Understanding Scan Behavior in Android

The Goals

1. Understand the timing between scan requests within Android.

1. Scan Behavior

The two sources of periodic scans are the watchdog scans, and the periodic scans setup according to the periodic scan schedule within Android. Most of the code is from this file.

Watchdog

The watchdog is a mechanism to look for WiFi networks when the phone is not currently connected to a WiFi network:

```
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);
  }
}
```

So a full band scan is sent every time scheduleWatchdogTimer() is triggered. Looking at this function, we see that the timer expires `config_wifiPnoWatchdogIntervalMs` after setting it. Finding

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

Since we don't access to their RO data, we trust the docs which state that Which state that this is once every 20 minutes.

Periodic Scans

The majority of the scanning comes from the periodic scans set up within Android.

The handler for the timer is:

```
// line 1778
private void periodicScanTimerHandler() {
    localLog("periodicScanTimerHandler");

    // Schedule the next timer and start a single scan if screen is on.
    if (mScreenOn) {
        startPeriodicSingleScan();
    }
}
```

The first thing of interest is that the scan only occurs if the screen is on. The definition for startPeriodicSingleScan() is rather long, so instead of dumping all of it here, I'll go through it piece by piece and dump the full definition of the function in the appendix.

```
long currentTimeStamp = mClock.getElapsedSinceBootMillis();

if (mLastPeriodicSingleScanTimeStamp != RESET_TIME_STAMP) {
    long msSinceLastScan = currentTimeStamp - mLastPeriodicSingleScanTimeStamp;
    if (msSinceLastScan < getScheduledSingleScanIntervalMs(0)) {
        localLog("Last periodic single scan started " + msSinceLastScan
                + "ms ago, defer this new scan request.");
        schedulePeriodicScanTimer(
                getScheduledSingleScanIntervalMs(0) - (int) msSinceLastScan);
        return;
    }
}
```

In the first section, we check if our scan request timer left enough time between our last scan and the current one. If we didn't leave enough time between them, we just reschedule this scan to happen once enough time has occurred. Now the question is what "enough time" is. Inside `getScheduledSingleScanIntervalMs()` we see the following:

```
private int getScheduledSingleScanIntervalMs(int index) {
  synchronized (mLock) {
      if (mCurrentSingleScanScheduleSec == null) {
          Log.e(TAG, "Invalid attempt to get schedule interval, Schedule array is null ");

          // Use a default value
          return DEFAULT_SCANNING_SCHEDULE_SEC[0] * 1000;
      }

      if (index >= mCurrentSingleScanScheduleSec.length) {
          index = mCurrentSingleScanScheduleSec.length - 1;
      }
      return getScanIntervalWithPowerSaveMultiplier(
              mCurrentSingleScanScheduleSec[index] * 1000);
  }
}
```

The schedule is guarded by a lock, which makes sense since we want to make sure we leave the right amount of time between scans. Since the 802.11 protocol expects scans according to exponential backoff, having a race condition risks not following the expected procedure.

If a scan schedule has not been provided, then a default value is provided. The default schedule is defined as:

```
private static final int[] DEFAULT_SCANNING_SCHEDULE_SEC = {20, 40, 80, 160};
```

Note that these times are in milliseconds, despite the name containing seconds. Thus, without a specified scan schedule, a scan is expected to occur every 20 seconds.

If a user defined scanning schedule is provided, then we simply find the entry to use as our scan interval. If the index is greater than our scan schedule length, we just return the maximum value. If not, we just use specified index.

Once we have the index, we call `getScanIntervalWithPowerSaveMultiplier()`, for which the definition is:

```
private int getScanIntervalWithPowerSaveMultiplier(int interval) {
  if (!mDeviceConfigFacade.isWifiBatterySaverEnabled()) {
      return interval;
  }
  return mPowerManager.isPowerSaveMode()
          ? POWER_SAVE_SCAN_INTERVAL_MULTIPLIER * interval : interval;
}
```

So if power saving mode is enabled, we multiply our interval by a factor of POWER*SAVE*SCAN*INTERVAL*MULTIPLIER.

```
private static final int POWER_SAVE_SCAN_INTERVAL_MULTIPLIER = 2;
```

So if power saving mode is enabled, we double our previous scan interval. Now the question is whether Android uses a custom configuration or if it uses its default scan schedules. Android has 3 different usable scan schedules, defined much earlier:

```
private int[] mConnectedSingleScanScheduleSec;
private int[] mDisconnectedSingleScanScheduleSec;
private int[] mConnectedSingleSavedNetworkSingleScanScheduleSec;
```

One while connected to another network, one while not connected, and one while connected with only 1 saved network.

These scanning schedules must be initialized somewhere, so searching for them, we find the following function:

```
private int[] initializeScanningSchedule(int state) {
    int[] scheduleSec;

    if (state == WIFI_STATE_CONNECTED) {
        scheduleSec = mContext.getResources().getIntArray(
                R.array.config_wifiConnectedScanIntervalScheduleSec);
    } else if (state == WIFI_STATE_DISCONNECTED) {
        scheduleSec = mContext.getResources().getIntArray(
                R.array.config_wifiDisconnectedScanIntervalScheduleSec);
    } else {
        scheduleSec = null;
    }

    boolean invalidConfig = false;
    if (scheduleSec == null || scheduleSec.length == 0) {
        invalidConfig = true;
    } else {
        for (int val : scheduleSec) {
            if (val <= 0) {
                invalidConfig = true;
                break;
            }
        }
    }
    if (!invalidConfig) {
        return scheduleSec;
    }

    Log.e(TAG, "Configuration for wifi scanning schedule is mis-configured,"
            + "using default schedule");
    return DEFAULT_SCANNING_SCHEDULE_SEC;
}
```

So we can simply look up the configurations, we find that:

```
public static final int config_wifiBackgroundScanThrottleExceptionList=0x7f010000;
public static final int config_wifiConnectedScanIntervalScheduleSec=0x7f010001;
public static final int config_wifiDisconnectedScanIntervalScheduleSec=0x7f010002;
```

So our goal will be to find these values inside the read only data somewhere. I will leave this for a future section.

Back to our initial `startPeriodicSingleScan()`, we've gotten our scan interval and now we need to check whether the scan is needed. This is done by checking one of the following conditions:

1) Network is sufficient 2) Link is good, internet status is acceptable and it is a short time since last network selection 3) There is active stream such that scan will be likely disruptive

The code for this is:

```
if (mWifiState == WIFI_STATE_CONNECTED
        && (mNetworkSelector.isNetworkSufficient(wifiInfo)
        || isGoodLinkAndAcceptableInternetAndShortTimeSinceLastNetworkSelection
        || mNetworkSelector.hasActiveStream(wifiInfo))) {
    // If only partial scan is proposed and firmware roaming control is supported,
    // we will not issue any scan because firmware roaming will take care of
    // intra-SSID roam.
    if (mConnectivityHelper.isFirmwareRoamingSupported()) {
        localLog("No partial scan because firmware roaming is supported.");
        isScanNeeded = false;
    } else {
        localLog("No full band scan because current network is sufficient");
        isFullBandScan = false;
    }
}
```

What is considered "sufficient" and "acceptable" is left for now, but can be explored at a later point.

Now we know whether we need the scan or not. If we need the scan, we scan and then set the timer for the next scan. Otherwise, we just set the timer for the next scan:

```
if (isScanNeeded) {
    mLastPeriodicSingleScanTimeStamp = currentTimeStamp;

    if (mWifiState == WIFI_STATE_DISCONNECTED
            && mInitialScanState == INITIAL_SCAN_STATE_START) {
        startSingleScan(false, WIFI_WORK_SOURCE);

        // Note, initial partial scan may fail due to lack of channel history
        // Hence, we verify state before changing to AWIATING_RESPONSE
        if (mInitialScanState == INITIAL_SCAN_STATE_START) {
            setInitialScanState(INITIAL_SCAN_STATE_AWAITING_RESPONSE);
            mWifiMetrics.incrementInitialPartialScanCount();
        }
    } else {
        startSingleScan(isFullBandScan, WIFI_WORK_SOURCE);
    }
    schedulePeriodicScanTimer(
            getScheduledSingleScanIntervalMs(mCurrentSingleScanScheduleIndex));

    // Set up the next scan interval in an exponential backoff fashion.
    mCurrentSingleScanScheduleIndex++;
  } else {
    // Since we already skipped this scan, keep the same scan interval for next scan.
    schedulePeriodicScanTimer(
            getScheduledSingleScanIntervalMs(mCurrentSingleScanScheduleIndex));
  }
```

Initial scans are the first scan after WiFi has been enabled or turning on the screen when disconnected. Since Android has sensors in their phones that detect movement, initial scans may occur without any user interaction.

Since we call `startSingleScan()` with isFullBandScan set to false, we are scanning a set of channels which will be explored in the second section of this report on channels.

Thus, the only step left is to find the values for the scan schedules in different states. This is not trivial (imo), and I will have to look into this to see what I can find.

What Happens When a Scan Fails?

One thing we should be interested in is what happens if a scan fails, and what it even means when a scan fails? The function that schedules retries is:

```
private void scheduleDelayedSingleScan(boolean isFullBandScan) {
    localLog("scheduleDelayedSingleScan");

    RestartSingleScanListener restartSingleScanListener =
            new RestartSingleScanListener(isFullBandScan);
    mAlarmManager.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                        mClock.getElapsedSinceBootMillis() + RESTART_SCAN_DELAY_MS,
                        RESTART_SINGLE_SCAN_TIMER_TAG,
                        restartSingleScanListener, mEventHandler);
}
```

So we wait `RESTART_SCAN_DELAY_MS` before scanning again, which is set here:

```
private static final int RESTART_SCAN_DELAY_MS = 2 * 1000; // 2 seconds
```

Another note is that we'll only retry up to MAX*SCAN*RESTART_ALLOWED times, which is set here:

```
public static final int MAX_SCAN_RESTART_ALLOWED = 5;
```

So we try the scan every 2 seconds up to 5 times.

The other option for scans is to pass a sequence of channels to scan. If this is the case, then it is directly passed to lower level scanning functions. The channel shuffling must occur at the lower level somewhere since the same flags are passed each time for a particular device (most of the time) and yet the order in which channels are probed changes.

Where to go from here?

1. Look through lower level scanning code to find how the channel list is shuffled.
2. Find the scan schedules for several android devices using a Disassembler.

Sources