Big Nerd Ranch

Choosing the Right Background Scheduler in Android

Sean Farrell

May 6, 2016 • Android

Over the last few years, Google has introduced several new classes to Android for scheduling work that needs to run outside the scope of an application's lifecycle. Many of these new schedulers implement features that improve battery life, as do updates to older methods. And as of Android Marshmallow, these schedulers became even more efficient thanks to Doze Mode, which doesn't even require any effort from developers!

Let's take a look at the different methods for scheduling tasks in Android and discuss the pros and cons of each so that you can determine when to use each tool—*before* implementing an inappropriate one.

Scheduling Work Within Your Application Process

First things first. If the work that you need to schedule does not need to live beyond the scope of your application, Google recommends using the Handler class along with a Timer and Thread. For any timing operations that occur only during the lifetime of your application, it is best to use application resources rather than system resources to schedule tasks. Using a Handler is easier and more efficient than the following methods when it comes to scheduling tasks within your application process.

Using the AlarmManager to Schedule Tasks at the System Level

The AlarmManager provides access to system-level alarm services. Using the AlarmManager allows an application to schedule tasks that may need to run or repeat beyond the scope of its lifecycle. This allows the application to perform some function even after the application process or all of its Android components have been cleaned up by the system.

Typically, the AlarmManager is used to fire off a PendingIntent that will start up a Service in the future. The AlarmManager triggers Services based on an elapsed interval or at a specific clock time. Both of these options also have the ability to wake up the device when it is asleep if the alarm is urgent.

The benefits of the AlarmManager come into play when using **inexact** intervals or times to fire off Services. The Android system tries to batch alarms with similar intervals or times together in order to preserve battery life. By batching alarms from multiple applications, the system can avoid frequent device wake and networking.

One concern to consider while using the AlarmManager is that alarms are wiped out during device reboots. Applications need to register the RECEIVE_BOOT_COMPLETE permission in their Android Manifest and reschedule their alarms in a BroadcastReceiver.

Another concern is that a poorly designed alarm could cause battery drain. While the AlarmManager does have the ability to wake devices and set an exact time for an alarm, the documentation mentions that developers should be wary of these features when performing networking. Aside from draining a device's battery by avoiding batch alarms, setting an exact time for an application to sync with a server could put high strain on a server if every application installation tries to sync with the server around the same time! This can be avoided by adding some randomness to alarm intervals or times.

AlarmManager is a great candidate for scheduling if an application needs to perform a local event at an exact time or inexact interval. Alarm clock or reminder applications are great examples for AlarmManager usage. However, the documentation discourages using AlarmManager for scheduling network-related tasks. Let’s take a look at some better options for networking.

Job Scheduler

JobScheduler helps perform background work in an efficient way, especially networking. JobServices are scheduled to run based on criteria declared in JobInfo.Builder(). These criteria include performing the JobService only when the device is charging, idle, connected to a network or connected to an unmetered network. JobInfo can also include minimum delays and certain deadlines for performing the JobService. Jobs will queue up in the system to be performed at a later time if none of these criteria are met. The system will also try to batch these jobs together in the same manner that alarms are scheduled in order to save battery life when making a network connection.

Developers might be concerned about a scheduler that frequently delays firing off its JobServices. If jobs are frequently delayed and data stale as a result, it would be nice to know about such things. JobScheduler will return information about the JobService such as if it was rescheduled or failed. JobScheduler has back-off and retry logic for handling these scenarios, or developers could handle those scenarios themselves.

Subclassing JobService requires an override of its onStartJob(JobParams params) and onStopJob(JobParams params) methods. onStartJob() is where callback logic for jobs should be placed, and it runs on the main thread. Developers are responsible for threading when dealing with long running jobs. Return true to onStartJob() if separate thread processing needs to occur, or false if processing can occur on the main thread and there is no more work to be done for this job. Developers must also call jobFinished(JobParameters params, boolean needsReschedule) when the job is complete and determine whether or not to reschedule more jobs. onStopJob() will get called to stop or cleanup tasks when initial JobInfo parameters are no longer met, such as a user unplugging their device if that parameter is a requirement.

There might be a lot to think about when implementing a JobService, but it comes with a lot more flexibility than AlarmManager. Another handy feature is that scheduled jobs persist through system reboots.

There is at least one drawback to using JobScheduler. As of the writing of this post, it's compatable only with API level 21 and higher. Here you can find the distribution of Android devices running various API levels. While there is technically no backport of JobScheduler, a similar tool is GCM Network Manager.

GCM Network Manager

GCM (Google Cloud Messaging) Network Manager actually uses Job Scheduler behind the scenes for API level 21 and higher, and features support for previous versions of Android with Google Play Services. GCMNetworkManager has all of the nice battery saving schedule features from JobScheduler, but the backwards compatibility and simplicity of its Service counterpart will make it more appealing to most developers.

Tasks are created using OneoffTask.Builder() or PeriodicTask.Builder(). The Builder pattern makes it easy to append scheduling criteria to a task. The criteria are almost identical to those from JobInfo, and the system aims to optimize battery life through batching

Tasks scheduled with GcmNetworkManager. Criteria can include required network or charging states and desired windows, but the ability to create a Task as one-off or periodic takes a bit of work out of the rescheduling required in JobService. Tasks are scheduled by calling the schedule(Task task) method on an instance of GcmNetworkManager.

A subclass of GcmTaskService is required to receive scheduled Tasks. GcmTaskService requires an override of its onRunTask(TaskParams params) method, and this is where any work should be performed. onRunTask() is run on a background thread, a handy improvement over JobService's onStartJob() method.

One thing to note about GcmNetworkManager compared to JobScheduler is that application and Google Play Service updates will wipe out scheduled Services. The onInitializeTask() method of GcmTaskService can be overriden to be notified of this event and respond accordingly to reschedule the Service.

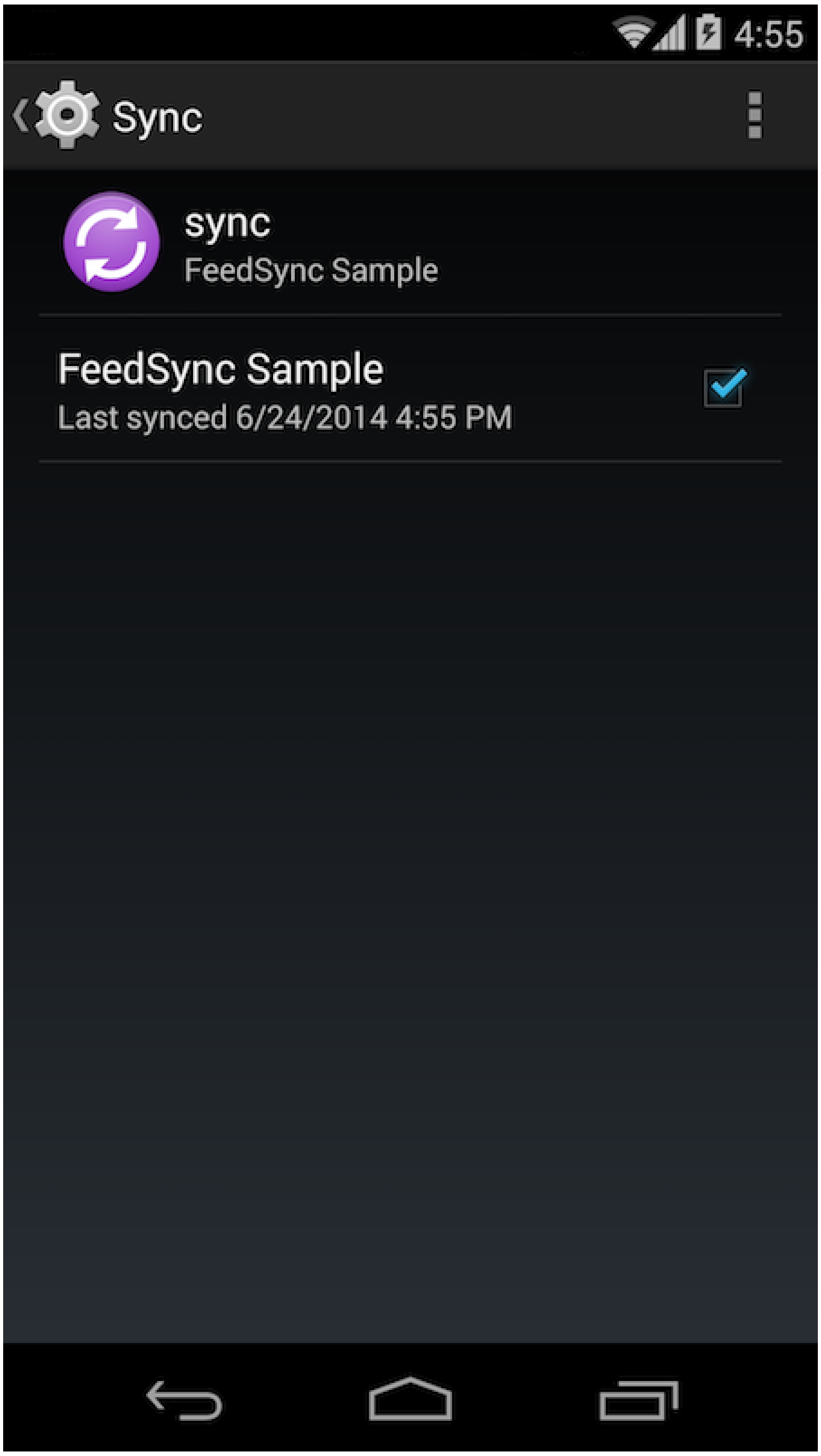
You should also be aware that GcmNetworkManager and JobScheduler were not built to perform tasks immediately or at a specific time. They are meant for performing repeated or one-off, non-imminent work while keeping battery life in mind. An alarm clock or reminder app that needed any sort of precision in timing should not use either option. This is especially true after the introduction of Doze Mode in Android Marshmallow, which we will discuss shortly.

Matt Compton wrote a great post on GCM Network Manager that goes into more detail. Check it out!

Sync Adapter

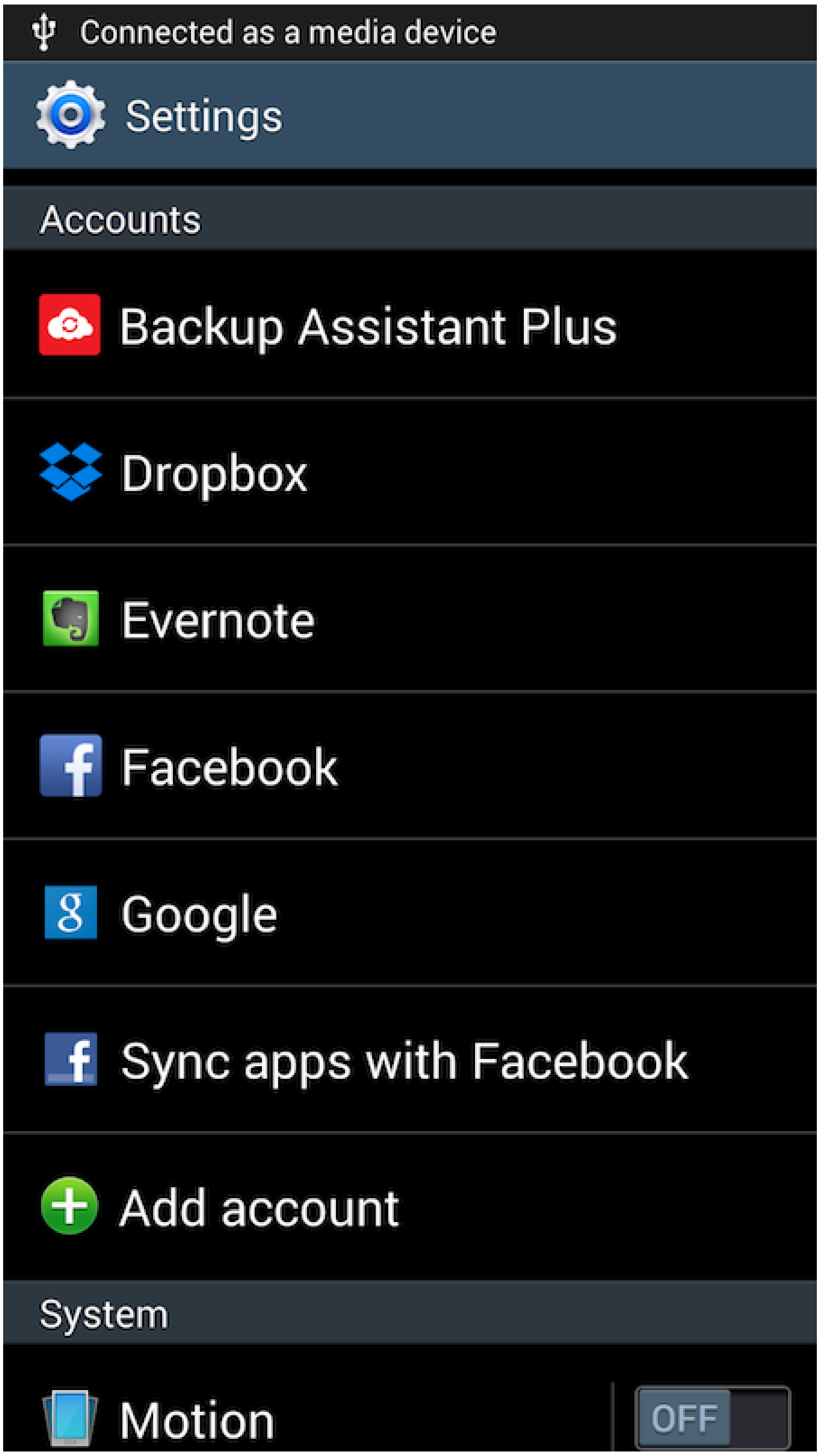
Sync Adapters were introduced with the intent of keeping Android devices and servers in sync. Syncs could be triggered from data changes in either the server or device, or by elapsed time and time of day. The system will try to batch outgoing syncs to preserve battery life, and transfers that are unable to run will queue up for transfer at a later time. The system will attempt syncs only when the device is connected to a network.

These concepts should sound pretty familiar by now. A nice optional feature included with Sync Adapters is the ability for users to see information about syncs and the ability to turn off syncing all together.



FeedSync Sample
Last synced 6/24/2014 4:55 PM ✓

In order to use a Sync Adapter, applications need an authenticator and a Content Provider. The expectation is that the application will have an Account and a database if syncing is needed between a device and server. This may not be true for this relationship all of the time, but Sync Adapters do require a subclass of AbstractAccountAuthenticator and ContentProvider in order to function. Applications that do not need these classes but still wish to use Sync Adapters can simply stub out a dummy authenticator and Content Provider. The stub classes are actually the bases for Google's handy Sync Adapter tutorial. Users can view Accounts in settings.



The SyncAdapter itself is declared in an XML file. Here you would specify the Account and ContentProvider required as well as whether or not the user can view the Sync Adapter in Settings, if uploading is supported, if parallel syncs are allowed, and if the Sync Adapter is always syncable.

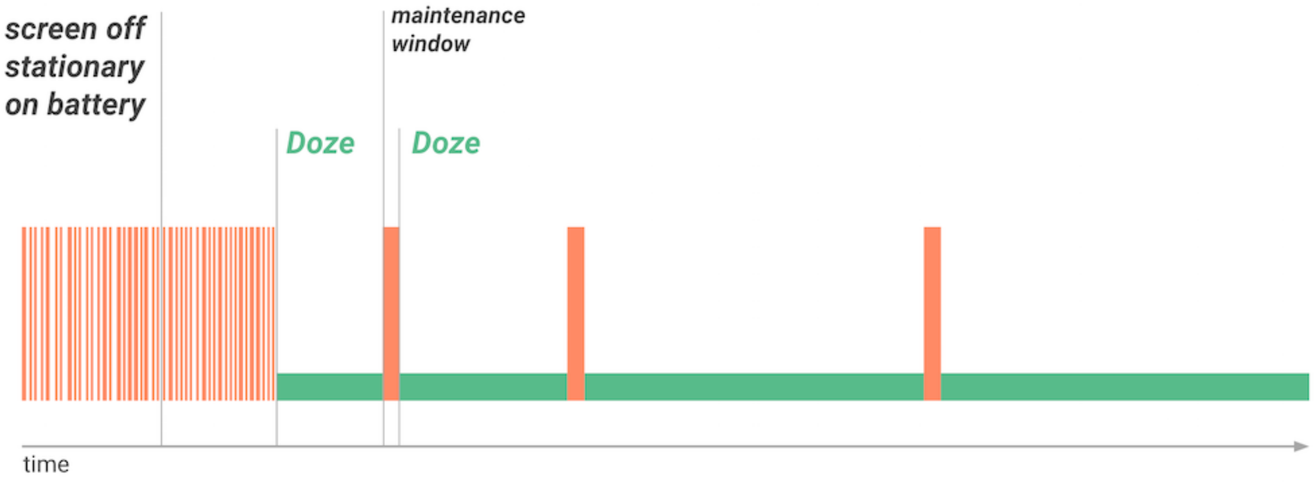
Syncs can occur automatically when changes occur locally if you create a ContentObserver subclass and register an observer for the Content Provider. When server data changes and you receive a notification through something such as a Broadcast Receiver, you can tell your local data to sync with the server by calling ContentResolver.requestSync(). Syncs can also be set to setSyncAutomatically() in order to sync whenever a network connection is already open, thereby frequently having up-to-date data, but never draining the battery by starting up the radio when it is not already running. Content Resolvers could have periodic syncs added to them with addPeriodicSync(), specifying an interval similar to those of AlarmManager. On demand syncs can occur by calling requestSync() on a Content Provider and passing in a flag specifying immediate sync or a more efficient sync that may wait a few seconds to batch other syncs. On demand syncing is discouraged due to battery drain, but is a common pattern seen on lists that are pulled downward to refresh.

Sync Adapters often require quite a bit more work to set up than the previous methods we discussed. However, if an application already supports Android Users and uses Content Providers, a lot of the boilerplate work is already done. Two huge upsides to Sync Adapter (compared with Job Scheduler or GCM Network Manager) are that it supports API level 7 and higher and does not require Google Play Services. Sync Adapters can leave you with the largest number of supported devices.

Doze Mode

Doze Mode was introduced in Android Marshmallow as a way to minimize battery drain while a user has been away from their device for a period of time. Doze Mode is automatically enabled on devices running Android API level 23 and higher. It is not a scheduler that developers need to implement. Rather, it affects the other schedulers we have already discussed.

Doze Mode is enabled after a period of time under these conditions: The user's screen has been off, the device has been stationary and the device has not been charging. When in Doze Mode, network access is suspended, the system ignores wake locks and standard alarms are deferred, including the running of Sync Adapters and JobSchedulers. There are maintenance windows that infrequently occur to allow all of these schedulers within the system to run at once, therefore preserving battery through infrequent and grouped network calls.



Notice that maintenance windows occur less frequently the longer the device is dozing. Once the user moves the device, turns on the screen or connects to a charger, normal activity resumes. You could imagine the battery benefits while users sleep or leave their phone unattended for a long period of time.

There are several scenarios developers need to concern themselves with following the introduction of Doze Mode. Alarms that need to fire off at an exact time can still do so with the AlarmManager using its setAndAllowWhileIdle() or setExactAndAllowWhileIdle() methods. Imagine an alarm clock app that did not wake its user in the morning due to Doze Mode! Another scenario would involve important incoming notifications, such as texts, emails or even tweets, if that is what users find important. For those cases, developers will need to use GCM with high-priority messages.

Notice the **"priority" : "high"** attribute-value pair below.

```
{
  "to" : "bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P1...",
  "priority" : "high",
  "notification" : {
    "body" : "Apparently this user thinks Tweets are important enough to disturb slumber
    "title" : "This cat's hair looks like Donald Trump's hair.",
    "icon" : "cat",
  },
  "data" : {
    "contents" : "https://twitter.com/trumpyourcat"
  }
}
```

Applications with core functionality requiring work to schedule outside of the Doze Mode maintenance windows can be whitelisted. Such whitelisting requires a special permission, and Google Play policies allow it only in certain cases.

These are currently the only scenarios in which developers can interrupt Doze Mode outside of the maintenance window. This isn’t usually a problem, since most scheduling should involve efficient, infrequent scheduling, but it is something to be aware of.

Summary

There are plenty of things to consider when trying to choose the right scheduler for Android applications:

- Will the scheduler need to exist outside the scope of the application lifecycle?
- Will the app need to schedule networking?
- Which scheduler will allow the support of the most users?

Let’s summarize the main pros and cons of the different methods we dove into:

- When using the AlarmManager, keep alarm frequency and device waking to a minimum, add randomness to the alarm’s timing when performing network requests so as to not overload a server, avoid clock time and precise timing unless the application needs to notify the user of something at a specific time, and consider another scheduler for networking.
- JobScheduler provides efficient background scheduling, but only if your user is running API level 21 or higher (that’s about 25% of users at the time of writing this post).
- GCM Network Manager provides efficient background scheduling—if your user has Google Play Services installed (this excludes Amazon devices). GCMTaskService is typically simpler to manage and more difficult to mess up than JobService.
- Sync Adapters are a great solution to sync local data with server data, especially if you already have User authentication and use Content Providers in your application. If you need to support devices running earlier than Android API level 21 in addition to devices without Google Play Services, Sync Adapter may be your best option.
- Doze Mode is wonderful in that developers do not need to do a thing to implement this system battery efficiency. AlarmManager and GCM Network Manager are the only options for interrupting a Doze.

Hopefully this post provides a good starting point in choosing the right path for background work on Android. With Android N already out for preview, and a planned release expected later in 2016, stay tuned for any updates about these topics. The documentation for behavioral changes mentions an update to Doze Mode and suggests JobScheduler as an option for handling new background optimizations.

MORE BY

Sean Farrell

RELATED POSTS:

Bluetooth Low Energy on Android, Part 2

Andrew Lunsford

Two-Way Data Binding on Android: Observing Your View with XML

Andrew Bailey

Recent Comments

comments powered by Disqus

Subscribe to Our Newsletter