BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

MOBILE APPLICATION FOR EXPORTING ANDROID DEVICE METRICS INTO A PROMETHEUS DATABASE

MOBILNÍ APLIKACE PRO EXPORT METRIK ANDROID ZAŘÍZENÍ DO DATABÁZE PROMETHEUS

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MARTIN PTÁČEK**
AUTOR PRÁCE

SUPERVISOR **RNDr. MAREK RYCHLÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

144035

Institut:	Department of Information Systems (UIFS)
Student:	**Ptáček Martin**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Mobile Application for Exporting Android Device Metrics into a Prometheus Database**
Category:	Mobile applications
Academic year:	2022/23

Assignment:

1. Familiarize yourself with the Prometheus database and its format for retrieving metric data when monitoring and loading various sources into the Prometheus database.
2. Explore the ability to monitor the status of Android mobile devices, both the status of hardware sensors and the status of running applications and user activity.
3. Design a mobile application for the Android system that will allow to export device status monitoring metrics to a Prometheus database instance, both continuously and in a batch. Also address possible issues of the export process in the case of temporary unavailability of a network connection (e.g., by temporary storing the metrics and subsequent export when online). Focus also on the energy efficiency of the application.
4. After consultation with the supervisor, implement the application. Also design and perform a set of tests to verify the usability and energy efficiency of the running application.
5. Evaluate the solution, suggest possible extensions, and publish the project as open-source.

Literature:

- Exporters and integrations. *Prometheus - Monitoring system and time series database* [online]. [cit. 2022-05-06]. Available at https://prometheus.io/docs/instrumenting/exporters/
- Turnbull, James: *Monitoring with Prometheus*. Turnbull Press, 2018. ISBN 9780988820289
- Brazil, Brian: *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. O'Reilly Media, 2018. ISBN 9781492034094
- Späth, Peter: *Learn Java for Android development :migrating Java SE programming skills to mobile development.* Fourth edition. ISBN 978-1-4842-5942-9

Requirements for the semestral defence:
Items 1, 2, and 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Rychlý Marek, RNDr., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	31.7.2023
Approval date:	18.10.2022

Abstract

This thesis deals with expanding the ecosystem of the time-series database Prometheus to enable monitoring for Android devices. The aim of this thesis is to implement a Prometheus exporter for Android devices. The final implementation of this exporter does not only support exposing metrics directly to Prometheus but also supports storing metrics on-device when offline and exporting them later in a batch. The application also includes a mode which can export metrics to Prometheus over a NAT or similar network barrier. Over 35 different metrics of the Android device are exported to the Prometheus database. The reader is presented with the results of tests regarding the energy efficiency of the implementation. The work is released as open-source software on the Github platform.

Abstrakt

Tato práce se zabývá rozšířením ekosystému databáze časových řad Prometheus, aby bylo možné monitorovat také mobilní telefony s operačním systémem Android. Cílem této práce je implementovat Prometheus exportér pro Android. Výsledná implementace tohoto exportéru podporuje nejen zpřístupnění metrik pro databázi Prometheus, ale také ukládání metrik do paměti zařízení v případě nedostupnosti připojení a jejich následný dávkový export. Aplikace také obsahuje režim pro export metrik do databáze Prometheus přes NAT nebo podobnou síťovou bariéru. Aplikace exportuje přes 35 různých metrik z Android zařízení do databáze Prometheus. Čtenáři jsou předloženy výsledky testů týkající se energetické náročnosti implementace. Práce je publikována jako software s otevřeným zdrojovým kódem na platformě Github.

Keywords

Prometheus database, Time series database, Android application, Prometheus Exporter, Metrics, Grafana, NAT traversal, Kotlin, Jetpack Compose, Device Monitoring, PromQL query language

Klíčová slova

Databáze Prometheus, Databáze časových řad, Android aplikace, Prometheus exportér, Metriky, Grafana, překonání NAT, Kotlin, Jetpack Compose, Monitorování zařízení, PromQL dotazovací jazyk

Reference

PTÁČEK, Martin. *Mobile Application for Exporting Android Device Metrics into a Prometheus Database.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Rozšířený abstrakt

Prometheus je populární databáze časových řad pro monitorování cloudových aplikací s otevřeným zdrojovým kódem. Funguje tak, že si periodicky stahuje metriky z monitorovaných aplikací, na každé monitorované aplikaci tedy běží HTTP server se zpřístupněnými metrikami. Podstatnou výhodou této databáze je, že má kolem sebe velký ekosystém instrumentačních knihoven pro různé platformy a programovací jazyky. Některé aplikace ovšem není možné instrumentovat přímo, a tak je pro monitorování potřeba program, který běží vedle monitorované aplikace a zpřístupňuje dostupné metriky aplikace ve správném formátu databázi Prometheus. Tomuto programu se říká Prometheus exportér.

Cílem této práce je implementace Prometheus exportéru pro Android zařízení ve formě nativní Android aplikace. Tento exportér umí nejen zpřístupňovat metriky pro databázi Prometheus, ale také obsahuje implementaci proxy klienta pro překonávání NAT a podobných síťových bariér. V případě nedostupnosti síťového připojení na Android zařízení umí implementovaný exportér také ukládat metriky lokálně a následně, jakmile je zařízení zase online, exportovat tyto metriky do databáze Prometheus.

Aplikace je implementována v programovacím jazyce Kotlin ve frameworku Jetpack Compose. Jetpack Compose je doporučený nástroj pro vytváření moderních uživatelských rozhraní pro nativní Android aplikace. Aplikaci lze z uživatelského hlediska konfigurovat buď v uživatelském rozhraní nebo přes konfigurační soubor ve formátu YAML. Po spuštění monitorování běží aplikace s nastavenou konfigurací na pozadí a znovu se spustí i po restartu zařízení. Lze konfigurovat 3 módy aplikace a to Prometheus exportér, proxy klient pro překonávání NAT a podobných síťových bariér a dávkový exportér pro uchovávání metrik offline a následný export do databáze Prometheus. Všechny tyto módy mohou také běžet paralelně.

První polovina práce se zaměřuje na detailnější popis databáze Prometheus a na rozbor současných možností pro monitorování Android zařízení. Ve druhé polovině práce je pak popsán design výsledného řešení, konkrétní architektura a použité technologie. Pro implementaci překonání NAT byla použita PushProx proxy, projekt s otevřeným zdrojovým kódem. PushProx proxy funguje na bázi klienta a serveru a PushProx klient je integrován přímo do finálního řešení.

Pro funkci exportování metrik v dávkách byl využit Prometheus Remote Write protokol. Tento protokol je typicky používán pro federaci více instancí databáze Prometheus, dá se však využít i pro dávkový export metrik přímo z aplikace. Odesílání metrik do databáze Prometheus přímo monitorovanou aplikací, ikdyž funkční, jde proti designové filosofii databáze Prometheus.

Implementovaný exportér zpřístupňuje především metriky z hardware senzorů, se získáním informací o aktivitě uživatele jsou problémy z hlediska systémových oprávnění. Dostupné metriky zahrnují data například z akcelerometru, gyroskopu nebo senzoru ambientní teploty. Dohromady je dostupných přes 35 různých metrik.

Byly také provedeny optimalizace z hlediska energetické náročnosti aplikace, a to zejména optimalizace síťového provozu. Čtenáři jsou prezentovány výsledky testů energetické náročnosti aplikace z hlediska síťového provozu.

Práce je zveřejněna jako software s otevřeným kódem na platformě Github. Pro jednoduchou vizualizaci metrik na webu byla také vytvořena a publikována tabule ve webovém vizualizačním nástroji Grafana. Součástí repositáře je také ukázková konfigurace serveru pro PushProx proxy, Prometheus databázi a instanci vizualizačního nástroje Grafana. Hlavním přínosem této práce je rozšíření ekosystému databáze Prometheus o možnost monitorování další platformy.

Mobile Application for Exporting Android Device Metrics into a Prometheus Database

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Marek Rychlý. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Martin Ptáček
July 31, 2023

Acknowledgements

I would like to express my thanks to my supervisor RNDr. Marek Rychlý Ph.D. for his support, patience and many valuable advice which helped me a lot during implementation of this project. I would also like to thank my family for emotional support.

Contents

List of Figures

Chapter 1

Introduction

Prometheus is a time series database mainly used for monitoring servers, production applications and other cloud infrastructure. It also allows visualization and alerting on gathered metrics. Prometheus exporters are applications, that gather metrics from monitored systems and expose them to Prometheus in the right format. There is a vast ecosystem of exporters for commonly used software systems available. The aim of this thesis is to implement a Prometheus exporter for the Android operating system.

The main focus of this work is to implement an exporter that will not only export available metrics to Prometheus, but also store them on-device during periods of network unavailability and export them in batch once the network connection is available. The solution also focuses on possible energy optimizations of the exporter.

This project is evaluated based on its usability and energy efficiency. I have conducted manual tests regarding the energy efficiency of the final solution.

Prometheus Android Exporter is released as an open-source project on Github.[1] To further increase the usability of the final solution and lower the barrier of entry for its users, the implemented solution can also traverse NAT and similar network barriers out-of-the-box. An example server configuration for such a use case is also provided.

This thesis aims to provide readers with an overview of the Prometheus metric format and its typical operation. Then confronts them with research on current Android monitoring solutions and describes a set of metrics available on Android devices. The rest of this thesis contains the technical design of the application along with explanations of more interesting parts of the implementation.

[1] https://github.com/birdthedeveloper/prometheus-android-exporter

Chapter 2

Prometheus database

Whenever a new system is deployed to a production environment, it is useful to monitor its functions, so that defects in system behavior can be determined and possibly fixed before the system users notice them. Although most monitoring is about system events, for each aspect of a system, the monitoring techniques are slightly different. This is a consequence of events having a context, for example, function call might have a call stack or an HTTP request might have a cookie set, etc. It would be ideal to store the full context of each system event, but this is not practical from the economical point of view as the monitoring system would have to store large amounts of data. Therefore we can roughly categorize monitoring into the four following categories[2]:

- **Profiling** is monitoring most of the applications events with most of the context for limited periods of time. This makes profiling a good tool for debugging software. An example of a profiling tool is tcpdump.
- **Tracing** does not take a look at all events, but rather at a proportion of them. Tracing will note individual functions in the stack trace and how long these functions took to execute. Tracing can be therefore useful for debugging latency in a system.
- **Logging** takes a look at a limited set of events and records some of the context of each event. Logging is typically not sampled.
- **Metrics** track aggregations over time of different types of events, for example, tracking the number of HTTP requests with status code 200 over time.

Originally developed at SoundCloud in 2012, Prometheus is an open-source metrics-based monitoring system written in Go.[2] It is a single statically linked binary that implements a database for received metrics, a query processor, and a subsystem that retrieves metrics from monitored applications. A typical setup consists of one Prometheus instance and multiple scrape targets. Scrape target is in Prometheus terminology an application that needs to be monitored. Applications are monitored by exposing metrics to Prometheus via an embedded HTTP server using a specific metric text format, that is further described in section 2.1. This practice of pulling metrics by the monitoring system rather than applications pushing metrics themselves can be referred to as the pull model.

Using the pull model allows Prometheus to keep information about what targets to scrape and how often in a single configuration file. While this is certainly an advantage, this also creates a need to update such scrape target configuration automatically when new applications are deployed. Such a process is in Prometheus terminology called service

discovery. Prometheus service discovery integration includes common open-source cloud application orchestrators, such as Kubernetes and HashiCorp Nomad, as well as common public cloud vendors, for example, AWS – Amazon Web Services and GCP – Google Cloud Platform.[6] There are also downsides to the pull model, as Prometheus must be able to initiate a TCP connection toward scrape targets, that means that NAT and other network barriers are a problem.

Prometheus has a vast ecosystem of application instrumentation libraries in different programming languages and metric exporters, that stands behind its wide usage in cloud monitoring today. A Prometheus exporter is a program that runs alongside the monitored application and exports application-specific metrics in the right format to Prometheus. An exporter is typically used when direct code instrumentation of the application is not possible, this is usually the case when using off-the-shelf software, for example, PostgreSQL database. Prometheus offers exporters for many popular open-source software projects such as databases and messaging systems.[7]

When an incident happens in a production system, it is practical if the monitoring system itself can send out a notification to humans about this incident. In Prometheus, this is called alerting. It is possible to create alerting rules, that are Prometheus queries that are evaluated continuously. Then thresholds can be set for the results of such alerting rules, and if conditions are met, an alert is generated. In a typical setup, alerts generated from a Prometheus instance do not immediately notify humans, but rather these alerts are processed by another program called Alertmanager, that takes care of grouping the alerts together and consecutive notifications to humans.

Nowadays, there is an ecosystem of projects around Prometheus to help with scaling and long-term storage of metrics. Examples of such projects are Grafana Mimir, that can scale up to 1 billion active metrics series and beyond[1], and Thanos, that is a Prometheus long-term storage backend. There are also commercial offerings of managed Prometheus instances in the cloud, such as Grafana Cloud.

2.1 Metrics format

Both Prometheus exporters and application instrumentation libraries expose Prometheus metrics on a given HTTP port in a text format. Each metric in Prometheus is composed of a name, value, that can be either an integer or a floating point number, and a set of labels. Labels are essentially key-value pairs that provide further information about the given metric. For example, when monitoring the total number of HTTP requests to a given server, the exposed metrics by the HTTP server would look like this:

```
# TYPE http_requests_total counter
# HELP http_requests_total Number of http requests received.
http_requests_total{path="/",method="GET"} 21
http_requests_total{path="/hello",method="POST"} 3
```

The metrics example above depicts metric `http_requests_total`, that is of a type counter, with two labels, `path` and `method`.

Every single combination of label values and metric names is counted in Prometheus as a new metric. Therefore, it is not recommended to have a value with high cardinality such as user IP address or user ID used as a metric label as this can increase the number of

[1] https://grafana.com/oss/mimir/

active metrics significantly. The number of active metrics increases the memory footprint of the running Prometheus instance, that is projected into the operation cost of running a Prometheus.

For the use case of running and monitoring multiple instances of HTTP servers from the previous example, Prometheus adds target labels to already present sets of labels to distinguish between them. Target labels are configurable in the Prometheus YAML configuration file.

The following example shows metrics from the previous example along with one target label `cluster`.

```
# TYPE http_requests_total counter
# HELP http_requests_total Number of http requests received.
http_requests_total{path="/",method="GET",cluster="foo"} 21
http_requests_total{path="/hello",method="POST",cluster="bar"} 3
```

Such usage of target labels by Prometheus suggests there is a set of reserved labels for use in target labels, that one should not use when instrumenting custom applications. These include `env`, `cluster`, `region`, `instance`, and a few more. Labels starting with an underscore are also reserved. A special case is the name of the metric, that is internally represented in Prometheus as a label `__name__`.

There are four metric data types in Prometheus[2]. The first two are simple metrics, they occupy exactly one line in the text format. The second two types of metrics are composed of the first two simple ones.

- **Counter** tracks the number of events. Therefore the counter is an integer. Counters can only go up. Counter metric names typically end with „total“, for example, `http_requests_total`.
- **Gauge** is a snapshot of some current state. For example, a gauge can be used to represent the actual number of items in a queue or current memory usage or current temperature. Gauges are floating point values and can go up and down. Gauge metric names typically end with the name of the used measuring unit, for example, `swap_memory_bytes`.
- **Summary** is useful for measuring average latency in software systems. For example, if a latency of a function `foo()` should be measured by using summary, two metrics would be exported, `foo_latency_seconds_count` and `foo_latency_seconds_sum`. The first metric is a counter describing the number of `foo()` function calls and the second metric is a sum of all the latency from the function calls. From these two data points, the average latency of function `foo()` can be calculated.
- **Histogram** is a useful metric type when further insight into system latency is needed. Histograms are made up of multiple counters, each of that counts how many events have fallen into a given bucket. They allow calculating quantiles, that can be helpful to determine whether for example, software latency is within its contracted SLA – Service Level Agreement, that is often expressed as 95^{th} percentile latency. Histograms are cumulative, that allows dropping histogram buckets if performance becomes a problem due to a high number of metrics, allowing quantiles to still be calculated. Histogram metric names end with measurement unit and „bucket“ and have label `le`, that determines what events does bucket count. An example of such a metric can

be `system_latency_seconds_bucket{le="0.3"}`. This bucket would count all the events that took less or equal to $300ms$.

Previously stated naming conventions for metrics are recommended although they are not mandatory.[8]

2.2 Prometheus exporters

Prometheus exporter is a program that runs alongside an application that should be monitored by Prometheus, but does not expose its metrics in Prometheus format. Usually, there is a need to run an exporter alongside an application when it is not possible to instrument the application directly. An example of such a exporter can be the Node Exporter, that exposes various metrics about the underlying operating system. This is also the case with the Android operating system. Writing a Prometheus exporter that will expose Android metrics is the goal of this thesis.

When writing Prometheus exporters, it is a good practice to follow the rules described in official documentation[11]. Many of these rules will directly apply to the Prometheus Android Exporter application design:

- The exporter should not require custom configuration by the user apart from telling it where the monitored application runs. If a user-provided custom configuration is inevitable for some reason, it is advised to include an example of such configuration with the exporter. Configuration should be in the YAML format.
- Metric names should be concise and prefixed with exporters name. They should also be provided in base units.
- Label names that are likely to conflict with target labels, such as `zone`, `instance`, and so on should be avoided.
- Metrics should make sense when summed or averaged, therefore do not include a metric that sums up all the other metrics with the same name.

2.3 PromQL query language

PromQL is the Prometheus query language. It offers the ability to do all sorts of aggregations, analysis, and arithmetic in order to better understand the performance of the monitored software systems.[2] Prometheus itself comes with a web-based user interface where users can write queries and visualize them in graphs. This UI also features autocomplete for queries and labels as well as autocomplete for present label values.

Describing the whole PromQL syntax is not a goal of this thesis, however, a simple illustration of its usage is in the figure 2.1. To get a deeper insight into PromQL, I would recommend reading the official documentation[9] and possibly trying it out yourself with a local Prometheus Node Exporter instance and a local Prometheus instance.

Suppose there is a metric `node_network_transmit_bytes_total`, that is a counter and two nodes are being monitored, thus there is a target label „node“. One can obtain the metric current values for both nodes simply by typing its name:

```
node_network_transmit_bytes_total
```

Figure 2.1: Prometheus web-based user interface with query and visualization capabilities.

This query will return two time series. Further one can filter out only one node by its label:

```
node_network_transmit_bytes_total{node="10.0.0.1"}
```

To calculate the amount of network traffic transmitted per second, one could use the following query:

```
rate(node_network_transmit_bytes_total{node="10.0.0.1"}[5m])
```

The `[5m]` in the query means to provide the `rate` function with 5 minutes of data, so the returned value will be the average of the last 5 minutes. To get the total rate of transmitted bytes per second for both nodes, one could use the `sum` function:

```
sum without(node) (rate(node_network_transmit_bytes_total[5m]))
```

Apart from querying Prometheus from its web-based UI, there is also an HTTP API that can be used for querying from other tools, for example, Grafana.

2.4 Grafana

When checking the health or performance of the software systems monitored by Prometheus, it might be convenient to have multiple graphs grouped together in a dashboard. Originally, Prometheus used to have its own dashboarding tool called Promdash. Prometheus developers later decided to use Grafana rather than keep developing their own dashboarding platform.[2]

Grafana is an open-source versatile web-based dashboarding tool being developed by Grafana Labs and its open-source contributor community. Although Grafana is open-source software, Grafana Labs offers Grafana as SaaS – Software as a Service and also offers many enterprise plugins for Grafana.

Grafana supports many data sources, such as common relational databases, metric databases, logs, and traces, and has many other integrations. Grafana allows its user to query useful data from the connected data sources and display that data using one of many available visualizations. These visualizations are put together into panels and these are further grouped into dashboards as can be seen in figure 2.2.

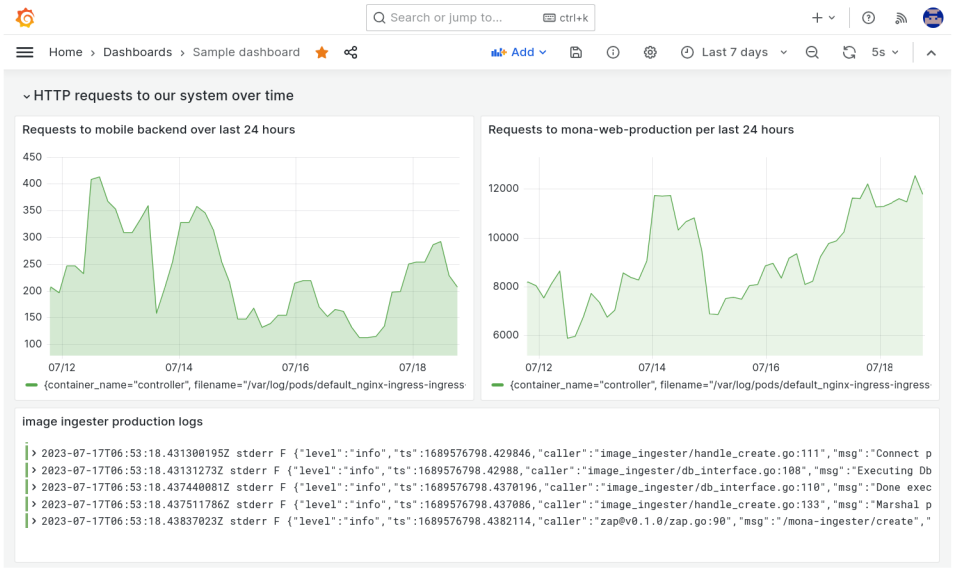

Figure 2.2: Example of a Grafana dashboard with metrics and logs.

To lower the entry barrier, Grafana has visual query builders for many of the data sources it supports. For advanced users who are capable of writing queries as code, Grafana helps by providing auto-complete and support tools such as a label browser, where a user can find all the labels of the data he or she is about to query.

One of the goals of this thesis is to create a Grafana dashboard one can use out-of-the-box with the Prometheus Android Exporter.

Chapter 3

Monitoring Android devices

The motives for monitoring Android devices vary greatly by the use case. When it comes to existing solutions for monitoring Android devices, they can be roughly categorized into the following four categories:

- Unified endpoint management solutions provide its users with visibility into what software is present on-device and how devices are used. It typically allows its users to enforce best security practices and further manage the devices in their inventory. As stated, with these solutions communication flows in and out of the monitored devices.
- Solutions focused on parental control, as parents might want to monitor or limit the activities of their children online.
- Tools for local collection of metrics such as CPU and memory usage via the mobile application UI.
- Tools that export local telemetry to a remote monitoring system, the thesis falls into this category.

3.1 Existing solutions

Bellow are described particular existing solutions for Android monitoring.

- **Home assistant** is an open-source home automation and monitoring tool. Apart from controlling smart devices at home, it is also capable of displaying data retrieved from installed sensors using the ZigBee or MQTT protocol. The whole system is interesting to this research as it also allows the possibility to monitor the hardware sensors of the Home Assistant companion application, that is available for Android. This sensor data is then sent to the Home assistant server.[1]
- **Prometheus Android exporter** is a mobile application that exposes a subset of device metrics in a Prometheus-compatible format. The application is freely available at the Google Play internet store.[2] It is the most similar monitoring solution to the outcome of this thesis.

[1] https://companion.home-assistant.io/

[2] https://play.google.com/store/apps/details?id=info.knacki.prometheusandroidexporter

The application offers a configurable HTTP port via its user interface. Exposed metrics include battery level and status, information about memory, information about available storage as well as brief information about CPU, and network statistics. This solution also does not seem to have any batch exporting or NAT traversal solutions implemented.

- **Miradore** is an enterprise MDM – mobile device management platform. It provides tools to manage the security of Android phones and other devices. Miradore provides its users with very basic information about each device in its inventory, such as a version of the operating system, security information and installed applications.[3]
- **CPU monitor** is a mobile application that locally monitors CPU usage, frequency, and temperature. Users can download it freely from Google Play.[4]
- **AirDroid Parental Control** is an example of a parental control application, one of the most important features of this application is tracking the location of the device and monitoring children's activities in installed applications.[5]

3.2 Available metrics

Various metrics are available on Android devices. Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions.[6] These sensor metrics can be accessed via the `SensorManager` API. Data from sensors such as light meters, proximity sensors, accelerometers, magnetic field detectors or gyroscopes is available.

In accordance with the Android platform's commitment for better user privacy, retrieving a user's location is a little bit more difficult as permission from the user needs to be obtained at runtime. User's location can then be retrieved from the `FusedLocationProvider` API, that is battery-efficient.

Information about CPU usage can be obtained from the `HardwarePropertiesManager` API. The retrieved array of `CpuUsageInfo` structures contains CPU active time and CPU time total for each CPU core. From this information, usage percentage for each CPU core can be calculated. Unfortunately, third party applications cannot tap into this API because of android permissions.

Regarding the user activity, information about that applications are currently running can be retrieved using the `ActivityManager`. Its function `getRunningAppProcesses()` provides an array of active processes with process names attached. Unfortunately, for non-system third-party applications, only their own package name is visible, rendering this method useless.

Information about the state of the network can be accessed via the `ConnectivityManager` API. This includes information about the state of the WiFi and cellular network.

Static information about the device such as the release name of the current version of the Android operating system can be accessed in the `Android.os.Build` package. Device name or manufacturer can be obtained in the same way.

[3] https://www.miradore.com/platforms/android-management/#analytics
[4] https://play.google.com/store/apps/details?id=com.glgjing.stark
[5] https://www.airdroid.com/parental-control/
[6] https://developer.android.com/guide/topics/sensors/sensors_overview

Chapter 4

Design

The goal of this thesis is to design and implement a mobile application for Android that will serve as a Prometheus Exporter. This application will also be capable of temporarily storing metrics on the device when the device is offline, and exporting them later to the Prometheus database when online again. The application will be able to traverse NAT – Network Address Translation and similar network obstacles when exposing data to Prometheus.

This research will be primarily about technologies to traverse NAT while still following the pull model and how to export metrics to Prometheus in batch. Final solution will be evaluated by its usability and energy efficiency.

4.1 Existing tools for building Android applications

As the assignment does not specify any support for other mobile operating systems, the project will be implemented as a native Android application. The application will be implemented in Jetpack Compose, as it is a recommended modern toolkit to build native user interfaces in Android.[1] Historically, Android applications have used Java, but Jetpack Compose supports only Kotlin, therefore the application will be written in Kotlin. Kotlin is a modern open-source programming language developed by JetBrains and its community of contributors. Android development has been Kotlin-first since Google I/O in 2019.[3]

4.1.1 The Kotlin programming language

Kotlin is a cross-platform, general-purpose statically typed programming language. It is mainly used for developing Android applications and server-side applications. Kotlin is fully interoperable with Java and runs on JVM – Java Virtual Machine, as well as a few other runtimes. This subsection illustrates the basics of Kotlin syntax with emphasis on differences from Java and structures used throughout the Prometheus Android Exporter codebase. More information about its syntax rules can be found in the official documentation.[3]

Variables in Kotlin are declared with `val` or `var` keyword. The difference between the two is that a variable created with `val` is immutable, and hence can be assigned only once. Variables created with `var` are mutable and can be assigned multiple times. When there is a need to initialize a variable after its declaration, the `lateinit` modifier, that marks the variable as initialized later in the code, comes in handy.

[1] https://developer.android.com/modern-android-development

Functions in Kotlin are declared using the `fun` keyword. The return type is specified after the list of parameters. Functions in Kotlin do support generics.

```
fun double(x: Int): Int {
    val coefficient : Int = 2
    return coefficient * x
}
```

Kotlin has strong syntax support for anonymous functions. Anonymous functions are functions that are not declared, but rather immediately passed on as an expression. Parameter declarations in the full syntactic form go inside the curly brackets and have optional type annotations. Jetpack Compose leverages this syntax a lot, that is described in the next subsection. Kotlin also provides a keyword `it` to reference the parameter of an anonymous function if such a function has only one parameter.

Kotlin aims to avoid null dereference exceptions by implementing null safety. With null safety, the type system distinguishes variables that can have a null value and variables that cannot. Declared variables can be marked as nullable by appending a question mark after the name of their type. If a variable is not explicitly declared nullable, it cannot be null.

When making network requests or other blocking calls from the UI application code, it is undesirable to freeze the UI for the user. Kotlin's approach to asynchronous code is using coroutines. Coroutines are essentially very lightweight threads. In other words, coroutine is a function that can suspend its execution and resume it later on, while not allocating any heavy system resources such as threads do. Kotlin coroutines are not preemptive.

In Kotlin, the way to enter asynchronous code from synchronous code is by using a `runBlocking { }` block, that blocks the thread until whatever is run in that block completes. To launch a coroutine inside `runBlocking { }`, one can use the `launch { }` block or the `async { }` block. The difference between the two is that `launch { }` behaves as fire-and-forget, it just runs the new coroutine and immediately moves to the next line. The `async { }` on the other hand returns the result of a new coroutine as `Deferred`, and one must use keyword `await` on it. When inside a coroutine, it is possible to start multiple coroutines at the same time using a `coroutineScope { }` block.

All asynchronous functions in Kotlin are marked with the `suspend` keyword. Calling suspending functions from other suspending functions is simple, there is no special syntax, so the function call looks like a synchronous one.

Kotlin handles runtime errors by throwing exceptions like in java. Exception classes in Kotlin inherit the `Throwable` class. Kotlin, unlike from java, does not have checked exceptions.

Kotlin has replaced static class members in java by having a `companion object{}`. Declaring any class member inside a companion object in Kotlin will have the same results as marking it `static` in Java. Kotlin has a `data class` modifier, that automatically generates `copyWith(...)` method, that is very useful when dealing with immutable objects.

4.1.2 Jetpack Compose

Jetpack Compose is a modern toolkit for building native Android user interfaces. Jetpack Compose simplifies and accelerates UI development on Android with less code, powerful tools, and intuitive Kotlin API.[2]

[2] https://developer.android.com/jetpack/compose

Jetpack Compose is built around composable functions. To create a composable function, just mark a function with a `@Composable` annotation. These functions are used to define the application UI programmatically by describing how it should look and providing data dependencies, rather than focusing on the process of creating the UI. Composable functions themselves should be side-effect free, as when and in what order composition happens is unpredictable. To provide an example, here is a composable function from the codebase, that renders a column with rendered text and a progress indicator.

```
@Composable
private fun LoadingPage(
    modifier: Modifier
) {
    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        modifier = modifier,
    ) {
        Spacer(modifier = Modifier.height(50.dp))
        Text(text = "Checking for configuration file")
        Spacer(modifier = Modifier.height(20.dp))
        CircularProgressIndicator(modifier = Modifier.size(size = 36.dp))
    }
}
```

The example can be confusing at first since the `Column` function has as the last argument a lambda function, Kotlin allows this lambda function to be specified outside of the parenthesis. This lambda function is also the way to add children composables to a composable function.

A state is any property of the application that changes while the application is running. The simplest way to create a state variable in a composable function is to create it with the function `mutableStateOf()`. The state variable data type should be immutable.

```
    val mutableState = remember { mutableStateOf("Default value") }
```

This approach is usable when making a single composable function stateful, such as a simple text field, but is not suitable for application-level state management. Furthermore, it makes the stateful composable function harder to test and to reuse. To solve this, state can be moved up the tree of composable functions to parent composable and state variables can be provided to the child composable function as parameters. This process of moving the state to parent composables is called state hoisting.

To separate business logic from UI code, the `ViewModel` class can be used. A custom view model is created by extending the `ViewModel` class. View model is used with `LiveData`, that holds the state. State can be then observed by calling the `observeAsState()` function on the Live Data object. Furthermore, a view model can contain all the functions that mutate the state. This helps with separation of concerns in the codebase.

4.2 NAT traversal solutions

With NAT – network address translation enabled routers, there is a problem with initiating a connection towards them from the outside world. [4]

When monitoring infrastructure such as a kubernetes cluster or a few servers, the monitored system and Prometheus instance are typically on the same network. This may not be necessarily the case with Android devices, as they are mobile and can get behind a network barrier such as NAT. Therefore this project should provide a solution to traverse NAT and other network barriers out-of-the-box.

Prometheus Pushgateway is a metrics cache for service-level batch jobs.[2] As batch jobs are usually not running continuously, Prometheus cannot scrape them. Therefore, instead of batch jobs exposing metrics to Prometheus, they actively push them to Pushgateway instead. Typical architecture is described in figure 4.1.

Figure 4.1: The Pushgateway architecture. Metrics are pushed to Pushgateway from batch jobs. Pushgateway caches these metrics and is scraped periodically by Prometheus.

The Pushgateway is not a way to convert Prometheus from pull to push. For example, if there are several pushes to the Pushgateway between Prometheus scrapes, the Pushgateway will only return the last scrape.[2] Therefore, even though the Pushgateway-based solution would effectively traverse NAT, as all connections would originate from the mobile device, it is not suitable for this task. Trying to convert Prometheus to push-model is undesirable.

Another possible solution would be to use remote-access VPN – Virtual Private Network. This approach would also enhance the security of the application, even though this is not the main aim of this thesis. In this case, the Android device would connect to the VPN to be on the same network as Prometheus instance, allowing it to scrape the device. A lightweight solution such as WireGuard could be used. WireGuard even offers a Android Java SDK.[3]

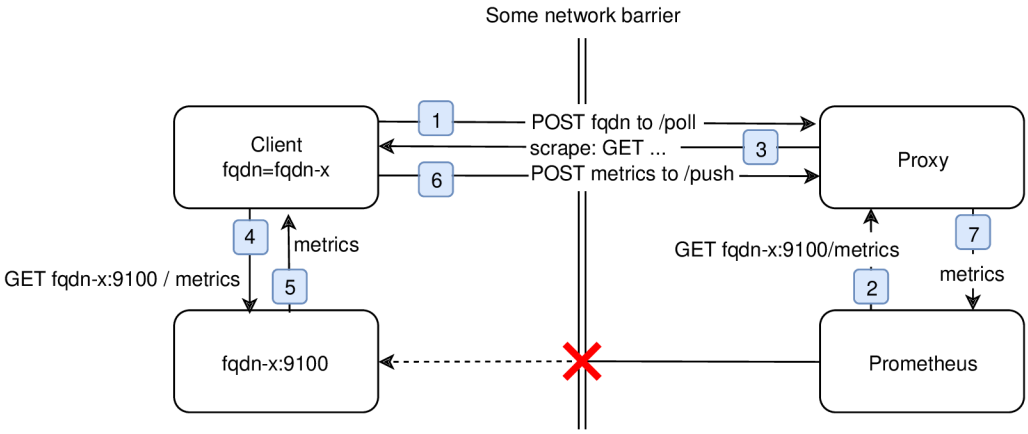

Figure 4.2: The PushProx architecture. Taken from [12]. PushProx traverses NAT by initiating a TCP connection by making an HTTP /poll request (1).

Prometheus PushProx is a client and proxy written in Go that allows traversing of NAT and other similar network topologies by Prometheus, while still following the pull model.[12]

[3] https://www.wireguard.com/

PushProx works by having the PushProx client initiate a TCP connection towards PushProx proxy, assuming that the PushProx client is run on the same network as the monitored application and the PushProx proxy is run besides Prometheus.

First, PushProx client initiates a TCP connection by sending the /poll request (1). PushProx proxy waits until Prometheus scrapes it (2). Then, the PushProx proxy responds to initial /poll request with the original scrape request from Prometheus as the request body. PushProx client then scrapes the monitored application (4) and pushes metrics to the PushProx proxy by sending an HTTP /push request (6). Subsequently, metrics are returned to Prometheus as a response (7) to the initial scrape request (2). The whole process is depicted in figure 4.2.

4.3 Application design

The mobile application is designed to work in these three modes:

- As a Prometheus exporter, by exposing metrics in text format on the configured HTTP port, the default HTTP port is provided.
- As a Prometheus PushProx proxy client, that can traverse NAT and other network barriers while still following the pull model. This mode requires a PushProx proxy server to be run besides Prometheus. The PushProx server URL and the fully qualified domain name must be configured on the mobile application for this mode to work.
- As a batch exporter, that is capable of scraping the device when offline and of subsequent export to the Prometheus instance in batches via Prometheus remote write protocol. Metrics are stored offline only in memory, they are not persisted to disk. There is a limit to how old the exported metrics can be. This limit is internally set by Prometheus to roughly one hour and is not configurable. In my opinion, this limit is connected to the way Prometheus internally stores metrics in its database.

These modes can be turned on simultaneously, with the one exception being that to turn on batch exporter, one of the other modes must be turned on as well. The batch exporter only scrapes the device when there were no successful remote scrapes in a configured time interval, and in order to determine this, PushProx client mode or exporter mode must be turned on.

4.4 Application usage

Application usage is very straightforward, with only minimal manual configuration required, as mentioned in the instructions on how to write Prometheus exporters.[11] The application is configurable either via its UI or via the local YAML configuration file. Once configured, the user can start the monitoring by tapping the start button. The monitoring worker then runs in the background, and even survives a reboot of the device.

I have decided to implement configuration UI to lower the barrier of entry to use the Prometheus Android Exporter and also to simplify proof-of-concept tests for potential users. How the user interface looks is displayed in figure 4.3. It is expected that users with more than a handful of devices to monitor will configure the Prometheus Android Exporter via the YAML configuration file. Even when using the YAML configuration file, monitoring has to be started manually by opening the application and tapping the start button.

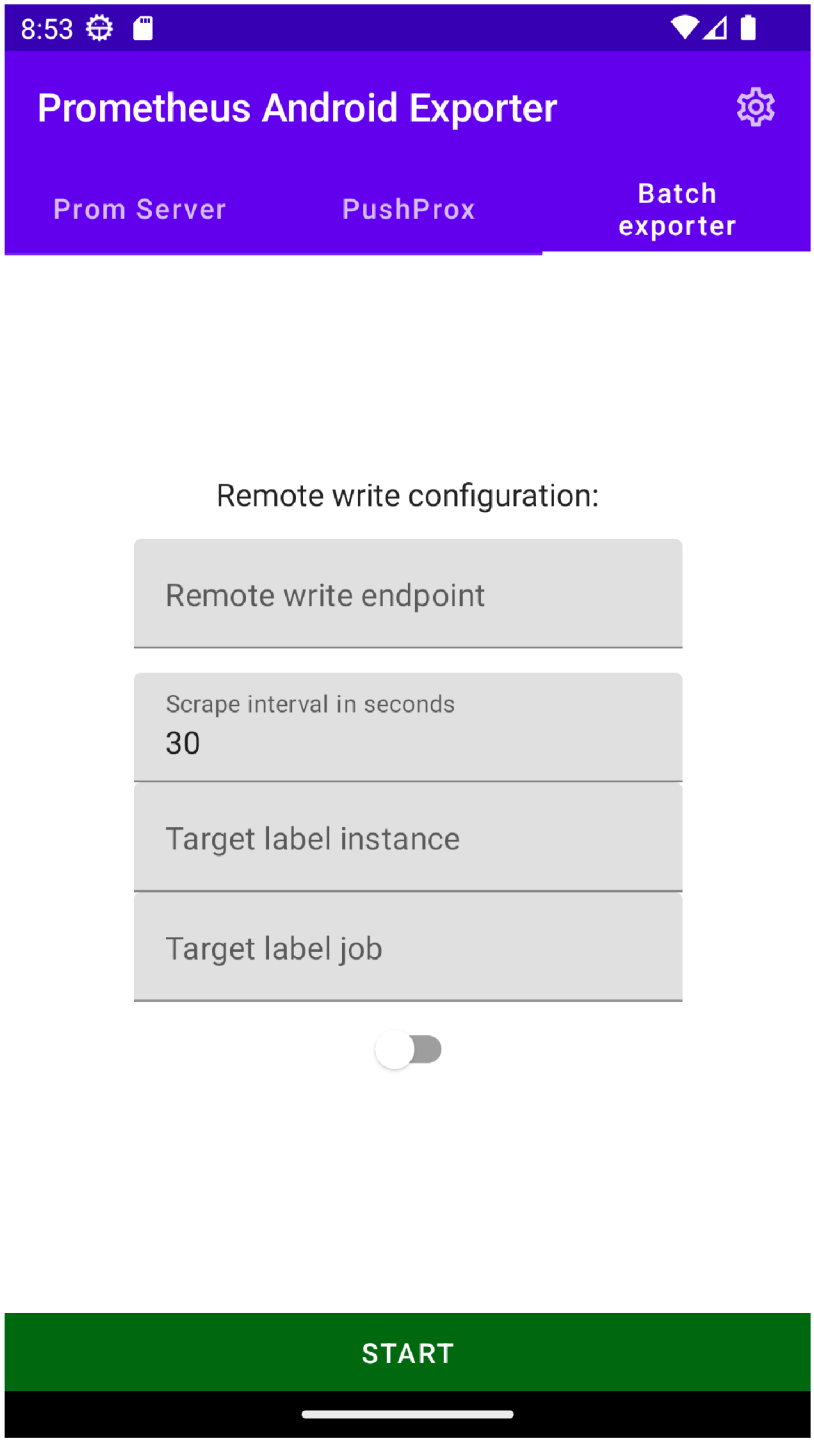

Figure 4.3: Prometheus Android Exporter user interface. The homepage is divided into three tabs, each tab contains configuration settings for one of the application modes. The batch exporter configuration is shown on this screenshot.

The user interface behaves differently if the local configuration file is present. If it is present, user can see only the loaded configuration. If it is not present, the user will be presented with a tab view with three tabs, one tab for each application mode.

Switch on each tab determines whether this mode should be turned on when the user turns on monitoring. Monitoring runs in the background using an Android WorkManager worker. The Android WorkManager is a standard part of Android Jetpack, that is a set of standard libraries around Jetpack Compose and is further described in section 5.1.

The YAML configuration file should be put at the following path and the name of the file should be `config.yaml`.

```
/data/user/0/com.birdthedeveloper.prometheus.Android.exporter/files/
```

Here is an example of such configuration file:

```
# file: config.yaml
# Configuration of prometheus exporter
prometheus_server:
  enabled: true
  port: 10101 # http port

# Configuration of the batch exporter, that uses the remote write protocol
remote_write:
  enabled: true
  remote_write_endpoint: "http://143.42.59.63:9090/api/v1/write"
  instance: "test"
  job: "Android phones"
```

In this particular configuration example, Prometheus metrics will be accessible on HTTP port 10101, also the application will check regularly whether there were successful scrapes. If a device network connection is lost or the Prometheus instance goes offline, the application will start scraping metrics itself and once online, the application will try to export metrics using the remote write protocol to the configured `remote_write_endpoint` with additional labels `job` and `instance`. A full list of options is provided in the code repository in the `config_file_structure.yaml` file.

Chapter 5

Implementation

Prometheus Android Exporter is written in the Kotlin programming language using Jetpack Compose. The application consists of three modes as described in chapter 4.3: Prometheus exporter, PushProx client, and batch exporter. This chapter describes parts of the implementation of these three modes that are most interesting or were more challenging to implement than the others.

5.1 Application architecture

State management in the application is done by a single top-level `ViewModel` implemented in the file `PromViewModel.kt`. This ViewModel stores and updates the configuration for the monitoring worker. This means knowing what modes should be turned on when the user taps the start button and so on.

All three modes are implemented by a single Android WorkManager worker. This worker runs in the background and can even survive a reboot of the device. Configuration passed on to this worker must be serialized, as Android WorkManager stores it in its database. Hence, this worker can be started at any time independently of the rest of the application. This worker is implemented in the file `PromWorker.kt`

Android WorkManager is the recommended solution for persistent work. Work is persistent when it remains scheduled through app restarts and system reboots. WorkManager is the primary recommended API for background processing on Android.[1]

Prometheus Android Exporter uses the Ktor HTTP server, used in Prometheus Exporter mode and Ktor client, used in the two other modes.

Thanks to Kotlin's interoperability with Java, the application is able to use the Prometheus Java client library to register metrics and convert them to Prometheus text format. The library provides usable API for adding all four types of Prometheus metrics

Hardware metrics are collected by leveraging the `SensorManager` API. There is no direct access to the sensors, but one can subscribe to events when the sensor value changes. Beware that on an Android emulator, an event is not generated when a listener is registered to the sensor, the sensor value has to be changed first. Other metrics are collected via their respective API as described in section 3.2.

While the set of metrics that Prometheus Android Exporter exposes is fairly large, the application design makes it easy to add other metrics. Custom metrics can be appended in the file `AndroidCustomCollector.kt` to function `collect()`.

[1] https://developer.android.com/topic/libraries/architecture/workmanager

The application leverages third party Java and Kotlin libraries for Prometheus instrumentation, Snappy compression algorithm, and YAML serialization.

On some versions of the Android operating system and with specific mobile phone vendors, it can be necessary to except the Prometheus Android Exporter from any battery optimizations. For example, LG with Android 9 requires this for the right functionality of the Prometheus Android Exporter.

5.2 Kotlin coroutines

To run multiple application modes simultaneously, the monitoring worker launches multiple Kotlin coroutines – coroutine for each active application mode – in parallel. Kotlin coroutines are not confined to one thread by default, in fact, they can be scheduled to multiple threads throughout their lifecycle. This creates potential race conditions at the boundaries in code where particular application modes meet.

The only information that is shared between application modes is counting successful metric scrapes. A successful metric scrape is basically an event, that is generated by either the Prometheus exporter or the PushProx client, and is sent to the batch exporter. Based on this information, the batch exporter determines whether it should start or stop scraping metrics locally.

I have decided to solve this problem not by adding a mutex, but by confining the coroutines to one thread only by creating a fixed count thread pool for all the coroutines. When thread count is set to 1, this effectively limits parallelism, as Kotlin coroutines are not preemptive. A code example is below:

```
val backgroundDispatcher = newFixedThreadPoolContext(1, "test_pool")
val threadContext =  backgroundDispatcher.limitedParallelism(1)

// launch multiple coroutines in parallel on one thread
withContext(threadContext){
    launch { print("Coroutine 1") }
    launch { print("Coroutine 2") }
}
```

The other argument for choosing thread confinement instead of mutexes was the fact that batch exporter, that is implemented by a class `RemoteWriteSender`, also launches two coroutines in parallel. One coroutine is used for controlling the conditions of whether the batch exporter should be locally scraping metrics, and if conditions are met, this coroutine also scrapes the metrics. The other coroutine takes care of sending metrics to the remote write endpoint, with an exponential backoff. There are shared resources between the two.

5.3 Traversing NAT using the PushProx proxy

I have chosen to implement NAT traversal using the Prometheus PushProx proxy, as it is the most straightforward solution, and it does work while still following the pull model. The PushProx client used in this project is a rewritten simplified version of the original written in the Go programming language, and is implemented in the file `PushProxClient.kt`.

PushProx client implemented in the application works by making a /poll HTTP request to the PushProx proxy server first. Then PushProx server waits for Prometheus to perform

a metrics scrape using it as a proxy. Metrics scrape is forwarded to the PushProx client, that then sends scraped metrics to the PushProx proxy server using the /push HTTP request. Then PushProx proxy server returns scraped metrics received from the client to Prometheus instance. The whole process is depicted in figure 5.1.

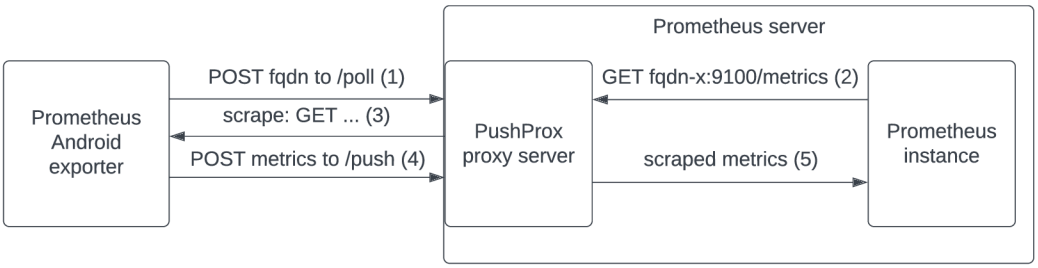

Figure 5.1: Depiction of order in which HTTP requests are made to perform one metrics scrape via PushProx proxy. This setup traverses NAT and other network barriers.

In the original implementation, there are 3 metrics for measuring the PushProx client itself. I have decided to keep these metrics. These metrics consist of three error counters that count poll errors, push errors and scrape errors.

This solution could be further optimized for energy efficiency by rewriting the PushProx proxy with WebSockets. The WebSocket protocol enables two-way communication between a client and a remote host. WebSocket protocol is a standalone protocol implemented on top of the TCP protocol, although it is negotiated via HTTP.[5] There is a one potentially unnecessary HTTP request with PushProx as essentially, assuming the connection was already established, only request from the Prometheus instance and consecutive response from the client with metrics is all that is needed. For this solution to be implemented, the PushProx proxy would need to be rewritten using WebSockets.

5.4 Prometheus remote write protocol

Prometheus remote write capability is used when there is a need to aggregate metrics data from multiple Prometheus instances. For example, there might be two datacenters, each with its own Prometheus instance, and there may be a need to aggregate the data from both datacenters into one Prometheus instance, on which the queries will be performed. To achieve this, there is a need for a protocol that will efficiently propagate scraped metrics from both datacenters to that one Prometheus instance. The remote write protocol is designed to make it possible to reliably propagate samples in real-time from a sender to a receiver, without loss. [10]

Prometheus is not the only metrics system that understands remote write protocol. In fact, most of the Prometheus long-term storage backends do support receiving metrics in the remote write protocol format. Typical usage of the remote write protocol is displayed in figure 5.2.

The official remote write protocol specification explicitly states that the remote write protocol is not intended for use by applications to push metrics to Prometheus remote-write-compatible receivers. This means the solution at hand is more of a hack and while it works, it is not officially supported. For example, I would not be surprised if there were any performance issues while a large number of Prometheus Android Exporters would be

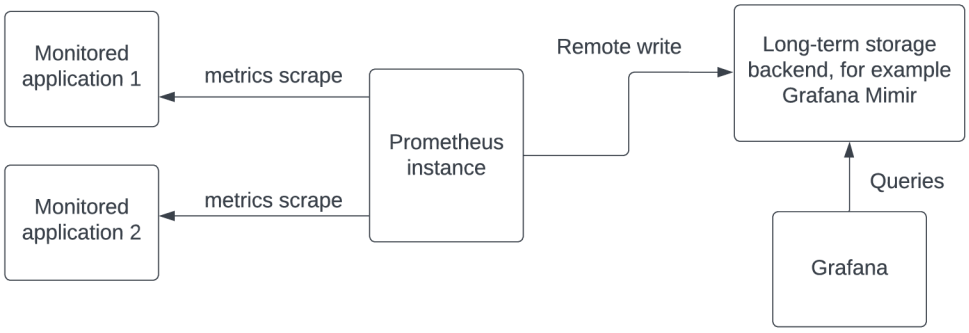

Figure 5.2: Typical usage of the Prometheus remote write protocol to transmit metrics from one Prometheus instance to another or to a long-term storage backend.

pushing metrics to a single Prometheus instance. Furthermore, the maximum age of batch-exported metrics is set to roughly one hour and this limit is not configurable. The pull model is deeply ingrained into the design of Prometheus and with batch exporting, I am essentially trying to convert it to push.

Prometheus remote write protocol communicates over HTTP protocol and the interface is described using protocol buffers. Remote write protocol messages are for better efficiency compressed using Google's Snappy algorithm. Protocol buffers are open-source language-neutral, platform-neutral extensible mechanisms for serializing structured data. There is official support for common programming languages such as C++, Go, Dart, Python, and many more. There is also a third party support for many other languages, this is the case with Kotlin.

With Google's protocol buffers, one can define how the data being transmitted between two services should be structured. Then with help from protocol buffers CLI – Command Line Interface tool, `protoc`, one can generate code in the target programming language. This generated code can be used to serialize and deserialize structured data.

Apart from serializing into JSON – JavaScript Object Notation, protocol buffers can also serialize to the wire format, which is a binary representation. This representation is obviously much more space-efficient and therefore is used by remote write protocol.

The data structure is defined in files with the `*.proto` extension. Protocol buffers use simple language to describe message structures. To describe the language a bit further, messages are structures enclosed in curly brackets and may contain simple data types such as `string`, `int32`, and `int64`, lists, that start with the keyword `repeated`, or other messages. Furthermore, message fields can be marked as optional with the `optional` keyword.

The numbers on the other side of the equation uniquely identify the given message field. As serialized protocol buffers data can be used for storage, it has language tools to preserve backward compatibility. For example, if a field with number 2 was used in the previous version of the protocol buffers structure, but is currently removed, one can reserve it using the `reserved` keyword. This ensures that no one in the future will create a message field with the same identifying number but different semantics.

Here is a concrete example of protocol buffers that describe the Prometheus remote write protocol.[10]

```
message WriteRequest {
  repeated TimeSeries timeseries = 1;
  // Cortex uses this field to determine the source of the write request.
  // We reserve it to avoid any compatibility issues.
  reserved  2;

  // Prometheus uses this field to send metadata, but this is
  // omitted from v1 of the spec as it is experimental.
  reserved  3;
}

message TimeSeries {
  repeated Label labels   = 1;
  repeated Sample samples = 2;
}

message Label {
  string name  = 1;
  string value = 2;
}

message Sample {
  double value    = 1;
  int64 timestamp = 2;
}
```

Remote write protocol messages consist of a list of time series, that each consists of a list of labels and a list of samples. A label is a pair of a name and string value and a sample is a pair of a value and a unix timestamp. The metric name is sent as the reserved label `__name__`.

5.5 Metrics batch export using the remote write protocol

My initial idea was to start scraping metrics on the device and save them to the memory when the device goes offline, e.g. when there are no successful scrapes for a given time interval. I have decided not to implement permanent storage for metrics, as the maximum age of a metric that Prometheus accepts via remote write protocol is about one hour. With a maximum age limit this low, permanent storage would not have much of an effect on the durability of the whole system.

The problem is that after the device has been offline for a while, there are lots of metric samples in the device memory, and when the device goes online again, that is detected by the application as a successful scrape, the metrics cannot be exported. The reason is that given the low-level implementation of the Prometheus database, time series can be only appended to – they are sorted by timestamps. So if a successful scrape happens, it essentially blocks all the locally stored metrics. HTTP request to remote write endpoint will return HTTP status code 400 – Bad Request in such case.

The solution I came up with is to differentiate between metrics regularly scraped by Prometheus and metrics exported in batch by adding a label. I have added a label `backfill`,

that is either „true“ or „false“ depending on whether the given metric was exported in batch or not. There is a serious downside to this solution that I have realized after I have implemented it. Adding another label with two values essentially doubles the number of active time series within the Prometheus database. If the user is using any commercial Prometheus hosting where he is billed by the number of active metrics, such as Grafana Cloud, the bill essentially doubles.

Now that the user is left with two metrics, each with a different `backfill` label value, instead of one. To merge these metrics, one can average them in PromQL like this:

```
avg(<name of the metric>) without(backfill)
```

Prometheus keeps the value of the last metric received via remote write for another roughly five minutes. I did not find the reason why it does this. To filter this out, one can use a bit more complex PromQL construct:

```
sum(<name of the metric>{backfill="false"}
or
on() <name of the metric>{backfill="true"}) without (backfill)
```

Unfortunately, the second solution increases the complexity of creating Grafana dashboards and generally writing PromQL queries significantly.

There is also a problem with target labels, metrics that Prometheus has received using the remote write protocol do not have any target labels as the Prometheus instance does not have any other information about the received metrics than what is already present in protocol buffer messages. I have partially solved this problem by adding `instance` and `job` labels into the configuration of the batch exporter.

The batch exporter determines whether to start scraping metrics locally by tracking the last time such a scrape has happened. If the last successful scrape has been more than `1.7 * scrape_interval` seconds ago, the batch exporter starts scraping metrics locally. To turn off local metric scraping, multiple successful scrapes must happen in time. This creates a hysteresis in the system as a protection against unstable network connections. Essentially, it is easier for the batch exporter to turn scraping on than it is to turn scraping off. The memory store caching the times of recent successful scrapes for the implementation of the hysteresis is implemented as a ring buffer.

After realizing all the problems this design had caused, if I had the opportunity to solve this problem again, I would have solved it differently.

The batch exporter consumes events from PushProx client and Prometheus exporter to determine whether there was a successful scrape. The better solution is to intercept this scrape and before an HTTP request would be sent, the batch exporter needs to export all stored metrics, if it has any. This solution only removes the need for the `backfill` label, target labels are still a problem with this solution. It is illustrated in figure 5.3.

5.6 Energy consumption considerations

One of the goals of this thesis is to optimize this application in regard to its energy efficiency. One way of optimizing the application is to look at its network usage.

In HTTP/1.0 protocol, each connection is established by the client prior to the request and closed by the server after sending the response. The newer protocol HTTP/1.1 have changed this, having persistent connections by default. This enables sending multiple HTTP

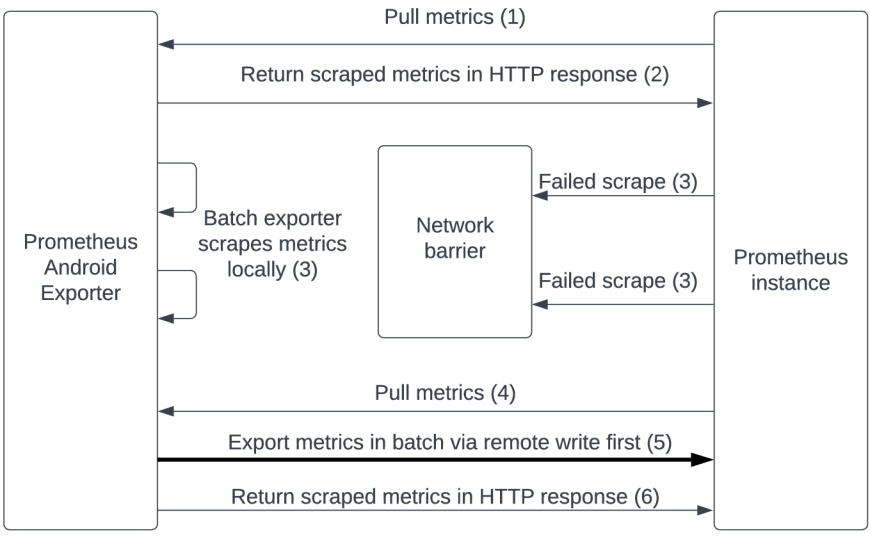

Figure 5.3: Better design of exporting metrics from the batch exporter. Before another regular successful scrape happens, the batch exporter needs to make sure all the metrics have been exported, as they need to be stored in the Prometheus database in chronological order. This solution is not implemented.

requests and responses over one TCP connection, thus greatly reducing the overhead of initializing TCP connections.

The Ktor `CIO` coroutine based HTTP client engine, that I have used initially, does not support persistent connections. This is not documented limitation of the `CIO` engine. The persistent connections do work with the Ktor `Android` client engine.

Searching for a cell signal is one of the most power-draining operations on a mobile device. A best practice for user-initiated requests is to first check for a connection using the `ConnectivityManager`. If there's no network, the app can save battery by not forcing the mobile radio to search.[2] Checks for a connection before making any HTTP requests are implemented in the PushProx client mode and the batch exporter mode.

5.7 Server example configuration

One of the measures of success for open-source projects is how easy it is for future contributors to run the code at hand. In order to lower the barrier of entry for possible future contributors, example server configuration for Prometheus, Grafana, and PushProx proxy server are provided as a part of the code repository. The general aim is to provide an Ansible playbook that can be run against a GNU/Linux server to install and configure everything needed at once. In order to describe the playbook, a few technologies must be explained first.

Docker is a container virtualization technology. So, it's like a very lightweight virtual machine. It addresses the problem of packaging software, as the binary and all dependent

[2] https://developer.android.com/training/connectivity/network-access-optimization

libraries can be packaged with docker into one container.[1] Example configuration consists of four containers, PushProx proxy server, Prometheus, Grafana, and Nginx reverse proxy, behind that all the other services run.

Docker compose is the tool used to run multiple containers at the same time. With docker compose, one can in a declarative manner define what containers should run on a system in a YAML configuration file.

Ansible is an open-source simple IT automation system. It handles configuration management, application deployment, and other automation tasks.[3] Ansible is used by creating so-called Ansible playbooks. Each playbook is a series of steps that are executed on a given server when such a playbook is run. Ansible playbooks are idempotent, meaning that a playbook can be run on the same server multiple times without ruining any already present configuration. Ansible uses YAML to describe the playbooks.

Both previously described tools are designed with the IaaC – Infrastructure as a Code design principle in mind. That means configuration is rather declarative than imperative. This has many benefits, for example, one can see exactly what state is the server is in just by looking at the code that was used to configure that server.

Ansible playbook that is present in the code repository in folder `./server` first installs docker on the new machine, then it creates a new user and copies configuration files over to the server. Then it creates and starts a systemd service to start the four docker containers on the machine boot using docker compose.

More detailed information on how to run the playbook can be found in `README.md`.

[3] https://docs.ansible.com/

Chapter 6

Testing

In this chapter, the performed energy efficiency tests are described as well as the Battery Historian tool, that was used to analyze test data.

Manual tests were conducted to prove the energy efficiency of the device and the benefits of the energy optimizations of this application. To ensure the functionality of the project, a continuous integration pipeline was set up on the CircleCI platform. Unit tests in the code are thus run in the pipeline in the cloud on every git commit.

Energy efficiency tests were conducted on a slightly modified application, exporting only one metric, to focus on carried-out energy optimizations. As energy optimizations are only related to networking and hardware sensors take a lot of energy, I was worried they would invalidate the tests.

Tests were conducted on an LG G6 real Android device with an Android 9 version of the operating system. During the tests, the Prometheus scrape interval was set to 15 seconds. For the tests, the application was run in a release mode and was using a WiFi network connection. All energy efficiency tests were run for 1 hour.

The energy efficiency tests did not only provide insight into battery usage but also proved the usability of the application, as from the one-hour windows when the tests were run, metrics were being continuously exported with no downtime.

Unfortunately, later manual functionality tests have shown that exporting sensor data is limited in the current configuration by the Android operating system to 10 minutes.

6.1 The Battery Historian tool

Battery Historian is a web-based UI program distributed in a docker container for analyzing reports from the battery. [1] It is used to analyze bug reports created via the ADB – Android Debug Bridge CLI tool.

Battery Historian also shows useful information about activities of the device that drain the battery, such as WiFi activity, audio activity, mobile signal strength, and many more. The information shown can be viewed for either particular processes or the system as a whole.

The data shown in the following test reports were gathered from this tool.

[1] https://developer.android.com/topic/performance/power/setup-battery-historian

6.2 PushProx client energy consumption

Tests were conducted for a variant of the application with HTTP keep-alive configured and without HTTP keep-alive configured to determine, what impact does it have on overall energy efficiency. The results of the tests are summed up in table 6.1.

HTTP keep-alive	Test duration	CPU user	CPU system	Estimated power use
enabled	1 hour	150 seconds	178 seconds	0.85 %
disabled	1 hour	286 seconds	224 seconds	1.00 %

Figure 6.1: Results of the PushProx client energy efficiency tests.

As it can be seen from the results, with HTTP keep-alive turned on, the estimated power use has dropped by 15 %.

6.3 Prometheus exporter energy consumption

Prometheus exporter ran with HTTP keep-alive enabled. The power usage results are more or less the same as the PushProx client with HTTP keep-alive enabled. Test results are summed up in table 6.2.

Test duration	CPU user	CPU system	Estimated power use
1 hour	229 seconds	188 seconds	0.86 %

Figure 6.2: Results of the Prometheus exporter energy efficiency test

Chapter 7

Conclusion

The aim of the work was to design and implement a Prometheus Exporter for Android devices, that would export available metrics both continuously and in a batch. The assumed functionality included storing metrics on-device in case of temporary unavailability of the network connection and exporting them to the Prometheus database later.

The mobile application for Android devices was successfully implemented according to the architecture proposed in this thesis. Implemented application is configurable using either its user interface or configuration file and can work in the following three modes. As a Prometheus exporter, that exposes metrics on a specific HTTP port, as a client for the PushProx proxy, this mode allows the application to traverse NAT and similar network topologies, and as a batch exporter. Exporting metrics in batch in case of network unavailability is unfortunately severely limited by the maximum age of ingested metrics set by Prometheus.

Work was evaluated using manual energy efficiency tests. Thanks to the energy optimization described in this thesis, in the PushProx client mode the estimated power use was lowered by 15 %. However, it has turned out that in the current configuration, the Android operating system limits retrieving data from sensors in the background to roughly 10 minutes. The implemented application exports over 35 metrics regarding hardware sensors, computing resources usage, system information, and user activity. Continuous integration is also set up to ensure the functionality of code in every git commit.

Work was published as an open-source project under the Apache 2.0 license on Github.[1] Sample Grafana dashboard for the metrics exported by this project is available as a file in the JSON format in the code repository and was published on Grafana dashboards as well.[2] The default HTTP port of the Prometheus exporter mode of the application was registered within the Prometheus foundation.

As possible extensions, I would suggest implementing root access for the application, to make it possible to gather more metrics from the device. I would also suggest using a custom websocket proxy instead of used open-source PushProx proxy, as it would remove one HTTP request on each scrape from the current implementation, making the application more energy-efficient. Expanding the on-device configuration of target labels for the batch exporter would be also a meaningful improvement in the usability of the batch exporter when monitoring a larger number of devices.

[1] https://github.com/birdthedeveloper/prometheus-android-exporter
[2] https://grafana.com/grafana/dashboards/19255-prometheus-android-exporter/

Bibliography

[1] Anderson, C. Docker [Software engineering]. *IEEE Software*. 2015, vol. 32, no. 3. DOI: 10.1109/MS.2015.62.

[2] Brazil, B. *Prometheus: up & running : infrastructure and application performance monitoring*. First editionth ed. Sebastopol, CA: O'Reilly Media, 2018. ISBN 978-1-492-03414-8.

[3] Kotlin foundation authors. *Kotlin for android: Google Play* [online]. 2021. 2023-05-11. Available at: https://kotlinlang.org/docs/android-overview.html.

[4] Kurose, J. F. and Ross, K. W. *Computer networking: a top-down approach*. Eighth editionth ed. Harlow: Pearson Education Limited, [2022]. ISBN 978-1-292-40546-9.

[5] Melnikov, A. and Fette, I. *The WebSocket Protocol* [RFC 6455]. RFC Editor, december 2011. DOI: 10.17487/RFC6455. Available at: https://www.rfc-editor.org/info/rfc6455.

[6] Prometheus Authors. *Configuration: Prometheus* [online]. 2014. 2023-07-04. Available at: https://prometheus.io/docs/prometheus/latest/configuration/configuration/.

[7] Prometheus Authors. *Exporters and integrations: Prometheus* [online]. 2014. 2023-05-31. Available at: https://prometheus.io/docs/instrumenting/exporters/.

[8] Prometheus Authors. *Metric and label naming: Prometheus* [online]. 2014. 2022-05-30. Available at: https://prometheus.io/docs/practices/naming/.

[9] Prometheus Authors. *Prometheus - Monitoring system & time series database* [online]. 2014. 2023-07-14. Available at: https://prometheus.io/.

[10] Prometheus Authors. *Prometheus Remote-Write Specification: Prometheus* [online]. 2014. 2023-06-05. Available at: https://prometheus.io/docs/concepts/remote_write_spec/.

[11] Prometheus Authors. *Writing exporters: Prometheus* [online]. 2014. 2023-05-31. Available at: https://prometheus.io/docs/instrumenting/writing_exporters/.

[12] PushProx authors. *Prometheus-community/PushProx: Proxy to allow Prometheus to scrape through NAT etc.* [online]. 2017. 2023-06-19. Available at: https://github.com/prometheus-community/PushProx.