EdgeCourier: An Edge-hosted Personal Service for Low-bandwidth Document Synchronization in Mobile Cloud Storage Services

Pengzhan Hao, Yongshu Bai, Xin Zhang, and Yifan Zhang
Department of Computer Science
SUNY Binghamton
Binghamton, NY

ABSTRACT

Using cloud storage to automatically back up content changes when editing documents is an everyday scenario. We demonstrate that current cloud storage services can cause unnecessary bandwidth consumption, especially for office suite documents, in this common scenario. Specifically, even with incremental synchronization approach in place, existing cloud storage services still incur whole-file transmission every time when the document file is synchronized. We analyze the problem causes in depth, and propose *EdgeCourier*, a system to address the problem. We also propose the concept of edge-hosed personal service (EPS), which has many benefits, such as helping deploy *EdgeCourier* easily in practice. We have prototyped the *EdgeCourier* system, deployed it in the form of EPS in a lab environment, and performed extensive experiments for evaluation. Evaluation results suggest that our prototype system can effectively reduce document synchronization bandwidth with negligible overheads.

CCS Concepts

•Networks → Cloud computing; •Human-centered computing → Mobile computing; Ubiquitous and mobile computing systems and tools; *Mobile devices;*

Keywords

Mobile computing; Edge computing; Cloud storage; File synchronization; Unikernel

1. INTRODUCTION

In this paper, we investigate achieving low-bandwidth document content synchronization for a popular scenario enabled by smart mobile devices (e.g., smartphones and tables) and cloud storage services: a user edits documents on his smartphones/tablets. The content changes made by the user are synchronized to cloud storage as they are generated, and can be further synchronized in real-time to other devices owned by the user or his collaborators. We name this common usage scenario as *cloud-storage-backed mobile document editing*.

Low-bandwidth sync is important for cloud-storage-backed mobile document editing. However, our motivation study (§3.2) shows that exiting cloud storage services incur high network traffic on mobile devices when synchronizing document files, especially for office suite documents [9, 10, 34, 47, 63], such as word processing, spreadsheet, and presentation documents, which constitute the most commonly used editable document formats in practice [18, 33]. Specifically, most cloud storage services transmit the whole document file from the mobile device to cloud storage every time when the user saves the file, even for a small change in the file (e.g., a single character addition). This behavior can cause a high amount of network traffic for mobile users considering that the "save" operation is common and happens frequently when users edit documents. We name this common problem as the *whole-file-sync problem* in the cloud-storage-backed document editing scenario.

To solve the problem, we develop *EdgeCourier*, a system to improve cloud-storage-backed document editing usage experience by significantly lowering network traffic generated by mobile devices when synchronizing office document file changes with cloud storage services. In the following, we introduce how *EdgeCourier* works by summarizing the *three* major challenges of solving the whole-file-sync problem, and our corresponding designs of addressing them.

First, the proper way to solve the whole-file synchronization issue is using an incremental synchronization approach, with which only the changes made since the last synchronization, instead of the whole file, are transmitted. However, existing techniques for this purpose, such as chunking/deduplication [5, 8, 16, 52, 57, 64] and delta-encoding [20, 36, 50, 58, 65], virtually have no effect on office documents. The reason is that, for most office applications, a small edit in an office document can result in a substantial change in its binary format. Moreover, the performance of existing incremental sync approaches highly depends on properly choosing the right configuration (e.g., chunk/block size) for the traffic of interest. However, the current incremental sync methods deployed on cloud storage services treat all sync traffic in the same way, and hence they also do not work well for synchronizing general document traffic. More details are discussed in §3.

To address the first challenge, we design and implement

SEC '17, October 12–14, 2017, San Jose / Silicon Valley, CA, USA

ISBN 978-1-4503-5087-7/17/10…$15.00

DOI: https://doi.org/10.1145/3132211.3134447

an *office-document-aware incremental sync* approach, which we name as *ec-sync*. Our approach is based on the observation that different types of office documents are all standard ZIP files (with different extensions), which are compressed archives containing a defined structure of sub-documents [68, 70]. When a small edit is made to a document on mobile device, the edit is recorded (in plain text) in one of the sub-documents. Thus, the general idea of *ec-sync* is to capture the actual edits by comparing the latest version of the sub-document containing the edits against the last-synced version, and transmit only the actual edits. Although the idea of *ec-sync* is straightforward, to make it work effectively and efficiently was not trivial. There were several notable difficulties. For example, when comparing the versions of the sub-document, using existing implementations of the `diff` utility [31] would contribute significant computation and time overheads to *ec-sync* if the document size is large. To address this problem, we developed `ec-diff`, which is a new diff tool specifically designed for the cloud-storage-backed document editing scenario and is able to achieve good performance regardless the sizes of the underlying files. We discuss the details of *ec-sync* and the associated difficulties later in §4.1.

The second challenge for *EdgeCourier* is that all the possible incremental sync approaches require adding the corresponding processing on cloud storage servers, which can greatly hinder them from being deployed in practice for two reasons: one reason is that these approaches would undoubtedly cause significant load increase on the cloud storage servers given their central processing nature; the other is that it is hard, and also would take long time, for all the cloud storage service providers to adopt such approaches.

Our approach of addressing the above challenge is to distribute the corresponding processing on cloud storage server to network edge nodes (e.g., wireless access points, in-home routers, and cellular towers). We propose the concept of *edge-hosted personal service* (EPS for short). An EPS runs on a network edge node, and performs a specific functionality for mobile users. A mobile user can start/stop his own EPS instance(s) on the edge node to which he is connected. One advantage of EPS is that it can help distribute the functionalities that are normally deployed in data centers to network edge, to enjoy the benefits of fast deployment and data center load reduction. For example, in the case of *EdgeCourier*, each mobile user has his own *EdgeCourier* EPS instance running on the edge. When the actual document edits captured by the *ec-sync* method are synchronized to a cloud storage service, they are transmitted to the user's EPS instance, which acquires the newest version of the document by applying the edits on the last-synced version of the file, and transmits the up-to-date file to cloud storage using the original interface provided by the storage service. This way, the incremental sync functionality can be easily deployed without requiring any changes on cloud storage services. There are many other useful usage scenarios and advantages of EPS. A in-depth discussion about these usage scenarios/advantages will be presented later in §2 and §4.2.

To implement EPS, two major considerations need to be taken into account. *First*, running an EPS instance should minimal computational resource from the hosting edge node. This is because, in addition to performing its own functionalities, each edge node may need to host dozens of EPS instances of difference purposes for different users. Given the fact that edge nodes are usually embedded devices with limited computational resources, achieving high resource efficiency is critical to EPS. *Second*, since multiple users' data are processed in the same edge node, an EPS runtime that can help protect users' data security and privacy is important. Our approach of fulfilling the above two goals is to use Unikernel as EPS runtime environment, since Unikernels are known for its properties of extreme lightweight and perfect resource isolation [43–46].

The third major challenge for *EdgeCourier* is that to deploy *EdgeCourier*, changes also needs to be made on mobile devices side. However, requiring modifications on either mobile apps or mobile OS is not practical for deploying *EdgeCourier* on millions of mobile devices already in use. To address this issue, we take advantage of our prior work StoArranger [12, 13], a practical system framework running on mobile devices that can rearrange, coordinate, and transform cloud storage accesses from mobile apps.

To summarize, our contributions are as follows.

- Through detailed measurement study, we identify the common whole-file-sync problem, which incurs high sync bandwidth in the popular cloud-storage-backed mobile document editing scenario, and we demonstrate that it cannot be effectively solved by the exiting incremental sync approaches.
- We design *ec-sync*, an effective office-document-aware incremental sync approach, and *EdgeCourier*, a system that integrates *ec-sync* and the associated tools (e.g., `ec-diff`) for low-bandwidth mobile document synchronization in cloud storage services.
- We propose the concept of edge-hosted personal service (EPS), which has many useful application scenarios and benefits. For example, we demonstrate that EPS can help deploy the functionalities of *EdgeCourier* without requiring changes on cloud storage services.
- We prototype the *EdgeCourier* system with real hardware. We explore using Unikernel to serve as the runtime environment of EPS, which is a promising approach to deploy EPS instances on resource-constrained edge nodes, and we report the corresponding experiences.
- We evaluate the *EdgeCourier* system with real-world experiments, which show that our system can achieve its goals with little overheads.

2. RELATED WORK

Exploiting network edge for efficient mobile computing. Edge computing research has been gaining attention rapidly [61, 71, 72]. Similar to our work, several notable efforts have been focusing on utilizing computational resources on network edge, as well as low latency, high-bandwidth wireless connection between mobile devices and the edge, to solve various problems or create new services.

- Cloudlets [28, 29, 60] have been used to help improve usage experiences on resource-constrained mobile devices. The idea is to *offload* computation from mobile devices to VM-based cloudlets located on the network edge. The low communication latency between mobile devices and cloudlets lends a great advantage over the approach of offloading to the cloud.
- AirBox [15] targets improving usage experience of existing cloud services on mobile devices. The idea is to *onload* those latency-sensitive cloud services from data centers to network edge, so that users can enjoy faster service response.

Table 1: Office suite applications used in the experiments and test document files generated.

Office suite / Operating system	Word Processing			Spreadsheet			Presentation		
	File	Format	File size	File	Format	File size	File	Format	File size
MS Office 2013 / Windows 10	W1	`.docx`	159 KB	S1	`.xlsx`	47 KB	P1	`.pptx`	274 KB
MS Office 2011 / macOS 10.12	W2	`.docx`	302 KB	S2	`.xlsx`	50 KB	P2	`.pptx`	293 KB
MS Office / Android 7.1	W3	`.docx`	168 KB	S3	`.xlsx`	49 KB	P3	`.pptx`	256 KB
WPS / Android 7.1	W4	`.docx`	153 KB	S4	`.xlsx`	46 KB	P4	`.pptx`	237 KB
OfficeSuite Pro / Android 7.1	W5	`.docx`	153 KB	S5	`.xlsx`	37 KB	P5	`.pptx`	218 KB
LibreOffice / Ubuntu 16.04	W6	`.docx`	151 KB	S6	`.xlsx`	42 KB	-	-†	-
LibreOffice / Ubuntu 16.04	W7	`.odt`	172 KB	S7	`.ods`	42 KB	P6	`.odp`	70 KB
iWOrk / macOS 10.12	W8	`.pages`	588 KB	S8	`.numbers`	911 KB	P7	`.key`	3334 KB

†: We skipped generating the `.pptx` file using LibreOffice since software did not work well with `.pptx` format.

• ParaDrop [41] allows third-party developers to create, deploy, and revoke new services for mobile users on network edge nodes. These services enjoy faster response time than the conventional approach of cloud deployment.
• The fast growing computation capability on network edge nodes, as well as the stable and low-latency connection between mobile devices and the edge, have also been used to improve video streaming experiences for mobile users [73].

We also take advantage of network edge nodes aiming to improve usage experiences for mobile users. We propose the concept of "edge-hosted personal service (EPS)", which have two main advantages: The first is that it enables developer-customizable communication protocol on the "last-hop" communication between mobile devices and the network edge. As we demonstrate with our *EdgeCourier* system, it brings two further benefits: one is that mobile end-users can enjoy the goods of the new protocol, such as less network traffic and longer battery life; the other is that it helps to deploy new functionalities without the need of changing the deployed cloud services. The second advantage of EPS is that it helps distribute cloud services for mobile users to network edge, so that they can be done on a *personalized* basis and thus achieve better performances. We defer the detailed discussion to §4.2.

Network bandwidth reduction for file transmissions. Existing solutions related to reducing network bandwidth requirement for transmitting files over networks is based on the general idea of transmitting the differences between different files or between different versions of the same file, instead of transmitting the whole file. These solutions use either the chunking/deduplication approach or the delta-encoding approach to achieve their goals.
• Solutions using the chunking/deduplication method [5, 8, 16, 52, 57, 64] store a file in the unit of chunks, which can be of fixed-size or varied-size. When transmitting a file, certain mechanism is applied to exploit chunk-level commonalities between the sender and the receiver, such that the sender only transmits the chunks not possessed by the receiver. Dropbox [23] and seafile [4], which we used in our motivation study (§3.2), belong to this category.
• Solutions using delta-encoding (a.k.a. delta-compression) [20, 36, 50, 58, 65] are also based on the idea of transmitting only the content not possessed by the receiver. But with delta-encoding, files are not necessarily stored in chunks. Sender uses certain approach, such as rolling checksum, to compare the similarity between the file it owns and that on the receiver side, and only encodes the difference into the packets sent to the receiver. Rsync [3], which we used in our motivation study (§3.2), belong to this category.

Improving performances of cloud storage services. Several recent studies focused on measuring and improving usage experiences for cloud storage services. Work by Drago et al. specifically investigated Dropbox [22]. Later, the same group of researchers went on performed another study comparing the system architecture and service capabilities of five popular cloud storage providers [21]. Work by Li et al. [37] studied the network transmission efficiency of cloud storage services. The authors define a metric, named Traffic Usage Efficiency (TUE) to quantify the bandwidth overhead caused by different cloud storage services. Moreover, the authors proposed batching and deferred transmission of of small file updates to improve transmission efficiency of cloud storage services. QuickSync [17] evaluated cloud storage transmission performance in wireless networks. To reduce upstream file synchronization time, the authors proposed a network-condition-aware file chunking mechanism based on the idea of balancing network transmission time and data deduplication computation overhead.

3. MOTIVATION STUDY

In this section, we first give the background of cloud storage services and our target mobile usage scenario, followed by our study demonstrating the problem and the inability of the state-of-the-art techniques to solve it.

3.1 Background

Cloud storage services refer to cloud services hosting user files. They allow a user to upload and store files to cloud-based storage, and access the files over the Internet from different devices owned by the user. Examples of popular cloud storage services include Dropbox [23], Google Drive [26] and OneDrive [49]. Recently, accesses to cloud storage services via mobile devices have experienced a significant increase. For instance, the numbers of installs of the official Android apps for Google Drive, Dropbox, and One Drive are now more than 1 billion [27], 500 million [24], and 100 million [48], respectively.

Target scenario: cloud-storage-backed mobile document editing. One key feature provided by cloud storage services that makes them popular is automatic file sync, with which user files are automatically synchronized between mobile devices and cloud storage as they are generated or changed [17, 38, 42]. In the meantime, smart mobile devices, such as smartphones and tablets, have been becoming people's favorite platforms of performing document markup and

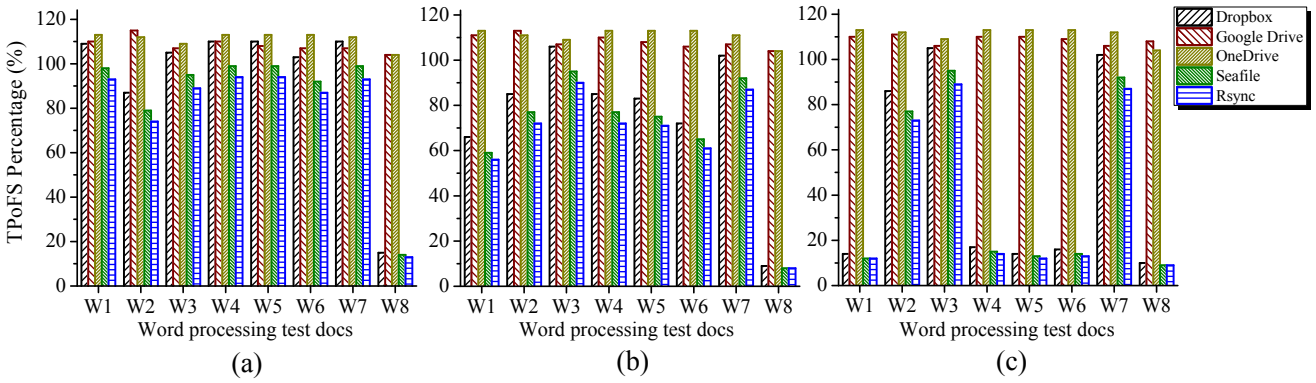

Figure 1: TPoFS performance of synchronizing a one-character-addition edit using three cloud storage services (Dropboxy, Google Drive, and OneDrive) and two file synchronization software (seafile and rsync). (a), (b) and (c) show the results of adding one character at the *beginning*, the *middle*, and the *end* of each word documents respectively.

editing, due to their highly convenient and intuitive user interfaces, excellent portability, and rich connectivity features [11, 54, 56, 62, 67].

Together, smart mobile devices and the automatic sync feature offered by cloud storage services enable a popular application scenario, where a user uses smartphone or tablet to edit a document file when he is on the move. The document file is shared with his collaborators via the cloud storage automatic sync functionality. Thus, the content changes made by the user are synchronized to the collaborators through the cloud storage service in real-time as they are generated (e.g., every time when user saves the document). We name this type of scenario as *cloud-storage-backed mobile document editing*.

Low-bandwidth synchronization is critical for achieving good usage experience in cloud-storage-backed mobile document editing for two reasons. *First*, document sync to/from cloud storage is running on cellular data when user is on the move, hence lowering network traffic generated by the sync can reduce the financial cost for the user. *Second*, lowering bandwidth requirement for the sync can be very helpful for prolonging device battery life, which plays an important role in providing good usage experience for mobile users. Therefore, we performed a series of experiments to study the network traffic performance when synchronizing document content changes with existing cloud storage services.

3.2 The experiments

Experiment setup. Our initial impression about network traffic generated between mobile devices and cloud storage services when synchronizing content changes in office suite documents was that most cloud storage services transmit almost the whole document file even when we make a small change to the document, and the traffic could vary for documents generated by different office applications. To perform automated and quantitative measurements, we developed a test tool that can perform cloud-storage-backed document editing, and can monitor, filter, and process, in an automated manner, the cloud storage related traffic generated during the editing process.

We chose eight popular office suite applications that spans four different operating systems. The first column of Table 1 shows the choices of these applications. We created three categories of test document files (i.e., word processing, spreadsheet, and presentation) using the eight office suite applications. We did this by first downloading a word document, a spreadsheet document, and a presentation document. Then we opened the documents and saved them as separate copies using the eight office suites respectively. Therefore, all the documents in the same category have the same content. But since different office suites have different proprietary implementations (while confirming the same document specification, details later), the documents are of different sizes. Table 1 lists the file sizes of file format extensions of the test documents.

To perform the synchronization operation, we chose three popular cloud storage services (i.e., Dropbox [23], Google Drive [26], and OneDrive [49]) and two well-known file synchronization software (i.e., seafile [4] and rsync [3], which implement the well-known network file transmission protocol LBFS [52] and the `rsync` algorithm [65] respectively). We are interested in the network bandwidth requirement when synchronizing *small edits* because of frequent save operations in a document editing process (and hence the content change to be synchronized each time is usually small). Therefore, when testing for each doc, our test tool first transmitted the original document (e.g., W1 in Table 1) to the storage server. Then the tool made a small edit to the doc (e.g., adding one character to W1), and saved it. The auto sync functionality of cloud storage services would automatically synchronize the edit to the storage server once the edit is made persistent on the disk. Our tool then measured the network traffic generated on the local device for synchronizing the edit.

Experiment results. Figure 1, 2, and 3 show the measurement results for the three categories of test docs. In the figures, the amount of sync traffic generated is represented as the metric of TPoFS (Traffic Payload over File Size percentage), which is defined as:

$$\text{TPoFS} = \left(\frac{100 \times \text{Payload of the traffic generated}}{\text{Size of the file synchronized}}\right)\% \quad (1)$$

The file sizes of all the test docs are listed in Table 1. Figure 1 (a), (b) and (c) present the results for the tests of adding a character at the beginning, the middle, and the end of the word precessing docs respectively. Figure 2 shows the results for the tests of modifying one cell at the end of the

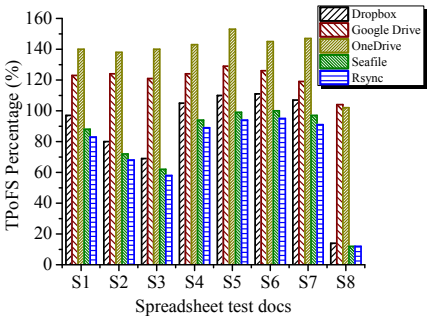

Figure 2: TPoFS performances of syncing a one-cell-modification edit at the end of the spreadsheet docs.

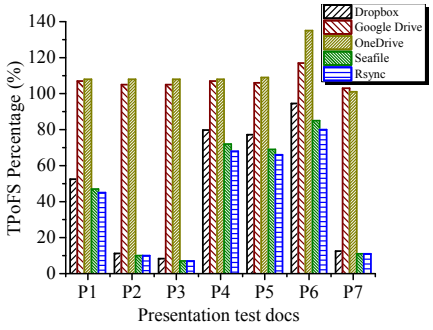

Figure 3: TPoFS performances of syncing a one-object-addition edit at the end of the presentation docs.

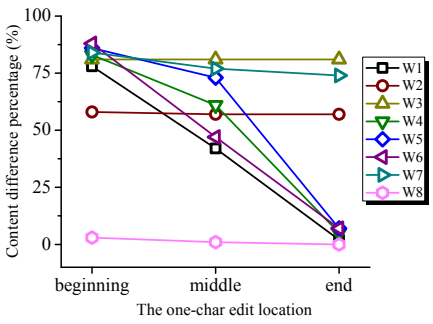

Figure 4: Binary difference percentages of the word processing docs after the one-character-addition edit.

spreadsheet docs. Figure 3 shows the results for the tests of adding one object at the end of the presentation docs. From the figures we can have the following three findings.

Finding-1. For all the edits in all the test docs, the TPoFS values of Google Drive and OneDrive are larger than 100%, suggesting that the two cloud storage services always perform whole-file transmissions when synchronizing edits (regardless of edit sizes) in office suite documents. This finding is in accordance with recent studies [17, 21, 37], which suggested most existing cloud storage services do not support incremental sync. Therefore, the whole-file-sync issue is not confined to just office suite docs, but applies to all types of files for Google Drive and OneDrive.

Finding-2. Although Dropbox is known to support incremental sync [21, 37], and seafile/rsync support LBFS-based [52]/`rsync`-based [65] incremental sync, small edits in many test docs still caused almost-whole-file transmission with Dropbox, seafile, and rsync, such as W1-W7 in Figure 1 (a) and (b), W2/W3/W7 in Figure 1 (c), S1-S7 in Figure 2, and P1/P4/P5/P6 in Figure 3.

Finding-3. Edit locations can affect sync traffic generated for word processing documents. For example, sync traffic reduced as the edit was moved toward the end of the docs for W1/W4/W5/W6. But for other docs, i.e., `.docx` file generated by MS Office on macOS (W2), `.docx` file generated by MS Office on Android (W3), and `.odt` file generated by LibreOffice on Ubuntu (W7), the one-character-addition edit always caused almost-whole-file transmissions with Dropbox, seafile and rsync. It is worth noting that edit location has no effect on sync traffic for spreadsheet and presentation documents as in our experiments.

One may have noticed that documents generated by the iWork office suite on macOS consistently incurred low TPoFS percentage. But please note that the file sizes of iWork docs are considerably larger than those generated by other office suites (because of the different way to construct the docs). Thus, the resulted sync traffic were just slight lower than or comparable to docs generated by other office suite applications. Lastly, We have also performed various other types of small edits, such as deleting/modifying one character, highlighting one line in word documents, adding a new row in spreadsheet documents, and adding/deleting a slide in presentation documents, all of which gave us the similar findings as shown.

Why the problem matters. The above problem that a small edit in office suite documents can cause whole-file or almost-whole-file synchronization even with incremental sync in place can greatly increase bandwidth requirement in a cloud-storage-backed mobile document editing process. This is because users tend to save the document frequently during the editing process (which is considered as a good habit), and each save is likely to cause a whole-file transmission. For example, according to De Lara et. al. [18], the average sizes of Word, Excel, and PowerPoint documents retrieved from the web are 196 KB, 115 KB and 891 KB. Suppose a user edits an office document with size of 500 KB, and he performs the save operation 5 times a minute. Then a 30-minute editing session would cause the user about 75 MB mobile data to synchronize the edits, which are likely to be small in reality.

3.3 Understand the findings

To understand the causes of the problem, we used the `diff` utility [31] to compare the binary content difference between each original test document and that after the small edit. Figure 4 shows the result for the word processing documents. From the figure we can obtain two observations. One is that that for W2/W3/W7, the one-character-addition edit caused over 50% percent binary difference regardless of the edit location. The other is that for W1/W4/W5/W6, adding one character at the beginning caused binary difference larger than 75%, and the difference reduced as the edit was moved towards the end of the docs. Both observations can echo those in the sync traffic experiments. We performed the same comparison for the spreadsheet and the presentation documents, and also observed that a small edit (regardless of its location) can usually cause a substantial binary change in the doc's binary format.

The reason for the above observations is that office suite documents are constructed based on certain standards, such as the Office Open XML standard [68] and the OpenDocument standard [68]. One common point of these standards is that they all specify an office document format as a ZIP archive containing a defined structure of sub-documents, which store different information of the document, such as content, styles, metadata, and application settings. An edit to an office document file leads to changes in at least two sub-documents: the one storing the doc content and the

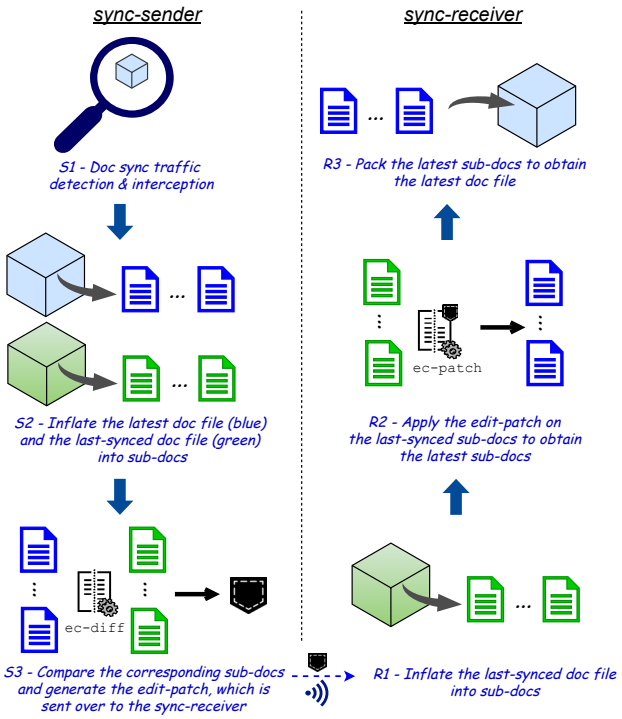

Figure 5: Overview of *ec-sync*, an office-document-aware incremental synchronization approach.

one storing metadata. When user saves the document, all the sub-documents are compressed into one ZIP archive and written to the persistent storage. Thus, a small edit usually leads to a substantial change in the doc's binary format.

For the reason discussed above, existing incremental sync techniques, such as deduplication as used in seafile and delta-encoding as used in rsync (see §2), are not suitable for implementing incremental synchronization for office suite documents. Another reason is that, the performance of deduplication-based or delta-encoding-based techniques relies on choosing the proper configuration for the transmission, such as chunk size suitable to the underlying sync traffic. As a result, they may not be able to work well on a general basis.

4. SYSTEM DESIGN

Out goal is to design an effective, practical, and easily-deployable solution, which we name as *EdgeCourier*, for solving the whole-file-sync problem in the cloud-storage-backed mobile editing scenario. In this section, we discuss the key designs that help us achieve the goal.

4.1 Office-document-aware incremental synchronization

An effective way of solving the whole-file-sync problem is incremental sync, with which only the contents that are different from the last-synced version of the file are transmitted. However, as we showed previously, existing techniques, such as those used in Dropbox, seafile and rsync, are not suitable for synchronizing office document changes. Our approach is *ec-sync*, an office-document-aware incremental synchronization approach.

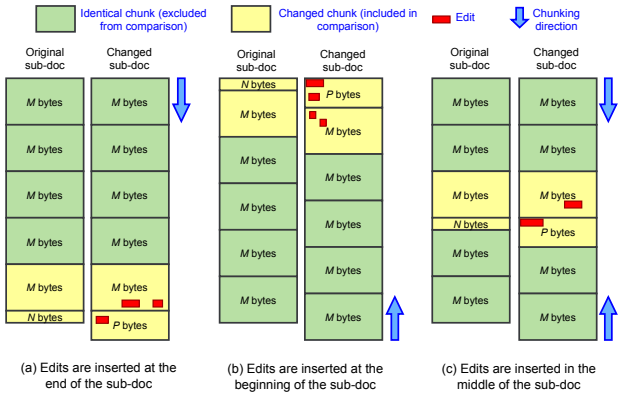

Figure 6: `ec-diff` working scenarios.

***ec-sync* overview**. *ec-sync* is based on the fact that office documents are all standard ZIP archives containing a defined structure of sub-documents (§3.3), and a straightforward idea: capture and send the changes of the sub-documents of an office document, and apply the changes on the receiver side to obtain the latest version of the document. Figure 5 summarizes the overall steps of our *ec-sync* approach. There are two participants in a *ec-sync* process: the sync-sender and the sync-receiver. The operations are as follows.

- The sender first needs to detect if there is an office document being synchronized to the receiver, and intercepts such a transmission if so (i.e., step S1).
- To capture the sub-document changes made on the latest version of the document being synchronized, the sender decompress the latest version of the document, as well as the last-synced version of the same file (i.e., step S2). Here we assume the sender has the last-synced version of the file. We introduce our implementation of achieving this next in §5.
- The sender compares the corresponding sub-documents of the latest version and the last-synced version of the document, and places the differences in a file called *edit-patch*, which is transmitted to the receiver (i.e., step S3).
- Upon receiving an edit-patch from the sender, the receiver first decompress the last-synced version of the document of interest into sub-documents (i.e., step R1; the details of receiver obtaining the last-synced version are discussed in §5).
- The receiver applies the edit-patch to the sub-documents to obtain their latest version (i.e, step R2).
- Finally, the receiver compresses the patched sub-documents into a standard ZIP file to obtain the latest version of the document (i.e., step R3).

Efficient sub-document comparison. Compared to the existing incremental sync approaches, *ec-sync* can effectively reduce network traffic generated when synchronizing office documents in mobile cloud storage services. However, it would cause synchronization delay since several extra steps are introduced in the sync process, such as intercepting the sync traffic on the client side, inflating the documents and generating the edit-patching by comparing the sub-docs on the client side, and applying the edit-patch on the receiver side. According to our experience, the step of generating the edit-patch can introduce the most significant delay, especially when the sub-documents to compare are large. For example, using the well-known `diff` utility [31] to compare two sub-docs with 1 MB text would take about 10 seconds,

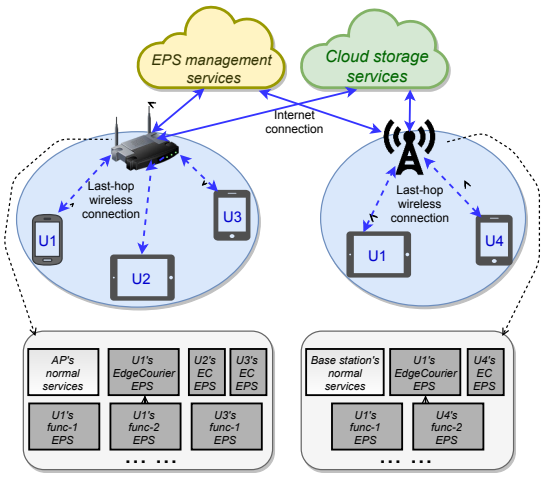

Figure 7: EPS deployment illustration.

which is un acceptably long. Therefore, it is important to find an efficient way to perform the edit-patch generation.

Our approach of solving the problem is `ec-diff`, a new text comparison tool designed for *EdgeCourier*. The `ec-diff` tool is founded on an important observation about the cloud-storage-backed mobile document editing scenario, which is, because of the frequent save operations occurring in the edit process, *the edits between two consecutive save operations (and hence also between two consecutive cloud storage sync operations) are small, and tend to close to each other in the document*. Based on this observation, `ec-diff` tries to reduces the comparison size by excluding the identical text in the two versions of a sub-document as much as possible. Figure 6 gives examples of the three working scenarios of `ec-diff`. In the examples, suppose the original sub-document has $5M + N$ bytes ($N < M$). All the edits cause the sub-doc's size increased to $5M + P$ bytes ($N < P < M$).

• If the edits are placed towards the end of the sub-document (Figure 6 (a)), the identical text in the changed sub-doc would be above the edits. In this case, `ec-diff` can perform chunking (with chunk size of M bytes) on both versions of the sub-doc from the beginning, calculates the checksum of the resulted chunks, and exclude those that are identical.

• If the edits are placed towards the beginning of the sub-document (Figure 6 (b)), the identical text in the changed sub-doc would be below the edits. In this case, `ec-diff` can perform chunking from the end of the files, and exclude those identical chunks.

• If the edits are placed in the middle of the sub-document (Figure 6 (c)), the identical text in the changed sub-doc would be on the both sides of the edits. In this case, `ec-diff` can perform chunking from the beginning and from the end of the file at the same time, and exclude the chunks with the same checksums.

Therefore, generally speaking, the `ec-diff` algorithm works by first performing chunking in three ways as discussed above. If the identical chunk percentage of the three ways are all below a certain threshold, `ec-diff` compare the two versions of the sub-doc entirely using an existing comparison tool. Otherwise, `ec-diff` adopts the way that generates the most identical chunks, and performs comparison on those different chunks via an existing text comparison tool.

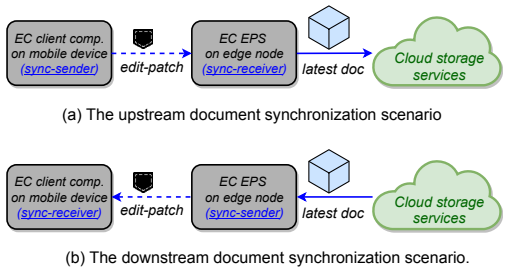

Figure 8: *EdgeCourier* EPS working scenarios.

4.2 Deploying the new incremental sync approach using edge-hosted personal services (EPS)

Deploying a new incremental sync mechanism (e.g., deduplication, delta-encoding, or the proposed *ec-sync*) on existing cloud storage services requires modifications on cloud storage servers. For example, the chunking/deduplication approach would need to run on both the sender and the receiver side of a transmission; *ec-sync* would need cloud storage server to apply the edit-patch to obtain the latest version of the document for the upstream doc synchronization scenario. However, it would be impractical to deploy *ec-sync* directly on cloud storage servers for two reasons. *First*, since a cloud storage server needs to serve a large number of users, adding extra handling would cause significant load increase on the server. *Second*, it would be hard and would take a long time to convince all the cloud storage service providers to deploy a new incremental sync approach on their services.

Edge-hosted personal services (EPS). We propose the concept of edge-hosted personal services (EPS) [30], which can help solve the above problem effectively. The idea of EPS is to utilize the fast-increasing computation resource available on edge nodes of wireless networks to deploy individual-based personalized services for mobile wireless users. Figure 7 shows an example of EPS deployment. On an edge node, such as an access point or a base station, EPSes are running to serve the mobile users within the local wireless network. Each EPS instance serves a single mobile user for a certain functionality. Specifically, each user can have multiple EPS instances running on the same edge node for different service functionalities. EPS of the same functionality can have different instances in the same edge node, with each instance serving a different user. Furthermore, mobile users' EPS instances are centrally stored and managed by the EPS management services locating in the cloud. Through the EPS management services, users can migrate, start and revoke EPS instances as needed.

EPS benefits. Generally speaking, EPS achieve two highly useful functionalities, which are discussed as follows.

• The first functionality is to help quickly deploy those communication protocols or mechanisms, which are beneficial to mobile users, but are hard to deploy in practice for the reason that modifications needed to be done on servers locating on the other side of the Internet.

For instance, the deployment of *ec-sync* can be benefited from running part of the *ec-sync* operations as an EPS, which is named as "*EdgeCourier* EPS", or "*EC* EPS" for short. Figure 8 depicts the two scenarios of deploying *ec-*

sync with the help of EPS. For the upstream document sync scenario (Figure 8 (a)), the *EdgeCourier* EPS performs the operations of an *ec-sync* receiver, which are accepting edit-patches sent from the corresponding mobile device, obtaining the latest version of the document by applying the patch to the last-synced version of the document, and uploading the doc to its destination (i.e., the cloud storage server) using the original interfaces provided by the cloud storage services. For the downstream document sync scenario (Figure 8 (b)), the *ec-sync* EPS carries out the operations of an *ec-sync* sender, which are intercepting cloud-storage document sync traffic, generating edit-patches by comparing the latest version of the document being synchronized with its last-synced version, and sending the patches to the receiver.

• The second functionality of EPS is to distribute cloud services or optimizations for mobile workloads to network edge, so that they can be done on a personalized basis for better performances. Moreover, after being distributed to the edge, these services and optimizations can enjoy much lower communication latency to/from mobile devices, because of the high-bandwidth wireless connection to the devices.

For instance, recent studies have shown that web caching performances on mobile devices are related to mobile users' personal web browsing habits and preferences [74]. Therefore, it would be more effective to perform personalized web caching using EPS than cloud-hosted web caching services like the one provided by Google's Flywheel [6]. More specifically, Flywheel performs cloud-based web data reduction (e.g., through compression and image transcoding), preconnecting and prefetching for mobile web. However, these optimizations are carried out in a user-preference/habit-agnostic manner. With EPS, it is possible to perform these optimizations on a more personalized basis to achieve better performances.

It is worth noting that although the idea of distributing cloud functionalities to network edge is not new [15], our idea of EPS is more focusing on how to take advantage of individual differences to better perform those functionalities. However, utilizing personalized info and EPS to improve mobile computing experience is out of the scope of this paper. We leave this for our future work.

Scaling and protecting EPS instances on edge nodes with Unikernels. The EPS concept poses two notable challenges on its implementation and deployment:

• One challenge is how to efficiently scale EPS instances on an edge node. Specifically, an edge node needs to be able to support tens of EPS instances at the same time, while maintaining its normal functionalities. To better scale EPS instances on an edge node, we need an efficient runtime environment for running EPSes.

• The other challenge is how to effectively protect user data and privacy on an edge node. Specifically, since an edge node runs EPS instances of difference users, we need an EPS runtime environment that can effectively protect and preserve users' data security and privacy.

Therefore, we need a lightweight yet secure runtime environment for running EPSes. Possible EPS runtime environment candidates can be, for example, traditional processes environment, where each EPS instance runs in a process; virtual machine environment [14,35], where each EPS instances runs in a VM; or container environment [19,40], where each EPS instance runs in a container.

Considering the above two challenges and the fact that

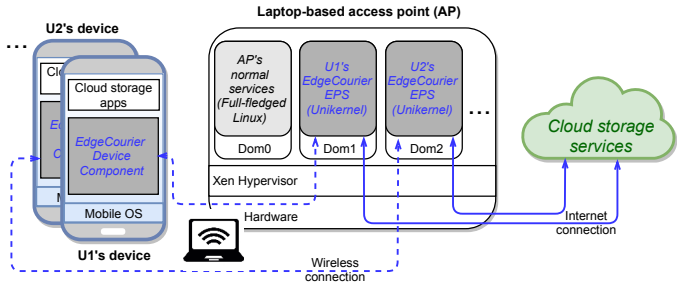

Figure 9: *EdgeCourier* implementation overview.

an EPS just needs to support one specific functionality, our choice of EPS runtime environment is Unikernel [43–46]. The development of Unikernel stems from the concept of library operating system [7, 25, 55, 66]. By directly compiling only the necessary OS features with user code, Unikernels are specialized and small OS images with very small runtime footprints. Unikernels are also extremely efficient in performing their designated tasks, since there is no distinction between user space and kernel space, which eliminates the overheads of trapping to kernel in traditional OSes. Moreover, since individual Unikernels can run directly on different VMs (as what we did in our prototype implementation), they can provide arguably the strongest resource isolation among all the candidates discussed. We introduce more details of using Unikernel as EPS runtime environment in our prototype system later in §5, and evaluate our Unikernel-based EPS runtime by comparing it to a container-based EPS runtime in §6.

4.3 Deploying the new incremental sync approach on mobile devices

Deploying the proposed *ec-sync* incremental sync approach on mobile devices faces the similar practical difficulty as deploying it on cloud storage servers, which is the need of deployment on existing mobile devices. The main challenge is to design and implement *ec-sync* sender's cloud storage traffic interception functionality *without modifying either the mobile OS or the mobile apps*. We utilize our previous work StoArranger [12, 13] to address this challenge. StoArranger is a practical system framework on mobile devices to coordinate, rearrange, and transform cloud storage traffic with the goal of improving cloud storage usage experience for mobile users. The proxy framework provided by StoArranger allows us to intercept cloud storage document synchronization traffic on mobile devices smoothly and without requiring changes to either the OS or the mobile apps.

5. SYSTEM IMPLEMENTATION

We have implemented the proposed *EdgeCourier* system and deployed it in a lab environment in the form of EPS. Figure 9 shows the high level overview of the prototype system. The *EdgeCourier* mobile device component has been implemented on two types of smartphones: Samsung Galaxy S4 and Google Nexus 5x. We use a laptop[1] equipped with a quad-core 3.4 GHz CPU and 32 GB memory as the edge node, which is normally functioning as a WiFi access point.

[1]We are also working on an implementation using ARM-based ODROID XU3 board [69] as the edge node. We hope to report our implementation experience in the near future.

The *EdgeCourier* EPS instances are implemented as Unikernels using the Rumprun Unikernel framework [59]. The *EdgeCourier* EPS instances directly run on virtual machines (DomU) created through the Xen hypervisor [39]. The normal services of the AP are performed in the full-fledged Linux-based host OS running in Domain 0. Next, we introduce the implementation details of the *EdgeCourier* mobile device component and the *EdgeCourier* EPS, and explain the associated difficulties as well as how we addressed them.

5.1 Components in the prototype system

There are two entities in our *EdgeCourier* prototype system: the *EdgeCourier*-enabled mobile device and the *EdgeCourier* edge node. Figure 10 shows the components in these two entities, and the relationship between them in the upstream document synchronization scenario. We describe the different components in these two entities in this subsection, and explain how they interact with each other later in §5.3.

***EdgeCourier*-enabled mobile device.** Mobile device is the host of *EdgeCourier* device component, which provides all the *EdgeCourier* related functionalities on the mobile device. The device component consists of the following parts.

• The EPS manager, which works with its counterpart on the edge node to create and revoke the user's EPS instances.

• The cloud storage traffic interceptor, which is to detect and intercept office document cloud storage sync traffic when the mobile device is acting as the sync sender. It is implemented based on a mobile device proxy framework, which we developed in our previous StoArranger work [12, 13].

• The *EdgeCourier* document database, which is used to store the previously-synced versions of office documents.

• The *EdgeCourier* client daemon, which performs the main logic of the device component. We explain how the client daemon works later when describing how the different parts work together in the upstream doc sync scenario.

The *EdgeCourier* device component is implemented in C++. The implementation has around 2,000 LOC, which resulted in a compiled binary of 300 KB.

***EdgeCourier* edge node.** There are generally two components in an *EdgeCourier* edge node:

• The EPS manager, which runs in the host OS, and creates/revokes EPS instances using the Xen tools according to the requests sent from the associated mobile users.

In our current prototype system, we implemented a simple interface between the EPS managers on mobile device and on edge node, which allows users to load their EPS instances from the persistent storage of the edge node, and run in newly created DomUs. When revoked, an EPS instance is saved back to the persistent storage of the edge node. We leave migrating EPS instances between edge nodes with the help of the cloud-based EPS management services (as we proposed in §4.2) to our future work.

• EPS instances, which run as Unikernels to provide the functionalities of *EdgeCourier* EPS. We used the Rumprun Unikernel framework [59] to develop the *EdgeCourier* Unikernel. Our *EdgeCourier* Unikernel implementation has about 1,000 lines of Python code, and is finally compiled as a 6.8 MB Unikernel image.

5.2 Obtaining and maintaining last-synced versions of documents

As discussed previously, both the mobile device component and the EPS instances need to maintain the last-synced versions of all the documents being synchronized to the cloud storage. These last-synced versions of documents are stored on the *EdgeCourier* document DBs on both the mobile device component and the EPS instance.

Table 2: Traffic generated by the mobile device for synchronizing a one-character-addition edit using OneDrive app (unit: byte).

Text size	Edit Loc.	File size after editing	Traffic (direct sync)	Traffic (sync with *EC*)
1K	Start	4,950	8,621	1,538
	Mid.	4,968	8,640	1,538
	End	4,950	8,618	1,538
100K	Start	79,874	83,551	1,538
	Mid.	79,912	83,587	1,538
	End	79,874	83,552	1,538
1000K	Start	760,214	763,894	1,538
	Mid.	760,215	763,894	1,538
	End	760,210	763,887	1,538

To obtain the last-synced version of the docs on the mobile device component initially, the document version manger locating inside the client daemon monitors office document *file open* operation performed by the office suite apps through the Linux `inotify` API [2], which allows user code to be notify when the specified file operations have been performed on the specified files. Once an office document is opened, the doc version manager adds the document into the document DB. From this point on, every time when the document is synchronized to or from the cloud storage, the latest-synced version will be added to the doc DB to serve as the last-synced docs in the *ec-sync* process next time when the same document is being synchronized to or from the cloud storage.

It is worth noting that to reduce the storage overhead on the *EdgeCourier* doc DB, it is better for the doc version manager to monitor *file write* operations instead of *file open*, because not every file open operation will lead to cloud storage document synchronization. However, our experience is that monitoring file write operations can lead to adding the file with incorrect content to the doc DB. This is because at the time when the doc version manager got notified about the write operation, the document file of interests may have already been changed. Adding the file into the doc DB to serve as the last-synced version can lead to edit(s) lost next time when the file is synchronized to the cloud storage via the *ec-sync* process.

To obtain the last-synced version of the docs on EPS instances initially, the EPS instances just monitor the cloud storage document synchronization traffic, and add a document to the doc DB on the first time when it is observed. Similar to the mobile device component, once a document is added to the doc DB, it is updated with the latest-synced version every time when the document is synchronized to or from the cloud storage, so that it can be served as the last-synced version of the document next time when the same file is being synchronized to or from the cloud storage.

5.3 Components interaction in the upstream document synchronization scenario

With the background of how the last-synced version of the docs are obtained on both the mobile device component and the EPS instances, we can use the illustration shown in Fig-

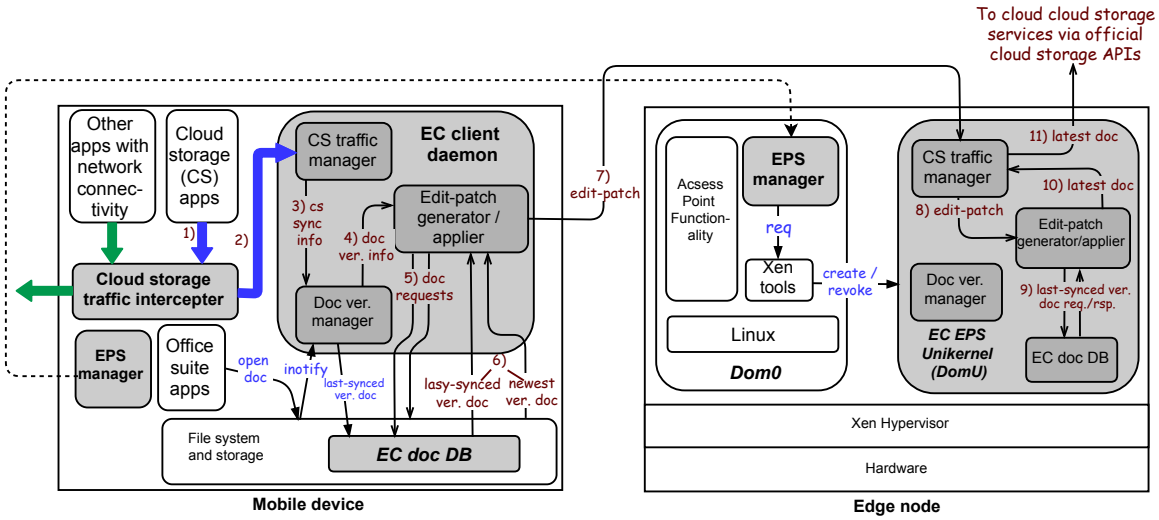

Figure 10: *EdgeCourier* implementation with an upstream sync example.

ure 10 to explain the upstream document synchronization process with our implementation.

• An upstream doc sync process is triggered by the auto synchronization feature provided by cloud storage apps. Once an office document is changed on the persistent storage, cloud storage apps will try to upload the latest files to cloud storage (i.e., step 1).

• The cloud storage traffic interceptor in the device component monitors all outgoing the HTTP traffic and redirects only the traffic related to cloud storage document synchronization to the *EdgeCourier* client daemon (i.e., step 2).

• The cloud storage traffic manager inside the client daemon is responsible for analyzing the cloud storage traffic redirected from the traffic interceptor, and sends the information of the document being synchronized to the document version manager (i.e., step 3).

• The doc version manager further processes the document info, and informs the edit-patch generator about the disk location of the file being synchronized, and where to find the last-synced version of the same document (i.e., step 4).

• Using the information provided by the doc version mananger, the edit-patch generator fetches the latest version of the document being synchronized directly from the file system, and the last-synced version of the same doc from the document DB (i.e., step 5 and 6).

• After getting the latest version and the last-synced version of the document, the edit-patch generator generates an edit-patch according to the process describe in §4.1, and sends the resulted patch over to the *EdgeCourier* EPS instance running on the edge node (i.e., step 7).

• The cloud storage traffic manager in the EPS instance relays the edit-patch to the edit-patch applier in the same EPS instance (i.e., step 8).

• The edit-patch applier uses the doc information in the edit-patch to obtain the last-synced version of the same document from the doc DB (i.e., step 9).

• Then the patch applier applies the patch to the last-synced version of the document according to the *ec-sync* process described in §4.1, and thus acquires the latest version of the document, which is sent to the the cloud storage traffic manager (i.e., step 10).

Table 3: Comparing `ec-diff` with existing solutions

Text size	`diff` [31]	`g-diff` [1,53]	`ec-diff`
1 KB	0.18 sec	0.046 sec	0.066 sec
100 KB	0.79 sec	0.181 sec	0.173 sec
1000 KB	10.52 sec	0.769 sec	0.415 sec
10000 KB	149.85 sec	8.63 sec	3.54 sec

• Finally, the cloud storage traffic manager sends the latest document to the cloud storage using the original interfaces provided by the cloud storage services (i.e., step 11).

6. SYSTEM EVALUATION

We have performed extensive real-world experiments to evaluate our *EdgeCourier* prototype implementation. To conduct the evaluation experiments in an automated manner and to obtain quantitative measurement results, we utilized the test tool used early in the motivation study (§3.2) to create cloud-storage-backed mobile document editing scenarios via the OneDrive Android app and the MS office Android app. Specifically, by using the test tool, edits were made on MS office suite documents through the MS office Android app and were synchronized to the OneDrive cloud storage via the OneDrive Android app. All the experiments were carried out using a Samsung Galaxy S4 smartphone as the mobile device, and the laptop-based access point (§5) as the edge node.

6.1 Evaluating document synchronization network bandwidth reduction

We first conducted an experiment to evaluate how effective our *EdgeCourier* prototype system can reduce document synchronization network bandwidth. In the experiment, we made three one-character-addition edits (one at the beginning, one in the middle, and one at the end) on each of the three word documents, which contained 1 KB, 100 KB and 1000 KB of text respectively. In total, 9 small edits were made on the three word documents. They were synchronized to the OneDrive cloud storage, and the amount of network traffic generated on the mobile device was mea-

sured. The experiment was performed twice: one using our prototype system and the other without.

Table 2 shows the results of this experiment. We can see that when using *EdgeCourier*, the network traffic generated for each edit was always 1,538 bytes, regardless the edit location differences and the file size differences. This is because the contents of all the 9 edits were the same (i.e., adding one character), and hence the resulted edit-patches were identical. By contrast, directly synchronizing the documents without using *EdgeCourier* caused whole-file transmissions. The experiment result suggests that *EdgeCourier* can effectively save network bandwidth for mobile users during a cloud-storage-backed document editing process. For example, suppose a user is working on the third document file (i.e., the one with 1000 KB of text), and he saves the document 5 times per minute on average. Then adopting *EdgeCourier* would save about 114 MB of network traffic for a 30-minute editing session.

6.2 Evaluating document synchronization time

Document sync time with *EdgeCourier*. Document synchronization time refers to the time difference between when the sender initiates the sync process and when the sender receives the confirmation from the receiver that the sync process has completed. For upstream document sync without using *EdgeCourier*, the document synchronization time consists of the following components:

- $T_{wireless}$: the transmission time in the local wireless network between the device and the AP/base station;
- T_{inet}: the transmission time on the Internet between the AP/base station and the storage server;
- T_{server}: the processing time on the storage server; and
- T_{other}: all other time remaining.

When using *EdgeCourier*, T_{inet} and T_{server} are not changed. $T_{wireless}$ is likely to be reduced because the smaller bandwidth requirement between mobile device and AP. However, the adoption of *EdgeCourier* also introduces two main components to the document synchronization time:

- T_{diff}: the time to generate the edit-patch using `ec-diff` on the client; and
- T_{patch}: the time to apply the edit-patch using `ec-patch` on the EPS instance.

We performed an experiment evaluate the practical impact that *EdgeCourier* can bring to document synchronization time. We first developed our own cloud storage sync app so that we can instrument the app code to get the overall document sync time. We also instrumented our prototype system to obtain the different components in the document sync time. In the experiment, we synchronized three documents with text size of 1 KB, 100 KB, and 1000 KB respectively, each of which contained an edit of one-character-addition. We performed the experiment 6 times, and report the average results in Figure 11. In the figure, the line-symbol plot shows the overall sync time when not using *EdgeCourier* (i.e., the direct sync time). Each stack column shows the breakdown of the document sync time when *EdgeCourier* is applied. Please note that since we have no control over the cloud storage server, we could only instrument the cloud storage traffic manager on EPS to obtain T_{inet} and T_{server} as one piece, to which we add $T_{wirelss}$ in the figure to show the impact of network transmission (i.e., both wireless and wired) to the sync time. We can have four observations from the experiment results.

- Our system achieved similar document sync time as direct synchronization. On average, our system only introduced around half a second delay.
- Network transmission is the most significant component in the breakdown of sync time when using *EdgeCourier*.
- For document with large size, the time spent on network transmission when using *EdgeCourier* is notably lower than direct synchronization. For example, for the document with 1 MB text, using *EdgeCourier* caused about 0.6 second less on network transmission than direct sync.
- As document text size increased, using `ec-diff` to generate the edit-patch on mobile device became another major source of time consumption. For example, with 1 MB of text, the time used on generating the edit-patch accounts for about 39% of the overall sync time. We believe there is still significant space for further reducing patch generation time. For instance, in our current implementation, the edit-patch generator reads both the latest version and the last-synced version of the document being synchronized from persistent storage. A promising improvement would be to prefetch and/or cache those frequently accessed documents (e.g., the ones that the user is currently editing) in memory. We leave this optimization to our future work.

Comparing `ec-diff` with existing solutions. The previous experiment shows that generating edit-patch has a non-negligible impact on document sync time, especially for documents with large amount of text. However, if we had not designed the `ec-diff` approach, the time overhead would be much bigger. We performed another experiment to further evaluate the performance of `ec-diff` by comparing it to the existing solutions.

Currently the best general-purpose diff algorithm is considered to be the one developed by Myer [53]. In our `ec-diff` implementation, we used an library [1], which implements Myer's diff algorithm, to perform the actual comparison of those non-identical chunks. Using the same library, we developed a diff tool, which is notated as `g-diff`, for this experiment. We compared the text comparison performances when using the Linux `diff` utility [31], the `g-diff`, and our `ec-diff`. In a comparison, one side is an original document, the other is the same document containing an one-char-addition edit. Table 3 shows the experiment results. From the result we ca see that `ec-diff` has the best performance among the three. As we discussed previously, the performance can be further improved by performing prefetching and caching of large and frequent accessed documents in memory.

6.3 Evaluating using Unikernel as EPS runtime environment

We conducted experiments to evaluate how our choice of using Unikernel as the EPS runtime environment in our prototype system can help scaling EPS instances on the Edge node. We used Docker [19], another well-known lightweight virtualization solution, as a comparison. To prepare for the experiments, we ported our *EdgeCourier* EPS implementation to Docker-based VM. We used two metrics for evaluating the "lightweight-ness" of Unikernel-based EPS runtime and Docker-based EPS runtime. They are 1) how the edge node's normal services can be affected and 2) how the performance of EPS can be affected, when the number of EPS instances is scaled up in the edge node.

For the first metric, we connected two WiFi nodes via the

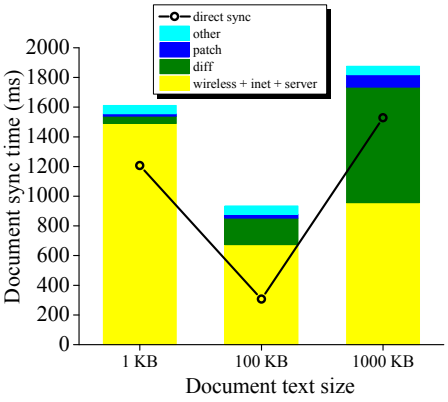

Figure 11: Document synchronization time (and breakdown).

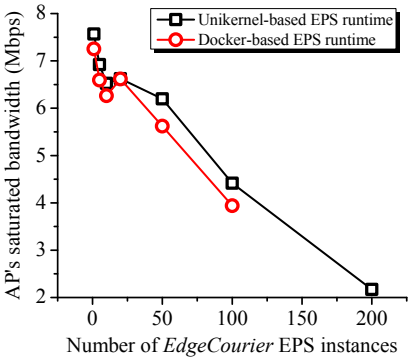

Figure 12: Unikernel-based vs. Docker-based EPS runtime: AP's saturated bandwidth.

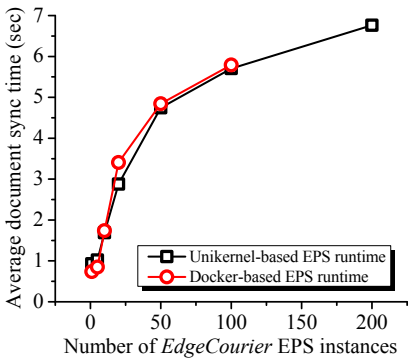

Figure 13: Unikernel-based vs. Docker-based EPS runtime: document sync time with *EC* EPS.

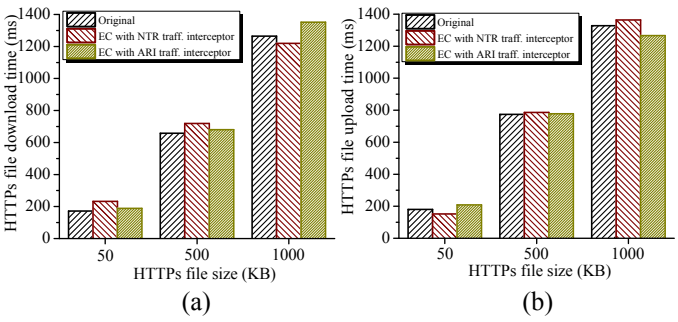

Figure 14: Time overhead for non-cloud-storage HTTP traffic in *EdgeCourier*-enabled mobile device.

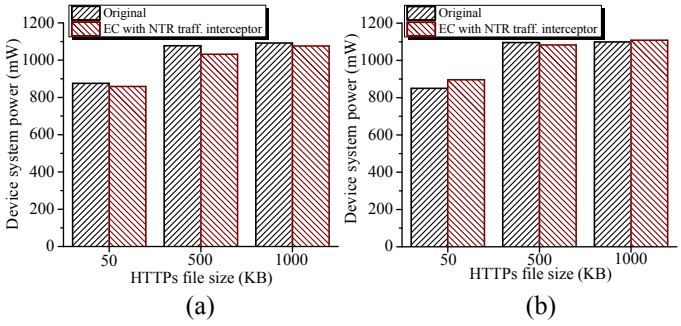

Figure 15: Power overhead for non-cloud-storage HTTP traffic in *EdgeCourier*-enabled mobile device.

edge node (i.e., an wireless access point), and established an iperf [32] UDP connection between the two WiFi nodes. We set the iperf connection bandwidth to a large value such that the saturated bandwidth on the AP was always reached. We increased the number of active EPS instances[2] on the edge node and observed how the saturated AP bandwidth changed in response to the EPS instance increase. In the experiment, we set the number of active *EdgeCourier* EPS instances to 1, 5, 10, 20, 50, 100, and 200. It is worth noting that in our experiment, the Docker-based EPS instances could not reach 200, which might support our expectation that Unikernel-based EPS runtime scales better than other lightweight virtualization techniques, such as Docker-based EPS runtime. Figure 12 shows the result of this experiment. We can see that the AP always achieved better saturated bandwidth with Unikernel-based EPS runtime than with Docker-based runtime, and the advantage was more obvious when the number of EPS instances was large.

For the second metric, we measured the average document sync time achieved by the active *EdgeCourier* EPS instances when scaling up the number of EPS instances. Figure 13 shows the result. We can see that Unikernel-based EPS instances achieved slightly better document sync time than Docker-based EPS instances.

Our experience and the experiment results (albeit the small advantage as seen from the results) can support our choice of choosing Unikernel as EPS runtime environment. Please note that the Rumprun Unikernel framework we used is in its early stage of development, and has much less community support than Docker has. By contrast, Docker is much more mature and has undergone many optimizations. We believe there are many opportunities, as well as challenges, to improve Unikernel used as EPS runtime environments, and we leave this to our future work.

[2]An active EPS instance means the EPS instance is actively performing its designated taks.

6.4 Evaluating overheads for non-cloud-storage HTTP traffic

As introduced previously, the implementation of the cloud storage traffic interceptor was based on the mobile device proxy framework developed in our previous work StoArranger [12, 13]. In our StoArranger work, we have two different implementations of the proxy framework: one implementation is based on network traffic redirection (short as NTR); and the other implementation is based on app runtime instrumentation (short as ARI). Further details can be found in [12, 13].

Since the cloud storage traffic interceptor is essentially a HTTP proxy, it intercepts all HTTP traffic, including those non-cloud-storage traffic. Therefore, non-cloud-storage HTTP traffic might suffer from certain performance degradation (e.g., needs more time, consumes more power). To evaluate this aspect, we developed an app that uploads/downloads files to/from an HTTPs server. Figure 14 and Figure 15 show the time needed, and the system power consumed (measured using a Monsoon power monitor [51]), to transfer non-cloud-storage files of different sizes using this app. In the figures, the columns labeled as "original" show the results of when not using *EdgeCourier*. From the results,

we can see that our prototype system only caused slightly longer transmission time and slightly larger system power for non-cloud-storage HTTP traffic.

7. CONCLUSION

In this paper, we have demonstrated and analyzed the problem of whole-file synchronization for office suite documents, which is common to the existing cloud storage services, in the common cloud-storage-backed document editing scenario. We proposed a system named *EdgeCourier*, aiming to address the problem effectively. We also proposed the concept of edge-hosted personal services (EPS), which enjoys many benefits including help deploy the proposed *EdgeCourier* system easily in practice. We implemented the proposed *EdgeCourier* system and deployed it in the form of EPS. Our extensive real-world experiment evaluation shows that our prototype system can effectively reduce document synchronization bandwidth with negligible overheads.

Acknowledgements

We thank the anonymous reviewers for their tremendously valuable feedbacks. This work was supported in part by NSF Award #1566375.

8. REFERENCES

[1] Diff, Match and Patch libraries for Plain Text. https://code.google.com/p/google-diff-match-patch/.

[2] inotify - monitoring filesystem events. http://man7.org/linux/man-pages/man7/inotify.7.html.

[3] rsync. https://rsync.samba.org/.

[4] seafile. https://github.com/haiwen/seafile.

[5] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *USENIX OSDI*, 2002.

[6] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's data compression proxy for the mobile web. In *USENIX NSDI*, 2015.

[7] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski. Libra: a library operating system for a jvm in a virtualized execution environment. In *ACN VEE*, 2007.

[8] S. Annapureddy, M. J. Freedman, and D. Mazieres. Shark: Scaling file servers via cooperative caching. In *USENIX NSDI*, 2005.

[9] Apache Software Foundation. Apache OpenOffice. https://www.openoffice.org/.

[10] Apple Inc. iWork. https://www.apple.com/iwork/.

[11] R. Z. Arndt. How to Get By Using a Tablet As Your Main Computer. http://www.popularmechanics.com/technology/gadgets/how-to/a16747/use-a-tablet-as-your-main-computer/.

[12] Y. Bai, X. Zhang, and Y. Zhang. Improving Cloud Storage Usage Experience for Mobile Applications. In *ACM APSys*, 2016.

[13] Y. Bai and Y. Zhang. StoArranger: Enabling Efficient Usage of Cloud Storage Services on Mobile Devices. In *IEEE ICDCS*, 2017.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SOSP*, 2003.

[15] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan. Fast, scalable and secure onloading of edge functions using airbox. In *IEEE/ACM SEC*, 2016.

[16] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *USENIX OSDI*, 2002.

[17] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao. Quicksync: Improving synchronization efficiency for mobile cloud storage services. In *ACM MobiCom*, 2015.

[18] E. De Lara, D. S. Wallach, and W. Zwaenepoel. Opportunities for bandwidth adaptation in microsoft office documents. In *USENIX Windows Symposium*, 2000.

[19] Docker, Inc. Docker. https://www.docker.com/.

[20] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *USENIX ATC*, 2003.

[21] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. Benchmarking personal cloud storage. In *ACM IMC*, 2013.

[22] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: understanding personal cloud storage services. In *ACM IMC*, 2012.

[23] Dropbox Inc. Dropbox. https://www.dropbox.com.

[24] Dropbox Inc. Official Dropbox Android app. https://play.google.com/store/apps/details?id=com.dropbox.android.

[25] D. R. Engler, M. F. Kaashoek, et al. Exokernel: An operating system architecture for application-level resource management. In *ACM SOSP*, 1995.

[26] Google Inc. Google Drive. https://www.google.com/drive/.

[27] Google Inc. Official Google Drive Android app. https://play.google.com/store/apps/details?id=com.google.android.apps.docs.

[28] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *ACM MobiSys*, 2014.

[29] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *ACM MobiSys*, 2013.

[30] P. Hao, Y. Bai, X. Zhang, and Y. Zhang. Poster: EPS - Edge-hosted Personal Services for Mobile Users. In *ACM MobiSys*, 2017.

[31] J. W. Hunt and M. MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories New Jersey, 1976.

[32] U. iPerf The ultimate speed test tool for TCP and SCTP. Opendocument technical specification. https://iperf.fr/.

[33] D. Johnson. The 8 most popular document formats on the web. http://duff-johnson.com/2014/02/17/the-8-most-popular-document-formats-on-the-web/.

[34] Kingsoft. WPS Office. https://www.wps.com/.

[35] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and

A. Liguori. kvm: the linux virtual machine monitor. In *Linux symposium*, 2007.

[36] T. Knauth and C. Fetzer. dsync: Efficient block-wise synchronization of multi-gigabyte binary data. In *USENIX LISA*, 2013.

[37] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang. Towards network-level efficiency for cloud storage services. In *ACM IMC*, 2014.

[38] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Y. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai. Efficient batched synchronization in dropbox-like cloud storage services. In *ACM Middleware*, 2013.

[39] Linux Foundation. Xen Project. https://www.xenproject.org/.

[40] LinuxContainers.org. Linux Containers. https://linuxcontainers.org/.

[41] P. Liu, D. Willis, and S. Banerjee. Paradrop: Enabling lightweight multi-tenancy at the networkâĂŹs extreme edge. In *IEEE/ACM SEC*, 2016.

[42] P. G. Lopez, M. Sanchez-Artigas, S. Toda, C. Cotes, and J. Lenton. Stacksync: Bringing elasticity to dropbox-like file synchronization. In *ACM Middleware*, pages 49–60. ACM, 2014.

[43] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. J. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, et al. Jitsu: Just-in-time summoning of unikernels. In *USENIX NSDI*, 2015.

[44] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *ACM ASPLOS*, 2013.

[45] A. Madhavapeddy and D. J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30, 2013.

[46] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *USENIX NSDI*, 2014.

[47] Microsoft Corporation. Microsoft Office. https://www.office.com/.

[48] Microsoft Corporation. Official OneDrive Android app. https://play.google.com/store/apps/details?id=com.microsoft.skydrive.

[49] Microsoft Corporation. OneDrive. https://onedrive.live.com.

[50] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *ACM SIGCOMM Computer Communication Review*, 1997.

[51] MonSoon Solutions Inc. Monsoon Power Monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[52] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *ACM SOSP*, 2001.

[53] E. W. Myers. An o (nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.

[54] R. Pegoraro. Can an iPad Pro or Surface Pro 4 Tablet Replace Your Laptop? http://thewirecutter.com/reviews/can-pro-tablets-replace-your-laptop/.

[55] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library os from the top down. In *ACM ASPLOS*, 2011.

[56] S. J. Purewal. 10 ways your smartphone has already replaced your laptop. http://www.greenbot.com/article/3006339/smartphones/10-ways-your-smartphone-has-already-replaced-your-laptop.html.

[57] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *USENIX FAST*, 2002.

[58] D. Rasch and R. C. Burns. In-place rsync: File synchronization for mobile and wireless devices. In *USENIX ATC*, 2003.

[59] Rump Kernel. Runprun unikernel. https://github.com/rumpkernel/rumprun.

[60] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4), 2009.

[61] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 2016.

[62] M. Smith. Can a tablet replace your laptop? We used an iPad for three months to find out. http://www.digitaltrends.com/computing/can-a-tablet-replace-your-laptop/.

[63] The Document Foundation. LibreOffice. https://www.libreoffice.org/.

[64] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. C. Bressoud, and A. Perrig. Opportunistic use of content addressable storage for distributed file systems. In *USENIX ATC*, 2003.

[65] A. Tridgell, P. Mackerras, et al. The rsync algorithm, 1996.

[66] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *ACM EuroSys*, 2014.

[67] J. Valcarcel. In Less Than Two Years, a Smartphone Could Be Your Only Computer. http://www.wired.com/2015/02/smartphone-only-computer/.

[68] R. Weir. Opendocument format: The standard for office documents. *IEEE Internet Computing*, 13(2):83–87, 2009.

[69] Wikipedia. ODROID. https://en.wikipedia.org/wiki/ODROID.

[70] Wikipedia. Office open xml. https://en.wikipedia.org/wiki/Office_Open_XML.

[71] S. Yi, Z. Hao, Z. Qin, and Q. Li. Fog computing: Platform and applications. In *IEEE HotWeb*, 2015.

[72] S. Yi, C. Li, and Q. Li. A survey of fog computing: concepts, applications and issues. In *ACM Workshop on Mobile Big Data*, 2015.

[73] J. Yoon, P. Liu, and S. Banerjee. Low-cost video transcoding at the wireless edge. In *IEEE SEC*, 2016.

[74] Y. Zhang, C. Tan, and L. Qun. CacheKeeper: A System-wide Web Caching Service for Smartphones. In *ACM UbiComp*, 2013.