Time and Space in Android-based Opportunistic Networks

Andre Ippisch, Tobias Küper, Kalman Graffi
Technology of Social Networks Group
University of Düsseldorf, Germany
{ippisch, tobias.kueper, graffi}@hhu.de

***Abstract*—Mobile devices like smartphones or other smart objects can create Opportunistic Networks with intermittent node contacts due to their wireless technology and their user's social behavior. In disaster, emergency and censorship situations, or if simply no internet connection is available, Opportunistic Networking can be used to connect devices to each other. We attempt to improve our Android-based network and take a look at time modification and buffer management issues. Syncing the time is a known problem due to the probable lack of a connection to the internet. Since the devices are user-controlled, changes in time have to be noticed and time differences between devices have to be exchanged. Also, we take a look at the special buffer which is not reserved entirely for the network but is shared with all applications on the device. For this paper, we developed functionality in form of multiple libraries to detect time changes and possible storage shortages on the devices, to synchronize time for our network, and to remove messages in a system in which there is not only one buffer but multiple virtual ones that belong to different applications. We conduct tests which detect when the Android system begins to notify about storage congestion and starts deleting buffer content. With these results, our libraries can improve the drop and queue policies that have to react before user data is deleted.**

***Index Terms*—Opportunistic Networks, Smartphones, Smart Objects, Android, Buffer Management, Clock Synchronization**

I. Introduction

Nowadays we have powerful mobile devices like smartphones or tablets which include great features like mobile communication with other people, or taking pictures and making videos. But with great devices comes great resolution and thus these files become larger and need a lot of space on the device. With chat applications and social media it is common to upload or share these files with friends and family. However, either the users have to upload and download those large files to and from a server which consumes a lot of quota on monthly data plans or these images and videos are decreased in their resolution.

This is only one, possibly the most prevailing, of many use cases for an Opportunistic Networking framework for mobile devices, called *opptain* [1], with which files and messages, from now on called *bundles* can be shared without an internet connection. It is a case in which the *opptain* framework only enables an additional networking channel in parallel to the common ones like mobile networks and a stationary 802.11 infrastructure mode network.

Another imaginable but undesired scenario is the possibility to communicate with other people if the infrastructure of the mobile network or the internet is not available or has been destroyed. This can happen because a region is struck by a natural disaster, due to censorship in a country or because there are no communication routes available like in developing countries. Normally, people in this region cannot communicate when they are in need for help or want to inform others of their survival and/or location. Today, most people have a smart device which can, without any reconfiguration, connect to the *opptain* network and send and receive data or messages as *bundles*.

opptain will serve as the foundation of this paper. If *opptain* is enabled on off-the-shelf smart devices not only the devices are connected but also the users behind those and messages can be sent to people that are not in the direct surroundings. With the before mentioned 802.11 based networking channel data can be transmitted over a distance of about 100 meters in average. This happens in the background without user interaction; the user only has to initialize the application once and the bundles are received or forwarded while the device stays in the user's pocket. The framework serves only as virtual network and transport layers in the stack, the application layer is represented by additional mobile applications that use *opptain*'s API to pass bundles from the application to the transport layer and back. When a bundle is created it is packed with essential information like creation time, Time-To-Live (TTL) and the bundle content which is a pointer to a file so that no duplicate storage space must be wasted. Possible third party applications, several of which are already implemented, are filesharing, chat, location, emergency beacon, and also gaming apps. The normal process is creating some content, packing it to a bundle, passing it to the framework, sending it to another device and then possibly to other devices and passing the bundle on the destination's device to the third party application. All content can be encrypted and signed with public key cryptography, but only the network streams are encrypted to address the duplicate storage space problem.

The framework which is already used in several domains started with the assumption of nearly synchronized clocks and without the need for a special buffer management. In the following paragraphs we want to introduce the need for looking at those issues.

Each bundle that is sent via the *opptain* network needs some extra information. For routing purposes, especially forwarding and dropping policies, the bundle contains information about

the creation time and TTL. For a local connection it is important that the clocks on two devices are synchronized that a bundle is not dropped before the TTL is reached or the bundle is kept to long.

In Opportunistic or Delay Tolerant Networks syncing the time with reliable methods like NTP is a known problem due to the probable lack of a connection to the internet. Also, since the devices are user-controlled, changes in time have to be adapted to the messages in the buffer. Furthermore, the time difference has to be published to other devices, so that relay devices can use the correct creation timestamps and TTL of the messages and the destination node can order the received bundle and bring it into some context.

For keeping the bundles on the devices, for relay purposes for example, it is important to have sufficient storage space. Nowadays, there are many device types which have a wide range of storage space. Some high end devices come with up to 128GB of storage and might never be filled completely but most of the devices, even the newer but maybe cheaper ones, still only come with 8 to 16GB and most of that storage is filled by the operating system and the applications. One question to answer is what happens if the device runs out of space because of the forwarding of large files in the network.

The buffer is not reserved entirely for the Opportunistic Networking application, instead, it is shared with all applications of the users and their personal data. For user-related devices like smartphones or smart objects, there must be a difference between persistent user data, non-persistent user data, and the relay data of our network. It is very important that users do not lose personal, perhaps important, data or get disturbed in their usual interaction with their device, especially in emergency situations.

In this paper, we take a closer look at both the time related issues, and our special case of buffer management on mobile devices. For the correct realization of the routing algorithms, the TTL, the order of bundles in user applications, and the drop policies, we must develop several mechanisms. First, methods to detect time changes and compare time in our network without being able to set the time on the devices, as seen as in Section III. Second, a method to check the buffer space for different user related virtual buffers, as shown as in Section IV.

II. Related Work

For our Android-based Opportunistic Network (OppNet) it is important to synchronize the device's time which is needed for delivering bundles that only stay in the network for a finite time. Many papers, like [2], handle the subject of clock synchronization and describe algorithms to reach a state in which there is almost the same time set in the network. But there are different ways to reach time synchronization. For example, the time can be synchronized at each node by getting closer step by step to the faster clock [3], by calculating the average of the clocks of all local nodes [4], [5] or by "gradually decreasing the clock error with neighboring nodes using an exponential weighted moving average" [6]. All these authors make the assumption that the synchronization via the Network Time Protocol (NTP) [7] is not possible. NTP is used to synchronize computer clocks over the internet and enables synchronization between the hierarchically arranged servers and clients. NTP is intended for connected internet nodes where the synchronization operation can be conducted continuously and frequently between the reference node and the clients. In OppNets devices are typically never connected to some centralized structure and a time synchronization is therefore not easy. Even if there were some centralized structure that would serve as a global clock, the devices are moving and have no fixed location. The keyword *location* introduces another possibility for nodes to retrieve an accurate timestamp, namely the Global Positioning System (GPS). While there is the possibility to synchronize clocks over GPS [8], it is not implemented on mobile devices and cannot be adapted without rooting or jailbreaking the devices which is not in the scope of our off-the-shelf approach. A new approach is needed to solve the time synchronization problem on mobile devices.

In our OppNet peers exchange data between each other via the store-carry-forward principle [9], which means that the data is forwarded to relay nodes in the network until the destination is reached. Each node has his own properties of processing power, disk storage or network bandwidth and bundles are routed on the best effort basis. If the node cannot forward the bundle due to a missing connectivity link, the bundles are buffered into a queue and scheduled for transmission once there is a connection. In case of congestion, if the buffer has reached his maximum, bundles are dropped based on a drop policy. There are many combinations of queue policies, drop policies and routing algorithms, most queue and drop policies consist of one or more of the following aspects: Arrival time, hop count, Time-To-Live, message size and randomness.

For different scenarios particular combinations of the aspects are useful for the policies and have distinctive impact on the routing protocol. The paper [10] takes a closer look at possible combinations and their impact on the routing protocol. Also the performance of different combinations is analyzed for the proposed routing protocol. The policies and their behavior are described in more detail in [11], [12], [13], [14], [15], [16], [17]. All these policies have in common that they are applied in a one buffer system. To apply those policies in our system, we first have to deal with the shared buffer on mobile devices.

III. Time

There are several issues with time management on mobile devices. First, the changing of time on the device. Bundles in the buffer means that there is a database for the header data and they are saved with a timestamp that is consistent with the current devices. If the time on the devices changes, all times in the database have to be adapted. The timezone changes can be noticed and processed accordingly.

A. Time Changes

One of the challenges in our system is to have the right or most correct time on each device, because there is no centralized server which can verify the time. On Android smartphones the time based on the Unix time which counts the seconds (without leap seconds) since January 1 in 1970. With this value each date can be identified and it is easy to calculate other specific times with a variety of built-in functionality or libraries. With the concept of time zones, other geographical areas have different current times than the local one and the time difference with the daylight saving time has to be handled as well. In our *opptain* framework the current time, according to our clock, is set to each bundle that a device creates together with an offset of this creation time that serves as the TTL. The current time cannot be changed by any application but only by the user in the settings or by the Android system itself. One of the questions is how an application like our framework can react on those time changes by the Android system. In the following different cases the time on a device can be set:

- after the device has booted where the time is set initially,
- automatically if the device has some connection where it can be synchronized with a time server,
- automatically after a time zone change,
- by the user manually,
- with GPS on a rooted device,
- after a long time without battery and charging where the time can be reset to the epoch (01.01.1970 00:00:00).

Usually, in Android devices the time is set automatically by synchronizing with a time server so that there should not be a significant difference between the sender and receiver time. In the case of the internet or the mobile network connection not being available for a longer time and the user changes their geographical area with a different time zone, the time on the phone is not change until the devices have a connection to the Internet or the the user change it by itself.

In some cases the user likes to set the time according to their needs, for example people that are constantly late might set their clock forward for some minutes to be more on time. Then, the time is set by the user at the device's settings and is not synchronized via the internet from then on.

The last option can occur if the Android device is booted after the device is take to another time zone. Only after a few minutes and a internet connection the time will be change otherwise the device have the old time. In addition it can occur a failure on the boot where a wrong time will be set by the system.

If the time is changed on an Android device the system sends out an *Intent* message which can be received by any application. However, this time change *Intent* does not contain further information about the time change like the previous or the new time, it only informs about *that* there was a change. To get the delta (δ) of the time change, we need a possibility to calculate the difference of the timestamp after and before the time change. The current timestamp can be retrieved from the system by built-in functionality but the old timestamp is not retrievable by default. We know from tests that the time on the device can not be changed while the device is powered off, except for the epoch time value. This is because even if the device is turned off there is actually some battery left to power precisely the clock. Nevertheless, after some more time being turned off, there is no energy left and the device will boot with the epoch time value.

After the device is turned on again there is a value present in the system that counts the seconds since the device has booted that is called *elapsed real time* on Android and *CPU time* on iOS devices. Directly after the device is booted up we calculate the delta (Δ) between the current time (t_{ts}) and the *elapsed real time* (t_{ert}) with Equation 1. Now, if the system detects a time change (t seconds after we calculated Δ) and we receive the *Intent* message, we can calculate this delta again as Δ_{new} with Equation 2 and calculate δ with the old and new Δ values as Equation 3 shows.

$$\Delta = t_{ts} - t_{ert} \tag{1}$$

$$\Delta_{new} = (t_{ts} + t + \delta) - (t_{ert} + t) \tag{2}$$

$$\begin{aligned} \Delta_{new} - \Delta &= ((t_{ts} + t + \delta) - (t_{ert} + t)) - (t_{ts} - t_{ert}) \\ &= t_{ts} + t + \delta - t_{ert} - t - t_{ts} + t_{ert} \\ &= \delta \end{aligned} \tag{3}$$

After the time change, the δ has to be set to Δ_{new} to be prepared for the next time change. Now we can change the creation time of all currently buffered bundles with the calculated δ. For the special case that after the boot of the device the time has been reset to the epoch time the next time change must be ignored and handled according to the new time after the time change. If there is an active connection while a device has a time change and the handshake was already realized the update of the buffered bundles is done after the connection finished.

The functions for reacting on time changes in a more useful manner are implemented in our *time library* and can not only be used by our Android-based Opportunistic Network but also by other applications.

B. Time Synchronization

As already mentioned in Section II there is no possibility to synchronize the time on current mobile devices if there is not connection to the mobile network or the internet. However, the local time of the device is actually not that quite important for creating a bundle if we assume the creation time to be a construct that is only valid on the originator's device. Since the Time-to-Live (TTL) is only saved as an offset of the creation time, time changes have no impact on the TTL if they also have no impact on the creation time of a bundle. If the creation time is now only valid on the originator's device, we can use a relative creation time value on each following device that receives the bundle. To calculate the relative creation time value, again, a time difference (δ) has to be saved, but this time it is the time difference between two devices. Directly after the handshake between two devices that get connected to each other, the devices' current timestamps (t_{ct}) are exchanged and

both devices can calculate their δ towards the other device as seen as in Equation 4.

$$\delta = t_{receiver} - t_{sender} \tag{4}$$

Now, the relative creation time t_{rct} value can be calculated with Equation 5.

$$t_{rct} = t_{ct} + \delta \tag{5}$$

Additionally to the TTL and the creation time, an *opptain* bundle saves its routing path while it moves through the network. The routing path contains not only the identification of each relay node but also each δ value on the way to the destination.

These approaches have advantages for all device types that keep a message; sender, relay and destination nodes. The original creation time and TTL are never changed and can be signed by the sending device. Similarly, each entry in the routing path list can be signed by the relay node. All relay nodes can have the original and relative creation time and still can use the TTL value for routing purposes. The destination node can use the combined delta values to calculate the combined delta value and bring context into the bundles. The delta value is passed with each bundle to the third party application that can use it, too.

Important to note is that the exchange of timestamps is beneficial for both sides since one connection is used to transfer bundles in both directions.

Let's look at an example where bundles with a TTL of 10 minutes are created and two devices have different times on their system. Without the time exchange Alice with the time of 1:00pm sends a bundle and Bob with a time of 1:11 pm receives the bundle. The bundle will be dropped because the TTL will be reached once the bundle has arrived at Bob's device. In a system with a constant TTL of 10 minutes all bundles that Bob receives from Alice would be dropped. To receive the bundles, Bob's time has to be between 1:00pm and 1:10pm; ideally at exactly 1:00pm, like Alice's time. With the time exchange, the calculated TTL on Bob's device will be the same as on Alice's device. Figure 1 shows three different stages, each one minute apart, of a connection between Alice and Bob. The global time in this example, shown at the top, is 12:00pm at the beginning. Alice's device has the local time of 12:10pm and Bob's device the local time of 12:29pm. The first step shows the exchange of the timestamps and the calculation of the delta value with Equation 4 after the two devices finished their handshake. In the second step a bundle is sent from Alice to Bob. The bundle's creation time from the sender Alice without the adjustment of the receiving device is shown at the left while on the right the adjusted value is seen. In the last step the reverse part of step two can be seen; bundles can be sent in both directions.

The last example shows the usefulness of the combined δ value for a third party application, in this case a chat message app. When exchanging messages with another person the messages must not necessarily arrive but should be displayed in the right order. The sum of all values gives us the time

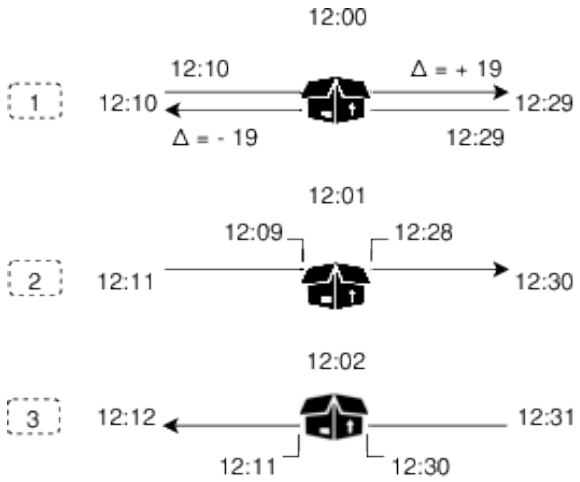
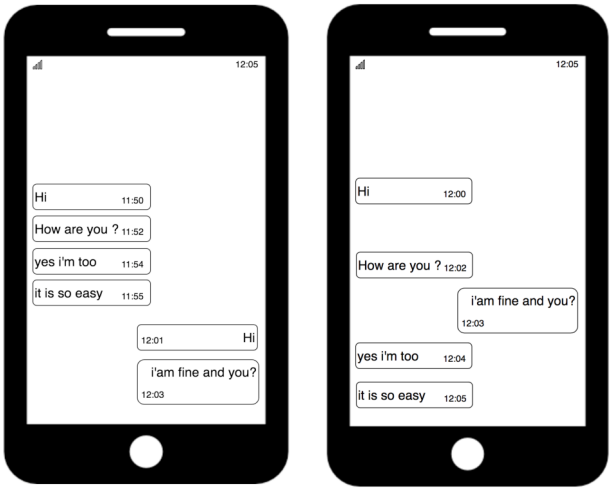


Fig. 1. This figure shows a connection between Alice and Bob.



(a) without reordering (b) with reordering

Fig. 2. Figure 2a shows the messages without reordering and Figure 2b shows the messages with reordering on a smartphone with calculated delta values.

difference between the originator's device and the receiver of the message. With this value the correct relative time can be calculated and the messages shown in the right order. Ion 6 the time differences of each pair of devices is added together.

$$\begin{aligned} \Delta_{path} &= \Delta_1 + \Delta_2 + \cdots + \Delta_{n-1} + \Delta_n \\ &= \sum_{i=1}^{n} \Delta_i \\ &= \sum_{i=1}^{n} (time_i - time_{i-1}) \end{aligned} \tag{6}$$

Figure 2 shows the difference between the possibility of reordering the message and its lack of it.

C. Transmission Delay

So far, in our approaches we have neglected one simple and small but important error. If a bundle is sent from one device

to another we learned that the creation time is adjusted and the TTL is checked against the new value. But, our adjustment doesn't contain the transmission delay of the timestamp exchange. The TTL values in our Android-based OppNet have a size of minutes and the duration of the transmission are in the magnitude of milliseconds or less. This value is negligibly small in Wi-Fi connections and was not added to the delta value before but actually can be guessed from the current network situation and included in all calculations. Also, of course, calculating and saving all necessary values takes time, too, but can also be neglected. One should note that other time syncing techniques, like NTP or GPS, also have those small errors.

IV. Space

We want to motivate this section with one sentence: On each smart device the storage space is limited.

A. Android Storage Options

On Android smartphones or tablets the storage for saving data can be divided into the three options [18] which are *internal*, *external* and *emulated storage*.

The *internal storage* is the one that is built into the smartphone. It is divided into two categories, the private persistence and the private cache. In the private persistence data is saved which only can be accessed by the application that owns the data. When the user uninstalls the application all the files are removed.. The private cache is similar to the private persistence but should only be used for temporary files. If there is not enough free storage space left on the Android system, the cache of each application can be cleared without prior notice. This is the main difference between the two categories.

The *external storage* is an option to save data on a removable medium like an SD card. This option has a private persistence and a private cache, too, but additionally, files can be saved in a public storage option where multiple applications can access those files. This public space can also be divided into a persistent and a cache space. The persistent space, for example, is used for *music*, *download* or *picture* directories that are accessible for the whole system. The cache space can also be cleaned by the system to free up some space.

The third option, the *emulated storage* is just a emulation of an *external storage* if there is no SD card present. The specific characteristic is that this storage is not removable and is stored on the same physical storage space. It has the same feature as the *external storage*, although, if the devices is connected to a computer the storage cannot be accessed by the devices because it is mounted to the computer.

B. Shared Buffer on Smart Devices

A network device can hold one of three roles towards a bundle that is present. First, it can be the originator, second, it can be a relay, and third, it can be one of multiple destinations. To prevent confusion: If the current device was the only destination of the bundle, it already would have unpacked the bundle and passed its content to the corresponding third party application that is responsible for this bundle type. To go on, bundle contents are only pointers to files that are present on the device. In case of a transmission to another device, the bundle is sent through the stream on the fly so that no copy of the bundle contents must be available on the device. This mechanism not only protects the buffer from wasting storage space but also makes the three before mentioned device roles very useful. If the device is the originator or one of the destinations there is not a decisive difference between keeping the bundle in the buffer or dropping it. This is because both, the originator and a device that is owner of one bundle copy but still has the intention to additionally forward it, have something in common: They would keep the bundle's content even if the bundle is dropped. Consequently, bundles whose content is saved permanently anyway have a higher priority than bundles that are only relayed on the current device. The latter are dropped according to their Time-To-Live (TTL) or the currently present drop policy.

Since there are multiple storage options, it is important to save the bundles at the right location. Furthermore the data of the device that does not belong to the *opptain* network has to be saved from deletion at all times. For bundles that are created on a device the third party application is responsible for the creation and saving of the content. As already mentioned the content is saved in a file and only the file pointer is handled by the framework. The file should be saved in a folder that is not categorized as a cache. For bundles for which the current device is the only destination, the bundle is passed to the third party application and this app is responsible for the saving of the content. For bundles for which the current device is one of multiple destinations the bundle is unpacked and stored on the public external persistent storage. Consequently, both the third party application and the *opptain* application can access the file. For bundles that are relayed on the current device, the private cache is the most fitting category, because the cache of Android is deleted first and other applications have no access to the content of the bundles. Also, the private application caches have a higher priority than the public cache.

What exactly happens if the space on an Android device is running low? The Android system sends an *Intent* which can, again, be received by every application on the device. The *Intent* informs that the system might have to remove files from the cache if the none of the applications free up space.

With this information, we now have an indicator when the buffer on the system is running out of space, and can act accordingly. For starters, each bundle of the framework can simply be saved in a non-cache directory, because from now on we manage our buffer completely ourselves and try to manage the user's buffer, too. If the *opptain* application receives the low storage *Intent*, it will free up space by applying the current drop policy on our bundles. Here we start with removing relayed bundles and remove other bundles afterwards.

The main difference between the buffer in traditional Opportunistic Networks and the one for our network is that our buffer is shared with many other applications. Files of the device's owner should be preferred to save and relayed bundles should

be dropped before bundles that are user-related.

C. Occupied Storage

The *opptain* framework and the Android system share one buffer, which is a special situation in an Opportunistic Network. Normally, the network can use the whole buffer without any limitation. But our framework is not the only application which run on the device. Therefore, the framework has to manage the buffer and to pay attention that the devices does not run out of space. Other applications can save files, that are only needed temporarily, in the cache. If only 10% free storage space is left the Android system shows a memory warning and sends the before mentioned *Intent* to all applications. It is not specified at which point the Android system starts to delete files from the cache. Even the 10% value from before is indeed only documented by Android but has not to be chosen by the manufacturers that install Android on their devices. Each manufacturer can override this value for their devices, which is why we ran a test on multiple devices to find the values which can be seen in Section IV-D.

There are two cases in which the framework has to react and drop bundles: due to an expired TTL or due to drop policies that are activated because the device runs out of storage. In the framework, bundles are saved both in a database and in the storage. In the database, the header information particularly creation time, relative creation time and TTL is saved, and in the persistent storage the content is saved as a file. The check if a bundle has to be dropped is realized by an *Alarm Manager* that wakes up the device if a bundle's TTL value is expired.

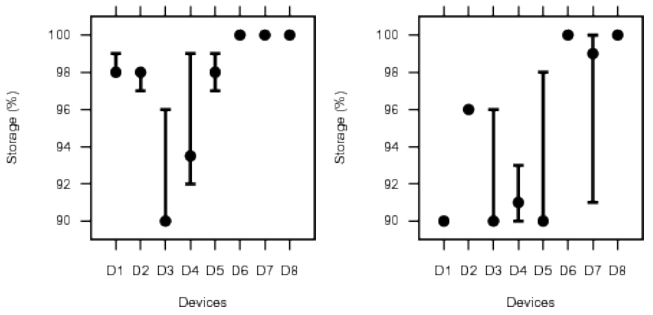
Drop policies have to act if the storage space on the device is running low. To see if the storage space is running low we have to find the right time to start removing bundles from the buffer.

Our proposed mechanism to detect storage space changes in the system is the following. In the beginning the remaining space on the device is saved and an initial time value t is set to the number of seconds it takes to fill up the device up to 90%. There are multiple ways to fill up the space on a mobile device quickly. Some examples are the shooting of a Full HD video, the downloading of a large file over a fast network connection or the connection to a computer to transfer files over a cable. All these write operations have an upper bound which is the speed of which randomly produced data can be written on the device. This speed is calculated by our *space library* once when the framework application is installed. The *Alarm Manager* is set to the time that is t seconds in the future. At this time the value t is calculated again and the *Alarm Manager* is set once again. If at one point the storage space is near the 90% mark the drop policy will act on the *opptain* buffer.

The *space library* is actually a bit more intelligent. There is a factor that is multiplied with t if the storage space has not change significantly in some time so that even less wake up times are necessary.

TABLE I
ANDROID TEST DEVICES WITH MODEL, MANUFACTURER, TYPE AND ANDROID-VERSION

Name	Model	Manufacturer	Android-Version	Device Name Test
Galaxy Nexus	Galaxy Nexus	Samsung	4.3	D1
Y3	Y360-U61	Huawei	4.4.2	D2
Desire 510	Desire 510	HTC	4.4.3	D3
Galaxy S4 mini	GT-I9195I	Samsung	4.4.4	D4
P9 lite	Huawei VNS-L31	Huawei	7.0	D5
Nexus 9 (LTE)	OP82200	HTC	7.1.1	D6
Nexus 5X	LG-H791	LG	7.1.2	D7
Pixel 2	G011A	Google	8.0	D8



(a) intent low storage (b) cache data

Fig. 3. In Figure 3a shows when the full storage *Intent* is sent. Figure 3b shows when the devices start to delete the cache data.

D. Device Tests

On Android devices it is defined but often not strictly implemented at which point the full storage space notification is sent and the cache storage is cleared. The threshold, defined by Android in general, is 90%, but this value can be changed by the device manufacturer. This value is not even documented for Android developers. After some testings on different device, see Section I, the threshold value is between 90% and 100%.

We developed two Android applications which test the behavior of the Android system when the full storage *Intent* is sent. The first application writes some temporary files, all with different sizes and different storage locations. The second application writes a large sum of temporary files to fill the storage of devices in order to simulate incoming bundles. Both applications monitor all files and log all written data, times, and deleted files by the system. It is important to use two applications instead of only one to check if there is a difference between cache files that are owned by the same application and cache files that are owned by another application. The first application is used to create some test files that are monitored. The second application now fills up the storage space and monitors at which point the full storage *Intent* is sent and at which time or times the cache files are removed from the system.

Table I contains the devices that we used for our tests and the devices' specifications. We can observe from Figure 3a that all *Intents* are sent out between 90% and 100%. The figure shows for each device defined in Table I the minimum value, the maximum value and the median as a black circle. The *Desire 510* and *Galaxy S4 mini* devices receive the *Intent* inside a large range of the spectrum. In a smaller range the

Galaxy Nexus and also the *Y3* and *P9 lite*, both from *Huawei*, receive there *Intent*. All other devices receive the *Intent* only at 100%, which is much too late to react to the full storage. The *Y3* and *P9 lite* force the user to free up space to be used again, all other devices can be used normally, even with full memory with the only restriction that no content can be saved anymore. Furthermore, some services of the system are stopped like the receiving of emails or SMS because those are services that mostly need free storage space. At some devices only the main function of a phone, the ability to make phone calls, is left. Those values give us important information, namely, that we should not rely on the full storage *Intent* but instead use our *space library* to check the currently available storage space.

The point of time the full storage *Intent* is received by the Android system is not the same as when the caches files are deleted. On some devices, if Android can delete enough cache files to get under the threshold value, the full storage *Intent* is not send. Also, the system does not bring up the hint or notification that it starts deleting some files. Figure 3b shows at which point the cache files are deleted on the test devices. The devices *Galaxy Nexus*, *Y3*, *Nexus 9* and *Pixel 2* have the same value for the point of time at which cache files are deleted for multiple test runs. The other devices' runs lay inside the same 90% to 100% range as in the *Intent* test before. Again, in Android there is no fixed point at which the cache directory is cleaned. Interestingly, no matter which file size, all files in the cache directories are deleted at once. To be sure that no files, even the ones in the cache directories, are deleted our framework application has to check the current storage space and react accordingly. Also, when a new device connection is enabled, it is important that the received files do not exceed the value of the 90% threshold.

V. Conclusion

In this paper we investigate some time and space related issues in Android-based Opportunistic Networks.

First, we address the time related issues which are the reaction on time changes and the time synchronization. We solve the problem of time changes by saving additional information on the system that can be used to our advantage if the time is changed. Also, we bypass the time synchronization issue by creating a relative creation time for our network bundles that depends on the time deltas on the path from sender to destination that are exchanged by the devices in the network.

Second, we address the space related problem of shared buffer space on Android devices. Our *opptain* app is not the only application that uses the buffer on the device. Other applications rely on free buffer space and if they save data in the cache directory it is possible that those data is deleted if our framework application needs too much storage space. Therefore the framework has to observe the available storage space and has to react accordingly if the device runs out of memory. In this case the framework has to drop bundles by applying a drop policy so that no cache data is deleted.

All functionality that was developed to solve these issues are not only interesting for Android-based OppNets but also for other networking applications that have to react on time changes, do time synchronization or have to use buffer space on a mobile device. All principles of this paper can be applied on both Android and iOS devices since these issues can arise on both systems. The *time* and *space libraries* as well as the test applications will be published as open source libraries and applications.

For Future Work, the algorithm for analyzing the speed to fill the buffer can be developed further. For example, several runs at different times can remove uncertainties that can arise when the memory of the device is loaded differently.

References

[1] A. Ippisch and K. Graffi, "Infrastructure mode based opportunistic networks on android devices," in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, 2017, pp. 454–461.

[2] "Distributed Clock Synchronization in Delay Tolerant Networks," *Communications ICC 2010 IEEE International Conference on*, vol. 23, pp. 1–6, 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5502781

[3] D. Zhou and T. H. Lai, "An accurate and scalable clock synchronization protocol for IEEE 802.11-based multihop ad hoc networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 12, pp. 1797–1808, 2007.

[4] Q. Li and D. Rus, "Global clock synchronization in sensor networks," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 214–226, 2006.

[5] "Gradient Clock Synchronization in Wireless Sensor Networks," *Information Processing in Sensor Networks, 2009. IPSN 2009. International Conference on*, pp. 37 – 48, 2009.

[6] "A distributed consensus protocol for clock synchronization in wireless sensor network," *2007 46th IEEE Conference on Decision and Control*, pp. 2289–2294, 2007.

[7] D. Mills, J. Martin, J. Burbank, and W. Kasch, "RFC 5905 - Network Time Protocol," *Internet Engineering Task Force (IETF)*, pp. 1–111, 2010. [Online]. Available: http://www.rfc-editor.org/info/rfc5905

[8] K. Behrendt, K. Fodero *et al.*, "The perfect time: An examination of timesynchronization techniques," in *In DistribuTECH*. Citeseer, 2006.

[9] S. Jain, K. R. Fall, and R. K. Patra, "Routing in a Delay Tolerant Network," in *Proc. of ACM Int. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*.

[10] S. Sati, C. Probst, and K. Graffi, "Analysis of Buffer Management Policies for Opportunistic Networks," in *IEEE ICCCN '16: Proceedings of the International Conference on Computer Communications and Networks*, 2016.

[11] A. Lindgren and K. Phanse, "Evaluation of Queueing Policies and Forwarding Strategies for Routing in Intermittently Connected Networks," in *2006 1st International Conference on Communication Systems Software & Middleware*. IEEE, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/1665196/

[12] K. Shin and S. Kim, "Enhanced buffer management policy that utilises message properties for delay-tolerant networks," *IET Communications*, vol. 5, no. 6, p. 753, 2011.

[13] S. Rashid, Q. Ayub, and A. H. Abdullah, "Reactive Weight Based Buffer Management Policy for DTN Routing Protocols," *Wireless Personal Communications*, vol. 80, no. 3, pp. 993–1010, 2014.

[14] D. Pan, Z. Ruan, N. Zhou, X. Liu, and Z. Song, "A comprehensive-integrated buffer management strategy for opportunistic networks," pp. 1–10, 2013.

[15] A. Krifa, C. Barakat, and T. Spyropoulos, "Message Drop and Scheduling in DTNs: Theory and Practice," *IEEE Transactions on Mobile Computing*, vol. 11, no. 9, pp. 1470–1483, 2012.

[16] ——, "An optimal joint scheduling and drop policy for delay tolerant networks," *2008 IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks, WoWMoM2008*, 2008.

[17] A. Rani, S. Rani, and H. S. Bindra, "Performance Evaluation of Mofo Buffer Management Technique With Different Routing Protocols in Dtn Under Variable Message Buffer Size," pp. 82–86, 2014.

[18] "Android Developers." [Online]. Available: https://developer.android.com/index.html