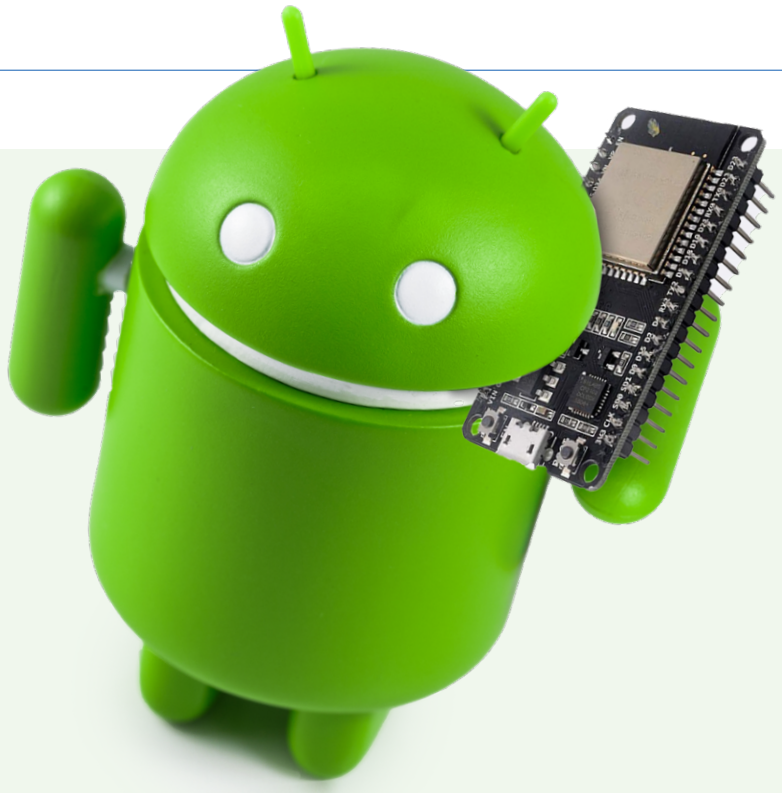
connecter un smartphone Android à un ESP32 ?

projet pratique avec l'API wifi d'Android



Tam Hanna (Hongrie)

Les développeurs d'applications Android savent que le code nécessaire pour se connecter à un réseau wifi n'est pas simple, surtout lorsqu'il s'agit de rendre l'ensemble du processus aussi convivial que possible. Les développeurs de logiciels doivent également s'assurer que leur code est compatible avec les différentes versions d'Android et prendre en compte les mesures de sécurité exigées par Google pour que l'application fonctionne sans problème. Cet article devrait vous aider à vous frayer un chemin dans la jungle.

Si vous n'êtes pas préoccupé par sa consommation d'énergie relativement élevée, le wifi est la norme radio de choix, en particulier pour les petits projets. On peut exclure le Bluetooth à cause des coûts de certification du Bluetooth SIG et des frais de licence. La communication peut se faire via une architecture d'interface REST et les émetteurs wifi sont disponibles un peu partout.

Il n'est guère surprenant que les petites cartes de développement ESP32 – avec leurs points d'accès wifi intégrés – soient si populaires auprès des développeurs d'applications Android.

Le système d'exploitation Android est conçu pour privilégier l'expérience de l'interface utilisateur plutôt que le temps de réponse aux stimuli externes. Cette latence élevée le rend inadapté à la gestion d'événements critiques. Android est doté de piles d'interfaces graphiques importantes et, pour les tâches complexes telles que l'affichage de graphiques, il existe des bibliothèques que vous pouvez charger avec Gradle.

Si un système nécessite une réponse rapide à des événements en temps réel et a également des exigences en matière d'interface utilisateur, il semble logique de répartir le traitement comme le montre la **figure 1**.

Souvent, le maillon faible de la chaîne est l'utilisateur final, qui doit entrer des valeurs dans le système. Si l'utilisateur final doit établir manuellement une connexion à un certain réseau wifi, votre équipe d'assistance produit ne manquera pas de recevoir d'innombrables appels de clients frustrés.

La situation serait plus confortable s'il était possible d'établir une connexion automatique à un réseau. Cependant, Google a des préoccupations en matière de sécurité fonctionnelle et de protection des données personnelles concernant une telle technique.

De quoi s'agit-il ?

L'objectif de cet article est de décrire le code Android nécessaire à l'établissement d'un lien de communication avec un réseau wifi cible. Un ESP32 utilise SoftAP pour établir un point d'accès wifi à distance, auquel l'appareil Android peut se connecter. En pratique, il est possible d'utiliser ce code avec de nombreux autres contrôleurs compatibles wifi (voir le téléchargement [1]).

Cet article est basé sur un projet que l'auteur a développé pour un client. Pratiquement toutes les cartes ESP32 qui exécutent une version du programme d'exemple, *esp-idf/examples/wifi/getting_started/* (inclus dans l'ESP-IDF [2]) configurées comme point d'accès conviendraient

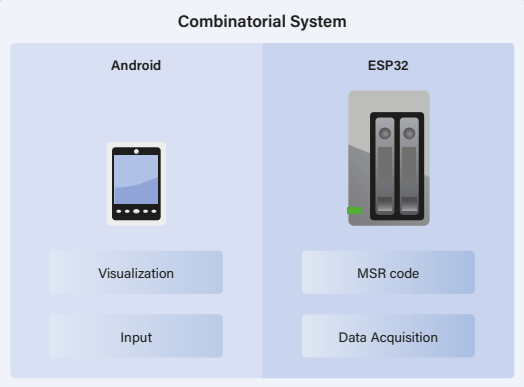


Figure 1. Le smartphone Android gère l'interface graphique, tandis que l'ESP32 est responsable de la communication matérielle.

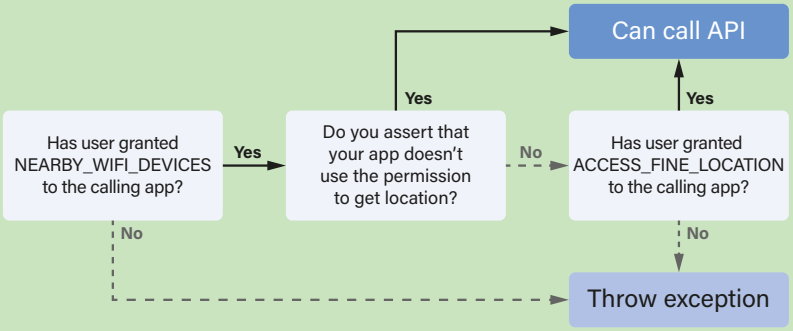


Figure 2. Des autorisations supplémentaires peuvent être requises pour la communication sans fil sous Android 13 (voir l'image [4]).

également. Nous supposons que vous avez une expérience préalable de l'environnement de développement ESP32.
Côté Android, vous devez disposer de deux smartphones : l'un fonctionnant sous Android build ≥ 10, et l'autre sous Android 9 ou antérieur. Android Studio [3] sert d'environnement de développement. Nous supposons également que le lecteur a une certaine expérience préalable de l'EDI Android.

Configuration

La compilation/configuration d'un projet Android est un peu compliquée. En général, vous trouverez une version du fichier *build.gradle* appartenant au module *app* et permettant de configurer tous les paramètres et dépendances nécessaires à l'application dans Android Studio. Ce fichier contient généralement des déclarations telles que :

```
android {
    compileSdkVersion 29
    buildToolsVersion "30.0.1"
    defaultConfig {
        minSdkVersion 26
        targetSdkVersion 31
```

Le champ `minSdkVersion` permet au développeur de spécifier la version minimale du système d'exploitation Android pour l'appareil cible ou final avec lequel l'application est compatible. La valeur 26 correspond à Android 8 (Oreo). `compileSdkVersion` indique la version du kit de développement logiciel (SDK) avec lequel l'application sera compilée, et `buildToolsVersion` indique la version des outils de construction Android qui sera utilisée pour compiler le projet. La ligne `targetSdkVersion` indique la version du système d'exploitation Android pour lequel une application est développée.
Cette distinction, qui semble compliquée à première vue, est nécessaire car le « comportement » de nombreuses API Android change en fonction de la version du système d'exploitation sous laquelle l'application est exécutée.
Si `targetSdkVersion` est défini à une certaine valeur, le système d'exploitation suppose que le développeur a pris en compte tous les changements introduits dans cette version lors de la conception de l'application - si la valeur est inférieure, le mode de compatibilité est activé. Malheureusement, Google a des directives strictes concernant les valeurs spécifiées dans `targetSdkVersion`. Si une application est conçue pour fonctionner uniquement sur des versions plus anciennes du système d'exploitation, Android Studio vous alertera pendant la compilation en affichant le message *« Google Play requires that apps target API level 30 or higher »*, indiquant que le backend du Play Store refusera de télécharger l'APK généré. Dans le futur proche, Google a l'intention de supprimer progressivement les « très vieilles » applications du Play Store.
En tant que développeur, vous devez être conscient de ces limites et difficultés éventuelles.

Autorisations

Le monde d'Android se marque par deux phases : avant et après l'introduction du nouveau système de gestion des autorisations et/ou du *Storage Access Framework*. Auparavant, les développeurs étaient libres de faire ce qu'ils voulaient, mais Google a ensuite pris au sérieux la question de la sécurité des données personnelles.
Pour le wifi, la situation est délicate, car les informations de localisation peuvent être obtenues à partir de la découverte des réseaux sans fil dans la zone (par exemple, le premier iPod touch d'Apple, dépourvu de GPS ou de capacité cellulaire, pouvait déterminer sa propre localisation ainsi).
Il est maintenant nécessaire de spécifier les déclarations suivantes dans le fichier *manifest* de l'application Android pour demander certaines autorisations.

```
<uses-permission android:name=
"android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name=
"android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name=
"android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name=
"android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name=
"android.permission.CHANGE_NETWORK_STATE" />
<uses-permission android:name=
"android.permission.ACCESS_NETWORK_STATE" />
```

Si vous souhaitez que votre application prenne en charge Android 13, vous devez apporter une modification supplémentaire : dans le cas du WLAN, la logique des autorisations d'Android 13 est décrite dans la **figure 2**.
Il est maintenant nécessaire de spécifier les déclarations suivantes dans

le fichier *manifest* de l'application Android pour demander certaines autorisations.

```
<uses-permission android:name=
"android.permission.NEARBY_WIFI_DEVICES"
android:usesPermissionFlags="neverForLocation" />
```

L'autorisation `android.permission.NEARBY_WIFI_DEVICES` est une demande faite par le développeur dans le but de permettre à l'application de rechercher et d'accéder à tous les appareils wifi à proximité, tandis que `android:usesPermissionFlags="neverForLocation"` est un mode spécial qui attribue un *flag* `"never for location"` à accorder à l'application. Cela signifie que l'application n'utilise les informations wifi qu'à des fins de configuration et non pour obtenir des informations de localisation. Dans ce cas, l'autorisation `android.permission.ACCESS_FINE_LOCATION` n'est pas requise sous Android 13.
Les versions antérieures du système d'exploitation requièrent cette autorisation, et dans ce cas, une configuration similaire à celle décrite ci-dessous est recommandée :

```
<uses-permission android:name=
"android.permission.ACCESS_FINE_LOCATION"
android:maxSdkVersion="32" />
```

Dans ce contexte, l'attribut `maxSdkVersion` garantit que la déclaration d'autorisation n'est reconnue que par les versions antérieures d'Android.

Développement à deux volets

Pour configurer l'activité chargée de « rechercher », dans l'environnement, un point d'extrémité wifi accessible, nous devons d'abord vérifier si le wifi du téléphone est activé :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
...
if (android.os.Build.VERSION.SDK_INT <
    Build.VERSION_CODES.Q) {
     wifiManager =
     (WifiManager) getApplicationContext()
    .getSystemService(Context.WIFI_SERVICE);
     if (!wifiManager.isWifiEnabled()) {
       wifiManager.setWifiEnabled(true);
     }
  }
```

Si nous utilisons une ancienne version d'Android qui n'est pas affectée par les modifications de l'API, notre programme peut activer le wifi sur le téléphone sans avoir à demander le consentement de l'utilisateur. Malheureusement, cela ne s'applique pas aux versions plus récentes d'Android - le consentement de l'utilisateur est alors requis.
Pour distinguer les deux cas lors de l'exécution, nous utilisons la constante `android.os.Build.VERSION.SDK_INT`. Elle fait référence à la version de l'API prise en charge par le micrologiciel du téléphone utilisé.

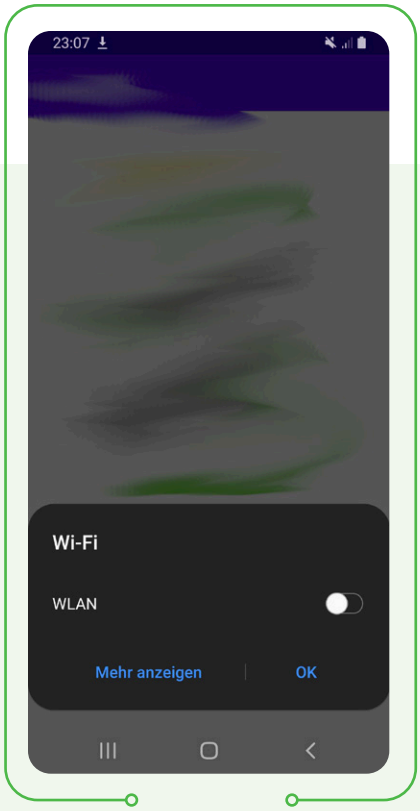


Figure 3. Google exige le consentement explicite de l'utilisateur.

Pour activer le wifi, nous devons envoyer un *Intent* de type `Settings.Panel.ACTION_WIFI` :

```
else {
    wifiManager =
    (WifiManager) getApplicationContext().
    getSystemService(Context.WIFI_SERVICE);
     if (!wifiManager.isWifiEnabled()) {
         Toast toast =
      Toast.makeText(getApplicationContext(),
           "PLEASE enable WiFi so that we can connect!",
           Toast.LENGTH_LONG);
        toast.show();
        Intent panelIntent =
      new Intent(Settings.Panel.ACTION_WIFI);
        this.startActivityForResult(panelIntent,
      ICY_WIFI_REQCODE);
     }
}
```

Le résultat de tous ces efforts est le message de la **figure 3** qui apparaît au moment de l'exécution. L'utilisateur doit basculer le bouton pour autoriser le processus d'activation.
Ensuite *Intent* est envoyé à l'utilisateur, l'informant de la nécessité d'activer le wifi.
Dans le cas contraire, il est nécessaire de rappeler à l'utilisateur la nécessité d'activer le wifi.
Pour les débutants en Android, il peut sembler étrange que `ICY_WIFI_REQCODE` soit transmis à `startActivityForResult()`, mais il s'agit d'un code de corrélation qui permet au point de réception final d'identifier le facteur de déclenchement à l'origine de la demande entrante.

Vieux mais précieux

Une fois que vous aurez terminé votre première application pour smartphone, pourquoi ne pas revenir en arrière et découvrir les principes fondamentaux sur lesquels Jeff Hawkins s'est appuyé pour créer Palm OS, qui fonctionnait sur les appareils Palm et Palm Pilot. Même si ces appareils sont aujourd'hui obsolètes, la philosophie qui a inspiré la conception de Palm OS est intéressante. Zen of Palm est assez court, et vous pouvez en trouver une version PDF sur [5].

Les codes de corrélation sont des entiers normaux auxquels le système d'exploitation n'attribue aucune valeur significative. L'auteur les déclare dans le programme avec le mécanisme suivant ; il est seulement important que le code généré soit unique.

```
public class MainActivity extends
AppCompatActivity implements
AdapterView.OnItemClickListener {
    private final int MY_PERMISSIONS_
   ACCESS_COARSE_LOCATION = 1;
    int ICY_WIFI_REQCODE = 4242;
```

La tâche suivante consiste à déclencher le processus de recherche de wifi. Cet article est limité ; ceci nous empêche de nous plonger dans des détails sur la pile de l'interface graphique d'Android, mais nous sommes convaincus que vous trouverez un moyen d'activer la méthode `getWifi()`.
La première étape du traitement des informations de localisation consiste à vérifier si nous disposons déjà d'une autorisation d'accès pour le `Manifest.permission.ACCESS_FINE_LOCATION`. Sous Android 13, vous devez également inclure la nouvelle permission dans le code d'analyse :

```
private void getWifi() {
    if (ContextCompat.
   checkSelfPermission(MainActivity.this,
   Manifest.permission.ACCESS_FINE_LOCATION) !=
   PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.
     requestPermissions(MainActivity.this,
     new String[]{Manifest.permission.
       ACCESS_FINE_LOCATION},
     MY_PERMISSIONS_ACCESS_COARSE_LOCATION);
    }
```

Rappelons que, sous Android 6.0 et plus, les autorisations « sensibles » ne sont activées par le système d'exploitation que lorsque l'utilisateur les accepte explicitement. Des dialogues sont utilisés à cet effet, comme le montre la **figure 4**.
Si nous avons déjà donné la permission, l'étape suivante est de changer encore de version. Si nous travaillons avec une version antérieure d'Android, nous appelons directement la méthode `getWifiWorkerOld()` :

```
    else {
        if (android.os.Build.VERSION.SDK_INT <
       Build.VERSION_CODES.Q) {
          getWifiWorkerOld();
        }
```

Dans le cas des nouveaux téléphones, et pour éviter un plantage, l'état d'alimentation de l'émetteur wifi est à nouveau vérifié à ce stade, car il est tout à fait possible que les utilisateurs éteignent le wifi avant que la méthode ne devienne active.

```
else {
    if (wifiManager.isWifiEnabled()) {
        getWifiWorkerOld();
    }
    else {
         Toast toast =
      Toast.makeText(getApplicationContext(),
           "PLEASE enable WiFi so that we can connect!",
            Toast.LENGTH_LONG);
        toast.show();
        Intent panelIntent =
      new Intent(Settings.Panel.ACTION_WIFI);
        this.startActivityForResult(panelIntent,
      ICY_WIFI_REQCODE);
    }
}
```

Une fois que nous avons vérifié l'état de l'alimentation de l'émetteur, nous activons le processus de recherche avec de la procédure suivante :

```
private void getWifiWorkerOld() {
    ...
    wifiManager.startScan();
}
```

`wifiManager` est une classe système que nous avons obtenue dans le cadre de l'activation de l'activité. La méthode `startScan()` se charge ensuite de lancer le processus d'analyse.
Le renvoi des informations collectées par le balayage s'effectue par *Broadcast*. Pour les recevoir, nous avons besoin d'un récepteur d'un *Broadcast Receiver*. Comme nous ne voulons pas le créer dans le

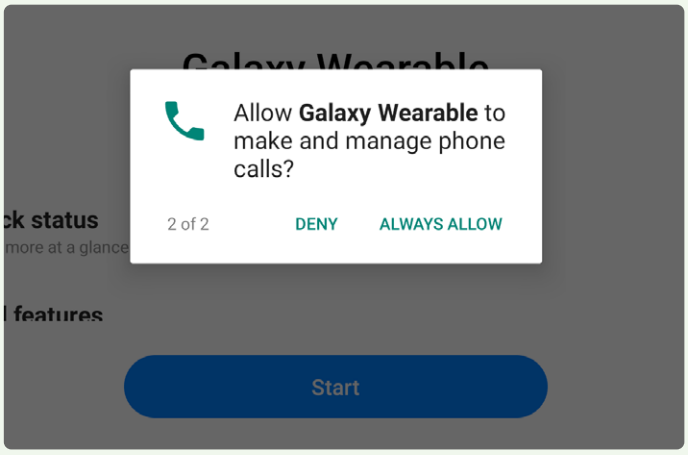


Figure 4. Tenez compte du nouveau système d'autorisations.

fichier *manifest*, nous l'écrivons dans la méthode `onPostResume()` de l'activité :

```
@Override
protected void onPostResume() {
    super.onPostResume();
    receiverWifi =
    new WifiReceiver(wifiManager, wifiList, this);
    IntentFilter intentFilter =
    new IntentFilter();
    intentFilter.addAction
  (WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
    registerReceiver(receiverWifi, intentFilter);
}
```

Afin d'assurer la conformité avec le Play Store, il est important que les *Broadcast Receivers* enregistrés manuellement soient également désenregistrés par le système d'exploitation. On peut le faire de la manière la plus pratique avec l'une des méthodes du cycle de vie de l'activité. J'ai décidé d'utiliser ici la méthode `onPause()` :

```
@Override
protected void onPause() {
    super.onPause();
    unregisterReceiver(receiverWifi);
}
```

Broadcast Receiver pour le traitement de données

Étant donné que la réception des informations réseau renvoyées par le processus de balayage wifi se fait via le système Broadcast d'Android, nous devons intégrer la classe `BroadcastReceiver` enregistrée ci-dessus avec le système d'exploitation.
Les *BroadcastReceivers* dans Android prennent la forme de classes dérivées de la classe principale `BroadcastReceiver`. Dans notre exemple, sans tenir compte des variables de l'interface graphique, l'en-tête se présente comme suit (nous n'incluons pas le constructeur nécessaire pour remplir les champs) :

```
class WifiReceiver extends BroadcastReceiver {
    WifiManager wifiManager;
    ListView wifiDeviceList;
    MainActivity myParentAct;
```

Maintenant, la question importante est de savoir comment traiter les informations réseau renvoyées par les émetteurs wifi. La réponse se trouve dans la méthode `onReceive()`, que le système d'exploitation activera à l'arrivée des informations diffusées sous forme d'intentions (*Intents*).

```
public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (WifiManager.
           SCAN_RESULTS_AVAILABLE_ACTION.equals(action)) {
            List<ScanResult> wifiList =
                wifiManager.getScanResults();
            ArrayList<String> deviceList =
                new ArrayList<>();
            for (ScanResult scanResult : wifiList) {
                 if(scanResult.SSID.
                     contains("NAMEOFTHING"))
                     deviceList.add(scanResult.SSID );
             }
             myParentAct.myArrayAdapter=
                new ArrayAdapter(context,
                  android.R.layout.simple_list_item_1,
                  deviceList.toArray());
               wifiDeviceList.
                  setAdapter(myParentAct.myArrayAdapter);
        }
    }
}
```

Dans la première étape, le code appelle `intent.getAction()` pour vérifier le nom de l'*Intent*. Le Broadcast Receiver peut théoriquement contenir n'importe quelle intention ; en vérifiant la chaîne `WifiManager.SCAN_RESULTS_AVAILABLE_ACTION`, nous nous assurons que nous traitons avec une intention « compatible ».
Puisque l'exemple de code dont l'auteur a tiré les extraits suivants contient une liste pour l'affichage, il est nécessaire de faire certaines modifications à l'étape suivante. Le but de cette démarche est de peupler la classe `deviceList`.
À ce stade, dans une application réelle, une liste serait remplie d'informations, mais vous pouvez procéder différemment, bien entendu, si vous ne considérez qu'un seul réseau wifi cible comme valide.
Un exemple de l'autre approche serait un appareil qui établit le contact avec son application de base lors de la configuration et qui l'indique par un nom wifi spécifique. Dans ce cas, nous pourrions immédiatement continuer après avoir détecté ce réseau.

Une configuration de connexion bien pensée

Une longue période d'attente dans les applications pour smartphones n'est pas du goût des utilisateurs. Il est donc logique d'afficher une barre de progression ou des symboles similaires qui assurent à l'utilisateur que l'application n'est pas plantée.
Dans l'application du client qui sert de base à cet article, elle a été implémentée sous la forme d'une activité dédiée : elle affiche le logo de l'entreprise et une barre de progression, ce qui rassure l'utilisateur. Elle est activée par l'envoi d'une intention :

```
public void onItemClick(AdapterView<?>
adapterView, View view, int pos, long anID) {
    String y = (String) myArrayAdapter.getItem(pos);
    Intent intent =
      new Intent(MainActivity.this,ConnActivity.class);
    intent.putExtra("WIFINAME", y);
    startActivity(intent);
}
```

Attention ! Google peut bloquer les demandes d'autorisation

À partir d'Android 11, Google a introduit les « autorisations uniques », qui permettent à l'utilisateur d'accorder à une application l'accès à une certaine fonctionnalité ou à des données une seule fois. Cela modifie également la façon dont les applications d'arrière-plan demandent des autorisations, ce qui peut entraîner le blocage de certaines demandes. L'objectif est de permettre aux utilisateurs de mieux contrôler leurs données et de renforcer la protection de la vie privée sur Android. Les développeurs doivent être conscients de ces problèmes éventuels.

Le code commence par déclarer un `AsyncTask` qui effectue un test de présence. Il s'agit d'une vérification (non décrite ici) que le pair de connexion fait partie de l'écosystème du client :

```
public class ConnActivity extends AppCompatActivity {
    PresenceTestAsynctask aT;
    ConnActivity mySelf;
```

Dans `onCreate()`, l'établissement de la connexion commence sans aucune intervention de la part de l'utilisateur. La première action officielle consiste à configurer certaines variables membres et à appeler les méthodes `getIntent().getExtras()` pour rendre les informations - « mises en paquet » plus haut par *Intent* - accessibles avec le nom wifi et d'autres données de support :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_conn);
    mySelf=this;
    Bundle b = getIntent().getExtras();
    String networkPass = "sdsds";
```

L'étape suivante est un autre changement de version, qui vérifie l'« état de la version » du matériel cible.
Si nous travaillons avec une version antérieure d'Android, l'étape suivante est le démarrage de la configuration de la connexion selon la procédure suivante :

```
if (android.os.Build.VERSION.SDK_INT <
Build.VERSION_CODES.Q) {
    WifiConfiguration conf =
    new WifiConfiguration();
    conf.SSID = "\"" +
    b.getString("WIFINAME") + "\"";
    conf.preSharedKey = "\"" + networkPass + "\"";
    WifiManager wifiManager =
    (WifiManager) getApplicationContext().
    getSystemService(Context.WIFI_SERVICE);
    wifiManager.addNetwork(conf);
```

La méthode `addNetwork()` prend en charge un objet `WifiConfiguration`. Sa tâche consiste à fournir un ensemble complet d'informations sur le réseau local sans fil, auquel le téléphone peut ensuite se connecter.
Dans cette dernière étape, nous devons établir la connexion selon le mécanisme suivant, avec trois méthodes : `disconnect()`, `enableNetwork()` et `reconnect()`. Ensuite vient la phase d'activation de l'AsyncTask - si la station distante est authentifiée avec succès, l'application client commence à configurer le matériel connecté :

```
/Can only come here from permission granted state
@SuppressLint("MissingPermission")
List<WifiConfiguration> list =
   wifiManager.getConfiguredNetworks();
for (WifiConfiguration i : list) {
    if (i.SSID != null && i.SSID.equals("\"" +
        b.getString("WIFINAME") + "\"")) {
      final ConnectivityManager connectivityManager =
        (ConnectivityManager)getApplicationContext().
        getSystemService(Context.CONNECTIVITY_SERVICE);
      ConnectivityManager.NetworkCallback
         networkCallback =
         new ConnectivityManager.NetworkCallback() {
        @Override
       public void onAvailable(@NonNull Network network) {
         super.onAvailable(network);
         //ONLY IF ANDROID 9
         if (android.os.Build.VERSION.SDK_INT ==
              Build.VERSION_CODES.Q) {
           connectivityManager.
                bindProcessToNetwork(network);
         }
       }
    };
    connectivityManager.
       registerDefaultNetworkCallback(networkCallback);

    wifiManager.disconnect();
    wifiManager.enableNetwork(i.networkId, true);
    wifiManager.reconnect();
    aT = new PresenceTestAsynctask
        (getApplicationContext());
    aT.myAct=mySelf;
    aT.execute("...");
    break;
  }
}
```

Avec les versions plus récentes d'Android, l'intervention de l'utilisateur est à nouveau nécessaire à ce stade. On commence par la création d'un objet `NetworkSpecifier` :

```
}else{
    final NetworkSpecifier specifier =
    new WifiNetworkSpecifier.Builder()
       .setSsidPattern(new PatternMatcher(
     b.getString("WIFINAME"),
     PatternMatcher.PATTERN_PREFIX))
       .setWpa2Passphrase(networkPass)
       .build();
```

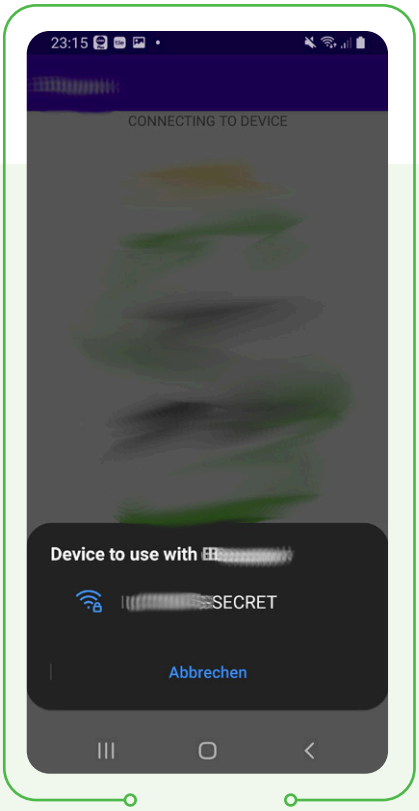


Figure 5. Android Q invite l'utilisateur à établir une connexion.

Les objets `NetworkSpecifier` servent de filtre de recherche pour aider l'interface graphique d'Android à identifier les pairs du réseau sans fil correspondant au cas d'utilisation.
L'étape suivante consiste à créer un élément `NetworkRequest`, comme suit :

```
final NetworkRequest request =
    new NetworkRequest.Builder()
        .addTransportType(NetworkCapabilities.
      TRANSPORT_WIFI)
        .removeCapability(NetworkCapabilities.
      NET_CAPABILITY_INTERNET)
        .setNetworkSpecifier(specifier)
        .build();
```

La configuration de la connexion proprement dite se déroule comme le montre la **figure 5**. L'utilisateur doit saisir le nom du réseau wifi qui sera la cible de la configuration.
Un inconvénient de cette procédure est que la connexion wifi doit être établie de manière asynchrone. Le retour d'information sur le succès ou l'échec est fourni via un rappel qui doit être créé dans l'application responsable de l'activation :

```
final ConnectivityManager connectivityManager =
(ConnectivityManager)getApplicationContext().
getSystemService(Context.CONNECTIVITY_SERVICE);
final ConnectivityManager.NetworkCallback
   networkCallback =
   new ConnectivityManager.NetworkCallback() {
    @Override
    public void onAvailable(@NonNull Network network) {
        super.onAvailable(network);
        connectivityManager.
            bindProcessToNetwork(network);
        aT = new PresenceTestAsynctask
      (getApplicationContext());
        aT.myAct=mySelf;
        aT.execute(". . .");
    }
    @Override
    public void onUnavailable() {
      super.onUnavailable();
    }
};
```

Si la méthode `onAvailable()` est appelée, la connexion a été établie avec succès. Dans ce cas, l'application relance l'AsyncTask afin d'authentifier la partie distante et d'effectuer d'autres actions si nécessaire.
La dernière action de notre méthode de balayage consiste à appeler `requestNetwork()`, qui active l'interface utilisateur illustrée ci-dessus.

```
            connectivityManager.
       requestNetwork(request, networkCallback);
        }
    }
}
```

L'effort en vaut-il la peine ?

En informatique, il y a souvent plusieurs façons de résoudre un problème. Certes, les systèmes d'exploitation en temps réel - je pense ici au RTOS Azure - ainsi que leurs piles d'interfaces graphiques intégrées, fonctionnant sur un microcontrôleur, permettront effectivement de réaliser des interfaces graphiques puissantes dans une solution à appareil unique qui conviendra à la plupart des cas d'utilisation avancés. Pour obtenir la même fonctionnalité que celle fournie par la configuration Android décrite ici, les coûts matériels seraient plus élevés - vous ne bénéficierez pas d'un écran tactile pour l'interface graphique, car la logique associée, telle qu'une mémoire tampon, augmenterait le coût. En outre, nous devons tenir compte du temps et des efforts nécessaires pour développer le système : si vous souhaitez réaliser, par exemple, le transfert d'images et de relevés de capteurs par courrier électronique dans un système d'exploitation en temps réel, vous devez prévoir quelques jours pour le codage et être conscient que la charge supplémentaire peut avoir un impact sur la latence du système. Android offre une interface graphique élégante sur un appareil que la plupart d'entre nous portent déjà dans leur poche. L'idée de sortir son téléphone et de se connecter à un point d'accès local est plutôt cool.
Si vous décidez de développer ce projet, j'espère que certaines des expériences et des ajustements décrits ici vous seront utiles. Si vous réussissez ou si vous trouvez des améliorations, nous serions ravis de le savoir ; envoyez-nous un courriel.

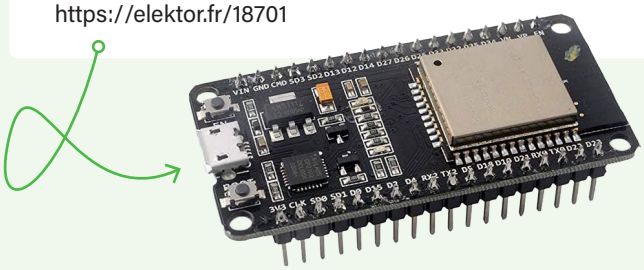
210229-04

Des questions, des commentaires ?

Envoyez un courriel à l'auteur (tamhan@tamoggemon.com) ou contactez Elektor (redaction@elektor.fr).

Produits

- **ESP32-PICO-Kit V4 (avec des connecteurs) (SKU 20323)**
  https://elektor.fr/20323
- **ESP32-DevKitC-32D (SKU 18701)**
  https://elektor.fr/18701



LIENS

[1] Page web de cet article : https://elektormagazine.fr/210229-04
[2] ESP-IDF : https://idf.espressif.com/
[3] Android Studio: https://developer.android.com/studio
[4] Android Documentation: Request permission to access nearby Wi-Fi devices : https://elektor.link/AndroidWiFiPermission
[5] Zen of Palm (2003) [PDF] : https://cs.uml.edu/~fredm/courses/91.308-fall05/palm/zenofpalm.pdf