Pervasive and Mobile Computing 81 (2022) 101550

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

Using mobile devices as scientific measurement instruments: Reliable android task scheduling

Alberto González-Pérez [*], Miguel Matey-Sanz, Carlos Granell, Sven Casteleyn

GEOTEC Research Group, Institute of New Imaging Technologies, Universitat Jaume I, Castellon, 12071, Spain

A R T I C L E I N F O

Article history:
Received 1 July 2021
Received in revised form 18 January 2022
Accepted 21 January 2022
Available online 29 January 2022

MSC:
6802
6804

Keywords:
Smartphone
Android
Task scheduling
Scientific measurements
Reliability

A B S T R A C T

In various usage scenarios, smartphones are used as measuring instruments to systematically and unobtrusively collect data measurements (e.g., sensor data, user activity, phone usage data). Unfortunately, in the race towards extending battery life and improving privacy, mobile phone manufacturers are gradually restricting developers in (frequently) scheduling background (sensing) tasks and impede the exact scheduling of their execution time (i.e., Android's "best effort" approach). This evolution hampers successful deployment of smartphones in sensing applications in scientific contexts, with unreliable and incomplete sampling rates frequently reported in literature. In this article, we discuss the ins and outs of Android's background tasks scheduling mechanism, and formulate guidelines for developers to successfully implement reliable task scheduling. Implementing these guidelines, we present a software library, agnostic from the underlying Android scheduling mechanisms and restrictions, that allows Android developers to reliably schedule tasks with a maximum sampling rate of one minute. Our evaluation demonstrates the use and versatility of our task scheduler, and experimentally confirms its reliability and acceptable energy usage.

1. Introduction

Today, mobile phones are sensor-packed devices, capable of capturing valuable real-time information about the users and their environment. The ubiquity of smartphones and their prevalence in broader society made them the measurement instrument of choice for a variety of use cases, ranging from participatory sensing applications, where crowd-gathered sensing data provides insights in environmental, social and urban phenomena [1], to highly personalized applications, where users and their environment are actively and passively monitored to provide value-added services, such as indoor positioning [2], mobility [3] or (mental) health services [4,5]. Particularly in mobile health applications, unobtrusive sensor data collection brings early detection and continuous monitoring for various health conditions, such as Parkinson, tremor, lack of physical activity and fall detection [6]. Smartphones are also widely recognized as the next wave in mental health treatment, capable of detecting, monitoring and treating mental health disorders [7,8]. Hereby, so-called "ecological momentary assessments" [9] and "ecological momentary interventions" [10] rely on prolonged data collection for the assessment of psychological symptomatology and corresponding real-time, in-situ interventions. In these health scenarios, where the smartphone is essentially used as a scientific measurement instrument, the accuracy of the derived health

[*] Corresponding author.
E-mail addresses: alberto.gonzalez@uji.es (A. González-Pérez), matey@uji.es (M. Matey-Sanz), carlos.granell@uji.es (C. Granell), sven.casteleyn@uji.es (S. Casteleyn).

https://doi.org/10.1016/j.pmcj.2022.101550

conclusions and provided treatments hinges on the capability of the mobile device to systematically and reliably capture mobile sensor data.

Despite the broad range of passive sensing applications described in literature and the essence of accurate data collection for mobile health applications, there are important yet under-reported technical issues with using mobile phones as scientific sensing devices: they are notoriously difficult to program and generally suffer from unreliability in terms of frequently capturing sensor data over prolonged time periods. Even though the reliability of mobile phones as sensing devices has been recognized as a critical issue early on [11], researchers continue to struggle with unforeseen and unexplained loss of scheduled sensor data capture. Recent studies in mobile health report up to 50% of unexplained data loss [12,13], and, not being the main focus of the study, researchers accept the unreliability [12] and/or only generally hint to possible technical issues, such as operating system and device model [13]. Far from questioning the scientific validity of such studies, we here only highlight the problem and the general lack of insight in its causes. Recent reliability studies on data collection with mobile platforms shed some light on the problem. In [14], the authors underline the problem of passively gathering sensing data in a reliable and continuous way. Without going deeper in the loss of data, the authors point to the source of problem: "in most cases, they [gaps in collected data] result from the restrictions of the operating system". Indeed, increasingly harsh energy-consumption and privacy policies enforced in mobile operating system's background services, significantly and artificially constrain the re-purposing of smartphones as passive sensor nodes [15]. Additionally, further restrictions and compatibility issues over different device manufacturers [16] are complicating cross-device reliability, and effectively impede mobile devices to be full-fledged scientific measurement instruments. Developer-driven initiatives such as "Don't Kill My App"[1] or "AutoStarter"[2] illustrate the extent of the issue. The recent study of Bärh et al. [16] evaluates the missingness of passively scheduled geolocation sensor data (with a 30 min sampling rate) in Android-based smartphones based on analysis of results of a previously performed social science study gathered from 625 participants. Results show that over 50% of scheduled data measurements are missing due various reasons, including a majority attributed to device or app specific issues, such as OS and manufacturer-related restrictive policies. Interestingly, statistical analysis shows that the likeliness of missing observations increases under specific power-saving conditions (display turned off) and restrictions (Doze standby mode), and with more recent Android versions.

Smartphones – more concretely Android phones – provide several mechanisms to perform tasks while the user is not actively using an app. On one hand, there are services, which are meant to execute arbitrary code without any execution time constraint. Services can, ideally, run forever (although real experiments prove this statement wrong [15]) or for a finite amount of time (i.e., until the task to be executed has finished). Starting from Android 8.0+, a new kind of service was introduced, the foreground service, created to provide visual feedback to the user (i.e., via a permanent notification) when an app is performing an operation which potentially can hurtle user's privacy (e.g., acquire the location of the phone). On the other hand, there are alarms, which are meant to delay the execution of any code. Alarm handlers (i.e., broadcast receivers) limit the execution time to a maximum of 10 s, notwithstanding, they can be combined with services to delay the execution of lengthy tasks. Each mechanism has its advantages and pitfalls.

In this article, we experimentally demonstrate the reported problem of unexpected loss of measurements in scientific longitudinal data collection tasks (i.e., missing scheduled task executions), and we discuss the technical (programming and engineering) challenges to reliably schedule prolonged and systematic (background) task executions, as for the collection and processing of passive sensor measurements on mobile (Android) phones. Then, we formulate guidelines for mobile phone developers to ensure reliable background (sensing) task scheduling in various usage scenarios. Finally, we present a novel software library to reliably schedule (sensing) tasks at various sampling rates, independent of the underlying low level operating system mechanisms. In doing so, we answer Regli's plea for computer scientists as toolsmiths [17] by providing a software tool, grounded in ingenious engineering and algorithmics, that addresses an obstacle for conducting solid science in a wide range of application domains that aspire the use of mobile phones as ubiquitous, reliable scientific instruments. Hereby, we focus the design and evaluation of our work on two key concepts: reliability of task scheduling (i.e., the ability to correctly execute scheduled tasks – i.e. sensor measurements – in a systematic manner over a prolonged time period, in terms of completeness and accuracy) and efficiency (i.e., assuring an acceptable energy consumption of the smartphone's battery).

To the best of our knowledge, such guidelines or implemented solutions for flexible and reliable background task scheduling on (Android) phones (e.g., sensor measurements) are currently not available in the literature. There exist specific data collection solutions, of which the AWARE [18] and Beiwe [19] frameworks are representative examples, or the CARP Mobile Sensing framework [20] for sensing and data handling tasks in Mobile Health. Yet, all of them come with a predetermined set of tasks (mainly data collection tasks), which in few cases can be extended or reduced (i.e., optimized). Our solution could be used to implement (the Android version of) any of them, leading to more reliable operation, as for all these solutions, online sources (i.e., bug reports, code repository comments, forums) report systematic loss of measurements in longitudinal data collection (background) tasks. Respective authors report to be working on ways to mitigate this problem, some partly overlapping with our proposal. However, the most popular workaround is the use of a continuously running foreground service, combined with alarms to restart it when killed. We discuss the disadvantages of such an approach (most importantly, battery drain) in Section 2.

Our software solution is released as open source [21] and can be used in Android-based applications requiring reliable and systematic scheduling of background tasks, such as sensor measurements.

1 https://dontkillmyapp.com/.
2 https://github.com/judemanutd/AutoStarter.

2. Scheduling tasks in a timely manner: pitfalls and guidelines

The development guidelines introduced in this section are the result of 3+ years of experience with SyMptOMS, an ecological momentary intervention framework to provide smartphone-based psychological treatment for a range of mental health disorders based on real-time patient monitoring [22]. Under the umbrella of SyMptOMS, we developed an Android-based smartphone application which passively collects sensor data at various frequencies, with a highest collection rate of 1 min. Throughout its lifetime, the SyMptOMS mobile app suffered from increased restrictions in four consecutive Android versions, provoking continuous updates and rigorous testing to maintain operation.

Our solution works on a heterogeneous set of phones, from a variety of manufacturers, even in those running the latest and most restrictive Android versions (tested up to Android 11 — API 30), while keeping support for old phones (Android 4.4 — API 19 and onwards). It uses an alarm-based mechanism to schedule data collection tasks over time, produces a percentage of missing data considerably lower than reported in literature (which is about 50% [12,13,16]) and has proven to be successful in different case studies [23,24].

In the next subsections, we share our experience in reliable task scheduling in Android. First, we review the different Android mechanisms for scheduling tasks, comparing them and explaining known workarounds for common pitfalls. We then formulate development guidelines to reliably schedule tasks, followed by the necessary mobile phone configurations to ensure Android's energy-savings mechanisms do not interfere.

2.1. Android task scheduling: the ins and outs

In essence, Android supports three ways to run tasks while the user interface is not visible: using a foreground service, a background service or alarms. A foreground service performs an action, possibly long-lived, that is noticeable by the user (e.g., playing music). Foreground services have the advantage that they act as if the application user interface is visible, which means that they do not suffer from certain data access restrictions as background services do (e.g., ability to get frequent access to sensitive data, i.e., GNSS locations or WiFi fingerprints), but on the downside, they keep the app continuously running (causing battery drain) and display a permanent sticky notification on the phone's task bar, damaging the user experience and possibly causing more app uninstalls. On the other hand, a background service performs an action which is not noticeable by the user (e.g., a backup process, a sensor measurement). It has the advantage of unobtrusive monitoring, yet on the downside of lacking access to up-to-date values of what the OS considers sensitive data. In either case, both can get killed at any time by the OS if the device is running low on RAM [15].

Finally, alarms allow the developer to schedule time- and interval-based tasks, even outside the lifetime of the associated app. Alarms are more energy efficient, as no continuous foreground or background service needs to be running, but they suffer the main disadvantage that alarms do not trigger when the device is idle (i.e., in Doze mode). Alarms are early-available low-level mechanisms that, in successive Android versions, have gradually been made available to application developers through higher-level abstractions (e.g., Job Scheduler and WorkManager APIs). From Android version 6.0 onwards, battery life has been prioritized over reliability, by penalizing alarms below 15 min and not guaranteeing timely triggering of alarms, to the point where the provided high-level abstractions no longer allow scheduling them.

Under the increasingly restrictive policies in later Android versions, high frequency and time accurate alarms (prevented by Doze mode in Android 6.0+), jointly with silent sensitive data access in background (prevented by the limitations introduced in Android 8.0+) did not work anymore. A good workaround for doing background work began to gain traction, the use of always-running background (or foreground) services. However, not without some problems [15,25]: a continuously running foreground service, with the aforementioned downsides of battery drain, decreased user experience and caused unexpected shutdowns.

Unless real-time or close-to-real-time data or computations (<1-min interval) are needed, our suggestion is to use alarms instead of background (or foreground) services all the time. The key advantage of alarms is that they allow the device to sleep after the task is finished (battery gain). Furthermore, they can be combined with foreground services to overcome certain system limitations (frequent geolocation updates can only be reliably collected in foreground). Possibly, an experienced reader will argue that alarms cannot be reliably scheduled every minute over a prolonged time period, mainly after Android 6.0, when Doze mode was introduced. However, our experience shows that – with special care – alarms can be reliable.

First of all, attention needs to be paid with the periodicity of the alarm-scheduled task. By design, Android will not let an app trigger more than one alarm per minute (or will penalize the app that does so too often). This restricts the solution to a minimum time interval of 1 min,[3] and also causes problems when multiple tasks need to be run in short time intervals. However, this problem can be solved by grouping tasks by their proximity in time, using a single alarm that runs in a <1 min time span.

Secondly, alarm-scheduled tasks should not exceed their periodic frequency (e.g., for tasks scheduled every 60 s, their workload should not exceed 60 s), and as a good practice we recommend using timeouts for app tasks. If the task is

[3] Even though, within this one minute, multiple sensor readings can be scheduled — as long as the overall task time does not overlap with a future alarm execution.

longer – i.e., another alarm will fire before it finishes – the developer should try to split it into simpler tasks instead. From Android 8.0 and onwards, users will get a notification if an app is doing intensive work in background for a longer time period, and since Android 10, the system classifies (and penalizes) apps based on their battery usage in background. Hence, continuous background work should be avoided, and it is recommended to let the phone "breathe" from time to time when running intensive background tasks. Even more, the use of timeouts prevents faulty tasks from running unlimitedly in background, which will eventually lead to Android blacklisting the app.

Thirdly, the developer needs to keep in mind that alarms will not persist after a system reboot. Therefore, an app should keep track of the alarms that have been scheduled and those that have been triggered, in order to reschedule tasks that were not triggered due to a system boot.

Fourthly, the developer needs to be aware that Android – for reasons we have not been able to determine – sometimes stops triggering alarms of an app. This does not happen often, but we anyway recommend to implement some counter-measurements against this behavior. The developer can for example perform an alarm status check (i.e., verify the status of the alarm's pending intent) every time the app is launched by the user, as to re-set up alarms if needed, combined with an alarm watchdog, checking if alarms are still scheduled, over a longer time period. In our experience, a repeating alarm outside the battery savings danger area (every 15 min) – to avoid the watchdog itself to be penalized or canceled by Android – does the job, allowing relatively fast recovery in case of failure.

Finally, we advise to implement any optimization which avoids running a task unnecessarily. For example, in the common case of collecting geolocation measurements, it is not needed to continue collecting samples every minute while being stationary for a long time (e.g., the user is not carrying his/her phone, sitting down, sleeping). In such a case, an optimization could be to use Android's activity recognition API, which is highly energy-efficient and can notify activity changes, in order to stop/resume alarm scheduling during non-active states. The app will make a better use of the available resources and the users will appreciate the reduction in battery consumption. In general, while designing background tasks, take a moment to think of possible contextual triggers for your background tasks, along with the possible time triggers.

2.2. Development guidelines

After reviewing Android's capabilities for systematically scheduling tasks, with the associated pitfalls, solutions and recommendations, we now move on the concrete implementation. As our focus is on alarm-based background tasks, basically two issues arise: alarm scheduling and alarm handling. The first two guidelines cover alarm scheduling, while the last two are centered on alarm handling.

Guideline 1: use the right alarm scheduling mechanism

The Android system offers multiple alarm scheduling mechanisms. It is important to understand and use them well in order to obtain timely alarm triggers while the device is idle. The first version of the AlarmManager[4] introduced the set() method. This method allows scheduling highly reliable alarms in phones running Android 4.3 (API 18) or lower. However, starting from Android 4.4 (API19), alarms scheduled with the set() method became inaccurate, i.e., the system now decides when it is best to trigger them, instead of triggering them (exactly) when they were scheduled. In its place, Android 4.4 introduced setExact(), a new method to accurately schedule alarms.

Once again, in Android 6.0 (API 23), the AlarmManager was updated with the introduction of the Doze[5] power saving mode and the setExactAndAllowWhileIdle() method. This method allows to wake up a phone that is idle in Doze mode, with a maximum frequency of 9 min. If a higher frequency is needed, then the Doze battery optimizations need to be deactivated. However, at the same time (from Android 6.0), the setExact() method that was introduced in Android 4.4 – contrary to its name – is no longer reliable ("exact") to trigger alarms if the device is idle.

Given this mishmash of alarm scheduling methods and associated functionality under different versions of Android, it is imperative for the developer to select the correct alarm scheduling method, depending on the Android version of the client. Listing 1 shows how.

```
int alarmType = AlarmManager.RTC_WAKEUP;
long timeInterval = 60 * 1e3; // Trigger in 60 seconds
long triggerAtMillis = System.currentTimeMillis() + timeInterval;

PendingIntent receiverPendingIntent = PendingIntent.getBroadcast(context, 0,
        new Intent(context, Receiver.class), 0);
AlarmManager manager = (AlarmManager) context.getSystemService(Context.ALARM_SERVICE);

if (Build.VERSION.SDK_INT >= 23) { // Android 6 or higher
    manager.setExactAndAllowWhileIdle(alarmType, triggerAtMillis, receiverPendingIntent);
} else if (Build.VERSION.SDK_INT >= 19) { // Android 4.4 (KitKat) or higher
    manager.setExact(alarmType, triggerAtMillis, receiverPendingIntent);
} else { // Any version older than Android 4.4
```

4 https://developer.android.com/reference/android/app/AlarmManager.

5 https://developer.android.com/training/monitoring-device-state/doze-standby.

```
    manager.set(alarmType, triggerAtMillis, receiverPendingIntent);
}
```

Listing 1: Example setup to ensure accurate alarms over different Android versions (up to current latest version, Android 11).

Guideline 2: reschedule one-time alarms for recurrent alarms

In case of recurrent alarms, from Android 4.4 (API 19) "all repeating alarms are inexact", as stated in Android docs.[6] Therefore, systematic rescheduling is a better alternative. It can be done by setting the next alarm immediately after the current alarm got triggered, before executing the real task.[7] Even though this induces minimal time drift between alarms (i.e., a couple of hundreds of milliseconds per trigger) – not an issue for the vast majority of applications – it is better than delays in the order of tens of minutes which high frequency recurrent alarms will suffer.

Guideline 3: use broadcast receivers instead of services

In our experience, recent Android versions and certain phone manufacturers, may penalize a service being run because of an alarm trigger, and consequently delay the alarm trigger until the screen is turned on. Nonetheless, it is perfectly possible to start a service from a broadcast receiver. Therefore, we recommend to use broadcast receivers for triggers, as intermediary to run a service. Next to improved reliability, this also promotes separation of concerns, whereby each software component has its own responsibility. Broadcast receivers thus act as alarm handlers and services act as runners. For example, it allows to decide the kind of service to be executed depending on the task to perform (e.g., a foreground service for opportunistic location acquiring or a background service for inertial sensor sampling). It is also possible to start a runner service from a boot receiver to execute pending tasks at phone's restart.

Once the responsibilities of each software component have been defined, we have to deal with the fact that our code might be stopped in the middle of its execution and resumed a few minutes or hours later. Android docs state that an app started in background has approximately 10 s to do some work.[8] Once that deadline passed, the phone's processor enters in sleep mode until the next wake up. In our experience, in phones from certain manufacturers, these 10 s can be reduced to even 2 s or less.

Guideline 4: use wake locks to ensure proper task execution

After the alarm handler (i.e., the broadcast receiver) time execution window ($<$ 10s) finishes, the execution of any ongoing code will be paused and it will resume with the next alarm trigger or screen turn on. To prevent the processor from sleeping at this point, there is a low level mechanism, called wake lock.[9]

A wake lock allows to control the power state of the smartphone and force the processor to stay active. However, wake locks are a double-edged sword: when misused, they may lead to huge battery drain. Therefore, it is good practice to set an expiration timeout when acquiring a wake lock. Notwithstanding, depending on the variety of tasks to be performed by the app, it can be difficult to fix a time limit for staying awake at the moment the alarm is raised. For example, in applications where multiple tasks can be scheduled depending on several external factors and each task complexity (i.e., average execution time) is different, additional processing will be required right after the alarm trigger in order to estimate the suitable timeout for each case. It is important to know that wake locks can be acquired with and without a time limit. In addition, wake locks can be acquired multiple times but its acquisition is reference counted. This means that in order to release a wake lock the release() method needs to be called for as many times the acquire() method has been called first.[10] However, wake locks can be configured as non-reference counted to change this behavior, which leads to them to be released with the first release() method call or acquisition time expiration, even though one of the acquisitions is unbounded.

Therefore, we recommend to use non-reference counted wake locks, which can be acquired many times, but are released with the first call to the release() method. The proposed usage workflow is as follow: 1/ an alarm is raised, 2/ create the wake lock, set it as non-reference counted and perform an unbound acquire, 3/ obtain the information of the task(s) to be executed, calculate timeout time based on them and re-acquire the wake lock with a time limit (if the timeout happens, the wake lock will be released, even the first acquisition), 4/ perform the task and 5/ manually release the wake lock before the timeout rises, to save battery.

Note that it is important to avoid transitive wake locks. This means that if we have a recurrent task, our timeout should be placed before the next alarm gets triggered, in order to allow the smartphone to sleep (and save energy) at least for some time.

6 https://developer.android.com/reference/android/app/AlarmManager#setRepeating(int,%20long,%20long,%20android.app.PendingIntent).

7 Note to set it immediately, to avoid losing the iteration in case our task execution is interrupted or canceled — see guideline 4.

8 https://developer.android.com/reference/android/content/BroadcastReceiver.

9 https://developer.android.com/reference/android/os/PowerManager.WakeLock.

10 A time limit expiration counts as a release() method call.

2.3. Target system configuration

Once the background tasks workflow has been designed and scheduled, it is paramount to configure the target smartphone, in order to prevent Android's power saving measures from interfering with correct task execution. Depending on the phone, this process will be more or less straightforward.

In phones running Android stock layer, a special permission can be added in the application manifest[11] that allows the app to ask the user to exclude it from the default system battery optimizer through an in-app dialog. Of course, not every app can have this permission and Google will ban apps from the Play Store that ask for it when it does not correspond.[12] However if scheduling is part of the core functionality of the app, and the background task is well designed (see Section 2.1), the application will pass the filter quite safely. Unfortunately, many hardware manufacturers (e.g., Samsung, Huawei, Xiaomi, etc.) provide custom Android versions which, in an attempt to extend battery life, incorporate more stringent battery optimizers. These proprietary solutions are not generally accessible to app developers and hence, users have to manually disable them for specific apps from their phone settings.

The way to access those settings varies from one manufacturer to another, and sometimes even from different custom Android versions within the same manufacturer. Nonetheless, we can summarize them in two settings: allow the app to automatically start in background (i.e., auto-launch setting) and allow the app to use phone computational resources without restrictions (i.e. white-list app from battery savings).

While phone manufacturers do not provide developers with a mechanism to manage these settings programmatically, some community efforts provide alternatives such as "Don't kill my app"see footnote 1 and AutoStarter See footnote 2. The first one is a curated list of steps to take to disable these battery optimizations, classified by phone manufacturer. The second tries to discover undocumented manufacturer APIs to disable the optimizations via software. Efforts as the latter have uncovered obscure details which developers can deploy to access certain energy savings settings clandestinely, or bypass them. For example, Samsung allows an app to trigger alarms in a timely manner, if the app package name includes a special keyword (i.e., alarm, clock or alert); otherwise, it will restrict them to a highest frequency of 5 min.

In light of the variations of the Android OS between manufacturers, we recommend developers to include links to step-by-step manuals, such as provided by the "Don't kill my app" See footnote 1 community, in the installation process of their application, possibly combined with detecting phone manufacturer and Android version to provide more specific instructions. For that later purpose, there are approaches like "Doki"[13]

Fig. 1 presents a flowchart as a summary of the aforementioned guidelines, recommendations and configurations for the sake of helping developers on deciding which mechanisms and which optimizations to implement, depending on each specific use case. Nevertheless, keeping all these findings in mind and implementing them in an Android app is not a trivial task. Therefore, we have developed a software library which implements the aforementioned guidelines and mechanisms to conduct background tasks scheduling, abstracting from the underlying details (e.g., setting recurrent alarms, selecting the correct alarm setting method for each Android version, acquiring and releasing wake locks, etc.). Next, we present this library.

3. A library to schedule them all

The "NativeScript Task Dispatcher" (NTD) library [21] is a software tool that abstracts development complexities in the implementation of reliable task scheduling, needed for example in data collection apps. The library provides fundamental building blocks in the form of software components (see Section 3.1) for defining and scheduling (data collection) tasks overcoming the restrictions and issues posed by the newest Android versions (see Section 2). The library follows the development guidelines defined in Section 2.2, and requires the mobile phone configuration settings as defined in Section 2.3.

Based on the NativeScript[14] framework, the NTD library (or *plugin* in NativeScript's terminology) follows a hybrid approach, i.e. it combines web and native technologies. NativeScript is a mobile application development framework which allows to develop apps both for Android- and iOS-based devices. Common logic for both platforms is coded in TypeScript,[15] a JavaScript language super-set which adds types, interfaces and enumerations, among other elements, to the parent language and can be back-compiled to standard JavaScript code. NativeScript allows to code platform-specific functionalities using native languages (i.e., Java or Kotlin for Android and Objective-C or Swift for iOS), and also offers dynamic constructs to call native components from JavaScript code (i.e., Java/Objective-C classes) and vice-versa (via callback functions). Compiled JavaScript code runs on both platforms using the V8 JavaScript engine. Custom HTML views are transformed into native UI components to reduce user interaction latency. Although the NTD library does not support iOS devices yet, it has been designed to meet this requirement: all the code has been implemented in TypeScript, except for native interfaces for programming alarms written in Java. Therefore the rest of the section stitches to Android devices.

11 https://developer.android.com/reference/android/Manifest.permission#REQUEST_IGNORE_BATTERY_OPTIMIZATIONS.

12 https://developer.android.com/training/monitoring-device-state/doze-standby#whitelisting-cases.

13 https://github.com/DoubleDotLabs/doki.

14 https://nativescript.org/.

15 https://www.typescriptlang.org/.

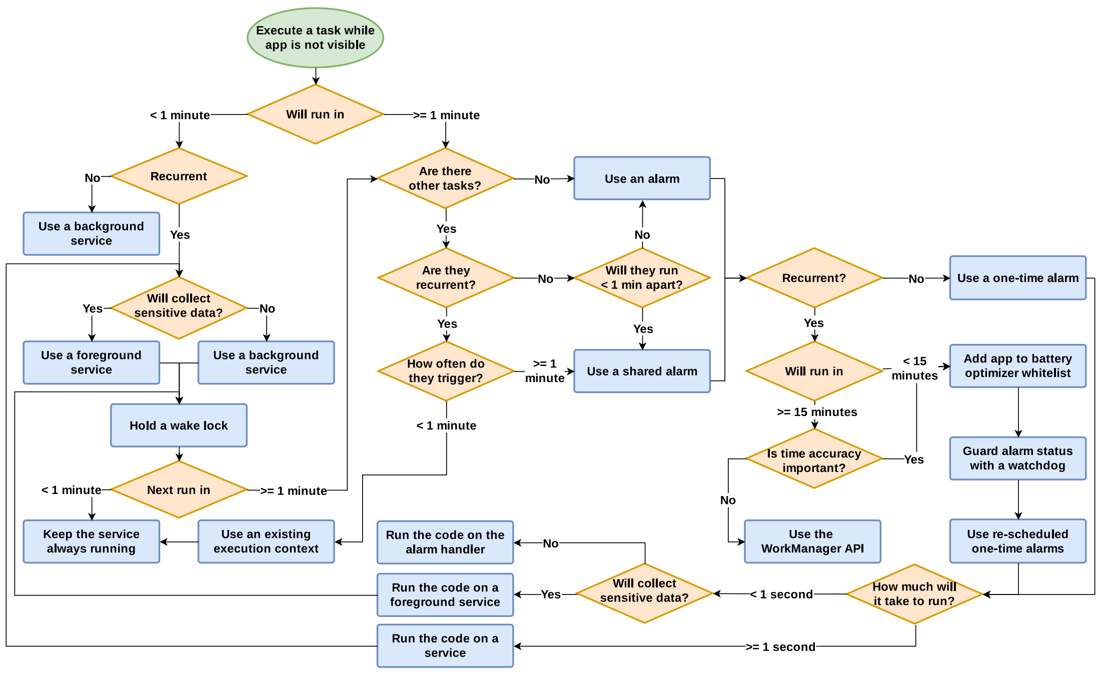
Execute a task while app is not visible
Will run in
< 1 minute
>= 1 minute
Recurrent
No
Yes
Use a background service
Will collect sensitive data?
Use a foreground service
Use a background service
Hold a wake lock
Next run in
Keep the service always running
Use an existing execution context
Are there other tasks?
Use an alarm
Are they recurrent?
Will they run < 1 min apart?
How often do they trigger?
Use a shared alarm
Recurrent?
Use a one-time alarm
< 15 minutes
>= 15 minutes
Add app to battery optimizer whitelist
Guard alarm status with a watchdog
Use re-scheduled one-time alarms
Is time accuracy important?
Use the WorkManager API
How much will it take to run?
< 1 second
>= 1 second
Run the code on the alarm handler
Run the code on a foreground service
Run the code on a service

Fig. 1. Background task design decision flow.

Before entering into detail in the next subsections, we summarize the main rationale for the library. It allows to create apps implemented as a workflow of event-driven tasks. Developers encapsulate custom code to be run in background in standalone units called tasks. Tasks are triggered by events and produce events as a result. A series of tasks are logically chained by events, which can be time-based events (i.e., triggered after a specified time), external events (i.e., emitted by the user interface, hardware or server side code) and task events (i.e. a result of a task). Therefore, tasks receive and produce events, ensuring that a task is never invoked directly by third-party components, but indirectly through external events. However, reversely, tasks can access external components. The motivation behind this behavior is to allow tasks to carry out certain side effects, which are strongly coupled to data collection processes, outside the plugin context. Beyond the resulting event, a task can for example store data locally, deliver a notification to the user, or submit collected data to a server, which are necessary actions in data collection apps.

3.1. Building blocks

The NTD library building blocks or essential components are tasks, events, the task graph and an entry point, which are covered in detail next.

3.1.1. Tasks

A task can be virtually any custom code block written by mobile application developers. A developer creates a new task either by extending the Task base class or by instantiating a new SimpleTask with a standalone function which represents the custom code.

```
export class DataProviderTask extends Task {
    constructor(
        name: string, // A way to refer to the task later
        private dataProvider: DataProvider // Dependencies can be injected via constructor
    ) {
        const taskConfig = {
            outputEventNames: ["dataAcquired"], // The event(s) which can be emitted
            foreground: false // Optional, specify foreground requirements
        };
        super(name, taskConfig);
    }

    async checkIfCanRun(): Promise<void> {await this.dataProvider.checkIfHasPermission();}

    async prepare(): Promise<void> {await this.dataProvider.askPermission();}

    protected async onRun(
18         params: TaskParams,
19         invocationEvent: DispatchableEvent
20     ): Promise<void | TaskOutcome> {
21         // Gracefully stop data acquisition if task gets canceled by a timeout. e.g.:
22         this.setCancelFunction(() => this.dataProvider.stopDataAcquisition());
23 
24         const data = await this.dataProvider.acquire();
25         return {eventName:"dataAcquired", result:data};
26         // Event name property is optional here, given task declared only one output event
27     }
28 }
```

Listing 2: Example which defines a task that obtains data from a sensor by using Task class inheritance.

Listing 2 provides an example of the former case, where a new task extends the Task base class by class inheritance. A task can specify external dependencies via constructor injection (ln 2–5). Task dependencies must be instantiated at the application start. Otherwise, lazy loading can be implemented at dependency level. A task also declares the event(s) that it produces (ln 7). This allows to check inconsistencies in the task graph at the entry point (see Section 3.1.4). By default, tasks are executed in background, but developers can also indicate foreground execution (ln 8). Task execution can be synchronous or asynchronous. Next, the checkIfCanRun() (ln 13) tests the task's pre-conditions for execution (e.g. lack of permission). Then, prepare() (ln 15) provides code to correct such unmet conditions (e.g. asking the user to grant permission). These last two methods can be asynchronous too (e.g., they can wait for user's answer to permission request). The core method of the task is onRun() (ln 17–27). Developers must override it with custom code that implements their specific task. The method gets as inputs the external parameters configured in the task graph (see Section 3.1.3) and the metadata and payload from the event which triggers the task. A task can declare a callback (ln 22) in case the task is canceled before finishing its job (e.g. due to a timeout). Other utility methods available are log(), remainingTime() and runAgainIn(), which respectively allow to add monitoring messages, get remaining time (i.e., time before timeout) and re-run the task at a given time.

The second way to define a task is by creating instances of the SimpleTask class. Unlike the Task base class, which is intended for sophisticated code with dependencies, SimpleTask instances are primarily used for data transformation. Developers can succinctly provide a function through the task constructor, at the expense of not relying on external dependencies or not specifying pre-execution checks. However, these tasks can still access globally available dependencies and utility functions such as log(), remainingTime() or runAgainIn(), which are part of the task context object that is passed to the custom function. The same holds true for the task parameters and the execution invocation event. Listing 3 shows the definition of three tasks using the SimpleTask instance creation method. The first task (ln 2–5) invokes two utility functions. The second task (ln 6–9) is triggered by an event (evt) with payload and manipulates it. The last one (ln 10–19) is a recurrent task with a variable execution interval. The interval for every subsequent execution is computed in the task itself and it is rescheduled by the runAgainIn() method. In general, tasks are independent of how they are scheduled. However, tasks that use the runAgainIn() method, such as the third example in Listing 3, are an exception; they need, by definition, to codify internally how the scheduling interval varies.

```
1  export const appTasks: Array<Task> = [
2      new SimpleTask("someTask", async ({ log, remainingTime }) => {
3              log(`Time left: ${remainingTime()}`);
4              // Other code to be executed
5          }, { foreground: false /* Optional */ }),
6      new SimpleTask("eventTriggeredTask", async ({ evt, log }) => {
7              log(`"someTask" result: ${evt.data.result}`);
8              return { result: `${evt.data.result} changed` };
9          }, { outputEventNames: ["dataProcessed"] }),
10     new SimpleTask("recurrentTaskVariableInterval", async ({ params, runAgainIn }) => {
11             const incrementFactor = 2; // Two minute increment per run
12             const execCount = params.execCount ? params.execCount : 1;
13 
14             // Do some processing
15 
16             // Reschedule again for next execution
17             const nextRun = toSeconds(execCount * incrementFactor, "minutes");
18             runAgainIn(nextRun, { execCount: execCount + 1 });
19         })
20 ];
```

Listing 3: Example defining a simple task, an event triggered task and a recurrent task with variable interval.

At this point, we have explained how to define tasks, we do not yet know how to schedule them, which is handled by the task graph (Section 3.1.3). An advantage of enforcing the separation of concerns between definition and schedule is that the testability of tasks is greatly improved.

3.1.2. Events

Events are the mechanism for providing the flow of data between tasks and between the library and external components. The event data model is composed of a name, an event chain ID, an expiration timestamp and the payload data. The name (a string, typically in camelCase) identifies a type of event. Event chain identifiers are version 4 Universal Unique Identifiers (UUIDv4), ensuring that the combination of event name and chain ID will always be unique (i.e., a chain cannot have more than one event with the same name). Event chain IDs are used to track event propagation, as we will explain below. The expiration timestamp is a UNIX timestamp specified for the first event in the task chain (see Section 3.1.3). It is used by the remainingTime() method to calculate the time available for execution before the timeout increases. Finally, the event payload data (encoded as a plain JavaScript object or array of objects) contains information shared by the event, either plain values or class instances (i.e., a JavaScript function instance).

Understanding how event chain IDs work is key to track event propagation. When a task (T1) is triggered by an event (evt1), and if evt1 is the first event in the chain (e.g. it is an external event dumped in the library context), a unique chain ID is created. Once T1 finishes its execution, it checks whether another task (T2) is interested in its output event (evt2). If so, T1 creates the new event evt2, which will trigger T2, but keeping the original chain ID. If no other task is interested in evt2, the task T1 emits a special event called "taskChainFinished" instead of its declared output event (evt2). That special event signals the end of a task chain. When all the ongoing task chains finish their execution, the phone goes back to sleep mode. There is a diagram illustrating this behavior on the public code repository.[16] There are exceptions to this behavior though, for example, a task chain can be divided in two children chains with their own chain IDs. The parent chain waits for all child chains to finish execution. However, these details are completely abstracted from the developer, but it is important to know them when analyzing the execution logs generated by the library.

Events can be created in the application and broadcasted through the NTD library context via the emitEvent() method placed at the library's entry point. We refer to Section 3.1.4 for more details.

The NTD library also includes utility functions to facilitate asynchronous testing of events: create(), emit() and listenToEventTrigger(). They allow to create events without immediately emitting them, to emit manually created events and to listen to events from outside the plugin context (i.e., to check if an event has been emitted by a task). These methods are for testing purposes only, as they are unpredictable when not used in isolation (as in test cases).

3.1.3. Task graph

The task graph allows the developer to define a task workflow of any complexity, agnostic from the underlying OS's technical details. It describes how the involved tasks are triggered and how they relate to each other. While a task is about blocks of code, the task graph is about scheduling the tasks. A developer creates a task graph by extending the TaskGraph base class to describe how and when each task is triggered. Per each task, a task graph describes how it is scheduled based on a wide variety of events such as time-based events (i.e., every minute, at a certain date, in a certain amount of time or a combination of recurrent and delayed execution), external events (e.g., button taps, built-in or external sensor changes, server sent events) and task events (emitted by other tasks).

Listing 4 shows an example of task graph. Each line typically starts with the on() method, which takes two arguments: the name of an event that will trigger a task, and the definition of how that task will be executed (i.e., immediately after the event trigger or at some point in the future), including any parameter if needed. To simplify the composition of the execution description of a task, there exists the run() function. It specifies if the task runs immediately (ln 7 and 10) or in the future (ln 4, 13, 16, 19–21 and 24), by means of the event-based scheduler or the alarm-based scheduler, respectively. Due to tasks can be scheduled in different ways, the run() method can be optionally chained with other methods: every() for recurrent tasks (ln 4), at() for tasks scheduled at a certain date (ln 13), in() within a concrete amount of time (ln 16), or a combination of thereof (ln 19–21 and 24).

```
1  class DemoTaskGraph implements TaskGraph {
2      async describe(on: EventListenerGenerator, run: RunnableTaskDescriptor): Promise<void> {
3          // Recurrent task at fixed interval, scheduled at event trigger
4          on("startEvent", run("recurrentTask").every(1, "minutes"));
5
6          // Single-run task, on event trigger. Receives a runtime parameter
7          on("startEvent",  run("instantTask", {param: "a param"}).now());
8
9          // Execution is immediate by default
10         on("recurrentTaskFinished", run("eventTriggeredTask"));
11
12         // Single-run task to be executed on a date, as long as the event triggers before
13         on("startEvent", run("dateTask").at(new Date(/* Future instant */)));
14
15         // Single-run task to be executed in 5 minutes, starting at event trigger
16         on("startEvent", run("delayedTask").in(5, "minutes"));
17
18         // Task to be executed on a date, then at recurrent interval
19         on("startEvent", run("dateRecurrentTask")
20             .at(new Date(/* Future instant */)
```

[16] https://github.com/GeoTecINIT/nativescript-task-dispatcher/blob/master/img/alarm-scheduler-lifycle.png.

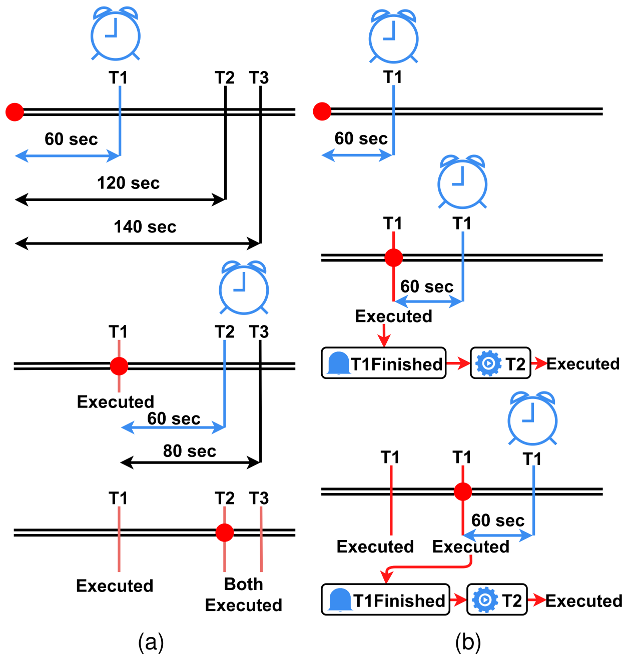

Fig. 2. Built-in task schedulers: (a) an example of an alarm-based scheduling in which T2 and T3 execution is grouped due to their proximity in time; (b) an example of alarm-based and event-based scheduling in which T1 is recurrently being scheduled and whose finalization triggers T2.

```
                .every(5, "minutes"));

        // Task to be executed in 5 minutes, then at recurrent interval
        on("startEvent", run("delayedRecurrentTask").in(5, "minutes").every(1, "minutes"));
    }
}
export const demoTaskGraph = new DemoTaskGraph(); // Generate a singleton instance
```

Listing 4: An example of a task graph describing how the tasks must be executed and how they relate to each other.

As pointed out above, the NTD library provides an alarm- and event-based scheduler. The alarm-based scheduler allows to plan the execution of one or more tasks in a concrete instant of time. Take the simple example of three tasks (T1, T2 and T3), each one runs once (one-time execution) and all are triggered by the same event. T1 is executed in 60 s, T2 in 120 s, and T3 in 140 s. Fig. 2(a) depicts how the alarm-based scheduler works in this example. T1 is set to run in 60 s from the start time (red dot), so the system alarm is scheduled to activate in 60 s (top-left diagram). After 1 min, the alarm-based scheduler triggers the execution of T1 and reschedules itself for the execution of T2 in 1 min (60 s) (central-left diagram). After 1 min more, the alarm-based scheduler triggers the execution of T2, but also T3 due to its proximity in time (bottom-left diagram). This relates to one of the recommendations in Section 2.1 in that tasks that run less than 1 min apart are scheduled together. By design, our alarm-based scheduler joins tasks when their execution time difference is less than 30 s, due to a limitation of Android alarms (see Section 2). If the time difference between the current time and the scheduled time of a task is less than 30 s, it will be executed in the current execution window. Consequently, some tasks can have their execution advanced or delayed up to 30 s. In other words, given four tasks scheduled at 60s, 90s, 91s and 120s respectively, only two alarms at 60s and 120s will be triggered; the first two tasks will be executed on the first alarm trigger and the other two tasks on the second. This behavior applies equally to recurrent tasks.

The event-based scheduler allows to trigger tasks as a reaction to an event, depending on internal (task result) or external events. This scheduler can be used in isolation (e.g., when a push notification arrives) or in combination with the alarm-based scheduler, allowing complex task configurations. Consider the following example: T1 runs recurrently every 60 s, while T2 depends on the result of T1, so the event-based scheduler is used every time T1 ends its execution, which produces an event. Fig. 2(b) shows the execution workflow for this scenario. T1 is set to run in 60 s from the start time (red dot), so the system alarm is scheduled for 60 s (top-right diagram). After one minute, the alarm-based scheduler triggers the execution of T1, and reschedules itself for the next execution of T1 (central-right diagram). When T1 ends the first execution, it emits an ending event which triggers the execution of T2 through the event-based scheduler. This is a typical situation for a periodic task that gathers data (T1) to be processed later (T2).

Notwithstanding, here are two important things to keep in mind. First, the execution of T1 may fail. In this situation, T2 will not run. Second, the developer needs to consider timeouts. Tasks triggered by external events have to finish in less than 3 min. In addition, recurrent tasks have to finish before their next execution. The due time is calculated and attached to the first event of the task chain. Tasks exceeding this time limit will be canceled, their cancellation code will be executed and the task chain will terminate prematurely due to the timeout.

The combination of scheduled tasks and event-driven tasks is a powerful mechanism to define tasks which, for example, systematically collect data over time and perform JIT analyses afterwards. The result of the data collection can be remotely sent or stored locally for further analysis or visualization. Furthermore, an event can be consumed by multiple tasks, thus opening up endless possibilities for task composition.

3.1.4. Entry point

The NTD library initialization and configuration is handled from a global class instance (singleton pattern), called entry point. It takes a list of tasks and the task graph instance every time it starts — even in background. Optionally, it allows to enable the built-in logging (i.e., console logging) or to pass by a custom logger to monitor the task executions. Listing 5 shows how to initialize the entry point with a list of tasks (e.g., Listing 3) and a task graph (e.g., Listing 4), which are placed in the application folder structure.

```
import {taskDispatcher} from "nativescript-task-dispatcher"
import {appTasks} from "./tasks"; // A list of tasks placed on another file
import {demoTaskGraph} from "./tasks/graph"; // A global instance of the app task graph

// Library initialization
taskDispatcher.init(appTasks, demoTaskGraph, {
    enableLogging: true
    // It is possible to inject a custom logger instead
})
```

Listing 5: Example of how to initialize the NTD library by providing the tasks and the task graph (i.e. init() method)

To ensure that all pre-execution conditions are met for the tasks to be executed, the entry point offers two methods: isReady() and prepare(). The isReady() method checks if the library and tasks meet all of the execution requirements (global requirements, e.g., battery optimizations are disabled). This is done by consulting the checkIfCanRun() method for all tasks (see Section 3.1.1). The prepare() method has the same semantics as the prepare() method of the Task context (Section 3.1.1): take appropriate steps to enable the execution of all tasks (e.g., showing up a permission dialog if permission is required).

In addition, developers are provided with two convenient features: the tasksNotReady property and the emitEvent() method. First, the property returns the list of tasks with unmet execution conditions, which is useful for example to display some preliminary information about the action before requesting user's permission. Second, the emitEvent() method allows injecting external events into the system by taking the event name, with an optional payload in the form of a plain JavaScript object. In case the latter is not provided, an empty object is used as payload. The method can be used to trigger the execution of tasks by an external event (e.g., a tap on a button, when a remote push notification is received, etc.). Listing 6 shows how to use these methods together to verify and prepare the application for the task execution.

```
import {taskDispatcher} from "nativescript-task-dispatcher"

const ready = await taskDispatcher.isReady();
if (!ready) {
    const notReady = await taskDispatcher.tasksNotReady;
    console.log(‘These tasks are not ready: ${notReady}‘);
    await taskDispatcher.prepare(); // Async call, wait for user intervention
}

taskDispatcher.emitEvent("startEvent", {withData: "start event payload (optional)"});
```

Listing 6: Example of how to check for tasks prerequisites (i.e., isReady() method), solve them (i.e., tasksNotReady property and prepare() method) and emit an event once ready (i.e., emitEvent() method).

Finally, it is important to note that as part of the pre-execution checks, the entry point automates the process of asking for disabling battery optimizations when necessary (i.e, there is at least one task to be run in less than 15 min in the application task graph), partly supporting the process described in Section 2.3. However, this process only works for devices running Android stock layer. For custom layers we provide a curated list of resources to deactivate OEM's proprietary battery optimizers.[17] This list can be used to guide users during app setup.

3.2. System architecture

Once the key components have been described, we outline the system architecture and how the low level components interact. Fig. 3 shows the overall architecture of the NTD library.

[17] https://github.com/GeoTecINIT/nativescript-task-dispatcher/blob/master/docs/disable-android-battery-saving.md.

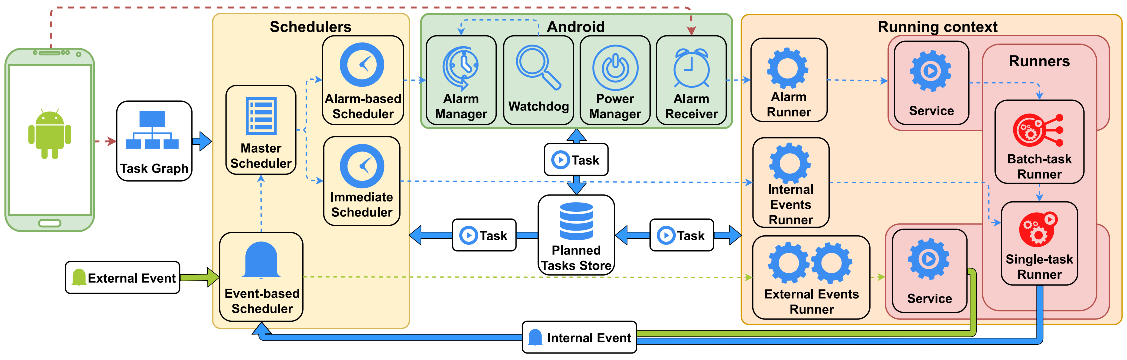

Fig. 3. NTD library architecture overview.

Starting with the `Schedulers` box, the task graph is interpreted and the required event listeners are registered in the `Event-based Scheduler`. When an external event is received (e.g., a button tap, a sensor change, etc.), a running context is bootstrapped, permitting to schedule tasks, even when the main application view is closed. Once the `External Event Runner` is running (within an Android service), it maps the received event to an internal event and passes it to the `Master Scheduler`, which schedules the tasks described in the task graph. At this point, two options are possible.

If a task is going to be executed in the future, it is sent to the `Alarm-based Scheduler`, which verifies the task and forwards it to the `Alarm Manager`. This manager schedules and stores the task metadata in the `Planned Tasks Store`. If a task runs immediately, it is sent to the `Immediate Scheduler`, which forwards the task to the `Internal Event Runner` for immediate execution. The `Internal Event Runner` bootstraps an instance of a `Single-task Runner`, which enforces timeout restrictions for the task being executed and stores metrics in the `Planned Tasks Store` before and after the execution of the task.

In the `Android` box, the Android-specific components responsible for the correct functioning of the alarm-based scheduling are represented. Previously named, the `Alarm Manager` is in charge of configuring the Android alarm mechanism following the suggestions listed in Section 2.2. Once an Android alarm is triggered, the `Alarm Receiver` forwards the control to the `Alarm Runner`, in order to run the tasks that need to be executed. These are obtained by querying the `Planned Tasks Store`, applying temporal filters.

The `Alarm Runner` bootstraps an Android service to run the tasks related to the alarm trigger, which in turn serves as a running context. The service is usually meant to run in background, but if any of the tasks to run or their dependents must run in foreground, the service will run in that mode instead. This service in turn bootstraps a `Batch-task runner`, to run all the tasks to be run by the alarm in parallel, using a `Single-task runner` instance for each one of them. Given JavaScript is single-threaded, tasks do not run truly in parallel, but they benefit from interleaving asynchronous calls.

Other components in the `Android` box are the `Power Manager` and the `Watchdog`. They are used to hold wake locks and ensure alarm setup every 15 min (i.e. outside the power manager danger area) respectively, thus meeting the rest of the recommendations from Section 2.2. Furthermore, the Power Manager is used to ask the user to disable the battery optimizations and ensuring these remain disabled – as long as the phone runs with Android stock layer – as specified in Section 2.3.

4. Does it truly work?: Experiment setup

We set up and ran three experiments to validate our approach. First, we conducted an initial exploratory experiment (experiment 0 — see Section 4.1) to scientifically demonstrate typical missingness of scheduled task executions (i.e. data measurements), as anecdotally reported in literature. We hereby rigidly followed the official Android docs to implement systematic background task scheduling. The two remaining experiments aim to evaluate the NTD library and the underlying guidelines (which are summarized in Fig. 1). The first experiment (Section 4.2) consists of the recurrent execution of a simple task to evaluate the two key concepts seen earlier in the introduction: reliability and efficiency. We use two criteria to address reliability: (O1) completeness — the extent to which all scheduled tasks are executed, (O2) accuracy — the extent to which scheduled tasks are executed at the time it was instructed. As regards efficiency, we computed (O3) battery usage — the power consumption of the proposed solution. The second experiment (Section 4.3) subsequently focuses on reliability under a more complex scenario, taken from a real-life case study, with multiple tasks scheduled at distinct time intervals. Based on the results, scientists/developers can decide whether the reported reliability is acceptable, depending on the requirements of the use case at hand.

Various Android-based mobile devices from different manufacturers were used (Table 1). As per our guidelines (see Section 2), battery saving mechanisms were disabled for all devices. During the experiments, the A1 and PO devices were

Table 1
Mobile devices used for experimentation.

ID	Device	Android OS	Exp. 0	Exp. 1	Exp. 2
NV	Nvidia Shield Tablet (Gen 1)	7.0	✓	✓	✓
BQ	BQ Aquaris V Plus	8.1	✓	✓	✓
A1	Xiaomi Mi A1	9.0	✓	✓	✗
H9	Honor 9 (STF-L09)	9.0	✓	✓	✓
PO	Poco F2 Pro (M2004J11G)	11	✗	✗	✓

Table 2
Percentage of data loss (lack of completeness) with a 1 min sampling rate over a 48 h time period. Average missing executions was 84%.

ID	No. planned executions	No. real executions	Missing executions (%)
NV	2,976	350	88.24%
BQ	2,975	165	94.45%
A1	2,973	1,199	59.67%
H9	2,880	181	93.72%

used as a personal phone, while the others were dedicated for the experiment. Note that not all devices have been used for both experiments. Research data and code supporting both experiments, along with execution instructions, are available on Zenodo, see [26].

4.1. Experiment 0: demonstrating the problem

To scientifically demonstrate the issue of missing data measurements (i.e., missing task executions) reported in literature, we designed a 2-day exploratory experiment, whereby a periodic task for unobtrusive data collection task was run every minute on different mobile phones (see Table 1). We hereby strictly followed the official Android documentation to schedule work in background.[18] Accordingly, two options are available: expedited works and exact alarms. The latter option – a repeating exact alarm, scheduled every minute – was chosen, as the former does not allow to schedule tasks more frequently than every 15 min. As prescribed, the repeating alarms were handled by a broadcast receiver, which locally logged the execution and the remaining device's battery life in a CSV file (hereby excluding any other point of failure, such as a failed sensor capture). As we used exact alarms, battery savings were not disabled because their execution is guaranteed according to the documentation. Therefore, according to the official Android guidelines, the task should be accurately executed every minute.

However, results demonstrated the lack of completeness and largely confirmed the approx. 50% data loss from literature (Table 2). In fact, the average missing executions exceeded 80%, which roughly means that only 2 out of 10 scheduled tasks are executed. For applications systematically correcting data, such behavior is not acceptable, regardless of the target application.

4.2. Experiment 1: simple task scheduling

For this experiment, we deployed two applications: 1/ an ad-hoc application written in Java that rigorously follows our task scheduling guidelines (Section 2), but without an alarm watchdog, 2/ a NativeScript application that uses the NTD library. Both applications ran a simple recurrent task – without logic or complex computations to produce a minimal execution overhead – every minute for 2 weeks.

In each task execution, we stored the "planning timestamp" and the "execution timestamp" in a local CSV file. The former is a UNIX timestamp which captures when a task was scheduled, i.e., when the alarm was triggered. The latter is a UNIX timestamp which captures when the task actually executed, i.e., the start of a task execution. In general, if given tasks were consistently scheduled every 60000 ms (i.e. one minute), subtracting two consecutive execution timestamps should exactly yield this value.

To examine completeness (O1), we computed the percentage of missing alarms (i.e., missing data) for each device-application pair by contrasting the actual number of tasks execution against the (theoretical) total number of scheduled tasks. Hence, the lower the missing data percentage, the better the reliability of the scheduling mechanism.

To measure accuracy (O2), the "delay" in task execution was calculated as the difference between two consecutive execution timestamps and subtracting the target alarm execution time (i.e., 60 s), obtaining an objective value that represents the real difference between the expected and the actual execution time. Examining this further, the delay in starting the execution after the alarm triggered was also calculated for both apps. To do so, we computed the time needed to schedule a task, by subtracting the "planning timestamp" of the next alarm from the "execution timestamp".

18 https://developer.android.com/guide/background.

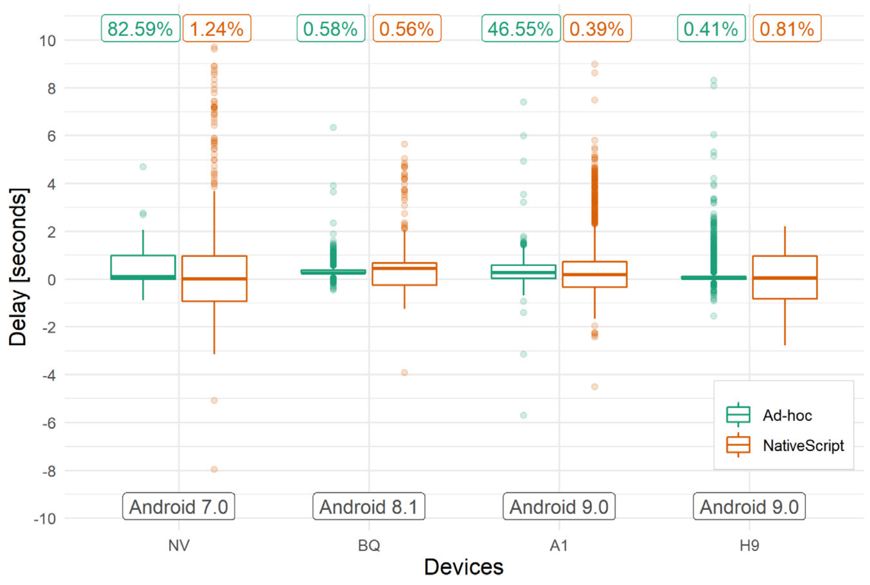

Fig. 4. Distribution of task execution delay per device and type of application (O2). Percentage of missing task execution at the top (lower is better) (O1). Devices are sorted from the oldest to the newest version of Android. Green represents ad-hoc application, orange the NTD-based application.

Consequently, we calculated the average delay between each alarm trigger and task execution for both apps by subtracting the corresponding timestamps. By comparing these delays of the ad-hoc and NTD application, we quantify the library's execution overhead.

To examine battery usage (03), in both applications, the task being executed was in charge of collecting the device's remaining battery percentage at task execution time. During the data analysis, we calculated the hourly battery consumption by taking the first remaining battery percentage every hour, and subsequently subtracted consecutive values pair-wise. This was done on each phone, each time running only one of both applications. This allows us to compute the energy consumption of both the ad-hoc and NTD-based application, and calculate the battery consumption overhead of the NTD library.

The ad-hoc application allows us to systematically assess the completeness (O1) and accuracy (O2) of the proposed guidelines, as well as the impact on battery consumption (O3). We hereby purposely did not implement a watchdog, as this is a process running in parallel which does not directly influence task scheduling, yet its absence allows us to detect Android's unexpected (and undocumented) interference with recurring tasks over a longer time period. On the other hand, the NativeScript application uses the NTD library, which implements all our guidelines, including the watchdog.

The premise is that the ad-hoc application should exhibit good completeness (O1) and accuracy (O2) over relatively shorter time periods - as per our guidelines - yet may fail (less complete - O1) over extended time periods (due to the lack of a watchdog). On the other hand, we expect the NTD application to exhibit better completeness overall (O1), due to the use of the watchdog, yet with unavoidable minor loss of accuracy (O2) and energy consumption (O3) overhead due to use of the NTD library (see above).

As regards the results, Fig. 4 shows box plots with the distribution of task execution delays for each pair of device-application. In addition, on top, percentages reflect the missing data for each device-application pair. Regarding completeness (O1), we observe that two of the four ad-hoc applications show a significant amount of missing data: the NV (82.59%) and the A1 (46.66%) devices. These devices abruptly stopped their execution – due to Android canceling the alarm planning and the lack of a watchdog in this application – before finishing the experiment and were unable to recover. In contrast, all NTD-based applications successfully finished the experiment, with only between 1.24% and 0.39% of missing task executions, demonstrating a completeness (O1) far exceeding the ones reported in literature (i.e., around 50%, see Section 1), maintained over longer time periods. Even the ad-hoc application, developed according to our task scheduling guidelines, shows better results than those reported in the literature, yet the need for a fail-safe mechanism such as the watchdog is clearly demonstrated by the two failing devices.

Regarding accuracy (O2), a total of 138,653 measurements were collected, from which 99.94% are represented in Fig. 4. The rest (77 values) were omitted for the sake of readability, from which a delay of 3+ hours (NV device with NTD-based application) and another one of almost 2 h (H9 device with NTD-based application) stand out. The first one happened over night, and the watchdog was able to recover the task (note that the ad-hoc application failed to complete on this device). The second one was due to a singular external event (a battery problem) which blocked the entire phone, and coincidentally only happened during the NTD-based run. All other outliers do not exceed 7 min. When interpreting the boxplot diagram in Fig. 4, we first need to emphasize that for the devices NV and A1, which both failed to complete

Table 3
Elapsed time from alarm trigger to task execution start comparison (unit: ms) (O2).

Descriptor	BQ		H9		BQ+H9	
	Ad-hoc	NTD	Ad-hoc	NTD	Ad-hoc	NTD
Minimum	16	49	11	29	11	29
25th percentile (Q1)	30	78	50	78	35	78
Median	35	85	58	82	**45**	**84**
75th percentile (Q3)	39	93	67	91	59	92
Maximum	708	953	1,692	1,990	1,692	1,990
Standard deviation	16	27	27	19	26	24

the experiment with the ad-hoc application, only the data until the failure of the application is represented. For these devices, we see a large spread above the 75th percentile for the NTD-based application (up to 10 s of delay), which is not present in the ad-hoc application simply due to the fact it prematurely halted its execution. That said, for these devices, the NTD-based application presents an average scheduling delay of 0.75 s for the NV device and 0.24 s for the A1 device.

Next to this, in general, we see that the median in all cases is around 0 (no delay), and data points generally fall well within acceptable limits for real-life measurement delays, both for the ad-hoc (between 0.02 and 0.48 s for the 25th–75th percentile; and 99.98% within 10 s) and NTD-based application (between −0.37 and 0.87 s; and 99.92% within 10 s). The data points for the ad-hoc applications tend to be more concentrated around the median. For the two devices (BQ and H9) on which both the ad-hoc and NTD-based application successfully finished the experiment, we cannot determine a clear pattern, with a larger spread outside the 75th percentile for the ad-hoc application for the H9 device, and the opposite for the BQ device.

To dig deeper into the accuracy (O2) of the BQ and H9 – the devices that successfully completed the experiment – we calculated an execution overhead metric to examine the time difference between the planned and the actual execution time of a task. This metric represents the execution expense, in milliseconds (ms), to ensure systematic scheduling. For the ad-hoc application, this consists of running the service and ensuring only one instance of the service is running at the same time (i.e., to handle Android mistakenly triggering alarms double, or crashed/blocked task executions). For the NTD-based application, this also includes all the library's execution overhead, such as deciding execution modality (foreground, background), assembling tasks to run and create the corresponding task execution chains, and finally, trigger their execution.

Table 3 compares the time elapsed from the alarm trigger to the execution of the task (execution start delay), for the BQ and H9 devices, both for the ad-hoc and NTD-based scheduling application, along with the overall task execution overhead for both devices. As can be observed, the median delay difference between the NTD-based application and the ad-hoc application results in an execution overhead of 39 ms, and 25th and 75th percentile show a similar difference (43 and 33 ms, respectively). Clearly, the benefits the NTD-based application brings in the long term (completeness and task scheduling flexibility) outweigh the minimal and acceptable penalty in accuracy and variability.

Next to the evaluation of the execution start delay of the NTD-based versus the ad-hoc application, we explored efficiency in terms of energy consumption during the experiment by calculating the average battery consumption per hour of the two solutions (O3). For a fair assessment, we hereby only considered the BQ device, which is the only one that successfully finished the experiment with both applications (excluding the NV and A1 devices) and ran the schedulers in isolation (i.e., the H9 was used as personal phone) as to avoid measurement interference from other applications.

Fig. 5 shows the hourly battery consumption, in percentage of the total battery capacity (3400 mAh), for the ad-hoc and NTD application running on the BQ device. The upper part (A) of the figure shows the result of calculating the difference of the first remaining battery capacity reading for each hour to the next (i.e., the hourly battery consumption/recharge), while the lower part (B) zooms in on the upper part (A) to look closer at the relative battery consumption in the range of 0 and 3%, i.e. excluding charging times (the large negative spike in A). We observe stable battery consumption usage between 0% and 1% per hour for both the ad-hoc and the NTD application.

Fig. 5 also shows the average hourly battery consumption (horizontal lines, annotated with the mean and standard deviation) when running the ad-hoc application and the NTD-based application respectively. Based on it, the NTD-based solution executing a task every minute consumes an average of 0.4% of the total battery power per hour (i.e., 9.6% per day) versus 0.31% per hour for the ad-hoc task scheduling application (i.e., 7.44% per day). In other words, the NTD-based solution consumes 0.09% of the total battery power more per hour than the ad-hoc scheduling application, i.e., 2.16% of the total battery power more per day. Hereby, note that no other applications were running on the device, so in both scenarios, only the regular OS battery usage and the task scheduling application's battery usage was included in our measurements. We argue that this can be considered an acceptable battery consumption, given a task was run every minute, and the benefits of using the NTD scheduler outweigh its consumption overhead. However, we also point out that developers need to pay attention to the task payload, which could considerably increase battery consumption, and needs to be balanced with the task scheduling frequency.

In summary, both the ad-hoc and NTD-based applications generally have excellent accuracy, with a median delay around 0 s (no delay) and the 25th–75th percentiles within 1 s of delay. Outside these percentiles, we observe various spreads (slightly more for the NTD-based solution), yet all within the 10 s delay. The NTD-based solution is generally

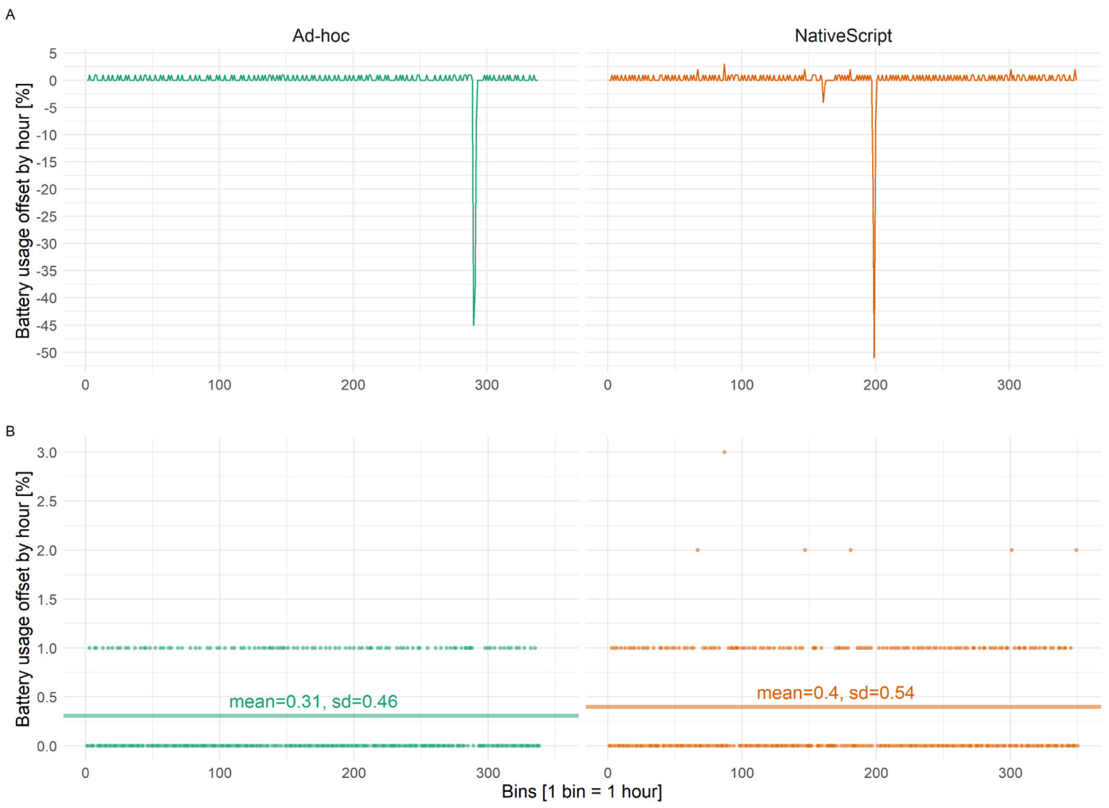

Fig. 5. (A) BQ device's hourly remaining battery usage (negative spikes mean charging time); (B) Same as A but omitting charging time to examine relative hourly battery consumption (O3).

shown to be more reliable and flexible than an ad-hoc solution with a very small extra cost in terms of execution overhead (39 ms median) and hourly battery consumption (0.09% of the total battery power on average).

4.3. *Experiment 2: complex task scheduling*

Beyond the simple task scheduling experiment, the second experiment mimics a more complex, real-life scenario to evaluate reliability of NTD-based apps. For that aim, we built a NativeScript-based data capturing tool – using our NTD library –, which includes a wide variety of tasks related to data gathering at different sampling rates, analysis, decision-making, and reporting. The application workflow is based on a well-known treatment in mental health, called exposure therapy, which consists of confronting patients with their fears and phobias in a controlled way. Often, such fears are related to specific places (e.g., agoraphobia, claustrophobia, acrophobia). During exposure, clinicians expose patients to these places, while (electronically) monitoring relevant behavioral and physiological parameters, such as movement, heart rate, anxiety level, etc. The capturing tool used in this experiment precisely covers this goal.

Fig. 6 represents the task graph for the experiment — the corresponding descriptive task specification is included in supplementary material. The first task collects the patient's location every two minutes. Once finished, it triggers another task to perform a geofencing analysis based on the location captured by the previous task. If the patient walks nearby (200 meters away) the area of interest for treatment, a new task is scheduled every minute to collect five (one every 10 s) location samples. This allows us to obtain greater precision of the patient's movement when within the area of exposure. Once the patient is in, a one-time task delivers a welcome message as a notification, and a recurrent task is scheduled every two minutes to prompt her with a single question questionnaire concerning her anxiety level. Once the patient answered the questionnaire, a task is triggered to properly process the response. Depending on the reported anxiety levels, the application decides whether the patient can leave the area (exposure completed) or not. If so, the application sends a notification rewarding the effort and inviting her to leave the area. When the patient leaves the exposure area (by 200 m), the multiple location gathering task stops and the application invokes again the single-location gathering task (first task above), scheduled every two minutes.

Fig. 6 also identifies each type of task. Recurrent (data collection) tasks are represented by blue solid-bordered boxes. Event-driven tasks are represented by green dot-bordered boxes. Arrows represent events launched by tasks, whereby

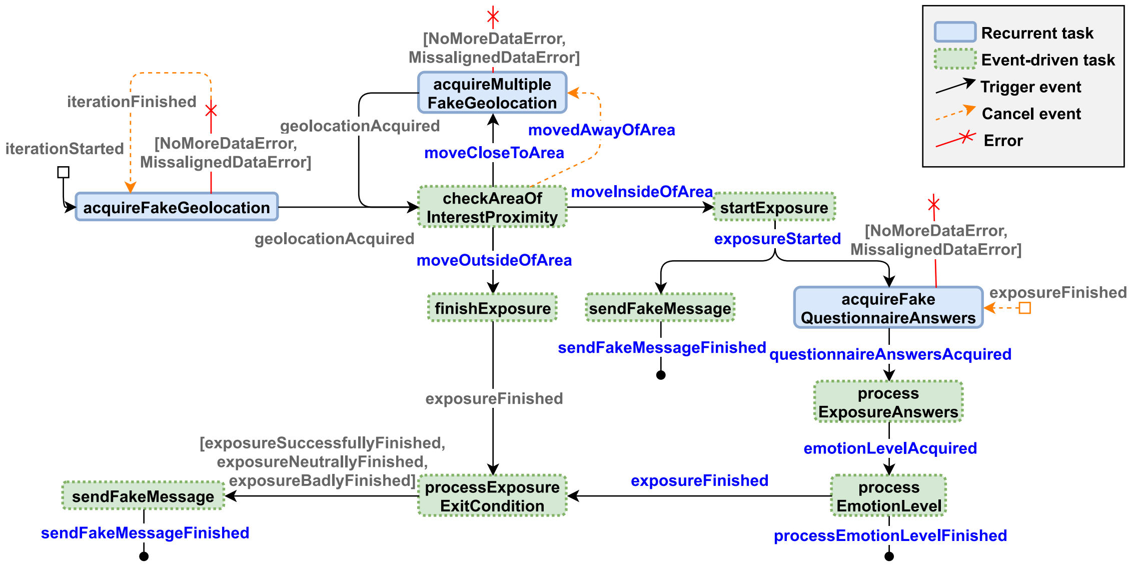

Fig. 6. Task graph used by the data capturing tool and the simulation application.

Table 4
Percentage of missing key events (blue-colored in Fig. 6) during the simulation. Bold event marks exposure's therapy end.

Key events	Missing events (%)			
	NV	BQ	H9	PO
movedCloseToArea	0	1.28	0	0.85
movedInsideOfArea	0	0	0	0.21
exposureStarted	0	0	0	0.21
sendFakeMessageFinished	0.74	1.06	0.21	3.72
questionnaireAnswersAcquired	1.49	2.29	0.21	5.62
emotionLevelAcquired	1.49	2.29	0.21	5.62
processEmotionLevelFinished	1.47	2.30	0.19	5.28
exposureFinished	**1.70**	**2.13**	**0.42**	**9.36**
movedOutsideOfArea	0	0	0	0.21
movedAwayOfArea	0.21	0	0.21	0.21

black-solid arrows are events that trigger other tasks and orange-dashed arrows events that cancel tasks. Sharp blue label events are analyzed to evaluate the reliability below.

We developed a simulation tool to simulate sensor readings and patient responses according to the task graph in Fig. 6. The data capturing tool was used once (on the BQ device) to collect real data for about 45 min (typical duration of an exposure therapy). Next, the captured data was cleaned and normalized to remove any device-specific bias. The processed data was then used to iteratively run simulations using the data capturing tool in combination with the pre-recorded data, on different phones (see Table 1) uninterruptedly for two weeks (470 exposure therapy iterations per device). During simulation, the application stored execution metadata of each task in a local database (i.e., task name, triggering event, execution timestamp, task duration, event emitted and outcome data). At the end of each iteration, the content of the database was exported to a JSON file. With this setup, we were able to stress our library and devices with a complex configuration of scheduled tasks for an extended period of time.

Table 4 shows the results of the experiments in terms of the percentage of key events missed on each device for the 470 simulated exposure iterations. Given real exposure therapy with a patient in clinical practice typically occurs twice per week, having 235 simulations per week signifies a considerable stress test. The results show that all events were triggered with high success. However, some differences between devices were identified. For example, the PO device showed the worst completeness, with 5.62% lost events in the acquisition of questionnaire answers and subsequent processing tasks, and 9.36% lost "exposureFinished" events. Two reasons may explain this: the device runs the latest Android version, possibly with more restrictive battery usage policies, and it was being used as personal phone during the experiments.

At the other end, the H9 device showed the best results, with only 0.21% of lost events related to user data acquisition and processing events and 0.42% lost of "exposureFinished" events. Finally, the NV and BQ devices showed in-between, generally good results, with missing rates around 1.70% and 2.13% respectively for the previous events.

Although the reliability of an application based on our library is partially dependent on the device's OS and vendor, as the previous results suggest, they show that the NTD library is able to schedule and handle multiple tasks in complex and stressful configurations with high degree of reliability in real-world scenarios. Furthermore, as evidenced by the relative simplicity of the associated descriptive task representation, the NTD library significantly eases the development of mobile applications requiring such complex systematic (background) task scheduling, such as (scientific) monitoring applications.

5. Limitations

Despite the good completeness and accuracy of the NTD library and the underlying guidelines, there are some limitations regarding usage requirements, functionality and the evaluation method.

In terms of usage requirements, while the guidelines are applicable over all Android versions and variants, the library is only available as a NativeScript plugin, and thus only supports applications written in NativeScript. However, NativeScript code can be interwoven with fragments in e.g. Java or Kotlin, so developers can benefit from the NTD while still writing their custom code (such as the task payload) in their preferred programming language. Furthermore, while NativeScript is inherently multi-platform, the NTD library currently only supports Android native instructions. Support for other platforms is future work.

From an end-user point of view, as explained in Section 2, the NTD library relies on disabling battery optimizations and enable auto-launch settings in non-stock Android devices, for which there is no universal solution. It requires end users to manually apply these settings, and lack or failure in doing so is a major source of systematic task scheduling failures.

Regarding functionality, some improvements for more dynamic use of the library are possible. First of all, the library does not allow updating the task graph at run-time. For re-configuration, a restart is required. Secondly, a technical limitation is the inability to stop the execution of a task through more than one event. Similarly, a single task cannot be triggered by multiple events. All of the above technical limitations are due to low-level implementation details, which we plan to overcome in future work.

Finally, we acknowledge limitations in the size and scope of the experiment. Even though this article reports on several months of experiments, further experimentation exhausting Android versions and device models are needed to confirm completeness and accuracy of our solution over a broader Android market segment. Concretely, we have shown that our solution successfully works in phones running Android stock and some non-Android stock systems, i.e. EMUI (Huawei and Honor phones) and MIUI (Xiaomi and POCO phones) OS layers, yet correct functioning of our solution using other custom layers needs to be experimentally confirmed.

6. Conclusion

Increasingly, mobile phone manufacturers introduce more restrictive energy-saving and privacy policies in subsequent Android versions and variants, complicating their use as systematic, scientific monitoring instruments. Based on our 3+ years experience with developing mobile phone based monitoring applications, in this article we addressed a significant and under-reported problem over various application domains: missing sensor readings due to unreliable background task scheduling in Android-based systems.

In this article, we scientifically demonstrated the problem, and showed an average of over 80% missing task executions over 48 h, using 4 different Android devices and a sampling rate of 1 min. Then, we presented a set of recommendations and best practices to overcome these problems, within certain restrictions (i.e., a maximum scheduling frequency of 1 min). Based on these guidelines, developers can implement their own reliable scheduling application, yet we also present a NativeScript task scheduling library, which abstracts from the low level implementation details. It offers developers an easy-to-use API to schedule complex task scenarios, based on an descriptive task graph specification.

The presented software library was evaluated in terms of completeness and accuracy, both in simple and complex scenarios, each with experiments running over two weeks. The results show excellent completeness (between 0.39% and 1.24% missing events in simple scenarios; between 0% and 3% in complex scenarios, with outliers at 5% and 9% on one particular device) and accuracy (with execution delays for the 25th–75th percentile between −0.37 and 0.87 s; and 99.92% of them below 10 s), both when applying the guidelines and when using our library. This far exceeds the scarce practical results reported in literature. Furthermore, we show an acceptable battery consumption, with an hourly 0.31% battery usage on average (3400 mAh battery) for an ad-hoc implementation applying a 1-minute sampling rate, and 0.4% for the NTD library (i.e., an overhead of 0.09% of the total battery power per hour), while yielding significant benefits in terms of reliability and scheduling flexibility.

To conclude, while we understand privacy and battery-life concerns of mobile manufacturers, we regret the negative impact their increasingly restrictive and mostly undocumented task scheduling policies have on using mobile devices as a reliable (scientific) measuring instrument. While some authors advocate a pragmatic solution (e.g., a dedicated Android version for scientific experiments [15]), we think these forego the ubiquity of smartphones as a day-to-day consumer product. Instead, we call upon smartphone manufacturers to acknowledge the status of a priority class of applications (such as scientific or medical ones), subjecting them to additional fierce scrutiny, yet granting them runtime privileges to bypass restrictive policies that impede their correct functioning — evidently with explicit confirmation of the user.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Alberto González-Pérez and Miguel Matey-Sanz are funded by the Spanish Ministry of Science and Innovation [grants FPU17/03832 and FPU19/05352]. This study was supported by grants RTI2018-099939-B-I00 (SyMptOMS) and PID2020-120250RB-100 (SyMptOMS-ET) funded by MCIN/AEI/10.13039/501100011033.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.pmcj.2022.101550.

References

[1] B. Guo, Z. Wang, Z. Yu, Y. Wang, N.Y. Yen, R. Huang, X. Zhou, Mobile crowd sensing and computing: The review of an emerging human-powered sensing paradigm, ACM Comput. Surv. 48 (1) (2015) 1–31, http://dx.doi.org/10.1145/2794400.
[2] J. Torres-Sospedra, J. Avariento, D. Rambla, R. Montoliu, S. Casteleyn, M. Benedito-Bordonau, M. Gould, J. Huerta, Enhancing integrated indoor/outdoor mobility in a smart campus, Int. J. Geogr. Inf. Sci. 29 (11) (2015) 1955–1968, http://dx.doi.org/10.1080/13658816.2015.1049541.
[3] T. Blaschke, G.J. Hay, Q. Weng, B. Resch, Collective sensing: Integrating geospatial technologies to understand urban systems—An overview, Remote Sens. 3 (8) (2011) 1743–1776, http://dx.doi.org/10.3390/rs3081743.
[4] D. Colombo, J. Fernández-Álvarez, A. Patané, M. Semonella, M. Kwiatkowska, A. García-Palacios, P. Cipresso, G. Riva, C. Botella, Current state and future directions of technology-based ecological momentary assessment and intervention for major depressive disorder: A systematic review, J. Clin. Med. 8 (4) (2019) 465, http://dx.doi.org/10.3390/jcm8040465.
[5] L.P. de Vries, B.M.L. Baselmans, M. Bartels, Smartphone-based ecological momentary assessment of well-being: A systematic review and recommendations for future studies, J. Happiness Stud. (2020) http://dx.doi.org/10.1007/s10902-020-00324-7.
[6] A. Trifan, M. Oliveira, J.L. Oliveira, Passive sensing of health outcomes through smartphones: systematic review of current solutions and possible limitations, JMIR MHealth UHealth 7 (8) (2019) e12649.
[7] V.P. Cornet, R.J. Holden, Systematic review of smartphone-based passive sensing for health and wellbeing, J. Biomed. Inform. 77 (2018) 120–132.
[8] I. Miralles, C. Granell, L. Díaz-Sanahuja, W. Van Woensel, J. Bretón-López, A. Mira, D. Castilla, S. Casteleyn, Smartphone apps for the treatment of mental disorders: Systematic review, JMIR Mhealth Uhealth 8 (4) (2020) e14897, http://dx.doi.org/10.2196/14897.
[9] S. Shiffman, A.A. Stone, M.R. Hufford, Ecological momentary assessment, Annu. Rev. Clin. Psychol. 4 (1) (2008) 1–32, http://dx.doi.org/10.1146/annurev.clinpsy.3.022806.091415.
[10] K.E. Heron, J.M. Smyth, Ecological momentary interventions: Incorporating mobile technology into psychosocial and health behaviour treatments, Br. J. Health Psychol. 15 (1) (2010) 1–39, http://dx.doi.org/10.1348/135910709x466063.
[11] M.M. Baig, H. GholamHosseini, M.J. Connolly, Mobile healthcare applications: system design review, critical issues and challenges, Australas. Phys. Eng. Sci. Med. 38 (1) (2014) 23–38, http://dx.doi.org/10.1007/s13246-014-0315-4.
[12] T.W. Boonstra, J. Nicholas, Q.J. Wong, F. Shaw, S. Townsend, H. Christensen, Using mobile phone sensor technology for mental health research: Integrated analysis to identify hidden challenges and potential solutions, J. Med. Internet Res. 20 (7) (2018) e10131, http://dx.doi.org/10.2196/10131.
[13] J. Torous, P. Staples, I. Barnett, L.R. Sandoval, M. Keshavan, J.-P. Onnela, Characterizing the clinical relevance of digital phenotyping data quality with applications to a cohort with schizophrenia, NPJ Digital Med. 1 (1) (2018) 1–9.
[14] O. Petter, M. Hirsch, E. Mushtaq, P. Hevesi, P. Lulugowicz, Crowdsensing under recent mobile platform background service restrictions: A practical approach, in: Adjun. Proc. Of The 2019 ACM Int. Jt. Conf. On Pervasive And Ubiquitous Computing And Proc. Of The 2019 ACM Int. Symposium On Wearable Computers, ACM, New York, NY, USA, 2019, pp. 793–797, http://dx.doi.org/10.1145/3341162.3344867.
[15] N. Klugman, V. Jacome, M. Clark, M. Podolsky, P. Pannuto, N. Jackson, A.S. Nassor, C. Wolfram, D. Callaway, J. Taneja, P. Dutta, Experience: Android resists liberation from its primary use case, in: Proc. Of The 24th Annual Int. Conf. On Mob. Comput. And Netw., in: MobiCom '18, ACM, New York, NY, USA, 2018, pp. 545–556, http://dx.doi.org/10.1145/3241539.3241583.
[16] S. Bähr, G.-C. Haas, F. Keusch, F. Kreuter, M. Trappmann, Missing data and other measurement quality issues in mobile geolocation sensor data, Soc. Sci. Comput. Rev. (2021) 0894439320944118, http://dx.doi.org/10.1177/0894439320944118.
[17] W. Regli, Wanted: toolsmiths, Commun. ACM 60 (4) (2017) 26–28.
[18] D. Ferreira, V. Kostakos, A.K. Dey, AWARE: mobile context instrumentation framework, Front. ICT 2 (2015) 6.
[19] J. Torous, M.V. Kiang, J. Lorme, J.-P. Onnela, New tools for new research in psychiatry: a scalable and customizable platform to empower data driven smartphone research, JMIR Ment. Health 3 (2) (2016) e16.
[20] J.E. Bardram, The CARP mobile sensing framework–a cross-platform, reactive, programming framework and runtime environment for digital phenotyping, 2020, arXiv preprint arXiv:2006.11904.
[21] A. González-Pérez, M. Matey-Sanz, "nativescript-task-dispatcher": A Reactive Android-Based Task Scheduler and Dispatcher, Zenodo, 2021, http://dx.doi.org/10.5281/zenodo.4530103, Data and documents are licensed under CC Attribution 4.0 International License. Code is licensed under the Apache License 2.0..
[22] A. González-Pérez, I. Miralles, C. Granell, S. Casteleyn, Technical challenges to deliver sensor-based psychological interventions using smartphones, in: Adjun. Proc. Of The 2019 ACM Int. Jt. Conf. On Pervasive And Ubiquitous Computing And Proc. Of The 2019 ACM Int. Symposium On Wearable Computers, 2019, pp. 915–920, http://dx.doi.org/10.1145/3341162.3346271.
[23] I. Miralles, C. Granell, A. García-Palacios, D. Castilla, A. González-Pérez, S. Casteleyn, J. Bretón-López, Enhancing in vivo exposure in the treatment of panic disorder and agoraphobia using location-based technologies: A case study, Clin. C. Stud. 19 (2) (2020) 145–159.
[24] L. Díaz-Sanahuja, I. Miralles-Tena, C. Granell-Canut, J. Bretón-López, A. González-Pérez, S. Casteleyn, D. Castilla, A. García-Palacios, Enhancing stimulus control in the treatment of gambling disorder using location-based technologies, in: ESRII 2019 Abstr. Book 6th Scientific Meeting 2019 Sept. 5-6, Copenhagen, 2019, pp. 63–64.
[25] M. Ciliberto, F.J.O. Morales, H. Gjoreski, D. Roggen, S. Mekki, S. Valentin, High reliability android application for multidevice multimodal mobile data acquisition and annotation, in: Proc. Of The 15th ACM Conf. On Embed. Netw. Sens. Syst., in: SenSys '17, ACM, New York, NY, USA, 2017, http://dx.doi.org/10.1145/3131672.3136977.
[26] C. Granell, M. Matey-Sanz, A. González-Pérez, S. Casteleyn, Reproducibility Package for "Using Mobile Devices as Scientific Measurement Instruments: Reliable Android Task Scheduling", Zenodo, 2022, http://dx.doi.org/10.5281/zenodo.5032027.