Investigating Android Scanning Procedures

The Goals:

1. Find out how Android calls the 802.11 scanning functions.
2. Find out how Android decide when to scan periodically (active scan).

Analyzing the Source Code

To analyze the scanning procedure, we will first take a look at how a single scan is performed, followed by background scanning.

The WiFi Scanner holds most of the information we are interested in. It contains information about which channels we are scanning, the callback once the scan is completed, has the entry point for starting single scans as well as background scans. One thing to note is that much of the background scan functionality has been depreciated: `* @deprecated Background scan support has always been hardware vendor dependent. This support * may not be present on newer devices. Use {@link #startScan(ScanSettings, ScanListener)} * instead for single scans.`

So background scans have seemingly been replaced with asynchronous single scans. This also suggests that background scans prior to this change were indeed uniquely identified by the network interface vendor.

The source code for starting a scan is: `java /** * starts a single scan and reports results asynchronously * @param settings specifies various parameters for the scan; for more information look at * {@link ScanSettings} * @param executor the Executor on which to run the callback. * @param listener specifies the object to report events to. This object is also treated as a * key for this scan, and must also be specified to cancel the scan. Multiple * scans should also not share this object. * @param workSource WorkSource to blame for power usage * @hide */ @RequiresPermission(android.Manifest.permission.LOCATION_HARDWARE) public void startScan(ScanSettings settings, @Nullable @CallbackExecutor Executor executor, ScanListener listener, WorkSource workSource) { Objects.requireNonNull(listener, "listener cannot be null"); int key = addListener(listener, executor); if (key == INVALID_KEY) return; validateChannel(); Bundle scanParams = new Bundle(); scanParams.putParcelable(SCAN_PARAMS_SCAN_SETTINGS_KEY, settings); scanParams.putParcelable(SCAN_PARAMS_WORK_SOURCE_KEY, workSource); scanParams.putString(REQUEST_PACKAGE_NAME_KEY, mContext.getOpPackageName()); scanParams.putString(REQUEST_FEATURE_ID_KEY, mContext.getAttributionTag()); mAsyncChannel.sendMessage(CMD_START_SINGLE_SCAN, 0, key, scanParams); }`

Before looking into how this function is called by other services, it may be beneficial to look into `ScanSettings`.

````java public static class ScanSettings implements Parcelable { /** Hidden network to be scanned for. */ public static class HiddenNetwork {

/** SSID of the network */ @NonNull public final String ssid;
````

/** Default constructor for HiddenNetwork. */ public HiddenNetwork(@NonNull String ssid) {this.ssid = ssid;} }

/** one of the WIFI_BAND values */ public int band;

/* one of the {@code WIFI*RNR} values./ private int mRnrSetting = WIFI*RNR*ENABLED*IF*WIFI*BAND*6*GHZ*SCANNED;

/* See {@link #set6GhzPscOnlyEnabled}*/ private boolean mEnable6GhzPsc = false;

/** list of channels; used when band is set to WIFI*BAND*UNSPECIFIED */ public ChannelSpec[] channels;

```
    . . .
```

````

Therefore each scan can either gives a set of bands to scan, or to provide a list of channels to scan. The possible combinations of bands are highlighted here:

```java /* *no band specified; use channel list instead */ public static final int WIFI_BAND_UNSPECIFIED = 0; /* 2.4 GHz band / public static final int WIFI_BAND_24_GHZ = 1 << WIFI_BAND_INDEX_24_GHZ; /* 5 GHz band excluding DFS channels / public static final int WIFI_BAND_5_GHZ = 1 << WIFI_BAND_INDEX_5_GHZ; /* DFS channels from 5 GHz band only / public static final int WIFI_BAND_5_GHZ_DFS_ONLY = 1 << WIFI_BAND_INDEX_5_GHZ_DFS_ONLY; /* 6 GHz band / public static final int WIFI_BAND_6_GHZ = 1 << WIFI_BAND_INDEX_6_GHZ; /* 60 GHz band */ public static final int WIFI*BAND*60*GHZ = 1 << *WIFI*BAND*INDEX*60_GHZ;

/* All combinations of the above below */ ````

One thing to note is that even if an Android scan specifies a particular band, the entire band may not be scanned if there are country restrictions imposed on which channels you may scan. This file contains the details necessary for anything related to the country code:

```java /** * Provide functions for making changes to WiFi country code. * This Country Code is from MCC or phone default setting. This class sends Country Code * to driver through wpa*supplicant when ClientModeImpl marks current state as ready * using setReadyForChange(true). */ public class WifiCountryCode { private static final String TAG = "WifiCountryCode"; private static final String BOOT*DEFAULT*WIFI*COUNTRY_CODE = "ro.boot.wificountrycode"; private final Context mContext;

```
  . . .
```

} ````

OK. Back to the WiFi scanner. Now that we know the API used for starting scans, we want to find the locations where scans are started, especially if it is a repeating service of some sort. Searching the code base for startScan yields the following: ```java public class WifiNetworkFactory extends NetworkFactory {

...

private void startScan() { if (mActiveSpecificNetworkRequestSpecifier == null) { Log.e(TAG, "Scan triggered when there is no active network request. Ignoring..."); return; } if (!mIsPeriodicScanEnabled) { Log.e(TAG, "Scan triggered after user selected network. Ignoring..."); return; } if (mVerboseLoggingEnabled) { Log.v(TAG, "Starting the next scan for " + mActiveSpecificNetworkRequestSpecifier); } // Create a worksource using the
```

caller's UID. WorkSource workSource = new WorkSource(mActiveSpecificNetworkRequest.getRequestorUid()); mWifiScanner.startScan( mScanSettings, new HandlerExecutor(mHandler), mScanListener, workSource); } ````

And

````java public class ScanRequestProxy {

...

/** * Initiate a wifi scan. * * @param callingUid The uid initiating the wifi scan. Blame will be given to this uid. * @return true if the scan request was placed or a scan is already ongoing, false otherwise. */ public boolean startScan(int callingUid, String packageName) { if (!mScanningEnabled || !retrieveWifiScannerIfNecessary()) { Log.e(TAG, "Failed to retrieve wifiscanner"); sendScanResultFailureBroadcastToPackage(packageName); return false; } boolean fromSettingsOrSetupWizard = mWifiPermissionsUtil.checkNetworkSettingsPermission(callingUid) || mWifiPermissionsUtil.checkNetworkSetupWizardPermission(callingUid); // Check and throttle scan request unless, // a) App has either NETWORK*SETTINGS or NETWORK*SETUP_WIZARD permission. // b) Throttling has been disabled by user. int packageImportance = getPackageImportance(callingUid, packageName); if (!fromSettingsOrSetupWizard && mThrottleEnabled && shouldScanRequestBeThrottledForApp(callingUid, packageName, packageImportance)) { Log.i(TAG, "Scan request from " + packageName + " throttled"); sendScanResultFailureBroadcastToPackage(packageName); return false; } // Create a worksource using the caller's UID. WorkSource workSource = new WorkSource(callingUid, packageName); mWifiMetrics.getScanMetrics().setWorkSource(workSource); mWifiMetrics.getScanMetrics().setImportance(packageImportance);

```
// Create the scan settings.
WifiScanner.ScanSettings settings = new WifiScanner.ScanSettings();
// Scan requests from apps with network settings will be of high accuracy type.
if (fromSettingsOrSetupWizard) {
        settings.type = WifiScanner.SCAN_TYPE_HIGH_ACCURACY;
} else {
        if (SdkLevel.isAtLeastS()) {
                // since the scan request is from a normal app, do not scan all 6Ghz chann
                settings.set6GhzPscOnlyEnabled(true);
        }
}
settings.band = WifiScanner.WIFI_BAND_ALL;
settings.reportEvents = WifiScanner.REPORT_EVENT_AFTER_EACH_SCAN
                | WifiScanner.REPORT_EVENT_FULL_SCAN_RESULT;
if (mScanningForHiddenNetworksEnabled) {
        settings.hiddenNetworks.clear();
        // retrieve the list of hidden network SSIDs from saved network to scan for, if en
        settings.hiddenNetworks.addAll(mWifiConfigManager.retrieveHiddenNetworkList());
        // retrieve the list of hidden network SSIDs from Network suggestion to scan for.
        settings.hiddenNetworks.addAll(
                        mWifiInjector.getWifiNetworkSuggestionsManager().retrieveHiddenNet
}
mWifiScanner.startScan(settings, new HandlerExecutor(mHandler),
                new ScanRequestProxyScanListener(), workSource);
return true;
```

} ````
```

Therefore these are the two access points for scanning functionality, we we must search for both of these in the code base and see what we find.

# Searching for WifiNetworkFactory

This file contains the `WifiConnectivityManager` class, which manages all the connectivity related scanning activities. Inside the file, we find functions related to periodic scanning.

First the function that provides the periodic scanning: `java // As a watchdog mechanism, a single scan will be scheduled every // config_wifiPnoWatchdogIntervalMinutes if it is in the WIFI_STATE_DISCONNECTED state. private final AlarmManager.OnAlarmListener mWatchdogListener = new AlarmManager.OnAlarmListener() { public void onAlarm() { watchdogHandler(); } };`

Next, we have several functions related to rescheduling / restarting scans:

````java // Due to b/28020168, timer based single scan will be scheduled // to provide periodic scan in an exponential backoff fashion. private final AlarmManager.OnAlarmListener mPeriodicScanTimerListener = new AlarmManager.OnAlarmListener() { public void onAlarm() { periodicScanTimerHandler(); } };

// A periodic/PNO scan will be rescheduled up to MAX*SCAN*RESTART*ALLOWED times // if the start scan command failed. A timer is used here to make it a deferred retry. private final AlarmManager.OnAlarmListener mRestartScanListener = new AlarmManager.OnAlarmListener() { public void onAlarm() { startConnectivityScan(SCAN*IMMEDIATELY); } };

// A single scan will be rescheduled up to MAX*SCAN*RESTART_ALLOWED times // if the start scan command failed. An timer is used here to make it a deferred retry. private class RestartSingleScanListener implements AlarmManager.OnAlarmListener { private final boolean mIsFullBandScan;

```
RestartSingleScanListener(boolean isFullBandScan) {
    mIsFullBandScan = isFullBandScan;
}

@Override
public void onAlarm() {
    startSingleScan(mIsFullBandScan, WIFI_WORK_SOURCE);
}
```

} ````

So we have the mechanism through which scans are scheduled if the phone is not connected to an access point, as well as the mechanisms related to starting the service.

The timer handler for the periodic scan is:

````java // Watchdog timer handler private void watchdogHandler() { // Schedule the next timer and start a single scan if we are in disconnected state. // Otherwise, the watchdog timer will be scheduled when entering disconnected // state. if (mWifiState == WIFI*STATE*DISCONNECTED) { localLog("start a single scan from watchdogHandler");

```
    scheduleWatchdogTimer();
    startSingleScan(true, WIFI_WORK_SOURCE);
}
```

} ````

Something I am unsure of is which restart service the periodic scan uses since we can see that the watchdog handler uses single scans. However, assuming that the scans do not fail, there will be a scan occurring every `config_wifiPnoWatchdogIntervalMs` milliseconds. Searching for this value, we get that:

```
java public static final int config_wifiPnoWatchdogIntervalMs=0x7f050040;
```

Which is ~2131 seconds, which is~36 minutes. This is far too large of an interval however, so there is likely something else going on.

# Finding the Discrepancy

There may be several reasons why we see such a difference between the observed scan pattern and the code:

1. If a scan does not return any results, it may be considered a failure and thus the scan is repeated according to exponential backoff.
2. On top of the watchdog scan, there are other scans occurring periodically.

I am currently looking into this and will update this document once I find any relevant information.

# Sources


https://stackoverflow.com/questions/15137247/how-does-getsystemservice-work-exactly

https://developer.android.com/guide/topics/connectivity/wifi-scan#java

https://cs.android.com/android/platform/superproject/+/master:packages/modules/Wifi/