

[Muthitacharoen et al., 2002] Muthitacharoen, A., Morris, R., Gil, T., and Chen, B. (2002). Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44. 5

[Parker Jr et al., 1983] Parker Jr, D. S., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B. J., Walton, E., Chow, J. M., Edwards, D., Kiser, S., and Kline, C. (1983). Detection of mutual inconsistency in distributed systems. *Software Engineering, IEEE Transactions on*, (3):240–247. 6

[Pearson, 2009] Pearson, S. (2009). Taking account of privacy when designing cloud computing services. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 44–52. IEEE Computer Society. 5

[Petersen et al., 1997] Petersen, K., Spreitzer, M., Terry, D., Theimer, M., and Demers, A. (1997). Flexible update propagation for weakly consistent replication. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 288–301. ACM. 6

[Philipp et al., 2011] Philipp, D., Durr, F., and Rothermel, K. (2011). A sensor network abstraction for flexible public sensing systems. In *IEEE MASS*, pages 460–469. IEEE. 28

[Reiher et al., 1994] Reiher, P. et al. (1994). Resolving file conflicts in the ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference-Volume 1*, pages 12–12. USENIX Association. 6

[Shepard et al., 2011] Shepard, C. et al. (2011). Livelab: measuring wireless networks and smartphone users in the field. *ACM SIGMETRICS Performance Evaluation Review*, 38(3):15–20. v, 2, 8

[Takabi et al., 2010] Takabi, H., Joshi, J., and Ahn, G. (2010). Security and privacy challenges in cloud computing environments. *Security & Privacy, IEEE*, 8(6):24–31. 1

[Veeraraghavan et al., 2009] Veeraraghavan, K., Ramasubramanian, V., Rodeheffer, T., Terry, D., and Wobber, T. (2009). Fidelity-aware replication for mobile devices. In *Mobisys*, pages 83–94. ACM. 5

[Ye and Chiu, 2007] Ye, C. and Chiu, D. (2007). Peer-to-peer replication with preferences. In *Proceedings of the 2nd international conference on Scalable information systems*, page 4. 17

Vita

Jilong Liao was born in Jintang, Sichuan Province, China in 1988. He obtained his Bachelor's degree in Communication Engineering from School of Communication and Information Engineering, University of Electronic Science and Technology of China in 2010. In Spring 2011, he enrolled in the doctoral program of Computer Science in Department of Electrical Engineering and Computer Science at University of Tennessee, Knoxville. In the meantime, he joined the Laboratory for Autonomous Interconnected, and Embedded Systems (Lanterns) leaded by Dr. Qing (Charles) Cao, where he worked as a research assistant. In summer of 2013, he changed his program to Master's and complete his Master's degree in Fall 2013. His major research focus on wireless sensor networks, mobile system, machine learning and distributed system.