Comprehensive Guide

gRPC WITH KOTLIN

Team Members:

- Ahmed Yehia
- Merna Islam
- Mohannad Hisham
- Noureldin Ahmed
- Rana Essam

Supervised By: Dr. Ahmed Shawky

Faculty of Computers and Artificial Intelligence, Cairo University

Table of Contents

Overview

This document serves as a comprehensive step-by-step guide for building a **modular, scalable, and distributed gRPC system using Kotlin** for Android. It is tailored for developers and researchers aiming to implement efficient communication and computation across multiple mobile devices.

The guide walks through the entire development process, including:

- Setting up a **gRPC-based architecture** using **Kotlin DSL**.
- Creating modular Android libraries for **Protobuf definitions**, **gRPC networking**, and the main **Android application**.
- Configuring tools such as **Protocol Buffers**, **gRPC-Kotlin**, and **multicast DNS (mDNS)** for service discovery.
- Building a **peer-to-peer mobile computing cluster** to demonstrate **distributed matrix multiplication** using gRPC.

This document is based on a real-world mobile HPC (High-Performance Computing) project and is designed to help others replicate or build upon similar architectures for distributed Android systems.

1. Introduction

gRPC **is an open-source RPC (Remote Procedure Call) platform** developed by **Google** that provides highly performant and efficient communication in any kind of environment and across data centers. Moreover, gRPC's pluggable support of load balancing, tracing, health check, and authentication makes it a good candidate to be used in distributed computing and microservices.

With gRPC, **a client application calls a method on a server application using generated stubs that hide the complexity of distributed applications and services**.

In this tutorial, we'll explore gRPC components, and we'll see how to implement a gRPC server and client in Kotlin.

2. Components

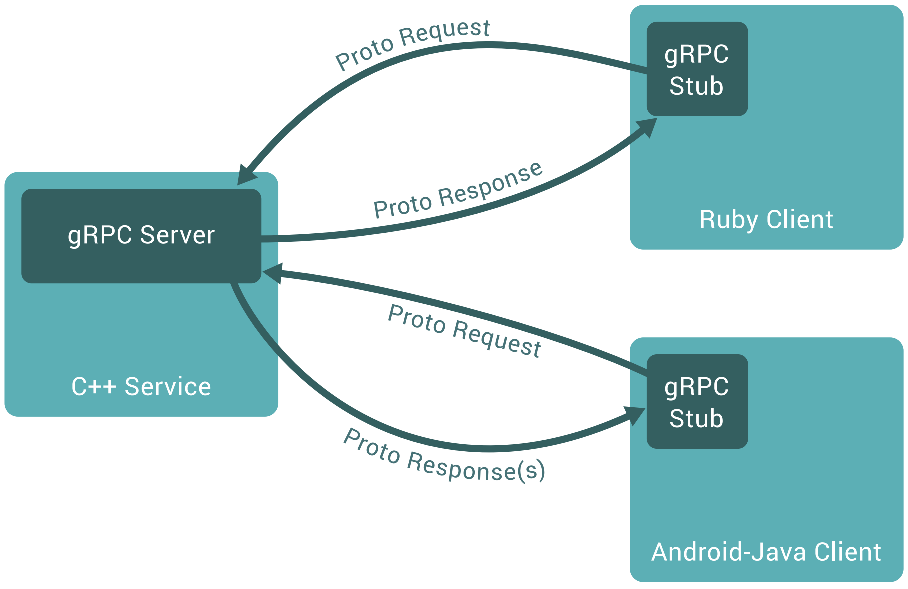
gRPC starts with a service definition. A service definition is an interface for the service and contains methods, parameters, and expected return types.

Then, based on the defined service interface, the clients use a stub to call the server. On the other side, the server implements the service for the interface and runs a gRPC server to handle client requests.

gRPC by default uses *Protobuf* as the interface description language to describe service definition and payload.

Now that we have some understanding of how gRPC works, let's take a look at the system architecture to visualize how components communicate.

3. System Architecture



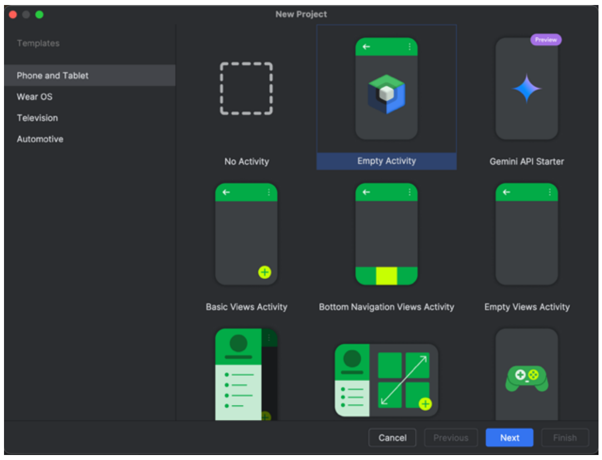
4. Project Setup

This section guides you through setting up a new Android project **(Kotlin DSL)** from **scratch** to support a gRPC-based system with Kotlin and Protocol Buffers.

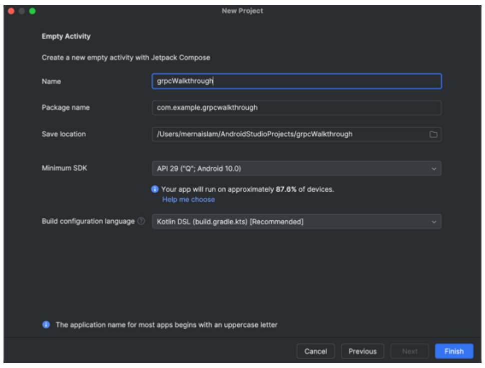
4.1. Setting Up the Root Project

To build a modular and maintainable gRPC project, we first need to configure our **root-level** Gradle setup, including plugin declarations and version management.

1. Create a New Project
    1. Open Android Studio.
    2. Select New Project > Empty Activity.



2. Click **Next** and choose the following:
    - Name: Your preferred project name.
    - Build configuration Language: Kotlin DSL.
    - Minimum SDK: **API 29 (Android 10)** (required for gRPC compatibility).
    - Finish the wizard to generate your project.


New Project

Empty Activity

Create a new empty activity with Jetpack Compose

Name: grpcWalkthrough

Package name: com.example.grpcwalkthrough

Save location: /Users/mernaislam/AndroidStudioProjects/grpcWalkthrough

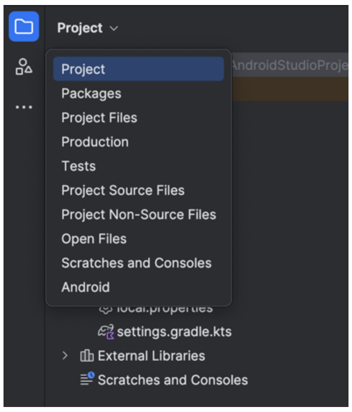
Minimum SDK: API 29 ("Q"; Android 10.0)

Your app will run on approximately **87.6%** of devices.
Help me choose

Build configuration language: Kotlin DSL (build.gradle.kts) [Recommended]

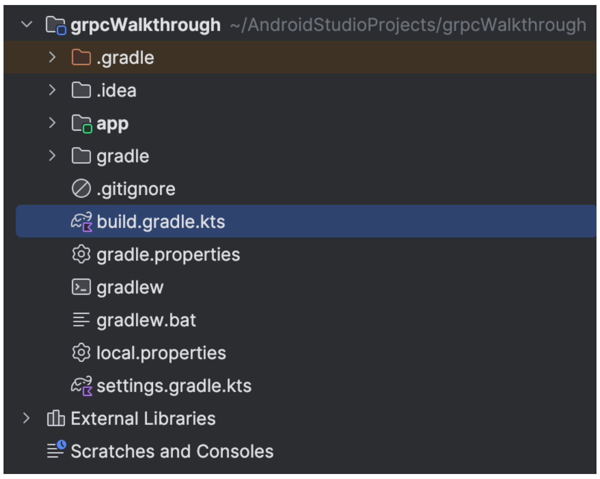
The application name for most apps begins with an uppercase letter

3. Choose Project view instead of Android



4. Configure Root **build.gradle.kts**

    Navigate to your **root-level build.gradle.kts** file and update the plugins block as shown below:



```
plugins {

  // Existing Project Plugins . . .

  kotlin("jvm") version "1.9.23" apply false
  id("com.google.protobuf") version "0.9.4" apply false

}
```

Plugin	Purpose
kotlin("jvm")	Enables Kotlin support for JVM modules like our protos module.
id(com.google.protobuf)	Adds the Protobuf plugin to generate Kotlin/Java code from .proto files.

Note that: Using **apply false** allows us to declare plugins globally without applying them immediately, keeping our build scripts modular and clean.

5. Define Version Variables in **gradle.properties**

   Rather than hardcoding library versions in multiple places, we'll centralize them in the gradle.properties file at the **root of the project**. This improves maintainability and consistency.

   Add the following lines to **gradle.properties:**

```
grpcVersion=1.60.0
grpcKotlinVersion=1.3.0
protobufVersion=3.23.4
```

These values will be referenced in your module-level Gradle files using **rootProject.ext["versionName"].**

4.2. Setup our protos

In this step, you'll set up a dedicated **protos module** to manage your **.proto** files, which define the structure and interface of communication across your distributed gRPC system.

4.2.1. Why Use a protos Module?

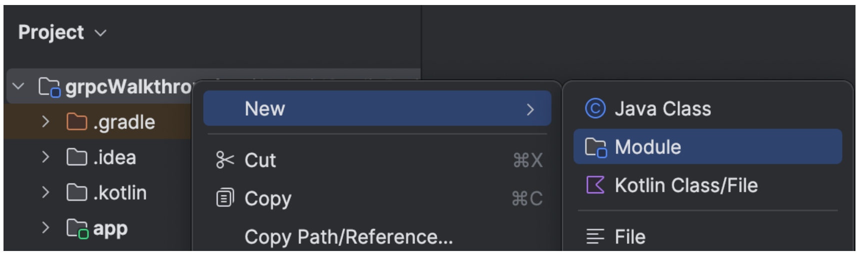
The protos module contains Protocol Buffers (.proto) files that define:

- **Messages**: Data structures (e.g., request and response types) shared between coordinator, workers, and clients.
- **Services**: gRPC service definitions with their method signatures.

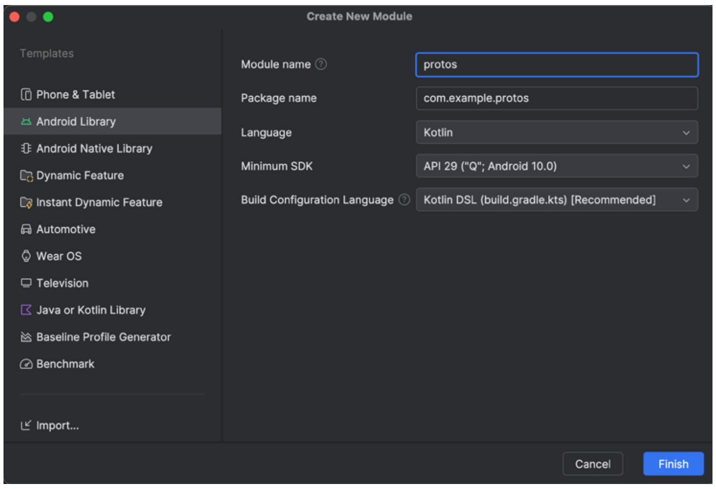
***Note**: Isolating **.proto** definitions improves modularity, reusability, and ensures consistency across components. It also simplifies code generation for multiple platforms (Android, JVM, etc.).*

4.2.2. Step by Step Guide

1. Create a new module
   - Go to **File > New > Module**.



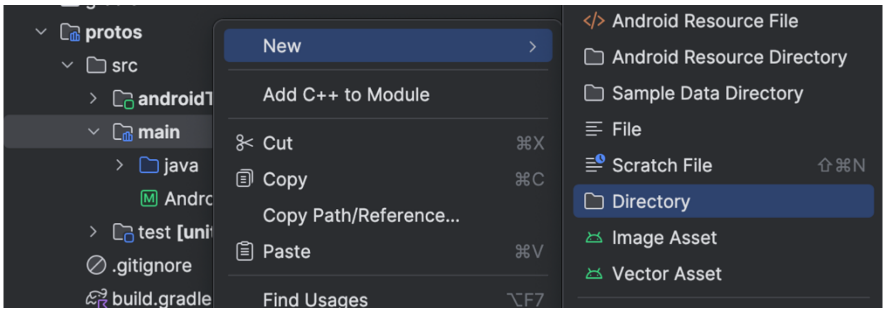
   - Choose **Android Library** as the module type.
   - Name it: protos



Module name: protos
Package name: com.example.protos
Language: Kotlin
Minimum SDK: API 29 ("Q"; Android 10.0)
Build Configuration Language: Kotlin DSL (build.gradle.kts) [Recommended]

2. Create the Proto Source Directory

   Inside **protos/src/main** create a new directory, name it **proto**

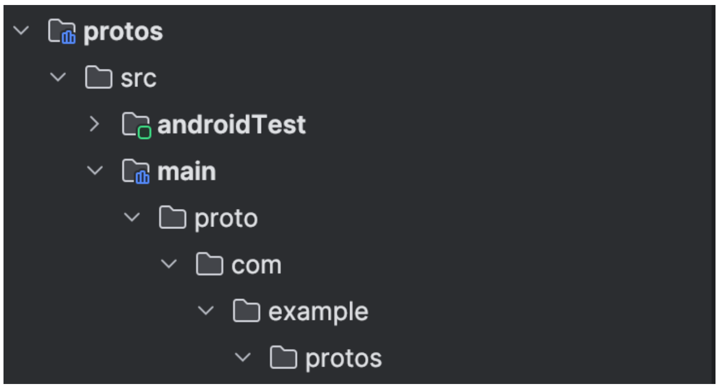


   ⚠️ This name must be exactly **proto**. It is the standard directory where the **Protocol Buffers** compiler (protoc) looks for **.proto** files.

3. Create a Package Directory for Protos

   - Inside **protos/src/main/proto**, create the following directory structure:

   protos/src/main/proto/com/example/protos



   This structure matches the package name used in your **.proto** definitions, ensuring correct namespacing during code generation.

4. Update **build.gradle.kts** for the **protos** Module

   - **Apply the Protobuf Plugin**

     At the top of **protos/build.gradle.kts**:

```
plugins {
  // Existing Plugins . . .
  id("com.google.protobuf")
}
```

- Set SDK Configuration

  Make sure your android block has minimum SDK of **29** and compile SDK of **35**:

```
android {
  namespace = "com.example.protos"
  compileSdk = 35

  defaultConfig {
     minSdk = 29
```

5. Add **gRPC** and **Protobuf** Dependencies
   - Inside the dependencies block:

```
dependencies {

 // Existing Dependencies . . .

 // gRPC + Protobuf
 implementation("io.grpc:grpc-okhttp:${rootProject.ext["grpcVersion"]}")
 implementation("io.grpc:grpc-stub:${rootProject.ext["grpcVersion"]}")
 implementation("io.grpc:grpc-protobuf:${rootProject.ext["grpcVersion"]}")
 implementation("com.google.protobuf:protobuf-java:${rootProject.ext["protobufVersion"]}")
 implementation("io.grpc:grpc-kotlin-stub:${rootProject.ext["grpcKotlinVersion"]}")
}
```

   These dependencies support **Java** and **Kotlin** code generation for **gRPC** services.

6. Configure the **protobuf** Plugin
   - Add this block **at the end of the file:**

```
protobuf {
   protoc {
      artifact = "com.google.protobuf:protoc:${rootProject.ext["protobufVersion"]}"
   }
   plugins {
      create("grpc") {
         artifact = "io.grpc:protoc-gen-grpc-java:${rootProject.ext["grpcVersion"]}"
      }
      create("grpckt") {
         artifact = "io.grpc:protoc-gen-grpc-
kotlin:${rootProject.ext["grpcKotlinVersion"]}:jdk8@jar"
      }
   }
   generateProtoTasks {
      all().configureEach {
         plugins {
            create("grpc")
            create("grpckt")
         }
      }
   }
}
```

This configuration enables automatic generation of both Java and Kotlin gRPC client/server stubs when building the project.

Once configured, you can create your .proto files inside.

protos/src/main/proto/com/example/protos/

For example:

protos/src/main/proto/com/example/protos/task.proto

These files will be compiled automatically, and gRPC classes will be generated for use in your app.

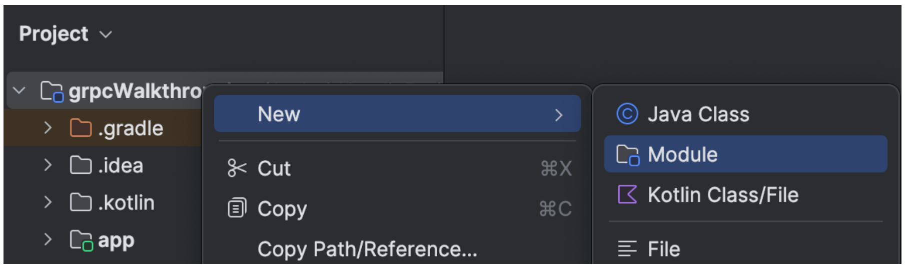
4.3. Setup our Network

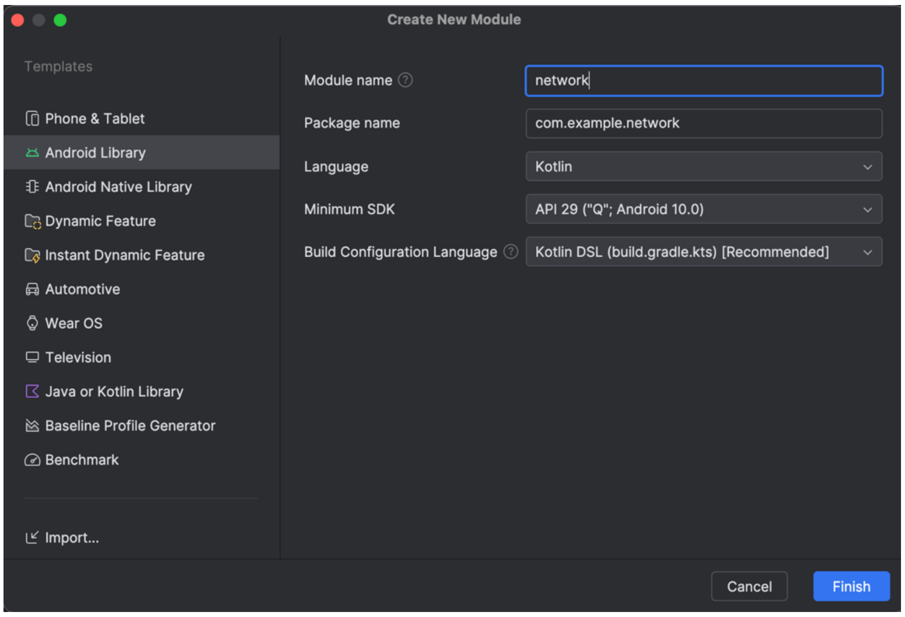
In this section, we will configure the network module, which includes our gRPC communication. This module will implement gRPC service logic and use the Protobuf definitions from the protos module.

---

1. Create the **network** Module

   - Go to **File > New > New Module**
   - Choose **"Android Library"** as the module template
   - Name it **network**





Module name: network
Package name: com.example.network
Language: Kotlin
Minimum SDK: API 29 ("Q"; Android 10.0)
Build Configuration Language: Kotlin DSL (build.gradle.kts) [Recommended]

2. Update **network/build.gradle.kts**

   - Add the Protobuf Plugin

```
plugins {
  // Existing Plugins . . .
  id("com.google.protobuf")
}
```

This enables gRPC code generation using Protobuf definitions from the upcoming **:protos** module.

- Set SDK Configuration

Ensure the SDK versions match your other modules:

```
android {
  namespace = "com.example.server"
  compileSdk = 35

  defaultConfig {
    minSdk = 29

  // . . .

}
```

Note that: The **minSdk** & **compileSDK** should be compatible with the other modules to avoid runtime issues.

3. Add Required Dependencies

```
dependencies {
  // Existing dependencies . . . .

   // Include generated Protobuf code from the protos module
  implementation(project(":protos"))

  // gRPC Core
  implementation("io.grpc:grpc-okhttp:${rootProject.ext["grpcVersion"]}")
  implementation("io.grpc:grpc-stub:${rootProject.ext["grpcVersion"]}")
  implementation("io.grpc:grpc-kotlin-stub:${rootProject.ext["grpcKotlinVersion"]}")
  implementation("io.grpc:grpc-protobuf:${rootProject.ext["grpcVersion"]}")
  implementation("com.google.protobuf:protobuf-java:${rootProject.ext["protobufVersion"]}")

  // Coroutines for async Kotlin logic
  implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.3")

 // Java multicast DNS for auto-discovery
  implementation("org.jmdns:jmdns:3.5.7")

}
```

Note that: Coroutines are essential for writing non-blocking gRPC service handlers in Kotlin.

4. Configure Protobuf Code Generation

   At the end of your Gradle script, add the following block:

```
protobuf {
   protoc {
      artifact = "com.google.protobuf:protoc:${rootProject.ext["protobufVersion"]}"
   }
   plugins {
      create("grpc") {
         artifact = "io.grpc:protoc-gen-grpc-java:${rootProject.ext["grpcVersion"]}"
      }
      create("grpckt") {
         artifact = "io.grpc:protoc-gen-grpc-
kotlin:${rootProject.ext["grpcKotlinVersion"]}:jdk8@jar"
      }
   }
   generateProtoTasks {
      all().configureEach {
         plugins {
            create("grpc")
             create("grpckt")
         }
      }
   }
}
```

   🛠️ This configuration tells Gradle to generate gRPC server stubs for both Java and Kotlin from your .proto files.

4.4. Setup our app

In this section, we will configure the app module of our project. This is where your write your code depending on your project idea. You can choose depending on the app context when you need to initialize your gRPC network.

---

1. Grant Network Permissions

   To allow your client to communicate over the network, open your app's manifest file:

   app/src/main/AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools">

<!-- Required Permissions for gRPC Networking-->
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_MULTICAST_STATE"/>

<application
        . . .
```

   ⚠️ Note that: **INTERNET** and **ACCESS_WIFI_STATE** are essential for connecting to other devices on the network during distributed computation.

2. Update **app/build.gradle.kts**

   - Make sure you have the same compile SDK as the previous modules (ex: 35)
   - Add the network module inside the dependencies

```
implementation(project(":network"))
```

3. Initialize your gRPC network

   Example we can add this code on app create:

```
private val viewModel: MainViewModel by viewModels {
    MainViewModelFactory(applicationContext)
}
```

---

5. Run the project

Before running your gRPC-based Android project, you must ensure a few essential tools are installed and configured correctly. This section walks you through installing the Protocol Buffers compiler (protoc), syncing your Gradle setup, generating source files, and building your project.

---

1. Install the Protocol Buffers Compiler (protoc)

   To compile **.proto** files into Kotlin/Java classes, you need the Protocol Buffers compiler (protoc) version 3.x or above.

   - Linux (Ubuntu/Debian)

   ```
   sudo apt update
   sudo apt install -y protobuf-compiler
   protoc --version     # Ensure compiler version is 3+
   ```

   - MacOS, using Homebrew:

   ```
   brew install protobuf
   protoc --version     # Ensure compiler version is 3+
   ```

   - Windows, using Winget

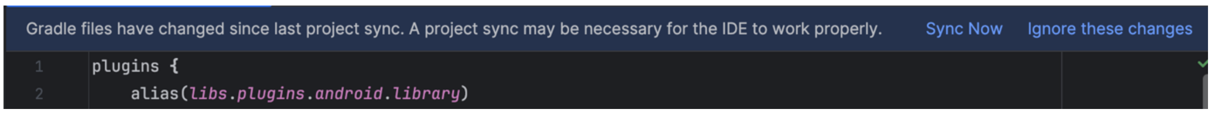
   ```
   winget install protobuf
   protoc --version     # Ensure compiler version is 3+
   ```

2. Sync Gradle

   After setting up your modules and **.proto** files, make sure Gradle is synced.

   In Android Studio:

   Click: **File > Sync Project with Gradle Files**
   or click the **"Sync Now"** prompt if it appears at the top of the IDE.



Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.

```
plugins {
    alias(libs.plugins.android.library)
```

3. Generating gRPC & Protobuf Files

   This step generates the required Kotlin and Java files from your .proto definitions. After adding your own .proto file you need or you can check the practical example below.

   Run the Command from the Root of Your Project

   Navigate to the root directory (where **gradlew** or **gradlew.bat** resides), then run the following based on your OS:

   💻 Run the Command from Project Root

   Navigate to your project's root directory (where the gradlew or gradlew.bat file is located).

   - MacOS/Linux

     ```
     ./gradlew generateDebugProto
     ```

   - Windows

     ```
     gradlew generateDebugProto
     ```

     📝 Note: For release builds, replace **generateDebugProto** with: **generateReleaseProto**

4. Build the Project

   Once the files are generated, you can build your project to ensure everything compiles correctly.

   - MacOS/Linux

     ```
     ./gradlew build
     ```

   - Windows

     ```
     gradlew build
     ```

You're Ready to Go!

Now that everything is set up:

- Your **.proto** files are compiled into usable Kotlin/Java classes.
- gRPC services are ready to be implemented.
- Your Android modules (app, network, and protos) are configured.
- You can now run your own app on multiple Android devices to begin testing your gRPC-based distributed system.

---

6. Practical Example: Distributed Matrix Multiplication using gRPC

Overview

This practical example illustrates the implementation of a **peer-to-peer** (P2P) mobile computing cluster using gRPC across multiple Android devices. The objective is to demonstrate how distributed computation can be achieved efficiently using mobile devices as cooperative nodes in a local network.

The use case focuses on **matrix multiplication**, where:

- A coordinator device splits a large matrix multiplication task.
- The task is distributed to worker devices, each responsible for computing a portion of the matrix.
- Once all partial results are computed, the coordinator aggregates them to produce the final output.

This setup demonstrates key concepts of high-performance computing (HPC) on mobile:

- Parallelism
- Networked coordination
- Efficient workload distribution

Architecture Summary

- All devices operate under a peer-to-peer architecture.
- Devices can dynamically join or leave the cluster.

- The system uses gRPC for communication, and Protocol Buffers for message serialization.
- Discovery is handled via multicast DNS (mDNS) to identify available devices on the same local network.

Source Code

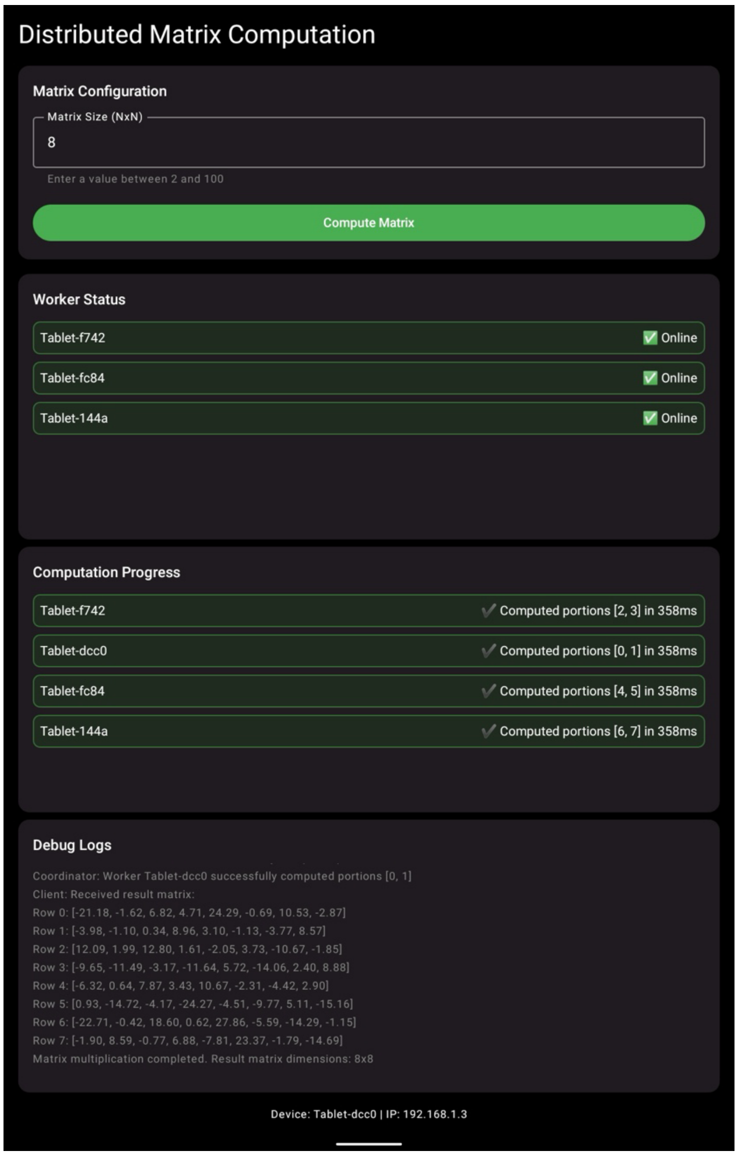
The complete implementation, including source code, documentation, and setup instructions, is available on GitHub:

View the full project on GitHub

---

7. Example Screenshots

Task Distribution



Distributed Matrix Computation

Matrix Configuration

Matrix Size (NxN): 8

Enter a value between 2 and 100

Worker Status

Tablet-f742	✓ Online
Tablet-fc84	✓ Online
Tablet-144a	✓ Online

Computation Progress

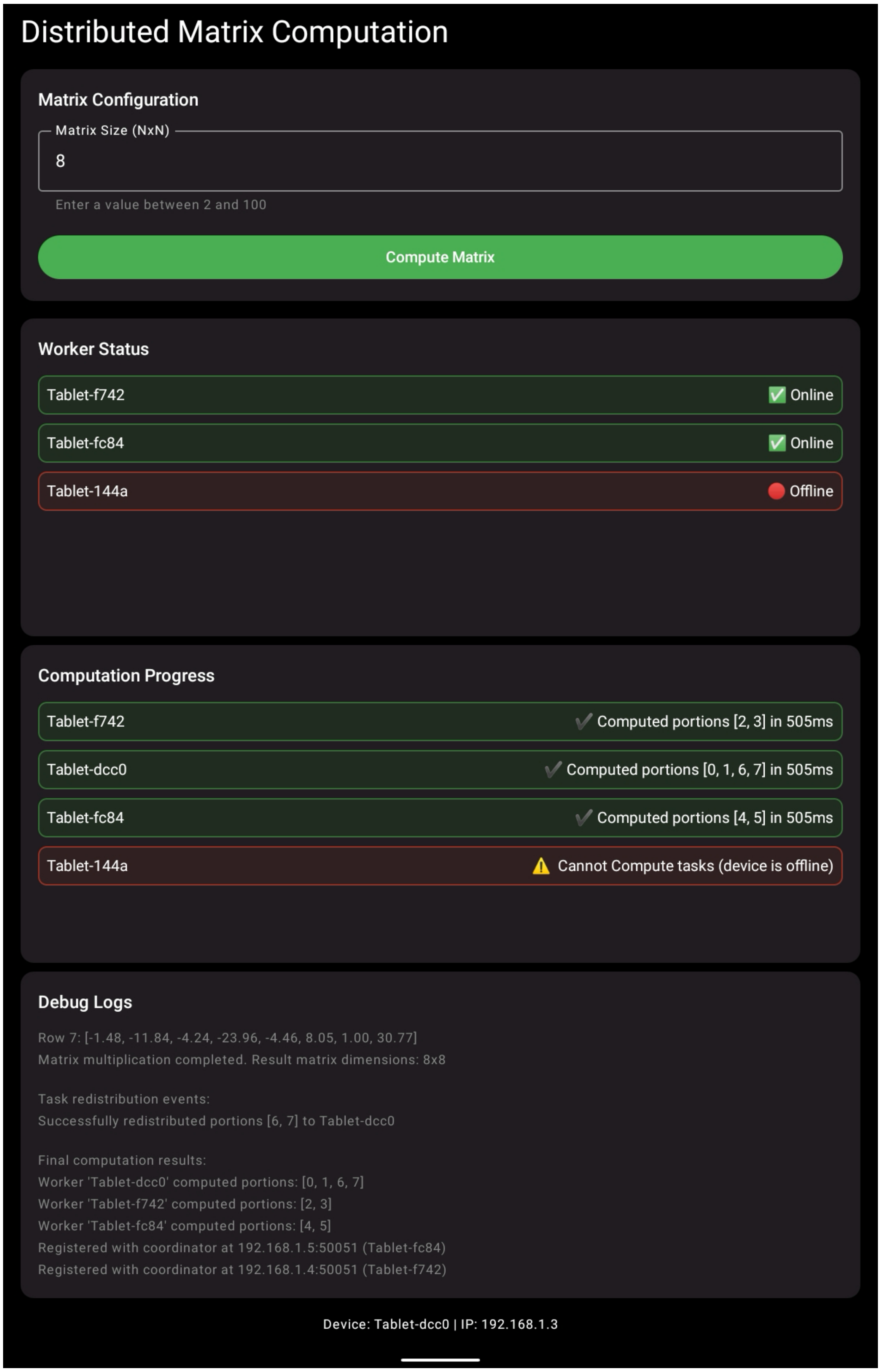
Tablet-f742	✓ Computed portions [2, 3] in 358ms
Tablet-dcc0	✓ Computed portions [0, 1] in 358ms
Tablet-fc84	✓ Computed portions [4, 5] in 358ms
Tablet-144a	✓ Computed portions [6, 7] in 358ms

Debug Logs

```
Coordinator: Worker Tablet-dcc0 successfully computed portions [0, 1]
Client: Received result matrix:
Row 0: [-21.18, -1.62, 6.82, 4.71, 24.29, -0.69, 10.53, -2.87]
Row 1: [-3.98, -1.10, 0.34, 8.96, 3.10, -1.13, -3.77, 8.57]
Row 2: [12.09, 1.99, 12.80, 1.61, -2.05, 3.73, -10.67, -1.85]
Row 3: [-9.65, -11.49, -3.17, -11.64, 5.72, -14.06, 2.40, 8.88]
Row 4: [-6.32, 0.64, 7.87, 3.43, 10.67, -2.31, -4.42, 2.90]
Row 5: [0.93, -14.72, -4.17, -24.27, -4.51, -9.77, 5.11, -15.16]
Row 6: [-22.71, -0.42, 18.60, 0.62, 27.86, -5.59, -14.29, -1.15]
Row 7: [-1.90, 8.59, -0.77, 6.88, -7.81, 23.37, -1.79, -14.69]
Matrix multiplication completed. Result matrix dimensions: 8x8
```

Device: Tablet-dcc0 | IP: 192.168.1.3

Failure Detection and redistribution



Tablet-f742	✅ Online
Tablet-fc84	✅ Online
Tablet-144a	🔴 Offline

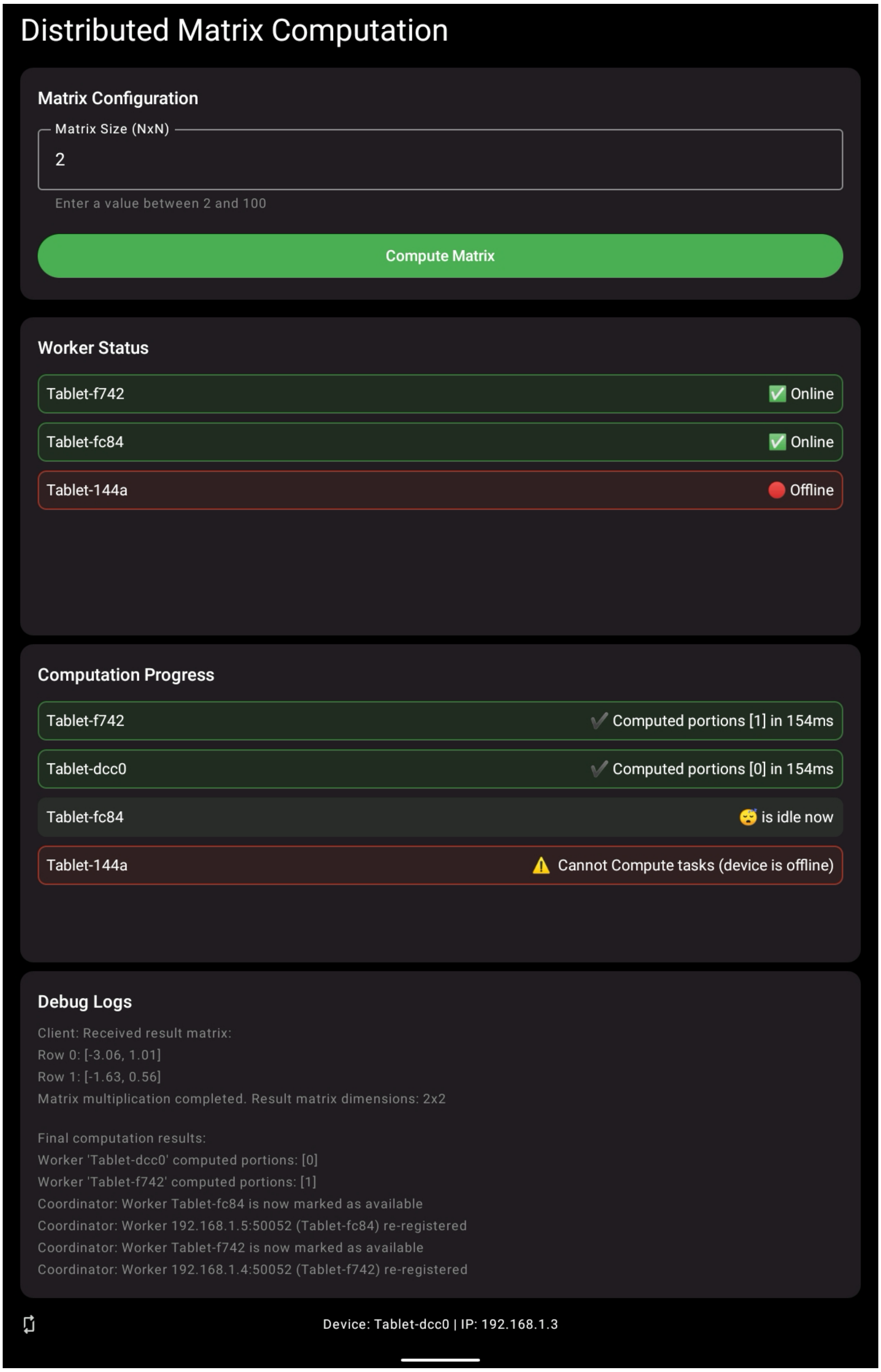
Tablet-f742	✔ Computed portions [2, 3] in 505ms
Tablet-dcc0	✔ Computed portions [0, 1, 6, 7] in 505ms
Tablet-fc84	✔ Computed portions [4, 5] in 505ms
Tablet-144a	⚠ Cannot Compute tasks (device is offline)

```
Row 7: [-1.48, -11.84, -4.24, -23.96, -4.46, 8.05, 1.00, 30.77]
Matrix multiplication completed. Result matrix dimensions: 8x8

Task redistribution events:
Successfully redistributed portions [6, 7] to Tablet-dcc0

Final computation results:
Worker 'Tablet-dcc0' computed portions: [0, 1, 6, 7]
Worker 'Tablet-f742' computed portions: [2, 3]
Worker 'Tablet-fc84' computed portions: [4, 5]
Registered with coordinator at 192.168.1.5:50051 (Tablet-fc84)
Registered with coordinator at 192.168.1.4:50051 (Tablet-f742)
```

Device: Tablet-dcc0 | IP: 192.168.1.3


Distributed Matrix Computation

Matrix Configuration

Matrix Size (NxN)

2

Enter a value between 2 and 100

Worker Status

Tablet-f742	✅ Online
Tablet-fc84	✅ Online
Tablet-144a	🔴 Offline

Computation Progress

Tablet-f742	✔️ Computed portions [1] in 154ms
Tablet-dcc0	✔️ Computed portions [0] in 154ms
Tablet-fc84	😴 is idle now
Tablet-144a	⚠️ Cannot Compute tasks (device is offline)

Debug Logs

```
Client: Received result matrix:
Row 0: [-3.06, 1.01]
Row 1: [-1.63, 0.56]
Matrix multiplication completed. Result matrix dimensions: 2x2

Final computation results:
Worker 'Tablet-dcc0' computed portions: [0]
Worker 'Tablet-f742' computed portions: [1]
Coordinator: Worker Tablet-fc84 is now marked as available
Coordinator: Worker 192.168.1.5:50052 (Tablet-fc84) re-registered
Coordinator: Worker Tablet-f742 is now marked as available
Coordinator: Worker 192.168.1.4:50052 (Tablet-f742) re-registered
```

Device: Tablet-dcc0 | IP: 192.168.1.3