



پروژه باریوم
شبکه های تلفن همراه

امیر فخار زاده ، هانا هاشمی

۱۵ تیر ۱۴۰۳

مقدمه ابتدا در این داک مقدمه ایی از پروژه هایی که در این organization ساخته شده داده میشود سپس به بررسی ساختار و پیاده سازی بخش های مختلف پروژه می پردازیم.

پروژه شامل دو بخش client ، server است . تعاملی که این دو بابکدیگر دارند به این نحو است که client میخواهد به سرور متصل شود . سپس با توجه به عملکردی که سرور دارد خدماتی از آن دریافت کند . این خدمات میتواند بسته به نیاز مشتری متغیر باشد . مشتری پس از وصل شدن به سرور ، باید مدام سیگنال دریافتی خود را از cell serving ایی که بن آن متصل است داشته باشد و یا لاگ بندازد . اگر از یک threshhold معین ، سیگنال دریافتی کمتر بود آنگاه به سرور در قالب SMS پیغامی از بدهد شامل :

اطلاعات cell serving ، سیگنال دریافتی بر حسب دسی بل و موقعیت مکانی

البته مشتری برای دریافت خدمات از سمت سرور باید از یک لایه امنیتی عبور کند . این لایه امنیتی داخل سرور تعبیه شده تا هر کسی نتواند به آن متصل شود . این لایه امنیتی را در این پروژه از جنس فرستادن یک پسوورد از سمت client به سرور طراحی کردیم که در صورتی که پسوورد درست باشد میتواند به سرور متصل شود.

نکته ایی که برای این پروژه قابل ذکر است این است که برای هر کدام از بخش های پیاده سازی شده ، برای شبیه سازی از دو گوشی استفاده کردیم . یک گوشی در نقش client و یک گوشی در نقش server . حال به جزییات ساختار طراحی می پردازیم:

بخش اول برنامه ساخته شده در بخش client شامل چندین بخش مهم است :

- **Location Helper** این بخش همان جایی ست که موقعیت مکانی client ثبت میشود. در این تابع ، با هر تغییر مکانی که در گوشی client اتفاق بیوفتد ، یک لاگ ثبت میشود از lat و long جدید

```

@Override fun onLocationChanged(location: Location) {
    Log.d(tag "LocationHelper", msg "Location changed: ${location.latitude}, ${location.longitude}")
}

```

شکل ۱: location onChange function

این دو متد شروع و توقف مانیتورینگ برای آپدیت های موقعیت مکانی می باشد در متد startListening ابتدا مجوز های مورد نظر چک میشود و در صورت دارا بودن مجوز درخواست آپدیت موقعیت مکانی داده میشود. در متد stopListening با پاک کردن آپدیت باعث توقف مانیتورینگ تغییر موقعیت مکانی میشود.

درمتمد آخر این کلاس ما داریم که بررسی محوز های لازم، اولین قدم برای دریافت آخرین لوکیشن اعلام شده است که پس از آن با داشتن محوز های لازم شما میتوانید آخرین لوکیشن ثبت شده را از یک GPS Provider دریافت کنید و در صورت معتبر بودن لت و لانگ آنرا برگردانید.

```
onGetKnownLocation() { String >
    if (ActivityCompat.checkSelfPermission(context, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED &
        ActivityCompat.checkSelfPermission(context, Manifest.permission.ACCESS_COARSE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
        return Location.permission_not_granted()
    }

    val location = locationManager.getKnownLocation(LocationManager.GPS_PROVIDER)
    return if (location != null) {
        "Lat: ${location.latitude}, Lon: ${location.longitude}"
    } else {
        "Location unavailable"
    }
}
```

- **SMS Service and Reciever** این سرویس در واقع همان کار ارسال SMS را برای ما انجام میدهد. دقت کنید که برای استفاده از این سرویس شما نیاز دارید که در فایل `Android.xml` این سرویس را امپورت کنید تا در کل پروژه بتوانید از آن استفاده کنید.

```
private fun sendSMS(recipient: String, message: String) {
    try {
        val smsManager = if (Build.VERSION.SDK_INT == Build.VERSION_CODES.S) {
            getSystemService(SmsManager::class.java)
        } else {
            SmsManager.getDefault()
        }
        smsManager.sendTextMessage(recipient, null, message, null, null, null, null)
        Log.d(tag, "SMSService", "Message sent to $recipient: $message")
    } catch (e: Exception) {
        Log.e(tag, "SMSService", "Failed to send SMS", e)
    }
}
```

در کلاس `SMSReceiver` هم متدی تحت عنوان `onReceive` وجود دارد که هنگامی که گیرنده یک `SMS-RECEIVED-ACTION` دریافت میکند، اطلاعات مربوط به فرستنده و پیغام را از `Intent` استخراج میکند. برای پردازش بیشتر یک متد دیگر در نظر گرفته شده است که دارای نام `processReceivedMessage` میباشد.

```

override fun onReceive(context: Context, intent: Intent) {
    if (intent.action == Telephony.Sms.Intents.SMS_RECEIVED_ACTION) {
        val messages = Telephony.Sms.Intents.getMessagesFromIntent(intent)
        for (message in messages) {
            val sender = message.originatingAddress
            val messageBody = message.messageBody
            Log.d(tag, "SMSReceiver", msg, "SMS received from: $sender, Message: $messageBody")
        }
    }
}

hannah-hashemi
private fun processReceivedMessage(context: Context, sender: String?, messageBody: String) {
    Log.d(tag, "SMSReceiver", msg, "Received message from $sender: $messageBody")
    // Implement logic to process the received message
}

```

شکل ۵: SMS Reciever

• **Signal Strength** می توان گفت اصلی ترین بخش پیاده سازی شده در app client مربوط به این فایل است . کاری که این بخش به صورت کلی انجام میدهد این است که با در نظر گرفتن یک treshhold، میزان سیگنال دریافتی را بررسی میکند . در صورتی که از مقدار آستانه تعیین شده کمتر بود برای سرور یک SMS به صورت خودکار ارسال میشود که حاوی اطلاعاتی ست که بالاتر به آن اشاره کردیم .

```

override fun onSignalStrengthsChanged(signalStrength: SignalStrength) {
    super.onSignalStrengthsChanged(signalStrength)

    val dbm = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
        signalStrength.cellSignalStrengths.firstOrNull()?.dbm ?: -1
    } else {
        val strength = signalStrength.gsmSignalStrength
        2 * strength - 115 // Convert to dbm
    }

    Log.d(tag, "SignalStrength", msg, "Signal Strength: $dbm dbm")

    if (dbm < -110) {
        val location = (context as MainActivity).getLastKnownLocation()
        val cellInfo = getCellServInfo()
        val message = "Signal strength: $dbm dbm\nLocation: $location\nCell Info: $cellInfo"
        (context as MainActivity).sendSMSToServer(message)
    }
}

```

شکل ۶: Cal SignalStrength and send SMS

متد `onSignalStrengthsChanged` هر زمانی که قدرت سیگنال تغییر کند صدا زده میشود یا با انجام یک سری عملیات به بررسی شرایط بپردازد. ابتدا مند سوپرکلاشش را صدا میزند تا از اجرای رفتار عادی مطمئن شود بر اساس ورژن sdk قدرت سیگنال را برای یک سری از ورژن های جدید به dbm محاسبه میکند و برای ورژن های قدیمی تر این مقدار را با محاسبه ی فرمولی که بخشی از آنرا از GSM گرفته است به دست میآورد پس از آن لاگ آنرا ثبت میکند و در صورتی که strength سیگنال از -۱۱۰ کمتر باشد آخرین لوکیشن ثبت شده و اطلاعات Cell را به دست میآورد و پیغامی حاوی این اطلاعات میسازد و آنرا برای کاربر SMS میکند.

```
private fun getCellServingInfo(): String {
    if (ActivityCompat.checkSelfPermission(context, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
        return "Permission not granted"
    }

    val cellInfo = telephonyManager.allCellInfo.firstOrNull { it.isRegistered }
    return when (cellInfo) {
        is CellInfoLte -> {
            val cellIdentity = cellInfo.cellIdentity
            "LTE - MCC: ${cellIdentity.mcc}, MNC: ${cellIdentity.mnc}, CI: ${cellIdentity.ci}, TAC: ${cellIdentity.tac}"
        }
        else -> "Cell info not available"
    }
}
```

شکل ۷: Get serving cell information

متد مهم دیگری که در این کلاس وجود دارد `getCellServingInfo` میباشد که از آن استفاده میکنیم که اطلاعات `Serving Cell` را بدست بیاوریم. ابتدا بررسی میکند که مجوز های مربوط به موقعیت مکانی را دارد این متد اطلاعات اولین سلول ثبت شده را از بین لیست همه ی سلول هایی که در دسترس دستگاه میباشد، را بر میگرداند. اگر اطلاعات سلول از تایپ `CellInfoLte` باشد یک سری اطلاعات خاص را استخراج و در خروجی بر میگرداند مثل `MCC / MNC / CI / TAC` و ... اما اگر اطلاعات سلول در دسترس نباشد یا تایپ `LTE` نباشد پیغامی مبنی بر در دسترس نبودن سلول نمایان میشود.

Main حال که مدل های مورد استفاده را خیلی کامل تشریح کردیم به `Logic Business` پروژه میپردازیم که در فایل `MainActivity.kt` حضور دارد. در این بخش در واقع جدای از کد مخصوص دسترسی، به یک شماره از گوشی نیازمندیم که نقش سرور را ایفا میکند و `SMS` ها را برای آن میفرستیم.

```
companion object {
    private const val SERVER_PHONE_NUMBER = "+989029518712" // Replace with actual server number
    private const val PERMISSION_REQUEST_CODE = 123
}
```

شکل ۸: properties

بخش اصلی دیگر، درواقع همان جایی ست که به `client` اجازه دسترسی داده میشود. در صورتی که دکمه `connect-to-server` کلیک شود این تابع فراخوانی شده و برای سرور پسونرد داخل `input`، `SMS` میشود.

```
connectButton.setOnClickListener { @View()
    val password = passwordEditText.text.toString().trim()
    if (password.isNotEmpty()) {
        sendSMSToServer(password)
        Toast.makeText(context, "Sending SMS with password...", Toast.LENGTH_SHORT).show()
        monitorInLayout.visibility = View.VISIBLE
        connectButton.visibility = View.GONE
    } else {
        Toast.makeText(context, "Please enter a password", Toast.LENGTH_SHORT).show()
    }
}
```

شکل ۹: properties

مهم ترین اجزای این بخش توضیح داده شد. بخش های دیگر که مربوط به main بیشتر به مانیتور کردن و ارسال پیام به سرور می پردازد که خروجی آن را در قسمت اجرایی این گزارش مطرح میکنیم.

بخش دوم این بخش مربوط به پیاده سازی همان service بک اندی ست که در phone-server اجرا میشود.

این بخش از برنامه UI خاصی ندارد و بیشتر به عنوان یه سرور عمل میکند. ما برای شبیه سازی این بخش از یک گوشی دیگری استفاده کردیم. برنامه که در گوشی اجرا میشود، حکم سروری ست که client به آن متصل است. کد بلاک مربوط به این سرویس اینگونه است:

```
// Hardcode constants
companion object {
    private const val PERMISSION_REQUEST_CODE = 123
    private const val CLIENT_PHONE_NUMBER = "+989029518712" // Replace with actual client number
    private const val PASSWORD = "123"
}
```

شکل ۱۰: properties

در این قسمت ما دو property مهم را داریم:

در property، چون از گوشی به عنوان سرور استفاده میکنیم که SMS های دریافتی از Clinet را بگیرد، شماره سرور را تعیین کردیم. در property دوم، پسورد را به صورت hardcode قرار دادیم. این پسورد باید با پسوردهی که client برای دسترسی به سرور میفرستد یکسان باشد در غیر این صورت اجازه دسترسی به سرور را نخواهد داشت.

```

1 package baritone
2
3 class SMSReceiver : BroadcastReceiver() {
4     & hashCode: hashCode
5     override fun onReceive(context: Context, intent: Intent) {
6         if (intent.action == Telephony.Sms.Intents.SMS_RECEIVED_ACTION) {
7             val messages = Telephony.Sms.Intents.getMessagesFromIntent(intent)
8             for (message in messages) {
9                 val sender = message.originatingAddress
10                 val messageBody = message.messageBody
11                 Log.d(tag, "SMSReceiver", msg: "SMS received from: $sender, Message: $messageBody")
12
13                 if (sender == CLIENT_PHONE_NUMBER && messageBody.trim() == PASSWORD) {
14                     // Send verification SMS back to client
15                     sendSMSToClient(message, "You have been verified", context)
16                 }
17             }
18         }
19     }
20
21     private final fun sendSMSToClient(
22         message: String,
23         context: Context
24     ): Unit {
25         com.example.baritone_server.MainActivity.SMSReceive
26     }
27 }

```

شکل ۱۱: BroadcastReceiver

عملکرد این تیکه از کد به شرح زیر است :

کامپوننت BroadcastReceiver متعلق به کتابخانه خود اندروید است و به این منظور استفاده میشود اگر در سیستم اندرویدی یا همان گوشی ، اعلانیه هایی مثل کم شدن باتری ، بوت شدن سیستم و یا زمانی که SMS ایی دریافت کند. اگر از هرکدام از این موارد اتفاق بیوفتد ، onReceive فراخوانی میشود . بعد از آن چک میشود که اعلانیه که در سیستم وجود دارد آیا از نوع SMS است یا نه . اگر این شرط برقرار بود شروع به چک کردن هر کدام از پیام های دریافتی میکند . در هر کدام از پیام ها اگر SMS دریافتی متعلق به شماره client بود ، آنگار محتویات داخل مسیج ها را چک میکند . سرور در محتوی پیام به دنیال همان کلمه رمزی که client فرستاده میگردد . اگر کلمه عبور وجود داشت و درست بود ، به client این sms را میدهد که verify شده و میتواند از سرور استفاده کند.

```

1 package baritone
2
3 private fun sendSMSToClient(message: String, context: Context) {
4     try {
5         val smsManager = SmsManager.getDefault()
6         smsManager.sendTextMessage(CLIENT_PHONE_NUMBER, null, message, null, null, null, null, null)
7         Log.d(tag, "SMSReceiver", msg: "Verification SMS sent to client: $message")
8     } catch (e: Exception) {
9         Log.e(tag, "SMSReceiver", msg: "Failed to send verification SMS to client", e)
10        Toast.makeText(context, "Failed to send verification SMS", Toast.LENGTH_SHORT).show()
11    }
12 }

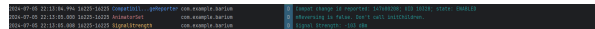
```

شکل ۱۲: server sends SMS to verify client

در آخر نیز وقتی client از سوی سرور verify شد ، برای SMS client ، ایی ارسال میکند که بیانگر verify شدن موفقیت آمیز client است .

همانجور که از پیاده سازی ها واضح است ، این قسمت پیاده سازی در سرور بیشتر جنبه سرویس امنیتی را دارد که برای هر client ایی اجازه دسترسی به دیگر اجزای سرور را ندهد .

مرحله اجرایی ابتدا باید به این امر توجه کرد که هر کدام از اپ های بخش client و server باید در گوشی های جداگانه و به صورت همزمان ران شوند که بتوان خروجی مد نظر را مشاهده کرد . وقتی اپ سمت client را اجرا میگیریم ، از شما پرسووردی میخواهد که بتوانید به سرور متصل شوید . پس از دریافت پرسوورد از سمت سرور مراحل مانیتور کردن client شروع میشود .



شکل ۱۳: SMS Reciever

برای تست کردن ارسال SMS، مقدار threshold را -db۱۰۰ قرار میدهم تا شرط فرستادن SMS برقرار شود . SMS ایی که از سمت client به سرور فرستاده میشود به این شکل خواهد بود :

Signal strength: -102 dBm
Location: Lat: 35.771387689746916, Lon:
51.324542658403516
Cell Info: LTE - MCC: 432, MNC: 35, CI:
45531661, TAC: 22474

Signal strength: -104 dBm
Location: Lat: 35.771387689746916, Lon:
51.324542658403516
Cell Info: LTE - MCC: 432, MNC: 35, CI:
45531661, TAC: 22474

شکل ۱۴: SMS Reciever

همان طور که مشاهده میشود، تمامی اطلاعات از جمله ناحیه TAC نیز برای سرور فرستاده میشود . اطلاعاتی دیگری مثل اپراتوری که به آن متصل است نیز در این پیغام آمده که MNC=۳۵ مربوط به اپراتور ایرانسل است .