Smartphone-based Data Collection with Stunner Using Crowdsourcing: Lessons Learnt while Cleaning the Data

Zoltán Szabó*, Vilmos Bilicki*, Árpád Berta†, and Zoltán Richárd Jánki*
*Department of Software Engineering
University of Szeged, Hungary
Email: {szaboz, bilickiv, jankiz}@inf.u-szeged.hu
†MTA-SZTE Research Group on AI
University of Szeged, Hungary
Email: berta@inf.u-szeged.hu

***Abstract*—The increasing popularity of smartphones makes them popular tools for various big data collecting crowdsourcing campaigns, but there are still many open questions about the proper methodology of these campaigns. Beyond this, despite the growing popularity of this type of research, there are familiar difficulties and challenges in handling a wide range of uploads, maintaining the quality of the datasets, cleaning the data sets containing noisy, incorrect data, motivating the participants, and providing support for data collecting regardless of the remoteness of the device. In order to collect information about the Network Address Translation (NAT) related environment of mobile phones, we utilized a crowdsourcing approach. We collected more than 70 million data records from over 100 countries measuring the NAT characteristics of more than 1300 carriers and over 35000 WiFi environments during the three year project. Here, we introduce our data collecting architecture, some of the most prominent problems we have encountered since its launch, some of the solutions and proposed solutions to handle difficulties.**

Keywords–smartphones; data cleaning; crowdsourcing.

I. Motivation

In recent years, smartphones have become part of our everyday lives. Their wide range of uses along with multiple sensors, networking and computational capabilities have also made them seemingly ideal platforms for research. One research area is data collection, with the collected datasets available for a wide area of analysis, including network mapping, discovering and analyzing various networks, and the network coverage of certain areas.

Different research teams from all over the world have discovered these new opportunities, and they employ smartphones as crowdsourcing tools in a wide variety of ways. Through crowdsourcing, they assign tasks to different users with different device types to collect data in real-life situations, or a monitored environment, providing huge amounts of realistic data. In recent years, we have seen a lot of successful, and interesting approaches to this methodology.

However, it is still a question of how exactly crowdsourcing campaigns should be implemented. Several research projects, such as SmartLab [1], the behaviour-based malware detection system Crowdroid [2], and the cross-space public information crowdsensing system FlierMeet [3] recruited a small number of users, who could be trusted, contacted if necessary, and provided the data taken from a known environment, specifically chosen, or created for the crowdsourcing project. This limited the variability and the amount of the data, but the results were of a high quality and easy to validate.

Another approach for recruitment is to upload the smartphone application to the Google Play store, or the Apple App Store, making it available for download by anyone world-wide, and opening up data collecting opportunities for anyone who agreed to the terms and services of the software package. With proper marketing, the results could include enormous datasets obtained from around the world. The NoiseTube project [4] for crowdsourcing noise pollution detection was downloaded by over 500 people from over 400 regions world-wide. The Dialäkt App [5], one of the most well-known crowdsourcing campaigns in recent years, was the most downloaded iPhone app in Switzerland after its launch, with wide media coverage, and over 78000 downloads from 58923 users by the time they had published their results. Many more datasets were collected in the Bredbandskollen project, later to be used by various smartphone-based research projects [6], which has collected network data from 3000 different devices and over 120 million records since its launch in 2007, and the OpenSignal [7] application, which between 2012 and 2013 collected over 220 million data records from more than 530000 devices and from over 200 countries.

However, collecting data using smartphones is not without its difficulties, and there are a number of challenges when smartphones are used as the prime source of information. Among these, battery consumption and network state are among the most important elements, as constantly accessing the state of the phone sensors and listening to specific events takes a heavy toll on the battery, making data collection inadvisable in certain situations (for example, after a device signalled a battery low event), and it is feasible, but pointless in other situations (the phone is on a charger while the user is asleep - the energy is there, but the valuable information is only a fraction of what we would get from an active user). Network state again has to be taken into account, as even today in many environments, we cannot ensure that a device will always have a connection strong enough to send the collected data to the server. Privacy is also an issue, since the data has to remain identifiable yet not contain any trace of personal information.

Aside from all of the above, if data collection was successful, we still have the problem of noisy, incorrect, disorganized data. We have to take into consideration the fact that there are different devices, different versions of the same OS, bugs, such as duplicated records uploaded by the client on network error and damaged records resulting from a similar event. We also have to take user interference into account, who may not wish to provide valuable data (e.g., by deliberately leaving the phone at home on a charger, having it switched off during specific hours, etc.). Their results will still be counted as valuable data, but this can severely distort the collected data set, as well as the results used in evaluations.

An even bigger problem, when crowdsourcing is a global campaign, is that of time synchronization. Not only do we have to find a good solution for the various time zones of the devices, but also the different time codes of the phone collecting and sending the data and the server storing this data, and the possibility that the user might have manually altered the date and time on the test phone as well.

All of the possibilities mentioned above result in a mixed

situation where the power and potential of the smartphones, as research tools cannot be denied, but to acquire correct, useful data is a challenge in itself. This requires careful planning, taking into account almost every possible cause of data distortion, well-defined filters and data cleaning algorithms before any actual research can be performed on the data collected. In this article, we are going to present our solutions with this type of data collection, and our solutions to the problems that emerged.

Our goal was to develop an Android app in order to collect important network information for research on the peer-to-peer (p2p) capabilities of smartphones, including the NAT type, network type and network provider. It does so by taking measurements on a regular basis as well as during specific events. When taking the measurements, the app sends a request to a randomly chosen Session Traversal Utilities for NAT (STUN) Server from a list, displaying useful network information, such as the IP address and NAT type to the user, while also storing the necessary data in an SQLite database, which later gets uploaded to a data collector server for analysis. The application called Stunner has been available for download from the Play Store since December 2013 [8].

Using the collected data, we will be able to define the graph model of a worldwide, peer-to-peer smartphone network. In this model, we aim to test various peer-to-peer protocols to measure the capabilities of a serverless network architecture, where the phones can slowly update their datasets and generate various statistics, without the data ever leaving this smartphone network. The ultimate goal of our research is the creation of an Application Programming Interface (API), through which developers can utilize these peer-to-peer capabilities to create various data collecting and processing applications (for example, general mood or health statistic researching applications for a specific region) without the need of a processing server.

II. Literature Overview

The challenges outlined above have been collected from the research results of other teams (Table I) - and nearly all of them offered good design viewpoints during the development of our own data collecting application.

Perhaps the best overview of the possible difficulties was provided by Earl Oliver [9]. While developing a data collecting application for BlueBerry, he defined five of the most common and serious problems, namely volatile file systems on mobile devices (as file systems can be easily mounted and unmounted on nearly any device), the energy constraints, the intervention of third-party applications running in the background, the non-linear time characteristics of the devices, and malicious user activity (file manipulation, simulated manipulation, etc.)

He solved these by exploiting many trends of BlackBerry users: the general maintenance of high battery levels, retrieving manifests of active applications, and data analysis for patterns of manipulation attacks. However, even he could not define a general solution for every problem, and these problems were not the only ones encountered by other research teams. In fact, they found other challenges to be rather common among data collecting applications.

The researchers at Rice University, while developing LiveLab [10], a methodology used to measure smartphone users with a similar logging technique, encountered the problem of energy constraints, with various optimizations needed to lower the high consumption of the logging application. They also recognized the problem associated with data uploading, namely the connectivity to the server which collects the data from the devices and updates them with new information. They chose rsync for its ability to robustly upload any measurement archive which failed earlier.

A similar method of re-uploading the failed archives was used by a research team at the University of Cambridge in their Device Analyzer project [11], which sought to build a dataset that captured real-world usage of Android smartphones, again with a similar event logging based solution. They found that repeated attempts at uploading caused duplicated data on the server, which could simply be removed by the server before saving it to a database. They also solved the above-mentioned problem of nonlinear time by timestamping every measurement with the device's uptime in milliseconds, recording the wall-clock time of the device when their application started, and later recording every adjustment to it by listening to the notifications caused by these adjustments. From these three elements, a simple server-side processing algorithm was able to reconstruct the exact wall-clock time of any given measurement.

Members of the Italian National Research Council [12] also confirmed these challenges (i.e., the scarcity of resources, difficulties with network monitoring and privacy) while also highlighting two more problems, caused by the participants using the devices - the much more complex control tasks in these types of research projects, and the issue of user motivation to get them carry out the tasks required to get valid data.

TABLE I. Comparison of Data Collecting Projects

Problem	*BlackBerry logger*	*Device Analyzer*	*Portolan*	*Livelab*
Energy constraints	OS callback based logging	Only 2% of the energy consumption	Computational and analyzation processes are run by the server and the collecting is not too energy consuming	The logging events are optimized, some of the data being collected directly from the system logs
Non-linear time	Dates are logged in a UTC timezone; datetime modifications recorded	Every measurement stamped with a device uptime in milliseconds; on startup, the device time is logged, like every modification on device time	Not described (there is a strict communication between client and server, probably kept in sync by this procedure)	Not described, the datetime is most likely to be among the logged data
Offline state, unsuccessful upload	-	Batched uploads only when the device is online and the charger is connected	Uploads are handled by proxy servers	Rsync protocol keeps trying until the upload is successful
Multiplicated data	-	Every device has a file on the server, multiple copies of data being detected by the server	This is solved by proxy servers	Not described most likely to be filtered by the server

In this article, we present our experiences with crowdsourcing-based data collection along with our methods and results of data cleaning on the present dataset.

- We propose a solution for the biggest challenge of the batched data uploads, namely the time synchronization among the different elements of the architecture, utilizing a 3-way logging solution, and lightweight log synchronization.
- We introduce heuristics to analyze incorrect NAT values, in order to decide which cases failed because of server side problems, and which cases originated from the client side.
- We also introduce a data cleaning algorithm to correct timestamp overlaps, using battery-based smartphone heuristics to detect anomalies among consecutive measurements, such as excessively rapid charging, or charging when the smartphone is in a discharging state or when no charger is connected.

III. Our framework

A. Architecture

Our goal with crowdsourcing measurements was to collect information about the network environment of mobile phones. This information is of key importance if one wishes to build a p2p network mobile phones. The crowdsourcing architecture consists of an Android based mobile app, several publicly available STUN servers and a data collecting server. The application collects the information described in Table II.

TABLE II. DiscoveryDTO Object

DiscoveryDTO	
batteryDTO	Energy supply data specified in the BatteryInfoDTO.
wifiDTO	WiFi connection data at the moment of measurement specified in the WifiIntoDTO.
mobileDTO	Mobile network data at the moment of measurement specified in the MobileNetInfoDTO.
publicIP	The public (external) IP address.
localIP	The local (internal) IP address.
timestamp	UNIX timestamp at the end of the measurement.
androidVersion	The version of Android running on the device.
discoveryResultCode	The result of the NAT measurement, defined by the DiscoveryResult enumeration.
connectionMode	The connection code used while taking the measurement is defined by the ConnectionType enumeration.
triggerCode	The event that triggered the measurement is defined by the DiscoveryTriggerEvent enumeration.
appVersion	The version of the application.
timeZoneUTCOffset	The difference between UTC and the device time in signed integer format.

Taking a measurement can be triggered by the events defined in the DiscoveryTriggerEvents enumeration. The event that triggered the measurement lives until the last running test is complete. The enumeration consists of the following items.

- USER: The user started the measurement using the user interface (UI).
- CONNECTION_CHANGED: The broadcast sent by the Android indicated changes in the connection and it triggered the taking of the measurement.
- BATTERY_LOW: The broadcast sent by the Android indicated a low charge level and it triggered the taking of the measurement.
- BATTERY_POWER_CONNECTED: The broadcast sent by the Android indicated a connection to a power supply and it triggered the taking of the measurement.
- BATTERY_POWER_DISCONNECTED: The broadcast sent by the Android indicated a disconnection from a power supply and it triggered the taking of the measurement.
- BATTERY_SCHEDULED: The scheduled battery status control triggered the taking of the measurement, which occurs every 10 minutes.
- BOOT_OR_FIRST_START: Taking the measurement was triggered by the first execution of the application or by the booting of the device.

A measurement starts with an Intent object. The Intent establishes a new DiscoveryService that uses the DiscoveryThreadHandler to start a new thread. The application uses a service implementation running in the background, which may be executed in parallel (with more threads) (Figure 1).

There are two different types of collection, namely online and offline. It is online if there is an active Internet connection on a WiFi or Mobile Network - in this case all the data types mentioned above can be measured and collected. In the case of no Internet connection, there is no guarantee that the network information will be initialized. The schedule in the offline case is set to 30 minutes.

The uploading process occurs when a device is online and it contains at least 10 non-uploaded measurements stored in the local database. After a successful uploading, the records get deleted from the local database in order to avoid duplication. The application did not store any index associated with the measurements, the only field available for this being the timestamp, stored with the discovery data. The storage time is very limited. In fact, after the very first measurement on a given day, the application deletes every record from previous days.

B. Statistics

Our application went live on 20th December, 2013. To promote the usage of the application we also launched a campaign, during which we provided 80 university students and users with smartphones, who agreed to download and provide data with the application for the duration of one year. On 21st March, 2017 the application had been downloaded and installed by 14,727 users on 745 different device types representing 1300+ carriers and 35000+ WiFi networks.

Although we have not released a new version since 5th January, 2016, the average daily installs have remained unchanged in recent months (the most popular phase being in 2015).

Our target API level was originally 19 (Android 4.4), but the application is still being downloaded and installed on more and newer devices, with Android 6.0 being currently the most popular on active installs.

We have also reached a wide variety of different types of devices, upon which the application got installed. The application also successfully reached hundreds of different mobile providers in different countries, which provided us with various, realistic NAT patterns and traces - which will be important later on, after the data cleaning phase is finished, and the analysis and usage of the data collected has commenced.

1) The collected data and the most important descriptive statistics: Based on the size of the dataset collected and our good track record since the 2013 release, it is safe to say that our application and data collecting campaign were both a success (with 70+ million records).

The chart (Figure 2) shows the number of uploaded records per device. The majority of users did not provide any measurements, but the decline of the slope lessened, indicating that the users who provided data were more likely to stay and keep providing data.

During the summer of 2015 we had to reassign some resources to other projects, resulting in an absence of data in the given time period. However, after restarting the server, our input declined only slightly, resulting in a steady amount of data arriving to this day despite the gap of a few months (Figure 3).

An interesting aspect is that although the daily uploads have been pretty steady since the hiatus, the number of active devices providing the uploads has been on a steady decline since early 2016. We hope that with our current developments, this decline can be reversed, and a new record in both the number of active devices and daily uploads obtained (Figure 4).

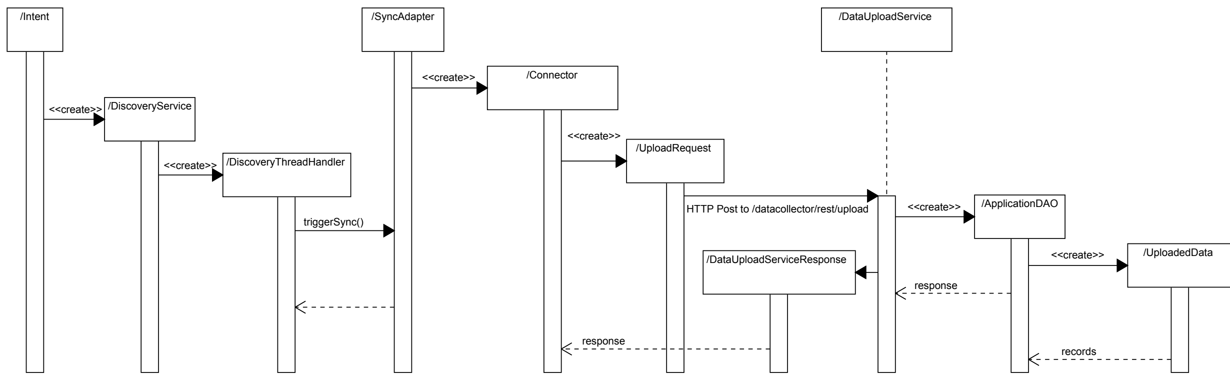


Figure 1. Upload process

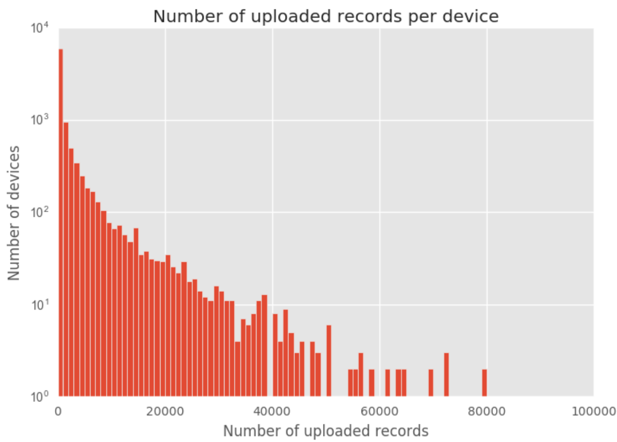


Figure 2. The plot shows the uploaded data per device

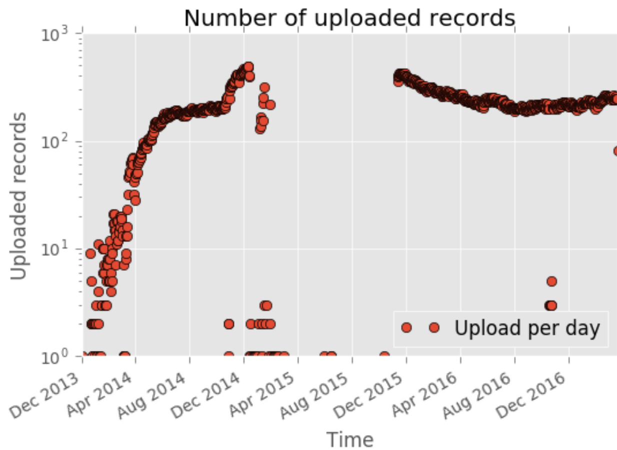


Figure 3. The plot show the uploaded data per day during the whole measurement period.

Following the hiatus, the first spike above shows all the collected data uploaded to the server at the same time. While the Wifi based tests closely follow the trends of active devices and daily uploads, the Mobile Operator-based measurements have been taken at a relatively low, but steady rate (Figure 5). The plot indicates that we can monitor more than 200 mobile networks and over 500 WiFi environments day after day. Here we have shown that a significant amount of data has been collected over the three year period. The real value of the data depends on the quality of the timestamps. Now, we will describe our findings in the area of data cleaning.

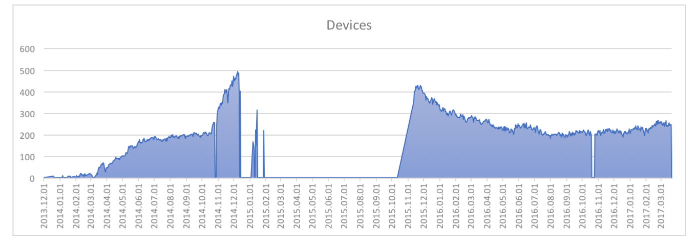


Figure 4. The plot shows the number of active devices per day.

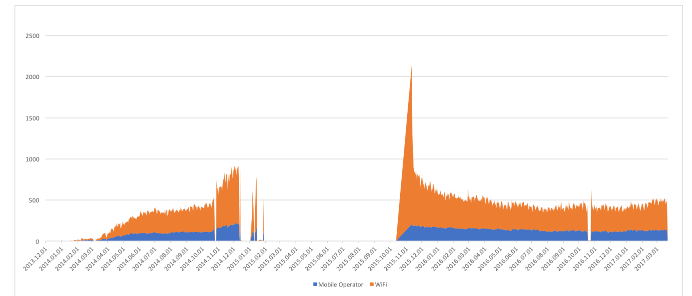


Figure 5. The measurements based on Mobile Operator and WiFi.

IV. Issues with Collected Data

A. Data Duplication

In spite of the theoretically sound software environment where the server-side logic was implemented in JEE with transactional integrity taken into account, it turned out that a significant proportional percentage of the dataset had been duplicated. We applied simple heuristics in order to filter out the duplicate measurement records by comparing only the client-side content and skipping the server-side timestamp and other added information. In practice, we utilized the Python Pandas framework duplicate filtering method shown in Figure 6, to remove the duplicates.

We found that out of the 70+ million rows only 30+ million rows were unique, while the remaining part were duplicates. We investigated the possible root cause of this phenomenon. Figure 7 shows the total submitted records per device versus the duplicated records per device. It clearly shows that there is a linear relationship between the two values. This is an evidence that this is a system-level symptom and not a temporal one related to the server overloading. The same is true if we check the temporal dimension of the duplicated

```
PSEUDO CODE:
data['duplicated'] = = data.duplicated(subset=[all client side columns], keep='first')
```

Figure 6. JSON sample

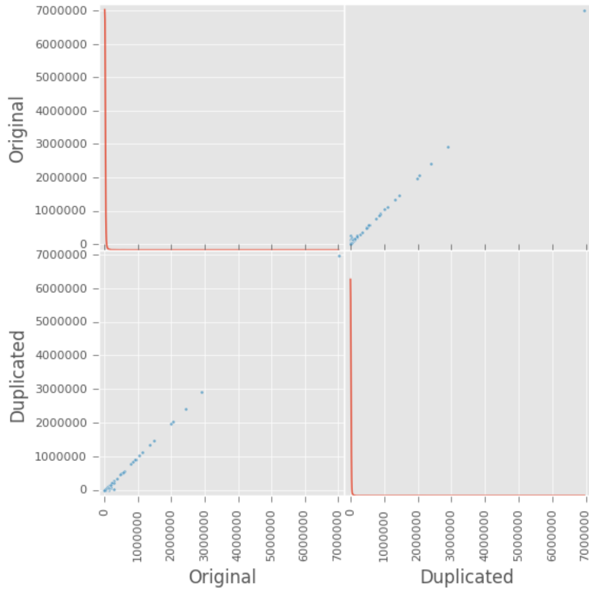


Figure 7. The plot shows the duplicated data per device vs the total number of records submitted by that device

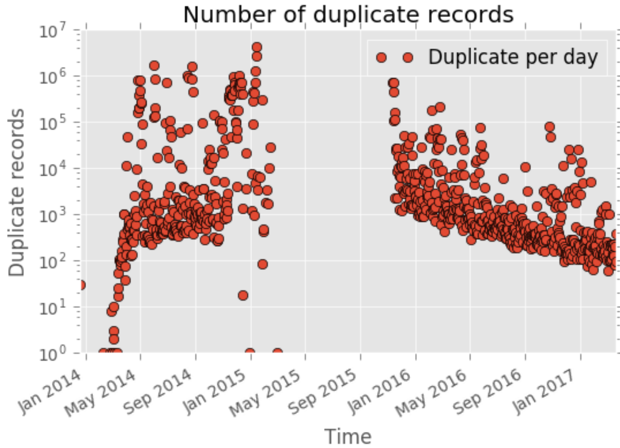


Figure 8. The plot concerning the duplicated data per day (server side)

records during the given period (Figure 8).

After an in-depth investigation of the client code, we found that the default HttpClient configuration contained a very robust upload model, with a default value of 3 retries for every HyperText Transfer Protocol (http) operation, if it failed with a timeout. This is a very useful method for simple data upload, but in our case, if the timeout chosen in the settings was too short, the client might have uploaded the same batch of records up to four times to the server, which would acknowledge and store all of them. In order to stop further multiple uploads, we will need to carefully look at the correct timeout and retry values and also identify the upload batches, so the server will be able to detect the retries on upload.

1) Overlap of the client-side timestamps: The actual unreliability of the client-side timestamps was a surprise for us. Figure 9 shows the difference between the Android timestamp and the date captured on the server side. A significant number of measurements have big differences between these two dates. The difference between the two timestamps is only an indication that there could be an error in the measurements as a week or weeks may pass by after capturing and uploading

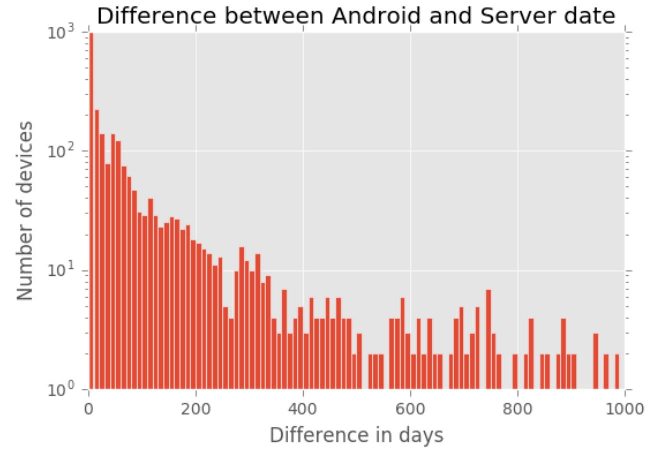


Figure 9. Difference between the Android and Server date

TABLE III. HEURISTICS FOR DETECTION

Name	*Description*	*Detection capability*
Fast change detector (ABC and SBC) (the first letter codes the ordering applied: A - android, S - server side, this coding being consistent among the different detectors)	We used the battery percentage and its sluggish behaviour to detect the fast changes. We defined the speed of change as the ratio of the two consecutive timestamps and the battery percentage difference between these two timestamps. We defined a threshold high enough to be able to recognize the measurement as an error.	For time-reset starting date estimation.
Rules based on charging and plugged state	This method focuses on the rules defined without time being included. Rules: Charging (more than 20% change) while not on charger (ACEU-SCEU) Charging (more than 20% change) while in discharging state (ACED-SCED) Big changes between consecutive elements (charging 20%, discharging 6%) (AP-SP) Charging (more than 20% change) while not on the charger and the phone is in a discharging state (ASC-SSC)	These methods could be applied in order to detect the beginning of a new measurement period (among the overlapped timestamps)

the data to the server in the case of missing or inadequate network conditions.

We started to examine the nature of the Android timestamp. First, we noticed records with timestamps that were significantly earlier (e.g., 01.01.1970) or later (01.01.2023) than our other measurements. Finding invalid time periods was trivial (like 2023), but it transpired during our in-depth investigation that several phones were reset to a valid date that lay within the observation period. In order to be able to properly detect this anomaly, we elaborated several simple heuristics for detection, these being shown in Table III.

We applied the anomaly detection heuristics mentioned above in order to compare two basic sorting approaches; namely, sorting by the server-side information (e.g., serial number) and the sorting based on the mobile timestamp. We observe that for about 6-7 thousand devices the number of errors is zero. So about 1/8 of the total devices are affected by the time overlap. Figure 10 shows the results of the fast change detector applied for the two ordering approach (it was run on a filtered dataset, skipping the valid data). The green line (server-side sorting) indicates fewer fast change errors in most cases (it was able to eliminate this error on about 40% of the affected devices). The scatterplot below (Figure 11) also shows a clear correlation between the two sorting approaches and the number of fast change errors. The slope of the correlation line (and the points under the line) tells us that the server side sorting was able to reduce the fast change errors in most cases. Based on these findings, one simple approach for time overlap fixing might be the hybrid sorting approach where a given number of records are located after a fast change error had been sorted

Figure 10. Fast change errors

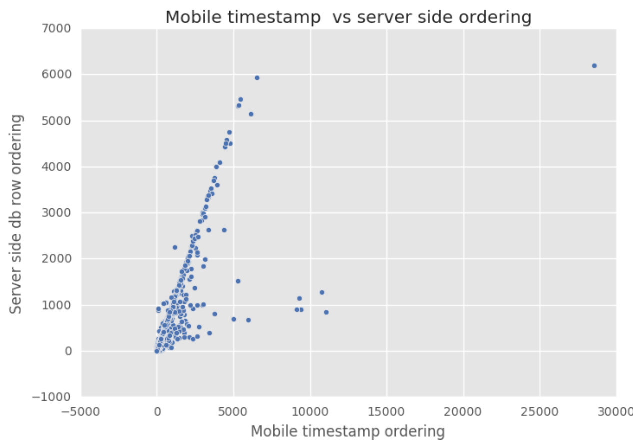


Figure 11. Mobile timestamp and server-side ordering

based on the server-side sorting.

The effectiveness of the simple server-side sorting is also shown in the Figure 12 concerning the correlation between different error detection heuristics and the sorting methods. It is apparent that server-side sorting can significantly decrease the error level for all error detectors (when comparing the same method with S and A sorting, most of the points are below or above the similarity line).

With the previously described heuristics, we were able to demonstrate that the server-side sort order can reduce the rows suspected of being in the wrong position to about 1/10 of the total dataset. A further decrease in the suspected errors could be achieved with a richer ruleset that incorporated different mathematical models for batteries. For our purposes, the current reliability level of the causality dimension of the data set is quite sufficient.

B. NAT discovery result code corrections

The main feature of our application is the discovery of the NAT type. Users can ask the application about their NAT information and public IP address. This method is based on User Datagram Protocol (UDP) message-based communication between the device and a randomly picked STUN server. A STUN server can discover the public IP address and the type of NAT that the clients are behind.

We were faced with a problem that was caused by the prefixed STUN server list. It contains a list of 12 reliable

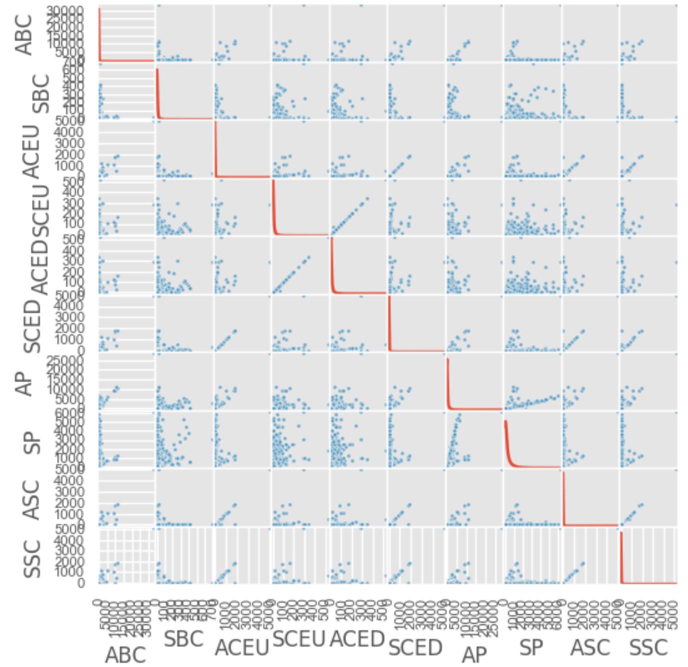


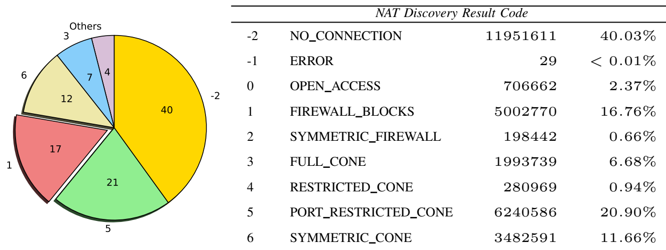
Figure 12. Error detectors

servers that are suitable for NAT detection, this list being embedded inside the application code. It allows the device to randomly pick a STUN server. As a result, every measured NAT type in the timeline is based on a different STUN server's NAT test. Hence it makes the measured data more trustworthy. This random pick approach has been well designed and worked very well initially. However, after a time four of the STUN servers went offline without any prior notice. Since then this four failed STUN server provide the same NAT discovery result code as firewall blocked connections. As a result of this error, some uncertainty exists in the NAT discovery result code. Therefore we propose a solution on how to correct it and make the collected data useable afterward. Quite significantly, another solution is needed to avoid connections to a failed STUN server.

Now we need to discuss the obscure NAT discovery result code. This is the 16.76% of the total measurement records. We have carried out this examination over the dataset, which has already been prefiltered and processed, the order being based on the approach defined above (Figure 13).

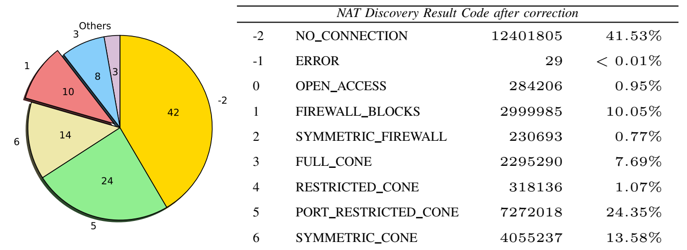
Firstly we need to discuss the FIREWALL_BLOCKS result code. This code is corresponding to NAT tests that has open communication channel but never get response from the STUN server. In normal case it means that firewall blocks the connection. Unfortunately records also has no response from failed server. Even though a part of those records may have online NAT type. Therefore these records are uncertain and further examination is needed.

Below, we present a method for filter the STUN server errors from FIREWALL_BLOCKS discovery result code. These set of records contains uncertain potential online states. The server fails with a 4/12 probability, and the event of consecutive repeated fails has an exponential pattern. Consequently we define sessions with consecutive repeated FIREWALL_BLOCKS discovery result codes and look at their distribution. If the distribution is roughly an exponential distribution, then we can interpret them as online and we can define their network properties. Otherwise, the others that do



NAT Discovery Result Code			
-2	NO_CONNECTION	11951611	40.03%
-1	ERROR	29	< 0.01%
0	OPEN_ACCESS	706662	2.37%
1	FIREWALL_BLOCKS	5002770	16.76%
2	SYMMETRIC_FIREWALL	198442	0.66%
3	FULL_CONE	1993739	6.68%
4	RESTRICTED_CONE	280969	0.94%
5	PORT_RESTRICTED_CONE	6240586	20.90%
6	SYMMETRIC_CONE	3482591	11.66%

Figure 13. Discovery Result Code



NAT Discovery Result Code after correction			
-2	NO_CONNECTION	12401805	41.53%
-1	ERROR	29	< 0.01%
0	OPEN_ACCESS	284206	0.95%
1	FIREWALL_BLOCKS	2999985	10.05%
2	SYMMETRIC_FIREWALL	230693	0.77%
3	FULL_CONE	2295290	7.69%
4	RESTRICTED_CONE	318136	1.07%
5	PORT_RESTRICTED_CONE	7272018	24.35%
6	SYMMETRIC_CONE	4055237	13.58%

Figure 14. NAT Discovery Result Code after correction

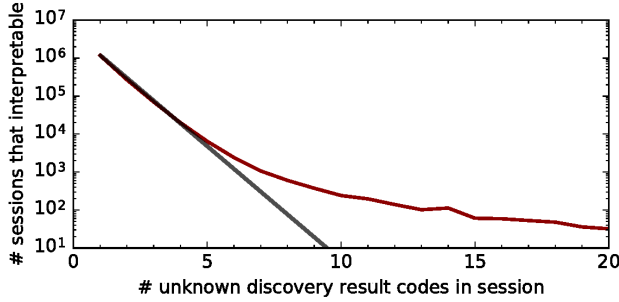


Figure 15. Discovery result code enclosed by sessions

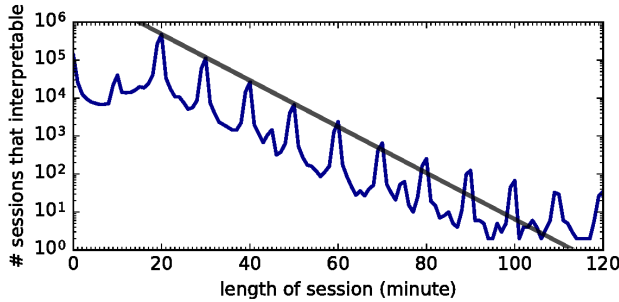


Figure 16. Length of candidate sessions

not have an exponential fit will remain FIREWALL_BLOCKS. This means that in this way we cannot prove the opposite (firewall blocks the connection). In general, we are looking for a session that begins and ends with the same network property and there are only uncertain online states between them. These sessions may be interpretable based on the begin-end enclosures. More specifically, the sessions must

- begin and end with the same NAT discovery result code
- begin and end with the same Service Set Identifier (SSID) in the case of a wifi connection
- begin and end with the same mobile operator in the case of a mobile data connection
- contain only uncertain online states
- contain a time gap between two records only in a range of 0 to 15 minutes based on the fact that the maximum time gap between two regular online records is almost 10 minutes. However, it is not very accurate because of the Android support scheduler with its inexact trigger time requirements.
- not be interrupted by trigger events that correspond to any potential change in network properties.

We show the above-defined candidate sessions in Figure 15 and Figure 16. Let us first take a look at how many uncertain discovery result codes are enclosed by these sessions in Figure 15. It is clear that the first four point seem to fit an exponential curve. Consequently, it is still open to interpretation and the rest of the points remain undefined. Next, Figure 16 shows length of the above-defined sessions. There are some peaks around every 10 minutes. These peaks correspond to the BATTERY_SCHEDULED trigger event, which is scheduled every 10 minutes and this is the most common trigger event. For example, if there is exactly one uncertain FIREWALL_BLOCKS value in the appropriate session and every taking of a measurement is triggered by this schedule event, then its length of time is around 20 minutes. Based on this example, an above-defined session that contains three unknown records lasts for 50 minutes. Accordingly, we examined the points from the first phase up to 50 minutes. Our examination revealed that it also had an exponential pattern. In contrast to the distribution in Figure 15, this distribution appears more complex, but it is still acceptable. Next we associate the two findings. More specifically, the intersection of the two sets is an above-defined session that contains less than five uncertain elements and it lasts no longer than 50 minutes. Based on this rule we can correct the network properties of 6.7% measurement records.

Next, we should mention some further minor errors associated with data collection. In a very few cases there was no network connection, but it still has some errors in the discovery result code (mainly code 0). We simply correct all of them to the no connection state (-2).

Now let us have a look at the final results of the NAT data correction in Figure 14 and Table ?. Records with FIREWALL_BLOCKS code are reduced to 10%, and the records with online state are expanded.

V. Lessons Learnt

Based on our findings, some of the challenges encountered proved to be quite trivial, and required only some small modifications to the algorithm, while others still have to be tested with our proposed solutions.

On the client side, we have found several elements where the default approach of Android development proved insufficient, and special consideration was needed for proper data collection. We found that the timeout value of the Android application should be increased in proportion to the connectivity quality with the data collector server, while the number of retries should be reconsidered and perhaps revised with upload batches accompanied by identifiers to make duplicate detection easier.

The detection of the NAT anomalies was made significantly easier through the NetworkInfo and WifiInfo objects of the Android system. When collecting network data, we found it highly advisable to include as many attributes from these rich objects as possible - such as SSID, whether the phone is in the

roaming mode and whether the network is connection metered -, since any of these could explain possible anomalies in the dataset. For example, the phone might be connected to a wifi network, but the router is not necessarily connected to the Internet; or, if it is located at a public establishment, it may redirect the requests to the establishment's login site instead of the original destination - all of which are serious problems, and they could go unnoticed without detailed information about the network.

Regarding the NAT problem, it is also advisable to reconsider storing the list of external servers in a constant array (a practice which is very common based on our experiences), because if one of those servers goes offline, it might generate huge amounts of incorrect data. A proxy which stores the server list, keeps it updated by using regular checkups, and forwards the list to the phones on request, would be a better solution here.

Also, while the deletion of previous data is a good practice to stop the application from taking up too much storage space, the 24 hour limit might be too short, since important events could get lost in that time period. The time limit for storage before deletion should be featured among the settings. Even after a delete, it is necessary to leave some trace of the deleted data - at least a log -, so the anomalies in the later, successful uploads could be interpreted.

The timestamp desynchronization between the server and the client remains perhaps the most challenging problem, with the battery based sortings providing some improvements in the dataset. One solution might be a lightweight log timestamping. In this case only a hash of the log would be sent to the server frequently (in order to minimize the mobile traffic and preserve the battery), where a reliable timestamp would be attached on the server side to this hash and saved in a permanent storage. In this way, we may define reliable milestones which are independent of the mobile side timestamps. On the mobile side, it is important to preserve the total order of the events. This could be achieved by using a simple increasing indexing procedure in the SQLite database.

We mentioned that even NAT types may be misleading, despite the quality of the connection. Once again, some of these incorrect values could be corrected by simply checking the actual state of connectivity during the upload. The NAT type in the remaining records is mostly corrected by a pattern recognition method. Hopefully, this problem may never occur again after the proposed changes have been made to handle a dynamic STUN server list.

VI. Conclusions

As the reader can no doubt see that our approach worked well in the above-mentioned areas of data cleaning. Since the application was launched in 2013, it has been downloaded by more than 14.000 users from over 1300 different carriers and 35000 different WiFi areas, to hundreds of different device types, which is providing enormous amounts of valuable data for the analysis of NAT traces, patterns, and later on, for the simulation of the above attributes.

Compared to other crowdsourcing projects, our crowdsourcing approach was a hybrid methodology, where we provided a certain number of users with smartphones, and released the app to the Play Store for wider availability, and took more data measurements from different parts of the world. We did not reach the volume of OpenSignal or Bredbandskollen with their 100-200+ million datasets, but this hybrid solution still provided us with a much bigger amount of valuable data than a closely monitored environment like FlierMeet or SignalLab that had roughly 40 devices, a shorter collection time period, and operated in a restricted environment like a university campus or a development environment.

Lastly, though we have encountered some of the most challenging problems of the smartphone-based data-collection, our data cleaning approach successfully handled the incorrect, distorted data, and turned the dataset into a clean, organized state, which is ready for further use and processing. From this clean data, we are ready to start the graph modeling of the worldwide peer-to-peer smartphone network. In this simulated network we will be able to test the dynamics, attributes and capabilities of smartphones with real-life measurements, and later, the results of concrete peer-to-peer protocol applications. Our next major step will be the development and in-depth testing of peer-to-peer algorithms, the determination of the speed and stability of smaller applications running in this simulated environment.

Acknowledgment

This research was supported by the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 ("Internet of Living Things").

References

[1] G. Chatzimilioudis, A. Konstantinidis, C. Laoudias, and D. Zeinalipour-Yazti, "Crowdsourcing with smartphones," IEEE Internet Computing, vol. 16, no. 5, 2012, pp. 36–44.

[2] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, October 2011, pp. 15–26, aCM.

[3] B. Guo and et al., "FlierMeet: a mobile crowdsensing system for cross-space public information reposting, tagging, and sharing," IEEE Transactions on Mobile Computing, vol. 14, no. 10, 2015, pp. 2020–2033.

[4] M. Stevens and E. D'Hondt, "Crowdsourcing of Pollution Data using Smartphones," in Workshop on Ubiquitous Crowdsourcing, held at Ubicomp '10, September 2010, pp. 1–4.

[5] A. Leemann, M. J. Kolly, R. Purves, D. Britain, and E. Glaser, "Crowdsourcing language change with smartphone applications," PloS one, vol. 11, no. 1, 2016, pp. 1–25, e0143060.

[6] T. Linder, P. Persson, A. Forsberg, J. Danielsson, and N. Carlsson, "On using crowd-sourced network measurements for performance prediction," in In Wireless On-demand Network Systems and Services (WONS), 2016 12th Annual Conference on. IEEE, January 2016, pp. 1–8.

[7] A. Overeem and et al., "Crowdsourcing urban air temperatures from smartphone battery temperatures," Geophysical Research Letters, vol. 40, no. 15, 2013, pp. 4081–4085.

[8] Á. Berta, V. Bilicki, and M. Jelasity, "Defining and understanding smartphone churn over the internet: a measurement study. In Peer-to-Peer Computing (P2P)," in 14-th IEEE International Conference on. IEEE, September 2014, pp. 1–5.

[9] E. Oliver, "The challenges in large-scale smartphone user studies," In Proceedings of the 2nd ACM International Workshop on Hot Topics in Planet-scale Measurement, June 2010, p. 5, aCM.

[10] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "LiveLab: measuring wireless networks and smartphone users in the field," ACM SIGMETRICS Performance Evaluation Review, vol. 38, no. 3, 2011, pp. 15–20.

[11] D. T. Wagner, A. Rice, and A. R. Beresford, "Device Analyzer: Large-scale mobile data collection," ACM SIGMETRICS Performance Evaluation Review, vol. 41, no. 4, 2014, pp. 53–56.

[12] A. Faggiani, E. Gregori, L. Lenzini, V. Luconi, and A. Vecchio, "Smartphone-based crowdsourcing for network monitoring: Opportunities, challenges, and a case study," IEEE Communications Magazine, vol. 52, no. 1, 2014, pp. 106–113.