To Offload or Not to Offload? The Bandwidth and Energy Costs of Mobile Cloud Computing

Marco V. Barbera, Sokol Kosta, Alessandro Mei, and Julinda Stefa
Department of Computer Science, Sapienza University of Rome, Italy.
Email: {barbera, kosta, mei, stefa}@di.uniroma1.it.

***Abstract*—The cloud seems to be an excellent companion of mobile systems, to alleviate battery consumption on smartphones and to backup user's data on-the-fly. Indeed, many recent works focus on frameworks that enable mobile computation offloading to software clones of smartphones on the cloud and on designing cloud-based backup systems for the data stored in our devices. Both mobile computation offloading and data backup involve communication between the real devices and the cloud. This communication does certainly not come for free. It costs in terms of bandwidth (the traffic overhead to communicate with the cloud) and in terms of energy (computation and use of network interfaces on the device).**

In this work we study the feasibility of both mobile computation offloading and mobile software/data backups in real-life scenarios. In our study we assume an architecture where each real device is associated to a software clone on the cloud. We consider two types of clones: The off-clone, whose purpose is to support computation offloading, and the back-clone, which comes to use when a restore of user's data and apps is needed. We give a precise evaluation of the feasibility and costs of both off-clones and back-clones in terms of bandwidth and energy consumption on the real device. We achieve this through measurements done on a real testbed of 11 Android smartphones and an equal number of software clones running on the Amazon EC2 public cloud. The smartphones have been used as the primary mobile by the participants for the whole experiment duration.

I. Introduction

The advances in technology of the last decades have undoubtedly turned yesterday's must-have devices into today's stock. Think of the phones with aerials of the late '80, or the Pentium 4 PCs of a few years ago. None of them is comparable to the power of nowadays smartphones, whose recent worldwide market boost is undeniable. We use smartphones to do many of the jobs we used to do on desktops, and many new ones. We browse the Internet, send emails, organize our lives, watch videos, upload data on social networks, use online banking, find our way by using GPS and online maps, and communicate in revolutionary ways. New apps are coming out at an incredible pace. Apple iPhone commercial's call to action "There's an app for everything" says a lot on this matter. Nonetheless, the more eager we get when using our smartphones by installing new apps, the less happy we are with the lifetime of the battery. The problem is that we rely upon a number of crucial pieces of information that are only stored in the device (phone numbers, addresses, notes, appointments, etc.), or, in some cases, that can be got only by using the Internet on the fly as many of us are used to do. It is so important to keep our smartphone operational that everyday we pay attention to our battery and try to save it by reducing the number of phone calls, or by avoiding to watch too many videos, just enough to be able to reach home and recharge it. But that means that we cannot use our device to the fullest.

Alessandro Mei is supported by a Marie Curie Outgoing International Fellowship funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n. 253461.

This work has been technically supported and partially funded by Telecom Italia within the Working Capital project.

This work has been performed in the framework of the FP7 project TROPIC IST-318784 STP, which is funded by the European Community. The Authors would like to acknowledge the contributions of their colleagues from TROPIC Consortium (http://www.ict-tropic.eu).

Many researchers believe that cloud computing is an excellent candidate to help reduce battery consumption of smartphones, as well as to backup user's data. Indeed, many recent works have focused on building frameworks that enable mobile computation offloading to software clones of smartphones on the cloud (see [1], [2], [3] among others), as well as to backup systems for data and applications stored in our devices [4], [5], [6]. Both mobile computation offloading and data backup involve communication between the real device and the cloud. This communication does not come for free, in terms of both energy consumption (utilization of network interfaces to send the data) and bandwidth [7]. Several works consider the trade-off between the energy spent to offload specific application modules and the energy saved thanks to the cloud [2], [3], [8], [1], [9]. Also do exist analytical models that aim to predict approximately these costs in the case of mobile computation offloading [10], [11]. However, all these works are limited to consider best case scenarios—ideal WiFi connectivity and WiFi interface always switched on. To the best of our knowledge there is no study on the cost of keeping updated a clone or a backup on the cloud in a real setting with real mobile smartphone users, where Internet connectivity is often guaranteed by WiFi only at home and at work, by 2G/3G when moving outside these areas, and where coverage is often not present (think of subways, for example).

Our goal in this work is to study the feasibility of both mobile computation offloading and mobile software/data backups in real-life scenarios. In our study we assume an architecture as the one in [9], [1], where each real device is associated to a software clone on the cloud. For the mobile computation offloading to work, the status of the applications running on the clone needs to be in sync (as accurately as possible) with the ones running on the real device. In order to allow users

978-1-4673-5946-7/13/$31.00 ©2013 IEEE

to offload mobile computation on the fly, the clone must run the same applications the real device is running. In addition, also the state of both the real application (the one running on the device) and the cloned application (the one running on the clone), has to be the same. We will refer to a clone that is used to offload computation on the fly as an *off-clone*, and to a clone that is used to backup user's data and to enable restore as a *back-clone* (backup clone). Since the goal of the back-clone is to restore user's data and the system (including installed apps) in case of system/data corruption or loss, it does not need to keep up with any single application's status in time, as in the case of the off-clone. Yet, it needs to be up to date with all sort of data generated/received by the user like notes, pictures, videos, contacts, calendar entries, messages, emails etc.

We give a precise evaluation of the feasibility and costs of both off-clones and back-clones in terms of bandwidth and energy consumption on the real device. Our contribution is as follows:

- We study the network availability (2G/3G, WiFi etc.) as well as the signal quality in a real testbed of mobile smartphone users, with reference to the requirements of offloading and backup on the cloud;
- we study the data communication overhead that is necessary to achieve different levels of synchronization (once every 5min, 30min, 1h, etc.) between devices and clones in both the off-clone and back-clone case;
- we report on the costs in terms of energy incurred by each of these synchronization frequencies as well as by the respective communication overhead.

To achieve all the above we design and build *Logger*, an Android app that runs in the foreground and collects data on the utilization of the device. Logger also handles the communication between the real device and the cloud. We run Logger on a testbed of 11 smartphones (associated to 5 different carriers) that make use of clones running on the Amazon's EC2 cloud platform. The paper is organized as follows: Section II presents the related work in the area; Section III explains, in details, the components of Logger, the system we built to collect the data for our study; Section IV describes the results obtained from the real testbed, while Section V concludes the paper.

II. Related Work

The proliferation of smartphones has undoubtedly spurred on developers to build a large variety of apps that allow users to better exploit their powerful devices. Testimonial of this effect is the plenitude of new apps that are built and uploaded to official and unofficial markets every day. However, we are still far from utilizing our last-generation smartphones to the fullest. The real limitation is the battery. Recently, researchers have explored the idea that one promising way to make the battery of our devices last longer is to offload part of the computation from the mobiles to the cloud. A number of recent works propose different methodologies to offload computation for specific applications [12], [13]. Other works describe frameworks that enable offloading for applications in general, as long as the application builder correctly identifies and tags either code binary or methods as "off-loadable" [14], [3], [9], [2]. Simultaneously, lots of works discuss on how to pre-compute/estimate the actual gain in terms of energy that comes from using this technique, independently from the specific application [10], [11]. In [15] the authors argue that previous frameworks that enable offloading are inaccurate in estimating the energy of a certain workload in the system. This is because they do not operate at kernel level. With this in mind they present AppScope, an Android-based energy estimator implemented as a kernel module that uses an event-driven monitoring method. By operating at kernel level their framework yields energy consumption estimation values very close to those of the Mobile Device Power Monitor[1], a tool used in other works in the area [3], [16].

Computation offloading is not the only thing the cloud comes to use: Arguing on the importance of data that nowadays users store on their mobile devices, the authors in [4] develop a remote control system for lost handsets that aims to protect personal information of users. Nonetheless, the remote control system has to be triggered so that to lock/unlock real device's functionalities/access to data, and eventually backup the data on an online server. If the thief abruptly cancels the data by formatting the SD card and re-installing the OS, the user will never be able to get her data again. So, backup/restore systems that regularly send user's data to remote servers for backup are very valuable in this context. The system proposed in [5], besides from backup/restore, also allows for sharing information in smartphones among groups of people. The authors test their system in terms of time needed (on the phone) to backup three different data types: SMS, calendar events, and contacts. In [6] the authors argue that not only contacts and emails—synced by e.g. Google sync on Android OS—but also application settings, game scores etc., are important to users. With this in mind they build ASIMS, a tool that has the goal of providing a better application settings integration and management scheme for Android mobiles. ASIMS is based on SQLite, it stores other applications' settings and syncs them to the Internet. Its interface makes it possible for other applications to store settings in one common place and for users to select which applications they want to sync. In addition, SociableSense [17] shows how also social related applications can benefit from cloud offloading and CloudShield [18] exploits sociality among smartphone clones to stop worm propagation in smartphones. Similarly, also systems like [19], [20], [21], [22] might benefit from offloading computation to the cloud.

The work in [10] presents energy models that trade-off the energy consumption on the mobile device versus the energy needed to send the data to the cloud, to encrypt it, and to manage other operations related to this process. The authors of [7] discuss on the main factors that affect the energy consumption of mobile apps in cloud computing, and deem that such factors are workload, data communication patterns,

[1] http://www.msoon.com/LabEquipment/PowerMonitor/

and usage of WLAN and 3G. In a more recent work [11] the authors present an analytical study to find the optimal execution policy. This is identified by optimally configuring the clock frequency in the mobile device to minimize the energy used for computation and by optimally scheduling the data rate over a stochastic wireless channel to minimize the energy for data transmission. By formulating these issues as a constrained optimization problem they obtain close-formed solutions which give analytical insight in finding the optimal offloading decision.

None of the many previous works related to mobile cloud computing explicitly studies the actual overhead in terms of bandwidth and energy to achieve full backup of both data/applications of a smartphone, as well as to keep, on the cloud, up-to-date clones of smartphones for mobile computation offload purposes. In this work we address both issues, and, for the first time (to the best of our knowledge), we provide results with a real testbed of 11 Android powered smartphones and associated clones running on the Amazon EC2 platform.

III. The Logger

Studying the overheads incurred by the off-clones and the back-clones is not easy. It depends on how user actually uses her device, on the properties of the network technologies (WiFi/3G/ etc.) and on which of these technologies is actually available and used. So, we develop Logger, an Android app that continuously logs the events occurring in the device. These includes user and system generated events. Examples of user generated events are: Mail sending and receiving, phone calls (both incoming and outgoing), files exchanged over blue-tooth, device switching on and off, battery charging, installing and uninstalling applications. Examples of system generated events are: Access to network interfaces (e.g. regular heartbeat pings to Google's servers), access to GPS radio (e.g. Google Latitude or Facebook), generation and editing of internal files by apps, automatic updates, and so on.

The Logger service is structured into sub-components, each responsible for logging data coming from a given resource. While some information about the device state can be logged passively by our Logger using standard facilities provided by the Android OS, collection of other statistics require the Logger to actively and recurrently send requests to the OS. In the remaining of this section we give details on both the passive and active aspects of our Logger. Also, we describe the difficulties we faced in building this tool and how we overcame them.

A. Passive data collection

Some of the Android OS components are loosely coupled (e.g. network interfaces and the browser). To make them communicate the Android OS provides the Intents—instances of the *android.content.Intent* class. Intents are asynchronous messages and can be used to perform all sorts of operations: Requesting the system to launch applications, asking the user to select a WiFi network to connect to, sending notification messages to other applications, signalling the Android system that a certain event has occurred (e.g. message received, connection available, etc.). In this latter case, a component interested in a specific event does not have to actively send the system requests on the occurrence of the event. It suffices that the component registers to the specific event through the so called intent filters, and it will be notified by the system as soon as the selected event occurs.

Our Logger uses Intents to be notified about changes in the device connectivity (2G, 3G, WiFi), of the battery status (charging, discharging, current battery life), of the screen state (on/off) etc.

B. Active data collection

Data collected passively through Intents are not enough for our study. Statistics as network data usage, for example, cannot be obtained through Intents. In this case, the Android OS has to actively be sent specific requests by the interested component. In Android systems this is achieved through Alarms—actions scheduled to be executed recurrently, even when the device is in sleep mode.

So, we built a set of alarms and included them in our Logger architecture in order to log data exchanged through the network interfaces, the set of currently running applications, incoming and outgoing emails, SMSs, phone calls, and so on.

C. Filesystem activity

Monitoring the filesystem's activity includes logging information on when and how each file or directory was created, deleted, accessed, and modified. This information is clearly important in both the study of the off-clones (as far as files internal to single applications are concerned) and back-clones (user-generated files). The Alarms are not suited to achieve this: They would require very frequent scan of the whole filesystem, which would be inaccurate, aside from being very expensive in terms of battery consumption. Fortunately, the underlying Linux kernel on which Android OS is based includes *Inotify*—a filesystem monitoring service with a similar approach to that of Intents. Inotify allows applications to add so called *watches* to directories. Whenever one of the watched directories, or its content, is modified, the kernel promptly notifies the "watching application". The Android OS provides user-level applications with a Java interface to the Inotify subsystem.

The notifications are sent only when something actually happens on the filesystem. So, the interface to the Inotify subsystem allowed our Logger to get the most accurate information possible about the filesystem changes at a minimal cost in terms of resources. Inotify does not allow to watch for changes happening in the subdirectories of a directory. To overcome this problem the Logger, when started, adds watches to all the directories in the filesystem tree of the device. In addition, whenever a new directory is created/deleted the watch is added/removed automatically by the Logger. It is worth noting that adding each directory of the filesystem tree to the Inotify watch list is not expensive in terms of memory

and computational resources, and it did not affect the user experience in a noticeable way.

D. File permissions

An Android device storage is usually composed of an external storage unit, in the form of an SD card, and one or more internal storage units. The external SD card is where most of the user files are stored (mp3s, pictures, etc.). Conversely, system files like config files and system binaries are stored in the internal storage unit. The SD card storage (usually a FAT32 filesystem) is the one with the highest capacity (for example 10 GB) and is accessible in read/write mode by all the apps without restrictions. While it is possible for an app to have its data stored on the SD card (this is very useful for large apps like games or other media-based ones), most of them are installed in the internal storage space.

As opposite to the external SD card, the access to the internal storage (an ext3 or ext4 filesystem) is restricted: A user-level app is only allowed to access its private data (usually stored in the /data/data/ directory). This prevents any user application (including our Logger), to add Inotify watches to the private directories of other apps. To overcome this limitation we root the devices of our experimental testbed. By rooting a device, an application can execute commands using the root permissions, thus going past the restrictions of user applications, including those on filesystem access. Our Logger, even though running on a rooted device, is not allowed to execute its own code with root permissions. It is however enabled to execute shell commands as root. This permits us to use the Unix shell included in Android OS to temporarily change the permissions of private application files. The permission changing step takes place during the starting phase of the Logger. The old permissions are saved into a database (internal to Logger), and are restored whenever it is stopped by the user. To make the permission changing phase as independent as possible from the Android platform we used BusyBox[2], a popular lightweight collection of Unix tools largely used and optimized for embedded systems. Finally, with these modifications, the Logger is enabled to add Inotify watches to files that are private to other apps.

E. Dealing with file modifications

Once Inotify sends a notification to our Logger about the modification/editing of a directory/file, we need to understand the amount of data that we should upload to the cloud so to keep the clone up-to-date with the user's device. Of course, uploading has to be performed in an efficient way—we certainly do not want to upload the whole file/directory each time it is modified. A standard technique is to send just incremental modifications of the file, instead of the whole file. Unfortunately, nor does Inotify provide details on the amount of data that was modified, neither does it keep track of the modified file portion. So we had to find a method to identify the file portions involved in a modification that was fast and lightweight. Fast, so that it could cope with frequent modifications occurring in the system. Lightweight, so that its overhead is low and it would not affect the user experience. For all these reasons we decided to apply the *rolling hash* [23] technique, widely used on popular backup/sync software *rsync*[3]. Rolling hash is one of the possible implementations of the so called *binary-diff*, used also by *Dropbox* to update users' modified files yet not wasting users' network bandwidth[4]. This technique allows to compute very quickly (in linear time with respect to file sizes) hashes of all possible file blocks (the block size is a parameter of the rolling hash technique). So, it makes possible to quickly compute the amount of data (in terms of blocks) that make two file versions differ. A very detailed and clear description of the rolling hashes technique (including how to compute it and how to deal with hash collisions) can be found in [23].

We built from scratch a Java implementation of the rolling hash technique, and included it as one of the modules of our Logger app. Old rolling hashes of files are maintained into a SQLite database (stored in the phone's SD card) so that they are promptly available to be compared with the new rolling hashes each time files are modified. The block size we used in our implementation equals $8KB$, the same used by rsync.

Our implementation of the rolling hash technique turned out to be very efficient in computing and comparing hashes. Nonetheless, we believe that an implementation written in native code (C/C++) and running as a root process would dramatically improve the performances. We leave this as future work.

F. Logger output

The data collected by each component of the Logger is continuously written to a log file (a simple text file) that is periodically rotated, compressed and stored onto a special directory within the SD card reserved to the Logger. The user, on request, can also trigger sending emails with the content of this directory, from time to time, to a custom gmail account we created for this purpose. If the user does so, the log files that are successfully sent by email are deleted from the device.

IV. Experimental results

Our experiments are based on a testbed of real users. In this section we describe the testbed, the results, and the observations of our study on the cost of keeping updated both off-clones and back-clones in the cloud.

A. Experimental setting

To gather data related to the device usage we set up a testbed of 11 smartphones running our Logger app. In addition, the testbed consists of an equal number of software clones—customized AMIs of the Android x86 OS [1]—running on the Amazon's EC2 platform. Logger makes the device communicate with the clones to collect data related to networking.

[2]http://www.busybox.net/

[3]http://rsync.net/

[4]https://www.dropbox.com/help/8/en

Number, type & OS	CPU	RAM
7×Samsung Galaxy S Plus (Android 2.3)	1.4 GHz Scorpion	512 MB
2×Samsung Galaxy S (Android 2.3)	1 GHz Cortex-A8	512 MB
1×Samsung Galaxy Note (Android 2.3)	1.4 GHz dual-core Cortex-A9	1 GB
1×Samsung Galaxy Nexus (Android 4.1)	1.2 GHz dual-core Cortex-A9	1 GB

TABLE I
PHONE SPECIFICATIONS.

The details of the mobile devices involved in the experiment are shown in Table I. The devices were used as primary smartphones for the whole duration of the experiment (the experiment lasted 3 weeks) and involved people living in the city of Rome, Italy, and Cambridge, UK. The profiles of the participants are heterogeneous in age and occupation—university students, faculties, and part-time and full-time workers that are completely external to the university. So is the technology they use to connect to the Internet with their mobile devices: One of the participants does not have a cellular data traffic plan, so he connects only through WiFi networks (home or work). Another participant does not have an available WiFi network neither at home nor at work, so he only makes use of 2G/3G technology. The remaining participants use habitually both cellular data traffic and WiFi interchangeably.

During the experiment the participants used their own contract with the service provider of their choice. The data we gathered includes the four major cellular service providers in Italy, namely, TIM, Vodafone, Wind and 3, as well as O2, a major provider in UK. It is worth to note that, though each of these carriers relies on its own network infrastructure, they all provide cellular data traffic plans with the same upload and download speed in best case scenarios. The actual speed and availability depends, of course, on the quality of the signal and of the load of the cells that are used by the participants during the three weeks of the experiment. As this is an element that might impact the performance of mobile cloud paradigms, we start our study with network availability.

B. Mobile data traffic and network availability

Certainly one of the most important component to keep both off-clones and back-clones updated is the ability of the real devices to communicate often and efficiently with the cloud. Often, so that the user can send data to the cloud anytime she needs. Efficiently, so that the overhead of this communication is as low as possible and does not impact the usability of the system. Indeed, if the networking process becomes a bottleneck to usability then people will not be happy about it, and they will less likely accept and use mobile cloud systems.

The first network related result that we present is the average amount of time per day, in percentage, users spend connected either to 3G/2G or to WiFi networks (see Figure 1). The first observation that we make is that the percentage of the daily time period during which the devices are connected is quite high (more than 90%), and the trend is similar for all the days of the week. The period of disconnection are either due to

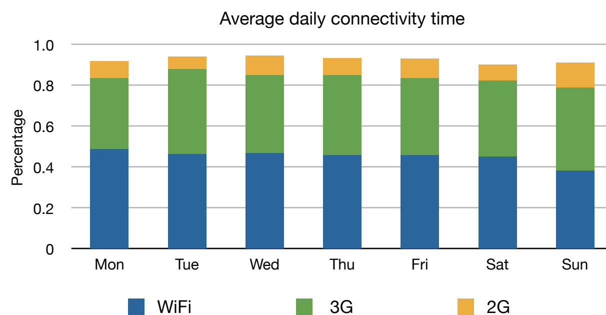
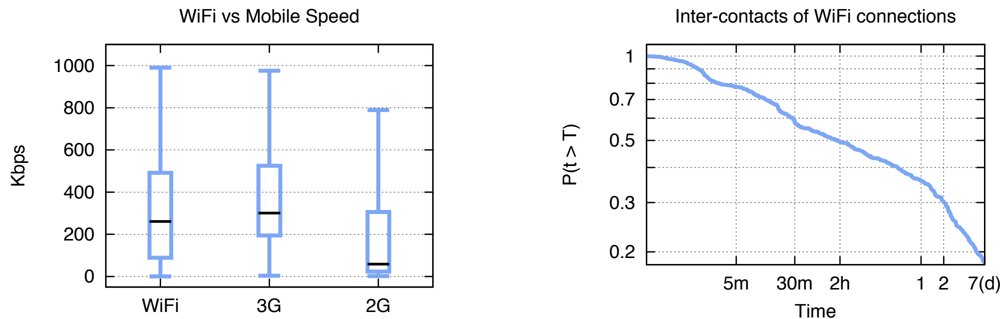


Fig. 1. Average daily connectivity percentage for various technologies.

areas with no signal at all (e.g. subways), or due to periods in which the devices are switched off. The time of connectivity using the three different networking technologies (WiFi, 3G, 2G) is also quite stable from day to day: Around 45% of WiFi, around 40% of 3G, and the remaining of 2G. This partitioning is particularly positive being WiFi and 3G the two connection technologies with larger bandwidth.

Keeping off-clones and back-clones up-to-date requires the device to upload data to the cloud. For this reason we measure, separately for 3G/2G and WiFi, the average speed in upload during the real life conditions of our testbed. We achieve the measurement by making Logger send a file of 300kB from each device to its clone on the Amazon platform every 30 min, keeping track of the technology used to send the data, and averaging the results. We compute the average speed for every user (per day). Figure 2(a) shows the average as well as a representation of the distribution of the values: Minimum, 25th, 50th, and 75th percentiles, and maximum. By doing so we are able to depict in the graphic results related to scenarios where the user happens to be in zones that are not well covered by wireless technology.

Let us first focus on comparing 3G with 2G connectivity. When using 2G/3G technology, smartphones switch from 3G to 2G whenever the 3G signal is not present or lower than a certain threshold, which depends on the smartphone type. As expected, the 2G upload speeds are much lower than the 3G ones (see Figure 2(a)). Fortunately, this 3G to 2G switch typically happens only for short periods of time (see Figure 1). WiFi connections are faster than 3G ones, as far as download is concerned. However, the mobile cloud paradigm requires that the majority of traffic travels from the real devices to the cloud, to keep the clone up-to-date. The results in Figure 2(a) show that the average upload speeds are higher in the 3G case with respect to WiFi. This is not surprising: When accessing the WiFi users typically set up their devices to automatically connect to known and trusted WiFi routers that are at their home or work locations. These in Italy are generally ADSL WiFi routers, and thus they do not provide high speed in upload. This last consideration makes think that, when a device is being charged, it is more likely to be connected to a WiFi network—usually we charge our devices at home or at work where WiFi is available. Not to mention that ADSL connection providers make users pay fixed prices for unlimited bandwidth. So, exploiting WiFi networking to keep clones up-to-date is more likely to come for free in both terms of costs (no cellular



(a) Average (per user) daily upload speed. The graphics include the minimum and maximum speed value as well as the 25th, 50th and 75th quartile.

(b) Cumulative distribution of WiFi connection inter-contact times.

Fig. 2. Network properties.

data traffic is used) and energy (the device has more chances to be charging).

Figure 1 shows that the overall WiFi connection time per day is high, around 45% of the time. However, it is important to see how this period of time is distributed over the day. This is an important question to address. Indeed, as long as the device is under WiFi coverage, keeping off-clones and back-clones updated is cheap and therefore users might configure the system in such a way to use WiFi only to sync. So, we are interested in checking whether it is frequent that users are without WiFi connection for long period of times. A technical way to check this property is to compute the distribution of the intervals between the end of a period of WiFi coverage and the start of the next period of WiFi coverage. We refer to this time as the WiFi inter-contact time. The results of our experiments to this respect are shown in Figure 2(b).

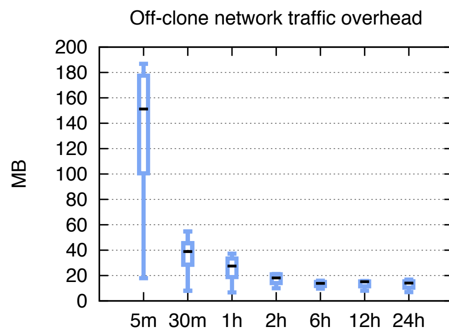
From Figure 2(b) we can see that 20% of the WiFi inter-contact times are lower than 20 min, 40% are lower than 30 min, 50% are lower than 2h. Just 35% of them are higher than 1 day. Considering that among the participants there are also people that do not use WiFi at all, this result is excellent—50% of WiFi inter-contact times are as low as 2h and that means that 50% of the times users can rely on WiFi connection to sync their device's clones as frequently as every 2 hours or more. And, as we already discussed, this makes users save their cellular data traffic, and, possibly, save battery (if during the WiFi connectivity they are charging their device at home or at work.).

C. Overhead of Mobile Cloud Computing

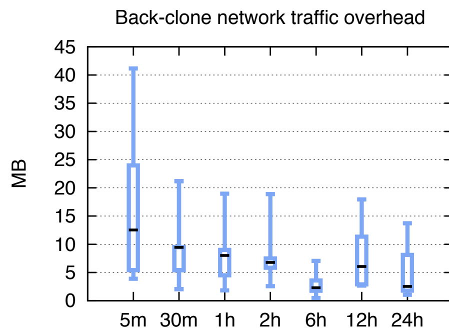
To keep off-clones and back-clones up-to-date devices need to recurrently access the network interfaces to send the latest user-generated data and app statuses to the respective clones. Clearly, the more often the device syncs with the clone, the better the clones represent the status of the device. Though, frequent syncs incur higher bandwidth and energy overhead to the real device. The overhead depends on the network technology involved in the device-cloud communication (WiFi vs 2G/3G), as well as on how the device is used—for example, the more pictures a user takes with her device's camera the more data have to be transferred to the user's back-clone. The user should be able to tradeoff this overhead with the synchronization level of the clone of her device, and decide the rate of synchronization that better suits her needs.

Off-clone synchronization requires information on the user apps. For a given app this information includes the app's state, its internal private files, its settings specified by the user, and so on. Back-clone synchronization requires informations on the apps that are installed in the device and user's data (contacts, calendars, pictures, music, notes, emails, SMS etc.). As we already anticipated in Section III, the Logger computes, at the end of specific time-intervals that correspond to sync frequencies, the amount of data that the device should send to the cloud in order to update each clone type. In particular we recall that, when a file is modified, we use the efficient rolling hash technique to compute the difference between the new and the old version of the file. The sync intervals that we have considered are 5 min, 30 min, 1h, 2h, 6h, 12h, and 24h. To keep it as real as possible, the Logger computes the amount of data to be sent at the end of each interval only if either 2G/3G or WiFi connectivity is available.

1) Network bandwidth overhead: In Figures 3(a) and 3(b) we plot, for both clone types, the quartiles of the distribution of the average (per user) daily traffic needed to be sent to the cloud in dependence of the sync frequency in order to keep off-clone and back-clone updated. The first simple observation that we make is that the traffic overhead incurred by the synchronization decreases as the sync interval increases. This phenomenon is straightforward—while the device is running there is a large number of files/directories generated and then destroyed in the system (temporary files). Typically, this happens more frequently with system or app private files rather than with user generated files. Indeed, the results in Figure 5, that show the complementary cumulative distribution of the file lifetime for both clone types in the device, confirm this intuition—more than 90% of off-clone files last less than 1s, while more than 90% of back-clone files last longer than 5 days. If the device-cloud synchronization interval is long, most of the temporary files generated during the interval do not last till its end—when the file diffs are computed. On the contrary, if the sync interval is short, it is more likely that temporary files are still "alive" at the end of the interval, so are involved in the file diffs. This is what boosts up the overhead traffic incurred by small synchronization intervals



(a) Average (per user) off-clone traffic overhead.



(b) Average (per user) back-clone traffic overhead.

Fig. 3. Bandwidth overhead for clone synchronization in dependence of the sync frequency. The graphics include the minimum and maximum speed value as well as the 25th, 50th and 75th percentile.

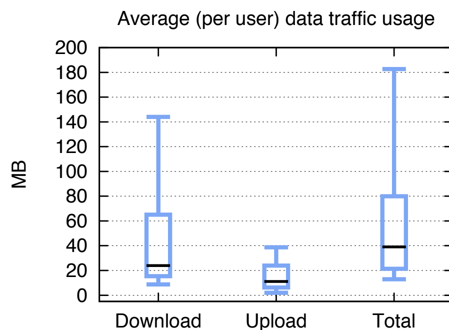


Fig. 4. Average (per user) data traffic sent/received per day.

even in the presence of a smart incremental backup system like the one we implemented. As this is more likely to happen with system/app private files, the difference between the overhead traffic generated by the synchronization for different sync interval lengths is smaller for back-clones—these clones involve more user-generated data (like sent/received emails, texts, calls, and so on) which is rarely deleted by the user. Another important observation is that, for small sync intervals, the back-clones incur much less overhead (around 4 times less) than the off-clones. Again, this is due to the fact that the user generates data with much less frequency with respect to the system, and typically do not delete their data. The overhead difference between the two types of clones is attenuated when the interval of syncs increases (see Figures 3(a) and 3(b)).

Lastly, for both types of clones, the bandwidth overhead incurred is not excessive, if we compare it to the data traffic normally generated by the user's device to receive/send mail,

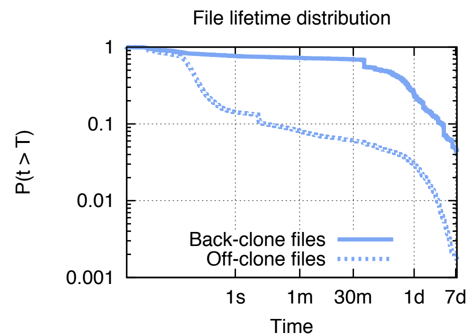


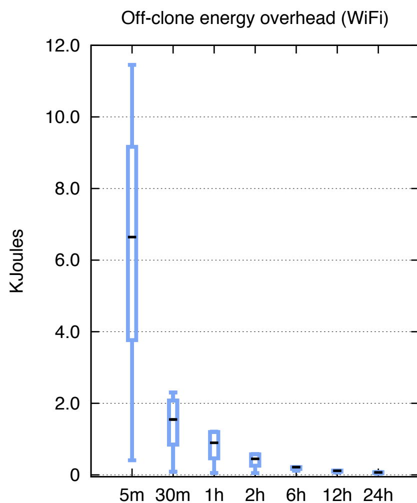
Fig. 5. Distribution of file lifetime relative to off-clones and back-clones.

access Facebook/Twitter accounts etc. For the sake of comparison we have plotted, in Figure 4, the quartiles of the average traffic generated normally by the users per day.

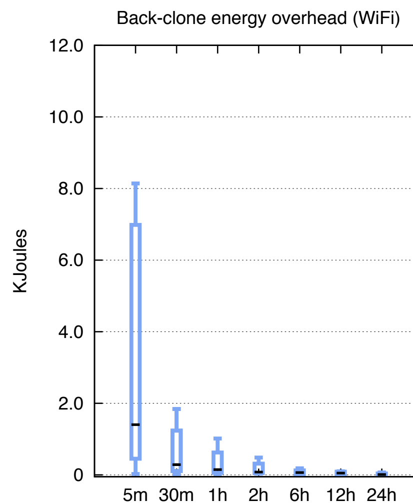
2) Energy overhead: To measure the energy overhead incurred by the sync of back-clones and off-clones we make use the Mobile Device Power Monitor, an external power meter widely used to validate offloading frameworks esteems [3], [16], [15]. The device samples the smartphone's battery with high frequency (5000 Hz) so to yield accurate results on the battery's power, current, and voltage. Clearly, we could not do the measurement when the devices were with the users. So, at the end of the experiment, we gathered the devices and re-simulated the device-cloud communication for each smartphone with the device connected to the Power Monitor. The simulation included re-computation of the amount of data to be sent to the clones at the end of the time-interval considered, as well as the data sending process. The clones involved are customized Amazon AMIs (Amazon Machine Image) of the Android-x86 OS and run on the Amazon EC2 platform.

The simulation is run 20 times for every device: 10 times using WiFi connectivity and the other 10 times using 3G connectivity. Figures 6(a) and 6(b) show the trend of the energy overhead in the case of WiFi connectivity for both off-clones and back-clones in dependence of the sync frequency. Similarly, Figures 6(c) and 6(d) show the trend of the energy overhead in dependence of the sync frequency in case of 3G connectivity. There are several considerations to be made. First, let us consider WiFi sync compared to 3G sync of the same clone type. See, for example, Figure 6(a) and Figure 6(c) that depict the results for the off-clones. When the same clone type is considered, the energy overhead difference is determined by the communication technology used. Indeed, the energy dissipated to compute the file diffs is the same, being that we are comparing the same clone types among them. However, the energy consumed in computing the rolling hashes of the files dominates the energy required to actually send the diffs. As a result, the overall energy overhead of the sync through WiFi is slightly lower than the overhead of the sync done through 3G. The difference is of the order of tens of Joule.

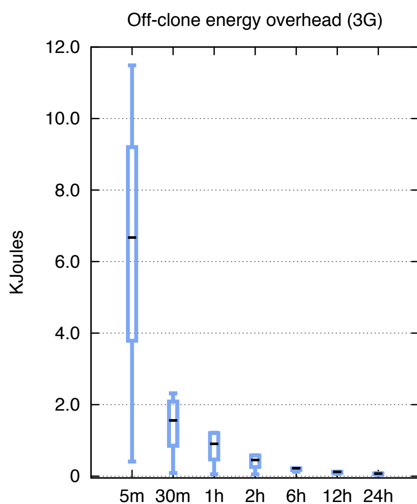
Now, let us focus on one communication technology, say, WiFi, and compare the energy overhead to sync off-clones (Figure 6(a)) with the energy overhead to sync back-clones



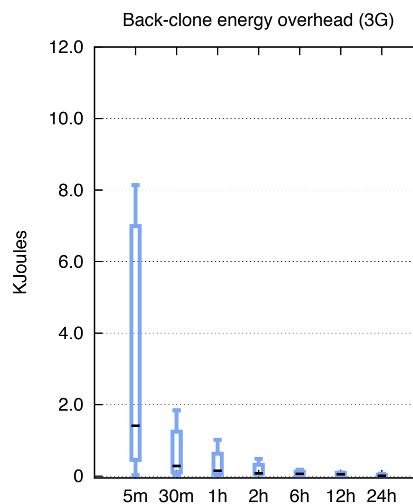
(a) Off-clone energy overhead (WiFi) per day.



(b) Back-clone energy overhead (WiFi) per day.



(c) Off-clone energy overhead (3G) per day.



(d) Back-clone energy overhead (3G) per day.

Fig. 6. Energy overhead for clone synchronization through WiFi (a and b) and through 3G (c and d) in dependence of the sync frequency. The graphics include the minimum and maximum speed value as well as the 25th, 50th and 75th percentiles.

(Figure 6(b)). It is worth noting that, although off-clones incur higher energy costs than back-clones, the difference in energy requirement is much more attenuated than the difference in traffic overhead. This is so for the following reason: As shown in Figure 5, off-clones generate many small temporary files (the system files). On the other hand, the files generated by the back-clones are user-generated files. Typically, these are bigger, are generated with a lower frequency, and last longer. So, the computation load of the file diffs is somehow balanced in the two cases—For the off-clones the rolling hashes are computed very frequently times on small files; for the back-clones, the rolling hashes are computed much less frequently on bigger files. The energy for this computation dominates the energy required to send the file diffs. So, even though back-clones generate 4 times less traffic overhead than off-clones, the difference of the overall energy spent to compute the diffs and to send them to the cloud for both clone types is attenuated. The same observation holds for the case when 3G technology is used in the device-clone communication.

That said, it is worth noting that mobile cloud computing does not impact much the life of the battery. The smartphones we used in our testbed are powered by Lithium–Ion batteries (1650 mAh, 3.7 V). These batteries, if fully charged contain 21.9 KJoule of energy. According to our experiments, synchronizing off-clones (back-clones) every 5 minutes, incurs, at max, around 11.8 KJoule (8 KJoule) of energy overhead per day (see Figure 6). This means that the sync cost is about 53% (36%) of the battery. These values certainly correspond to an extreme scenario—keeping the clone updated every 5 min certainly has its cost. As soon as lower synchronization frequencies are considered these values are drastically reduced. For 30 min (2h) sync intervals the synchronization cost drops to around 11% (2.7%) of the battery for the off-clone and around 8% (2.3%) of the battery for the back-clone.

Lastly, the differences in terms of energy overhead between off-clones and back-clones (being it WiFi or 3G) are attenuated

with the increasing of the synchronization interval. In addition, large synchronization intervals yield much lower energy overhead than short ones.

V. Lesson learned and conclusions

In this work we have described our experience with a testbed of real users of smartphones and a mobile cloud system of smartphone software clones in the cloud. The goal of the experiment is to understand the feasibility of mobile cloud systems in a real setting—a setting consisting of people (participants in the experiment) using the smartphone as their primary device during their normal life. In the experiment, the participants made use of their mobiles just as usual for three weeks. Here are some of the key observation we made during this experience:

- Most of the users are virtually always under coverage of some wireless technology. In particular, almost 50% of the time smartphone users are connected to a WiFi access point. Indeed, this is often the case at home and at work (recall that these numbers are computed as the average of all the participants);
- in more than 50% of the cases, users lose WiFi coverage for just for 2 hours at most. This is probably due to commuting between places with WiFi, like work and home. For a systems point of view, it means that sync operations can optimistically wait until the device is connected to a WiFi access points, and that most probably this is going to happen in a short period of time.
- Synchronizing back-clones (for backup purposes) requires less network traffic (down to 4 times less) and less energy overhead (around to 3 KJoule less) than synchronizing off-clones (that handle mobile computation offload).
- The difference in overhead incurred by the synchronization of the two clone types decreases drastically as the sync frequency decreases; reasonable sync frequencies like 30 min have a reasonable cost in term of energy spent on the device to keep off-clones and back-clones updated (11% of the battery for the off-clones and 8% for the back-clones sync). Recall that by paying this overhead, the user either has a very efficient backup system or can efficiently offload computation on the fly. This latter service has the potential, depending on the application, to reduce energy consumption by a factor that is much higher than the cost we computed in this experiment.
- Finally, WiFi technology incurs lower overhead with respect to 3G. However, the overall energy overhead, which depends also on the workload before the device-cloud communication, is almost the same with both communication technologies.

Our work supports the conclusion that mobile cloud computing can be sustained by continuous update of software clones in the cloud with a reasonable overhead in terms of bandwidth and energy costs, especially if the sync intervals are not too short.

References

[1] S. Kosta, C. Perta, J. Stefa, P. Hui, and A. Mei, "Clone2clone (c2c): Enable peer-to-peer networking of smartphones on the cloud," T-Labs, Deutsche Telekom, Tech. Rep. TR-SK032012AM, 2012.

[2] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading." in *Proc. of IEEE INFOCOM 2012*, 2012.

[3] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proc. of MobiSys '10*, 2010.

[4] I. Joe and Y. Lee, "Design of remote control system for data protection and backup in mobile devices," in *Proc. of ICIS 2011*, 2011.

[5] V. Ottaviani, A. Lentini, A. Grillo, S. D. Cesare, and G. Italiano, "Shared backup & restore: Save, recover and share personal information into closed groups of smartphones," in *Proc. of IFIP NTMS 2011*, 2011.

[6] C. Ai, J. Liu, C. Fan, X. Zhang, and J. Zou, "Enhancing personal information security on android with a new synchronization scheme," in *Proc. of WiCOM 2011*, 2011.

[7] A. Miettinen and J. Nurminen, "Energy efficiency of mobile clients in cloud computing," in *Proc. of HotCloud 2010*, 2010.

[8] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: Versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, Austin, Texas, December 2010.

[9] B. G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proc. of EuroSys '11*, 2011.

[10] K. Kumar and Y. H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *IEEE Computer*, vol. 43, no. 4, pp. 51–56, April 2010.

[11] Y. Wen, W. Zhang, and H. Luo, "Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones." in *Proc. of IEEE INFOCOM 2012*, 2012.

[12] G. Portokalidis, P. Homburg, K. Anagnostakis, and H.Bos, "Paranoid android: versatile protection for smartphones," in *Proc. of ACSAC '10*, 2010.

[13] E. Chen and M. Itoh, "Virtual smartphone over ip," in *Proc. of IEEE WoWMoM '10*, 2010.

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

[15] C. Yoon, D. Kim, W. Jung, and C. Kang, "Appscope: Application energy metering framework for android smartphone using kernel activity monitoring," in *Proc. of USENIX ATC 12*, 2012.

[16] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. of IEEE/ACM/IFIP CODES/ISSS '10*, 2010.

[17] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow, "Sociable-sense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing," in *Proc. of Mobicom '11*, 2011.

[18] M. Barbera, S. Kosta, J. Stefa, P. Hui, and A. Mei, "CloudShield: Efficient Anti-Malware Smartphone Patching with a P2P Network on the Cloud," in *Proc. of The 12th IEEE International Conference on Peer-to-Peer Computing (P2P 2012)*, September 2012.

[19] A. Mei and J. Stefa, "Give2Get: Forwarding in Social Mobile Wireless Networks of Selfish Individuals," in *Proc. of The 30th IEEE International Conference on Distributed Computing Systems (ICDCS '10)*, June 2010.

[20] A. Mei, G. Morabito, P. Santi, and J. Stefa, "Social-Aware Stateless Forwarding in Pocket Switched Networks," in *Proc. of The 30th IEEE Conference on Computer Communications (INFOCOM '11)*, April 2011.

[21] A. Mei and J. Stefa, "Give2Get: Forwarding in Social Mobile Wireless Networks of Selfish Individuals," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 4, pp. 569–582, 2012.

[22] M. Barbera, J. Stefa, A. Viana, M. D. Amorim, and M. Boc, "VIP Delegation: Enabling VIPs to Offload Data in Wireless Social Mobile Networks," in *Proc. of the 7th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS '11)*, June 2011.

[23] A. Tridgell and P. Mackerras, "The rsync algorithm," The Australian National University, Tech. Rep. TR-CS-96-05, 1996.