NTT DOCOMO CONFIDENTIAL
Copyright Ⓒ 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

Android アプリ作成ガイドライン

～効率的な通信制御に向けて～

第 2.1 版

2014 年 3 月 7 日
株式会社 NTT ドコモ

目次

1 章　はじめに - 5 -
1.1.　対象 - 5 -
1.2.　ドキュメントの構成 - 5 -
1.3.　関連文献 - 6 -
2 章　モバイルネットワーク概要 - 7 -
2.1.　Preservation 機能 - 7 -
2.2.　Fast Dormancy 機能 - 7 -
2.3.　ネットワークを効率的に利用するために考慮すべきこと - 8 -
3 章　ネットワーク効率的利用の具体策 - 10 -
3.1.　1アプリあたりの通信回数を減らす手法 - 10 -
3.1.1.　Polling の見直し - 10 -
3.1.2.　Push の利用を検討する - 11 -
3.1.3.　リトライ処理の見直し - 16 -
3.1.4.　携帯電話の状態に応じた制御の切り替え - 19 -
3.1.5.　キャッシュの活用 - 19 -
3.1.6.　不要な通信を見つけるための方法 - 20 -
3.2.　1携帯電話あたりの通信回数を減らす手法 - 20 -
3.2.1.　（補足）AlarmManager - 21 -
3.3.　通信を行う時刻を携帯電話間で分散させる手法 - 22 -
3.3.1.　AlarmManager 使用における注意点 - 22 -
3.3.2.　通信開始時刻の乱数による分散 - 22 -
3.3.3.　setInexactRepeating()メソッドのパラメータ - 23 -
3.3.4.　意図しないタイミングでの動作 - 23 -
3.4.　その他 - 27 -
3.4.1.　時刻設定 - 28 -
3.4.2.　スリープ状態からの WAKEUP - 28 -
3.4.3.　ウィジェットアプリにおける注意事項 - 28 -
3.4.4.　スクリーン ON 契機で処理を開始する場合の注意 - 29 -
3.4.5.　ファイルの圧縮による通信量の削減 - 30 -
3.4.6.　（補足）API Level 19 以降の AlarmManager の動作について - 30 -
4 章　サンプルコード - 31 -

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

商標について

- 「Android」は Google Inc. の商標または登録商標です。
- その他、掲載されている会社名、製品名、サービス名は各社の商標または登録商標です。
- 本書ではコピーライト及び商標・登録商標表記はしておりません。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

改版履歴

版	項目	種別	内容
1.0	—		初版作成
2.0	3.1.2.	追加	“Push の利用を検討する” を追加
	3.1.3.	変更	“3.1.2. リトライ処理の見直し” を 3.1.3.項へ移動
	3.1.4.	変更	“3.1.3. 携帯電話の状態に応じた制御の切り替え” を 3.1.4.項へ移動
	3.1.5.	追加	“キャッシュの活用” を追加
	3.1.6.	追加	“不要な通信を見つけるための方法” を追加
	3.4.4.	変更	“スクリーン ON 契機で処理を開始する場合の注意” の内容を追記
	3.4.5.	追加	“ファイル圧縮による通信料の削減” を追加
	4 章	追加	サンプルコードを追加(図 11 への追記、図 12 追加)
2.1	3 章	追加	API Level 19 以降での AlarmManager の動作変更に関する補足を追加
	3.4.6	追加	API Level 19 以降での AlarmManager の変更点を追加

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

1章　はじめに

このドキュメントでは、アプリの応答性とバッテリ消費量について、バランスをとりつつ改善するためのアプリ実装手法について主に通信制御の側面から解説します。ユーザがアプリを使用する上で心地よいと感じてもらうためには、アプリがユーザの操作に対して速やかに反応することや、情報を適時更新することなど、ユーザを待たせない工夫が重要です。しかし一方で、処理を頻繁に行うことは携帯電話に大きな負荷をかけ、バッテリ持ちが短くなるなどユーザビリティを低下させる可能性があります。このようにアプリの応答性確保とバッテリ消費量の低減は相反する場合もあるため、適切なバランスをとりつつ改善することが重要です。

アプリの応答性確保やバッテリ消費量の低減を行ううえでの工夫には多くの観点がありますが、本ドキュメントでは主に通信制御に注目して具体的な手法を紹介します。なぜなら、基本的に通信は携帯電話のバッテリを消費する主要因であるからです。アプリは提示する情報を更新するためにネットワークを随時利用する場合があります。この際、適切なタイミングで情報を更新しかつバッテリ消費量を抑えるには、携帯電話がネットワークとの間の接続をどのように管理しているのかの知識が役立ちます。本ドキュメントでは、一般的なアプリ開発者にはあまり馴染みがないと思われる携帯ネットワークに関する背景知識の説明をした上で具体的な改善手法を説明し、なぜそのように実装すべきなのかを理解できるよう記載しています。

本ドキュメントで紹介する手法を用いると、アプリが携帯ネットワークを効率的に利用できることになります。その結果、手法を採用したアプリの使用感が向上することはもちろんですが、加えて周囲にもよい影響を与えます。有限である携帯ネットワークを効率的に利用することによって、携帯電話で動く他のアプリが通信を行う余地が増えますし、その携帯電話ユーザの周囲にいる他のユーザが通信を行う余地も増えます。アプリを開発する際には、是非本ドキュメントを活用いただければ幸いです。

1.1. 対象

本ドキュメントでは Android 向けアプリを対象としています。したがって、このドキュメントでは、Android アプリケーション開発の基礎的な知識を有する読者を対象としています。

1.2. ドキュメントの構成

本ドキュメントでは、2章にて携帯電話とネットワークの間で行われる制御について解説します。2章の内容は、3章で解説する具体的な手法が有効であることを理解するための背景説明にあたります。結果としての手法のみを実装したい場合は、2章を読まずに3章から読み始めても構いません。3章では、具体的な手法を系統立てて解説します。最後に4章では、サンプルコードを提示し

ます。本ドキュメントの内容をよりよく理解するための参考としてお使いください。

1.3. 関連文献

本ドキュメントは Google の公式ドキュメントを参考に作成しております。
API の使用法などについては Android Developers を参考ください。
http://developer.android.com/develop/index.html

また、本ドキュメント以外にも、GSM Association から提供されている、スマートフォン向けアプリ開発に関するガイドライン「Smarter Apps for Smarter Phones」にも有益な記載があります。
http://www.gsma.com/newsroom/ts-20-smarter-apps-for-smarter-phones

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

2章　モバイルネットワーク概要

スマートフォンには、従来のフィーチャーフォンとは異なり、基本的に常時接続状態を維持しようとする特性があります（ここでいう常時接続とは、携帯電話～インターネット間において、いつでもIPパケットの送受信が可能(IP Reachable)な状態を指します）。しかしながら、常時接続の状態を維持するには無線ネットワークリソースを保持し続けている必要があります。無線ネットワークリソースの保持は、保持していないときに比べて多くのバッテリを消費します。
そのため、通信をしない時には無線リソースを解放することが望ましい状態となります。この解放処理は特別アプリが意識することなく、ネットワークと携帯電話によって自動的に実行されます。本章ではその仕組みの概要を説明したうえで、アプリ実装時に考慮すべきポイントを解説します。

2.1. Preservation 機能

携帯電話ネットワークでは、一定時間無通信の状態が続いた場合、ネットワーク側からの要求を契機に無線ネットワークリソースのみを解放する機能（Preservation 機能）が備えられています。Preservation 機能は、常時接続状態を維持したまま無線ネットワークリソースのみを解放するものです。この状態は Preservation 状態と呼ばれ、携帯電話のバッテリ消費量の面からも無線ネットワークリソースの有効利用の面からも効果的な状態です(図 1)。

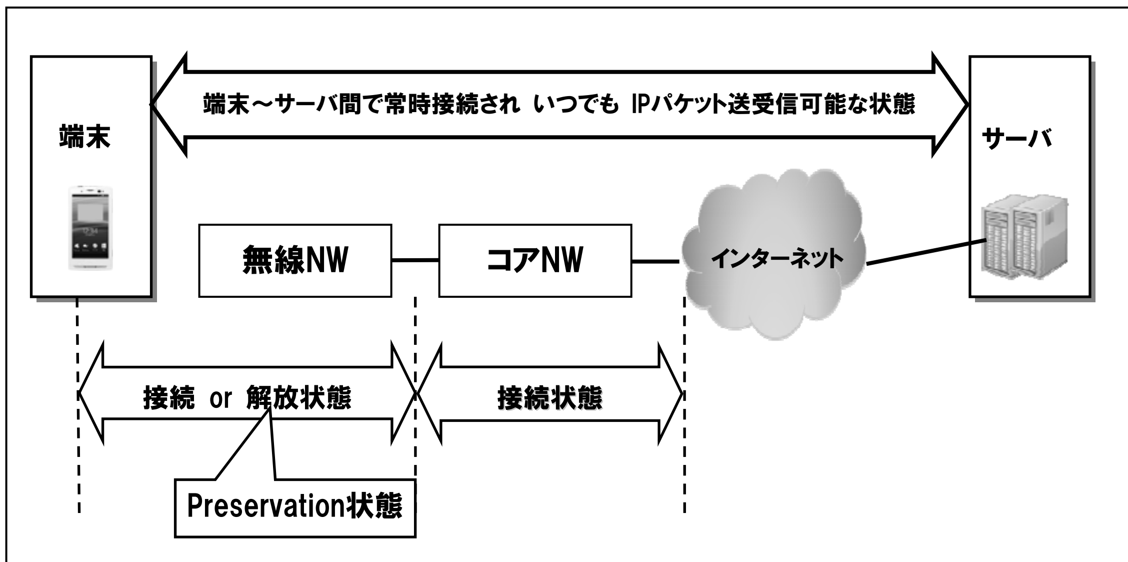


図 1 Preservation 状態

2.2. Fast Dormancy 機能

携帯電話の中には、更なるバッテリ消費量抑制や無線ネットワークリソースの効率的な利用を目的に、携帯電話からネットワークに対して、無線ネットワークリソースの解放状態（Preservation 状態）またはバッテリ消費が抑えられる無線確立状態（Battery Efficient 状態：Cell-PCH など）

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

への遷移の要求を行う Fast Dormancy 機能と呼ばれる機能を備えるものもあります。Fast Dormancy 機能は、無通信状態におけるバッテリ消費量抑制が期待できます(図 2)。

2.3. ネットワークを効率的に利用するために考慮すべきこと

先に述べたように、Preservation 機能および Fast Dormancy 機能によって、バッテリ消費量抑制や無線ネットワークリソースの効率的利用が期待できます。一度 Preservation 状態になった携帯電話は、再び通信が必要になったタイミングで再び無線ネットワークリソースの取得処理を行い、その後通信を行います。通信が終了すれば再び Preservation 機能および Fast Dormancy 機能が働き無線ネットワークリソースを解放します。このように、携帯電話の無線通信モジュールは、通信がないときは休み、通信が必要な時は起きて働くというサイクルを繰り返します。

問題となるのは、このサイクルが短時間で繰り返され、無線ネットワークリソースの取得処理および解放処理が多数行われる時です。無線ネットワークリソースの取得処理および解放処理では、携帯電話とネットワークの間で、複数回のネットワーク制御信号の送受信が行われます。1 回の取得および解放で送受信されるネットワーク制御信号の量はそれほど多くはありませんが頻繁に行われる場合は大きな負荷がかかる可能性があります。

そのため、携帯電話の通信はぱらぱらと複数回行うのでなく一度にまとめて実行するほうが、無線ネットワークリソースの取得処理および解放処理の回数を抑制でき効率が良いといえます。通信をまとめるには、1アプリの中で通信をまとめる方法の他、携帯電話内のアプリが同時に通信することで携帯電話全体としての通信をまとめる方法があります。

一方、アプリに必要なユーザビリティを損なわない範囲で、アプリの通信回数を下げることももちろん有効な手法になります。

また、アプリを実装する上で、複数の携帯電話で同時にアプリが通信を行わないようにすることも重要です。多くのユーザに利用されるアプリでは、同時に通信が発生した場合に、サーバやネットワークの処理が滞り、通信に通常より多くの時間がかかる場合があります。このような通信の集中がおこらないように通信開始時刻を分散させるアプリ実装にすることは、アプリの応答性とバッテリ消費量を削減する上で望ましい実装になります。

続く3章では、上記の考慮点を踏まえた上で有効な手法を紹介します。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

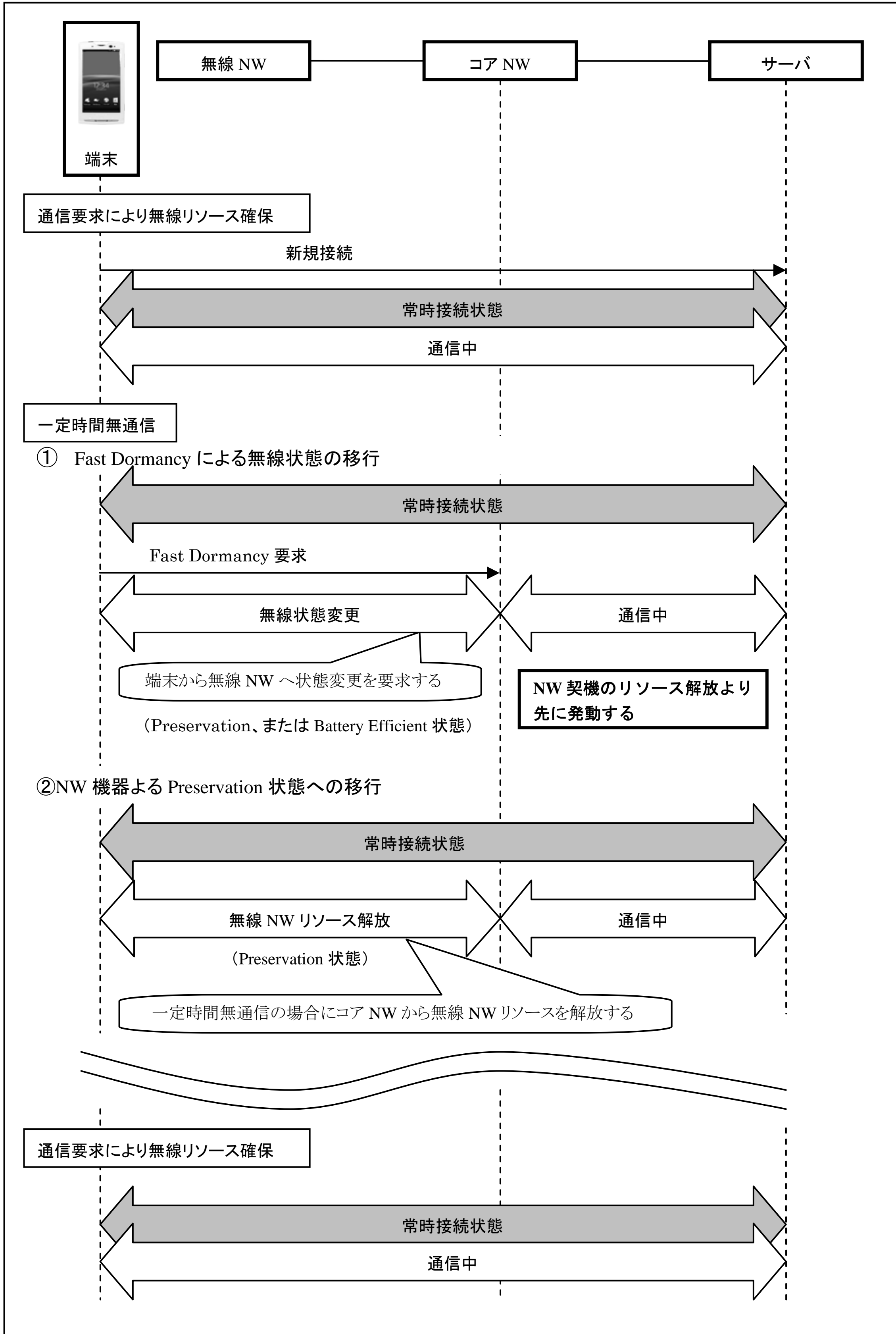


図 2 Fast dormancy 機能

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

3章　ネットワーク効率的利用の具体策

本章では、携帯電話ネットワークを効率的に利用しアプリの応答性とバッテリ消費量についてバランスをとりつつ改善するために、アプリ開発者が採用を検討することが望ましい手法について解説します。手法は3種類に大きく分かれます。

１、1アプリあたりの通信回数を減らす手法
２、他アプリと同時に通信をすることで1携帯電話あたりの通信回数を減らす手法
３、通信を行う時刻を携帯電話間で分散させる手法

なお、本章は、API Level 18(Andoroid 4.3)以下に関しての記述となっています。
API Level 19(Android 4.4)以上でのみ提供するアプリの処理については、動作が一部変更になっており、3.4 章に補足で記しました。

3.1. １アプリあたりの通信回数を減らす手法

3.1.1. Polling の見直し

ここでいう Polling は、携帯電話からサーバに対して一定の間隔で問い合わせのための通信を行う実装方法（PULL 型）を指します。この方法は主に擬似的な PUSH 処理の実現や、セッションを保持するなどの目的で用いられています。

基本的に Polling は、サーバ側に取得すべき新たな情報があるかないかに関わらず通信を行うため、その確認頻度が高い場合には携帯電話とネットワークへの負荷が大きい処理になります。Polling を行うアプリを作成する場合には、Polling 間隔が短くなりすぎていないか注意してください。
実際にサーバ側の情報が更新される頻度が少ないわりに、更新を即座にアプリで認識するために短い間隔で Polling を行っているアプリについては、Polling の使用を止めることを検討すべきです。代わりに、Push 型の情報通知の仕組みを用いることが多くの場合、リアルタイム性の面でも効率性の面でも適切になります。Android においては、Google Cloud Messaging（GCM）と呼ばれる Push 通知の仕組みが提供されています。

3.1.1.1. Polling 間隔を適切に設定する

Polling を行うパケットサイズが小さい場合でも、3 分～5 分程度といった比較的短い間隔で Polling 処理による通信を行うことは、バッテリ消費量の顕著な増加につながります。パケットが小さくとも、通信を開始するにあたっては、無線ネットワークリソースの取得するために、ネットワーク制御信号の送受信が起きる可能性があるためです（2 章参照）。ネットワーク制御信号の送受信が起きる場合には、携帯電話の各種処理の中でも比較的多くのバッテリを消費します。

また、取得対象となるデータの更新頻度に比べて短い間隔でPolling処理が行われた場合、結果として有効なデータを得られる見込みのない通信が発生することが想定されます。
そのため、Polling による通信を行う場合には、有効なデータを得られる見込みのない通信の発生頻度を減らすよう、Polling によって取得するデータの更新頻度を考慮して Polling 間隔を決定するようにしてください。
さらに、Polling 処理で有効なデータを得られなかった場合には、それまでと同じ間隔で Polling 処理を行うのではなく、有効なデータを得られなかった Polling 処理の実施回数や、ユーザの最終操作時刻からの経過時間に応じて、段階的(初回 1 分、2 回目 10 分、3 回目 100 分、など)に Polling 間隔を広げるなどの制御を実装することを検討してください。段階的な Polling 間隔の制御方法については 3.1.3 節で詳しく説明します。

3.1.2. Push の利用を検討する

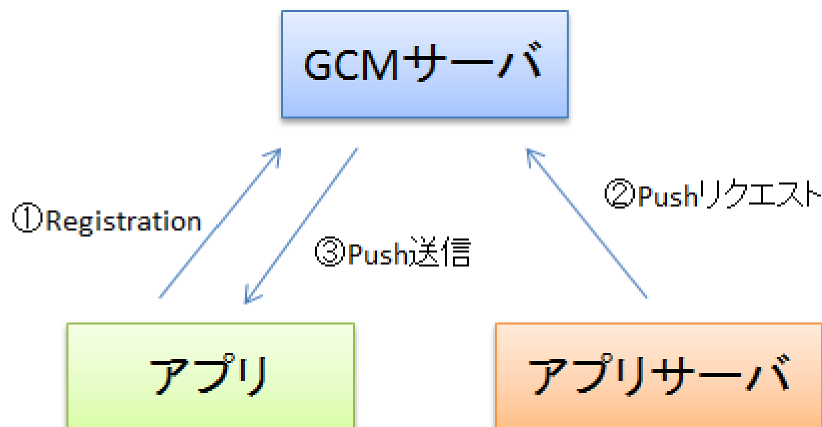
Polling を利用した擬似的な PUSH 型ではなく、サーバから携帯電話へ通知を行う PUSH 型機能として、Android では Google Cloud Messaging(GCM)機能が提供されています。Android2.2 以上の Google Play 搭載端末にてご利用いただけます。
GCM を利用すると、任意のタイミングで携帯電話上のアプリに通知メッセージを送ることができるため、Polling 処理によって一定間隔で通信を行うよりも効率的にデータの送受信を行うことができます。アプリの処理内容によっては Polling 処理ではなく GCM を利用することを検討してください。
GCM の詳細は以下の Google のサイトを参照してください。
http://developer.android.com/guide/google/gcm/index.html

3.1.2.1. Push を適用すべきケース

サーバ側の情報の変化を取得したい場合はPollingではなくPushを使うようにしてください。ただし、サーバ側のデータ更新頻度が非常に高い場合は、その更新頻度で Push を送信することは避けるべきです。このような場合は複数回の更新頻度に対して 1 度の Push 送信にするなど工夫して下さい

3.1.2.2. GCM の概要

GCM による Push を受け取るためには、予めユーザ毎にアプリを GCM サーバへ登録しておく必要があります。GCM の概要図は下図のようになります。



事前に GCM を利用するアプリから GCM サーバへ登録(Registration)をします(①)。
GCM サーバはアプリサーバから Push リクエストを受ける(②)と登録済みの Android 端末へ Push を送信します(③)。このとき Android 端末の在圏状況などにより Push を送信できなかった場合は GCM サーバへ一時保管され、端末が受信できるようになると再送されます。未達の場合の GCM サーバでの保管期間や再送の有無は開発者側でコントロールできます。

3.1.2.3. ネットワークトラフィックを考慮した GCM の使い方

GCM を利用する際に気をつけなければならない点は、不必要な Push 送信を極力防ぐため、Push リクエストの頻度を調整することです。なぜならば、Push 送信を減らすことはネットワークトラフィックと端末の電力消費の削減に貢献するからです。

GCM による Push の目的は二通りに大別できます。

(1) 更新や同期の促しのための Push 通知
(2) ペイロードを含む Push

(1)の例としては、メールの新着通知、サーバの更新通知などがあげられます。
ユーザにとってこれらの通知は最新の 1 件だけを受け取れば十分であり、同じ内容のメッセージが何通も届いてしまうとネットワークトラフィックやユーザビリティの観点では好ましくありません。
したがって、このような目的の Push はメッセージをグルーピングして常に最新 1 件のみが Push されるようにしてください。これにより不必要な Push 送信を防ぐことができます。
メッセージのグルーピングについては、後述の collapse_key をご参照ください。

(2)の例は、個々の Push のペイロードに意味のあるデータが格納されている場合です。
たとえば、インスタントメッセンジャーアプリのメッセージなどが該当します。
このようなメッセージはすべてのメッセージが Push されるべきであり、グルーピングしてしまうわけにはいきません。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

アプリサーバから GCM サーバへ Push をリクエストする際、メッセージの保管期限やグルーピング要否を指定することができます。
Push リクエスト時において重要なパラメータは以下の3つです。
GCM を利用する際は、メッセージの特性や目的に合わせて下記のパラメータを適切に設定して下さい。

① collapse_key
同じ collapse_key を指定することでメッセージはグルーピングされます。同グループのメッセージがサーバに複数ある場合は最新のもののみが送信されます。下記②の time_to_live を指定した場合は必須項目となります。

② time_to_live
端末へ即時配信できなかったメッセージは一旦 GCM サーバで保管され、端末の状態が Active になったタイミングで送信されます。この保管期間を指定するパラメータが time_to_live です。time_to_live を0に指定すると、即時配信できなかった場合は即時破棄されます。Push のリクエストを行ったタイミングで送信されないと意味をなさないメッセージに対しては0を指定するようにしてください。

③ delay_while_idle
この値を true にすると、端末がアイドル状態のときには Push は送信されません。端末は Push を受信すると Wakeup し電力をより消費します。したがって、即時性が求められないメッセージの場合はこの値を true に指定してください。

3.1.2.4. GCM の使い方

GoogleCloudMessaging API を利用することにより簡単に GCM を利用するアプリを作成することができます。GoogleCloudMessaging は GooglePlayService に含まれており、AndroidSDKManager より取得できます。
※GCMBroadcastReceiver や GCMBaseIntentService など gcm.jar に含まれる API は現在では非推奨となっています。
http://developer.android.com/google/gcm/client.html

クライアントアプリの作成手順は下記の手順です。

① Android プロジェクトに GooglePlayService のライブラリを追加する。
② GCM への Registration/Unregistration 処理を実装する。
・GoogleCloudMassage のインスタンスを取得する。
・register()メソッドで、GCM サーバへ Sender ID を渡して Registration を行う。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

・unregister()メソッドで Unregistration を行う。

③GCM サーバからの Push 受信時に Android システムが発行する BroadcastIntent を受信する BroadcastReceiver を実装する。
・GoogleCloudMassage のインスタンスを取得する。
・メッセージタイプによって行いたい処理を実装する。

④ GCM を利用するための設定を AndroidManifest に記述する。
・実装した Activity と BroadcastReceiver を記述する。
・GCM 利用時に必要なパーミッションを記述する。
・③で作成した BroadcastReceiver に GCM 利用時に必要なインテントフィルタを記述する。

```
/**
 * Registration の実装例
 * AsyncTask を利用し、Registration 処理を非同期で実行します
 */
private void registerBackground() {
    new AsyncTask<Void, Void, String>() {
        @Override
        protected String doInBackground(Void... params) {
            try {
                if (gcm == null) {
                   // GoogleCloudMessaging のインスタンスを取得
                    gcm = GoogleCloudMessaging.getInstance(context);
                }

                /**
                   * register()メソッドで SENDER ID を渡し Registration を行う。
                   * 戻り値として RegistrationID を受け取る。
                   */
                regid = gcm.register(SENDER_ID);

            } catch (IOException e) {
                //Registration ID の取得失敗時の処理
            }
            return regid;
        }

        @Override
        protected void onPostExecute(String regid) {
            //払い出された RegistrationID をメインスレッドで扱う場合はここで実装
        }
    }.execute(null, null, null);
}
```

```
/**
* Push 受信時に発行される Intent を受け取る BroadcastReceiver の実装例
*/

@Override
public void onReceive(Context context, Intent intent) {

   //GoogleCloudMessaging のインスタンスを取得
   GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(context);

   //Push のメッセージタイプを取得
   String messageType = gcm.getMessageType(intent);

   // メッセージのタイプによって処理をわける
   if(GoogleCloudMessaging.MESSAGE_TYPE_SEND_ERROR.equals(messageType)) {

      // エラー時の処理


   }else if(GoogleCloudMessaging.MESSAGE_TYPE_DELETED.equals(messageType)){

     // GCM サーバ上でメッセージが削除されたときの処理

   }else if(GoogleCloudMessaging.MESSAGE_TYPE_MESSAGE.equals(messageType)){

     // 通常のメッセージを受信したときの処理

   } else{

     // 上記以外の場合の処理

   }
}
```

```
<!—GCM 利用時の AndroidManifest.xml の例(GCM 関連部分を抜粋) -->

<uses-sdk
<!-- 必須:GCM を利用するためには、minSdkVersion は”8”以上でなけらばならない -->
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <!-- 必須:GCM のメッセージを受け取るためのパーミッション -->
    <uses-permission
android:name="com.google.android.c2dm.permission.RECEIVE" />

    <!-- 必須:インターネットに接続するためのパーミッション -->
    <uses-permission android:name="android.permission.INTERNET" />

    <!-- 任意:Android4.04 未満の端末では Google アカウントが必須 -->
    <uses-permission android:name="android.permission.GET_ACCOUNTS" />

    <!-- 任意:GCM からのメッセージを受信する際に WakeLock が必要な場合 -->
    <uses-permission android:name="android.permission.WAKE_LOCK" />

<!--
       任意:他のアプリから GCM 機能の利用をされないようにするための独自パーミッション
        name 属性は<パッケージ名>.permission.C2D_MESSAGE でなければならない。-->
    <permission
        android:name="com.example.newgcm.permission.C2D_MESSAGE"
        android:protectionLevel="signature" />
    <uses-permission
              android:name="com.example.newgcm.permission.C2D_MESSAGE" />


        <!-- Push 受信時に発行される BroadcastIntent を受け取る receiver -->
    <receiver
        android:name=".GcmBroadcastReceiver"
        android:permission="com.google.android.c2dm.permission.SEND" >

        <!-- Intent フィルターを定義。category にはアプリのパッケージ名を指定 -->
        <intent-filter>
           <action android:name="com.google.android.c2dm.intent.RECEIVE" />
           <category android:name="com.example.newgcm" />
        </intent-filter>
    </receiver>
```

3.1.3. リトライ処理の見直し

アプリが通信を行おうとした際、何らかの理由で通信に失敗することがあります。このとき、すぐに通信のリトライ処理を行ったり、通信が成功するまでリトライ処理を繰り返したりといった実装は避けるべきです。無線環境が不安定なことが通信失敗理由である場合、データ送受信を行うことができないにもかかわらず、携帯電話は無線ネットワークリソースの取得を試みることになるためです。既に解説したとおり、無線ネットワークリソースの取得処理は比較的バッテリ消費量の大きい処理となります。そのため、短い間隔でのリトライ処理や、リトライ処理を繰り返すような実装はバ

ッテリ消費量の増大につながります。

特にバックグラウンドで動作するアプリにおける処理などユーザを明示的に待たせることがない処理に関する通信エラー発生時には、Exponential Backoff 制御を行うことを推奨します。Exponential Backoff 制御とは、初回のリトライ処理は 1 分後、次のリトライ処理はその 2 分後、次のリトライ処理はさらに 4 分後にというようにリトライ回数が増えるにしたがって、リトライ処理を行う間隔を徐々に広げていく制御です。

また、通信が成功するまでリトライ処理を繰り返すといった実装はせず、リトライ回数制限や経過時間期限を設定し、条件を満たした場合に以後のリトライ処理を行わないようにする制御も検討してください。

以下に Exponential Backoff 制御でのリトライ間隔の例を示します。

表 1 Exponential Backoff 制御によるリトライ間隔の例

リトライ回数	リトライ間隔	経過時間
—	—	0 分(通信失敗)
1	1 分	1 分
2	2 分	3 分
3	4 分	7 分
4	8 分	15 分
5	16 分	31 分
…	…	…

以下の図 3 では、`calcWaitTime()`メソッドで、リトライを何度行ったか表す値を引数として待ち時間を計算しています。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

```
/** スロット時間 msec */
static final int SLOT = 1000;

/** 指数上限  */
static final int EXP_MAX = 10;

/** 最大リトライ回数 */
static final int RETRY_MAX = 15;


/**
 * 通信処理を含む何らかの処理を行います。
 */
private void doSomething() {
    int retry = 0; // リトライ回数
    while (retry <= RETRY_MAX) {
        boolean ret = doCommunicate();
        if (ret) {
            // 通信に成功したらループを抜ける。
            break;
        }

        // ★待ち時間を取得する。
        double wait = calcWaitTime(retry);

        try {
            Thread.sleep((long) wait);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        retry ++;
    };
}

/**
 * バックオフ制御のための待ち時間を計算する。
 * @param count リトライ回数。
 * @return 待ち時間。
 */
private double calcWaitTime(int count) {
    return SLOT * Math.pow(2,
            ((count < EXP_MAX) ? (double) count : (double) EXP_MAX));
}

/**
 * 通信処理を行う。
 * @return 通信が正常終了したら true。エラーが発生したら false。
 */
private boolean doCommunicate() {
    return true;
}
```

図 3 Exponential Backoff 制御例

3.1.4. 携帯電話の状態に応じた制御の切り替え

通信処理は携帯電話の処理の中でバッテリ消費量が比較的多いものです。そのため、通信を行う前に携帯電話の状態を確認し、通信の適否・頻度を適切に制御するようにしてください。

主な制御例を示します。

- アプリ状態に応じた制御
  - ユーザが利用していない(バックグラウンドでの動作中)状態では通信頻度を下げる。
  - 最終利用時刻からの経過時間に応じて通信間隔を広げる(Exponential Backoff 制御)。
- ネットワーク状態に応じた制御
  - ネットワーク接続が存在しない状態では通信処理を開始しない
  - Wi-Fi 環境なら通信量を多く、3G/LTE 環境なら通信量を少なくする。
- バッテリ状態に応じた制御
  - 充電中なら最大限通信する
  - 非充電中なら通信頻度を下げる。
  - バッテリ残量が少ないときはさらに通信頻度を下げる。

アプリ状態に応じた制御例のうち、ユーザが利用していない状態であることを検知する方法には以下のようなものがあります。

- アプリが最前面に表示されているか判定する方法
  `Activity` やアプリが切り替わったり、`Dialog` が表示されたりした場合など、最前面の `Activity` が切り替わりフォーカスの状態が変化すると、`Activity.onWindowFocusChanged(boolean hasFocus)`メソッドが呼び出されます。`Activity` が最前面での表示でなくなった場合には、引数に `false` が設定されてこのメソッドが呼び出されます。アプリでは、このメソッドの引数に設定された値に応じて通信処理の制御を行います。
- 携帯電話の画面が ON の状態かを判定する方法
  `PowerManager.isScreenOn()`メソッドの戻り値が `false` であるかどうか、ブロードキャストの `ACTION_SCREEN_OFF` を受信したかどうか、などで画面の表示状態を検知します。

3.1.5. キャッシュの活用

Web サイトや Web アプリのように Web ベースのサービスの場合、ブラウザのキャッシュを有効利用することで通信量を削減することができます。特に下記の観点に留意してください。

・重複するデータを何度も取得していないか
・Cache-Control ヘッダは適切に利用されているか

また、Web サービスのオフライン化も通信量削減には有効な手段です。サービスの特性に合わせ

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

て利用を検討して下さい。

HTML5 では、Web サービスをオフライン化するための機能として主に下記の機能が利用できます。
・アプリケーションキャッシュ・・・Web コンテンツを明示的にクライアントにキャッシュしてオフラインで利用する機能
・WebStorage/ IndexedDB・・・クライアントのストレージを利用する API

3.1.6. 不要な通信を見つけるための方法

一般的なアプリ開発において、アプリ毎のネットワーク負荷や電力消費などを定量的に解析することは困難です。
AT&T 社が無料で提供している Application Resource Optimizer（以下、ARO）を利用することにより、これが容易になります。ARO はアプリ開発者に対して、非効率な通信を最適化する指針・気付きを与え、アプリの高速化・省電力化の実現を支援する開発者向けツールです。ARO の下記の機能を利用することにより、アプリによる不要な通信を見つけることが可能です。

・ネットワーク負荷状況の可視化
・無駄なデータ受信の検出
・Cache-Control の不適切な利用を検出

このツールを利用することで、3.1.5 のキャッシュ利用における留意点もチェックすることができます。ARO の詳しい使用方法は下記をご参照ください。

◇NTT ドコモ開発者情報 Blog
http://devlog.dcm-gate.com/aro/

3.2. 1携帯電話あたりの通信回数を減らす手法

2章で解説したように、無線ネットワークリソースが解放されている状態で通信を開始することは、ネットワーク制御信号を送受信することになり携帯電話に負荷をかける処理になります。一方で、一度無線ネットワークリソースを取得した後の実際のデータ送受信時のバッテリ消費量は、ネットワーク制御信号送受信時ほどのバッテリを必要とせず、その消費量は比較的少ないものとなります。

そのため、例えばあるアプリ A,B,C があり、それぞれがばらばらに 30 分間隔で通信を行うという場合に比べると、アプリ A,B,C がタイミングをあわせて 30 分間隔で通信を行うほうがバッテリ持ちが向上します。無線ネットワークリソースの取得・解放回数を抑えられる可能性があるためです。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

定期的に通信を行うアプリが他のアプリと同期して通信を行うには AlarmManager の setInexactRepeating メソッドを使う方法があります。setInexactRepeating メソッドを使用した場合には、多少時刻が不正確になりはしますが、Android が自動的に他のアプリと同じタイミングでアプリを起動してくれます。これを契機として通信を行うことで携帯電話とネットワークに負荷のかかる無線ネットワークリソースの取得・解放回数を抑えられる可能性があります。

3.2.1. （補足）AlarmManager

Android には AlarmManager という仕組みがあり、指定した時刻あるいは一定間隔で任意の処理を実行させることができます。AlarmManager はアラームやタイマー、定期的な処理を実行するなどの目的で利用されます。

AlarmManager の主なメソッドには以下のものがあります。

- set(int type, long triggerAtTime, PendingIntent operation)
  任意の時刻で実行する処理を登録します。
- setRepeating(int type, long triggerAtTime, long interval, PendingIntent operation)
  一定間隔で実行する処理を登録します。
- setInexactRepeating(int type, long triggerAtTime, long interval, PendingIntent operation)
  一定間隔で実行する処理を登録します。
  このメソッドは、setRepeating()メソッドに比べて、多少時刻が不正確となる代わりにバッテリ消費を抑えるようなスケジューリングが自動的に行われます。
  また、引数 interval に指定する値として有効な値は INTERVAL_FIFTEEN_MINUTES、INTERVAL_HALF_HOUR、INTERVAL_HOUR、INTERVAL_HALF_DAY、INTERVAL_DAY の 5 つで、それ以外の値を指定した場合は、setRepeating()メソッドと同等の振る舞いになります。
- cancel(PendingIntent operation)
  登録済みの処理を削除します。

```
/*
 * 定期処理を行う Service の Intent を作成し、
 * Service を開始する PendingIntent を取得する。
 */
Intent intent = new Intent(this, MyService.class);
PendingIntent pendingIntent = PendingIntent.getService(this, 0, intent, 0);

/*
 * 15 分毎の繰り返しアラームを設定する。
 */
AlarmManager am = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
am.setRepeating(
       AlarmManager.ELAPSED_REALTIME,
       SystemClock.elapsedRealtime() + 10000, // 最初のアラームは登録の 10 秒後。
       AlarmManager.INTERVAL_FIFTEEN_MINUTES,
       pendingIntent);
```

図 4 アラームへの時刻設定例

set()、setRepeating()、setInexactRepeating()の各メソッドには、実行する処理の

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

開始時刻を指定する形式として、以下の 2 種類があります。

- RTC 形式:携帯電話内の時計の時刻で指定する(絶対時刻)
- ELAPSED_REALTIME 形式:携帯電話が起動してからの経過時間で指定する(相対時刻)

3.3. 通信を行う時刻を携帯電話間で分散させる手法

多くのユーザに利用されるアプリでは、同時に通信が発生した場合に、サーバやネットワークの処理が滞り、通信に通常より多くの時間がかかる場合があります。このような通信の集中がおこらないように通信開始時刻を携帯電話間で分散させるアプリ実装にすることは、アプリの応答性とバッテリ消費量を削減する上で望ましい実装になります。

そのため、ある特定時刻に通信を行うようなアプリ実装は、可能な限り避けるべきです。アプリのユーザビリティを下げない範囲で携帯電話毎の通信時刻を分散するような実装としてください。
ただし、特定時刻に通信を行うような実装を明示的にしていない場合であっても、特定の条件がそろう場合には、意図せずして特定時刻に通信を行ってしまう場合があります。本節ではそのような状況を避けるための手法を紹介します。

3.3.1. AlarmManager 使用における注意点

AlarmManager(3.2.1 節参考のこと)を使う際に、時刻指定に RTC 形式を利用する場合には処理開始時刻(triggerAtTime)に定数を指定すべきではありません。処理開始時刻が分散するように計算した値を設定すべきです。たとえば、現在時刻から x 分後というように相対的に処理開始時刻を求めて設定するのは一つの方法です。ただしこの方法では基底となる現在時刻が特定の時刻に偏る事象が発生した場合には問題を生じますので注意が必要です(3.3.4 節にて後述します)。
可能であれば 3.3.2 項で解説するように乱数を使用する実装にし、携帯電話ごとにランダムに処理が開始されるようにしてください。

3.3.2. 通信開始時刻の乱数による分散

ある決まった時刻に定期的に通信を行う必要がある場合であっても、アプリが満たさねばならない仕様が許す範囲で携帯電話ごとの通信開始時刻を分散すること検討してください。
図 5 は、00 分 00 秒から 4 分 59 秒の間で処理開始時刻を分散する処理の例です。

なお、AlarmManager を使用した定時起動をしない場合であっても、Intent.ACTION_DATE_CHANGED や Intent.ACTION_TIME_TICK といった定時発生イベントをトリガとして通信処理を行う場合には、通信開始時刻を分散することが必要です。同様に適切な範囲で乱数によるオフセットをいれることで通信の分散を行ってください。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

```
/* 処理開始の目安時刻 */
long targetTime = (20 時 00 分);

/*
 * 分散のため乱数ジェネレータを生成し、遅延させる時間を取得する。
*/
Random random = new Random();
int offset = random.nextInt(5 * 60 * 1000); // ★処理開始を遅延させるオフセットの算出。

/*
 * 20 時 00 分から 20 時 05 分の間でランダムに処理開始時刻を設定する。
 */
AlarmManager am = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
am.set(AlarmManager.RTC,
       targetTime + offset, // 最大 4 分 59 秒、処理開始時刻を遅らせる。
       pendingIntent);
```

図 5 処理開始時刻分散例

3.3.3. setInexactRepeating()メソッドのパラメータ

3.2 項で解説したように、AlarmManager クラスの setInexactRepeating()メソッドは、setRepeating()メソッドに比べると、多少時刻に不正確さはありますが、バッテリ効率が良く実行待機中のバッテリ消費量が少ない特徴があります。また通信時刻集中の回避にもつながるため、一定間隔で実行する処理を登録する際には可能な限り setRepeating()ではなく setInexactRepeating()メソッドを使用するようにしてください。
なお、setInexactRepeating()メソッドに RTC パラメータを指定した場合、AndroidOS のバージョンによっては 00 分、15 分、30 分、45 分のようにある決まった時刻に処理が開始される場合があります。これにより、各携帯電話からの同時に通信が行われることになり、通信が集中する可能性があります。setInexactRepeating()メソッドを使用する場合には RTC パラメータではなく ELAPSED_REALTIME パラメータを使用するようにしてください。

3.3.4. 意図しないタイミングでの動作

意図的に特定時刻に通信を行う実装になっていない場合でも、携帯電話の状態によっては意図しないタイミングで通信が行われ、結果として特定時刻に通信処理が集中してしまう場合があります。

AlarmManager では、指定時刻になった際に携帯電話がスリープ状態であった場合のアラームイベントの通知方法として 2 通りの方法が提供されており、登録時にどちらかを選択します。提供されているアラームイベント通知方法は以下の 2 通りです。

- スリープ状態を解除(WAKEUP)してアラームイベントを通知する方法
  目覚まし時計やタイマーなどの機能を持つアプリのように、遅延なくアラームイベントが通知されることが期待されるアプリで多く用いられる方法です。
  AlarmManager への登録には、RTC_WAKEUP あるいは ELAPSED_REALTIME_WAKEUP を呼び出し時間の設定方法として指定します。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

- スリープ状態を解除せず、次に WAKEUP した際にイベント通知を通知する方法
  時間精度が求められないアプリで用いられる方法です。
  AlarmManager への登録には、RTC あるいは ELAPSED_REALTIME を呼び出し時間の設定方法として指定します。

例として、以下のようなアプリ A、アプリ B、アプリ C を考えます。

- アプリ A
  - 19:00 に WAKEUP する
  - 通信は行わない
- アプリ B
  - スリープ状態の場合 WAKEUP しない
  - 60 分毎の繰り返しアラームを以下のいずれかのメソッドで登録
    - setRepeating()
    - setInexactRepeating()
  - アラームによる起動後に通信を行う
- アプリ C
  - スリープ状態の場合 WAKEUP しない
  - 60 分後のアラームを set()メソッドで登録
  - アラームによる起動後に通信を行い再度 60 分後のアラームを set()メソッドで登録

図 6 は、これら 3 つのアプリが同時に異なるユーザ X とユーザ Y の携帯電話にインストールされている状況にて、どのタイミングでアラームによってアプリが起動するか通知されるかを表したものです。
スリープ状態の間は、アプリ B、アプリ C へのイベントは保留されます。その後 19:00 にアプリ A によって WAKE 状態になり、保留されたアプリ B、アプリ C へのイベントが同じタイミングで通知されます。
アプリ B は繰り返しアラームを登録しているので、その後の WAKEUP 状態の 19:00 台では、本来の分秒にイベントが通知されます。一方、アプリ C はイベント処理内で 60 分後のアラームを登録しているため、次回通知されるのは 20:00 となります。以降は毎時 00 分にアプリ C へイベント通知が行われます。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

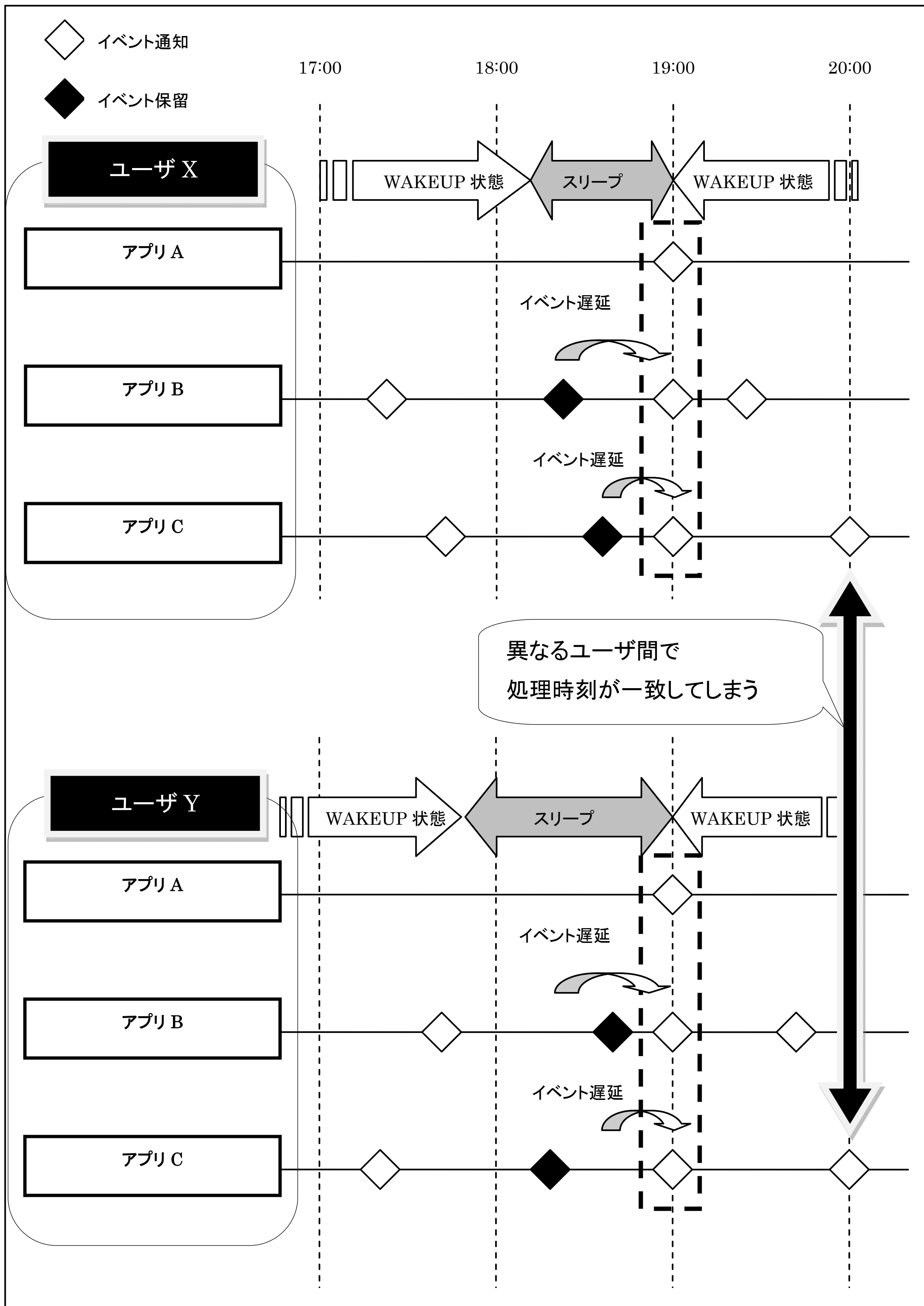


図 6 意図しない動作タイミングの集中

アプリ C のように、WAKEUP しない設定で次回起動時刻を処理実行の度に現在時刻にオフセットを加えて登録を行う場合、アプリ A のように特定時刻で起動するアプリが同じ携帯電話に存在してしまうと、アプリ C の初回処理開始時刻が携帯電話間で分散されていたとしても、アプリ A によって特定時刻で WAKEUP が起きてしまった以後は、多くの携帯電話で同じ時刻にアプリ C が通信を行うことになります。
このとき、アプリ B のように setRepeating()や setInexactRepeating()メソッドでアラーム登録を行っていると、異なる携帯電話でアプリ起動時刻が一致しにくくなります。例では、19:00 においては異なる携帯電話で同時に起動してしまいますが、それ以降は異なる携帯電話は異なる時刻で起動しています。

このため、定期的に処理を実行するために AlarmManager にアラーム登録する場合には、set()メソッドではなく、setRepeating()や setInexactRepeating()メソッドを使用するようにしてください。

また、アプリ B、アプリ C のような、スリープ状態の場合に WAKEUP しないアプリでは、異なる携帯電話が同時に通信(例では 19:00)を行わないようにするため、アラームイベントを受信しても即座に通信処理を開始するのではなく、ごく短時間でも処理開始時刻が分散されるような実装にすることを検討ください(3.3.2 項参照)。図 7 は、アラームイベント受信後の通信処理開始を、最大 60 秒遅延させる際の処理例です。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

```
/**
 * アラームを受け取るレシーバ。
 */
public class AlarmReceiver extends BroadcastReceiver {

    /**
     * アラームイベント通知の際に呼び出される。
     */
    @Override
    public void onReceive(Context context, Intent intent) {
        /*
         * 通信スレッドを開始する。
         */
        new CommunicateThread().start();
    }

    /**
     * 通信を行うスレッド。
     */
    private class CommunicateThread extends Thread {

        @Override
        public void run() {
            /*
             * 最大 60 秒間、処理開始を遅延させて
             * 通信処理開始時刻の分散を図る。
             */
            int offset = new Random().nextInt(60 * 1000);

            try {
                Thread.sleep(offset);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            // 通信処理を呼び出す。
            doCommunication();
        }

        /**
         * 通信処理を行う。
         */
        private void doCommunication() {

        }

    }
}
```

図 7 アラームイベント受信後の通信開始遅延

なお、AlarmManager を利用するアプリだけでなく、ACTION_DATE_CHANGED のようなブロードキャストアクションを受け取るアプリも、アプリ A のように他のアプリのイベント発生タイミングの同期対象となりやすいことに注意してください。

3.4. その他

ここでは、その他、アプリ開発時に考慮することが望ましい事項について解説します。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

3.4.1. 時刻設定

携帯電話には、携帯電話ネットワーク上の日付・時刻情報を基にして携帯電話の日付・時刻情報を自動的に設定・補正する機能があります。しかし、SIM カードが未挿入の状態やネットワーク圏外の場合にはネットワークから時刻情報を取得することができません。このようなとき、「0:00」など携帯電話固有の特定時刻が設定されます。また、自動時刻補正機能は携帯電話の電源を ON にした際にも動作します。

AlarmManager を利用する際などに携帯電話の日付・時刻情報を基にすると意図しないタイミングで処理が実行される場合があることに注意してください。たとえば、現在時刻(new Date()で取得)に一定のオフセット値を加えた値を RTC 形式で起動時刻として設定した場には, 時刻補正の成否によっては意図したタイミングで処理がされない可能性があります。そのため、現在から指定時間経過後に処理を実行したい場合には、RTC 形式でなく ELAPSED_REALTIME 形式で指定することがより好ましい実装です。このような場合には、現在システム時刻(SystemClock. elapsedRealtime()で取得)に一定のオフセット値を加えた値を ELAPSED_REALTIME 形式で起動時刻として設定ください。

3.4.2. スリープ状態からの WAKEUP

3.3.4 項で解説したように、AlarmManager では携帯電話がスリープ状態であっても指定の時刻で携帯電話を WAKEUP しアラームイベントをアプリに通知することができます。WAKEUP して通知することは時間精度が求められるアプリにとっては有効な方法ですが、多くのアプリがこの方法を利用していると携帯電話が WAKEUP する頻度が高くなり、バッテリ消費量の増大につながる可能性があります。

そのため、AlarmManager を利用するアプリでは、スリープ状態の携帯電話を WAKEUP する必要があるのかどうかを十分に検討し、不必要に WAKEUP するような実装は避けてください。なお、WAKEUP しない実装とする場合で、かつ通信を行うアプリでは、 節で解説した内容も考慮してください。

3.4.3. ウィジェットアプリにおける注意事項

ウィジェットアプリでも、不必要な通信処理を軽減するために、ウィジェットが画面に表示されていないときには、処理を行わない(処理を停止する)、あるいは頻度を減らすなどの実装をすることを推奨します。

なお、通信処理と同様に描画処理もバッテリ消費に影響を与えます。描画処理についてもウィジェットが画面に表示されていない場合に処理を行わないようにするなど、必要以上の描画処理を控えることでバッテリ消費量を抑えることができます。

なお、ウィジェットの定期処理は、Manifest ファイルに updatePeriodMillis 属性を指定することで定期処理が実装できますが、間隔が変更不可であり、スリープ中でも WAKEUP される実装となってしまうため柔軟なコントロールができません。そのため、ウィジェットの定期処理は可能な限り AlarmManager を用いて端末状態に応じた更新頻度となるような処理を作りこむことが望ましいです。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

3.4.4. スクリーン ON 契機で処理を開始する場合の注意

ユーザが端末操作を開始したタイミングで情報更新を行う等の処理をするために、スクリーンがON になったタイミングで通知される ACTION_SCREEN_ON インテントを使用しているアプリでは、意図しないタイミングで処理が行われる可能性があるため下記に注意ください。ACTION_SCREEN_ON インテントは、ユーザが端末のスリープを解除し操作を開始したタイミングのみならず、AlarmManager で起動した他のアプリがスクリーンを ON にしたタイミングでも通知されてしまいます。アプリとして、ユーザが端末操作を開始したタイミングでのみ処理したい場合は、間接的な ACTION_SCREEN_ON インテントを用いるのではなく、直接目的を達成できる ACTION_USER_PRESENT インテントを用いるべきです。ACTION_USER_PRESENT は、ユーザがロック画面を解除したタイミングで通知されます。ただし画面ロックの種別を《なし》に設定している端末では、ACTION_USER_PRESENT インテントが通知されません。したがって、ACTION_USER_PRESENT インテント受信契機の処理は実行されません。

このケースに対応するために、ACTION_SCREEN_ON インテント受信と同時にロック画面が表示されているかどうかを判定します。このときにロック画面が表示されなければ、画面ロックの種別を《なし》に設定している端末と判断し目的の処理を実装します。

```
/**
  *ユーザが端末操作を開始したタイミングを検知する Receiver の例
  */
@Override
public void onReceive(Context context, Intent intent) {
   String action = intent.getAction();
   KeyguardManager keyguard =
       (KeyguardManager) getSystemService(Context.KEYGUARD_SERVICE);

   //ロック画面が表示されているかを判定
   boolean isScreenLock = keyguard.inKeyguardRestrictedInputMode();

   //画面 ON 時にロック画面が表示されない場合
   if (action.equals(Intent.ACTION_SCREEN_ON) && !isScreenLock) {

  /** ロック画面を導入していないユーザが端末操作を開始したタイミングで情報更新を行う等の処理 */

   }

  //画面ロックが解除された場合
   if (action.equals(Intent.ACTION_USER_PRESENT)) {

   /** ロック画面を導入しているユーザが端末操作を開始したタイミングで情報更新を行う等の処理 */

   }
}
```

3.4.5. ファイルの圧縮による通信量の削減

ネットワーク経由で送受信するデータを圧縮することは通信量の削減に直接的な効果があります。特にテキストなどの高圧縮率が期待できるものについては大幅な通信量削減を見込めます。Android ではファイルを GZIP 圧縮する API が標準で用意されており圧縮・解凍処理は簡単に実装できます。

また、HTTP の GET リクエスト時にサーバに対して GZIP 圧縮要求することにより、レスポンスを GZIP に圧縮された状態で受け取ることができます。なおサーバが GZIP 圧縮に対応していない場合は、圧縮されていない通常のデータを受け取ることになりますので、クライアントアプリではその処理分けが必要となります。

アプリ側の実装上の注意点としては、圧縮・解凍処理は UI スレッドでは行わず、別スレッドで行うようにします。

3.4.6. （補足）API Level 19 以降の AlarmManager の動作について

API Level 19(Android 4.4)以降は、AlarmManager の一部メソッドの動作が電池持ちを考慮して変更になっているほか、新しいメソッドが追加されています。

targetSdkVersion を 19 以上に指定することで、これらの変更内容が有効になります。

- 新規に追加されたメソッド
  - setExact (int type, long triggerAtMillis, PendingIntent operation)
    - 発動時刻を厳密に指定して、アラームを登録するメソッドです。
  - setWindow (int type, long windowStartMillis, long windowLengthMillis, PendingIntent operation)
    - 発動時刻を特定の時間枠に指定して、アラームを登録するメソッドです。

- 動作が変更になったメソッド
  - set (int type, long triggerAtMillis, PendingIntent operation)
  - setRepeating (int type, long triggerAtMillis, long intervalMillis, PendingIntent operation)
    - 上記メソッドで登録したアラームは、targetSdkVersion19 以降では発動時刻が厳密ではなくなります。setInexactRepeating()メソッドと同様に、発動時刻が近接した他のアプリケーションのアラームを OS 側でまとめて実行するようになりました。

一定間隔で繰り返し実行する処理を登録する際は、setInexactRepeating() もしくは setRepeating()メソッドを用いてください。その際は、3.3.3 に従った実装を行ってください。厳密な時刻指定が必要な場合は、setExact()メソッドを用いてください。その場合においても、3.3.1・3.3.2 に従った実装を行ってください。

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

4章　サンプルコード

この章では、3 章で解説したネットワーク負荷軽減策を盛り込んだサンプルを示します。
サンプルの構成と概要は以下のとおりです。

- アラームを登録するアクティビティクラス
  定期的に処理を行うためアラームを登録するクラスです。
  処理開始時刻が分散されるように、乱数を用いて最初のアラーム通知時刻を決定しています。
  このサンプルでは、目覚まし時計のような時間精度は求めてないので、スリープ状態時に WAKEUP しない ELAPSED_REALTIME を使用しています。
- アラームから呼び出されるサービスクラス
  アラームイベントが通知されたら、通信スレッドを開始するクラスです。
  通信処理開始時刻の分散は通信スレッドで行うため、サービスクラスでは特別な実装はありません。
- 通信を行うスレッドクラス
  別スレッドで通信を行うクラスです。
  スレッドが開始されると、携帯電話状態を確認した後、最大 5 分の間でランダムなタイミングで通信処理を行います。通信処理に失敗した際には、Exponential Backoff 制御を用いて計算した間隔でリトライしています。
  doCommunicate()メソッドで通信処理を実装することを想定していますが、このサンプルでは実際に通信は行わず、"doCommunicate()!!!"というログを出力しています。
- ユーティリティクラス
  携帯電話の状態を確認する機能として、下記のメソッドを提供しています。
  - 携帯電話のネットワーク状態の取得
  - 携帯電話の充電状態の取得

  また、下記の汎用的なメソッドも提供しています。
  - GZIP 圧縮を行うメソッド
  - ホームアプリが表示中であるかどうかを判定するメソッド
- HTTP GET で GZIP 要求を行うクラス
  HTTP GET 時に GZIP 要求を行うクラスと使用例です。本クラスでは AsyncTask を利用し非同期で通信を行っています。

以下に、これらのサンプルコードを示します。

```
/**
 * アラームを登録するアクティビティ。
 */
public class SampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // アラームを登録する。
        registAlarm();
    }

    /**
     * アラームを登録する。
     */
    private void registAlarm() {
        /*
         * サービスを呼び出す PendingIntent を取得する。
         */
        Intent intent = new Intent(this, SampleService.class);
        PendingIntent pendingIntent
             = PendingIntent.getService(this, 0, intent, 0);

        /*
         * 30 分後から 35 分後までの間でランダムに開始する、
         * 60 分毎の繰り返しアラームを設定する。
         */
        AlarmManager am
            = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
        long trigger = SystemClock.elapsedRealtime()
             + (30 * 60 * 1000) + (new Random().nextInt(5 * 60 * 1000));
        am.setInexactRepeating(
            AlarmManager.ELAPSED_REALTIME,
            trigger,
            AlarmManager.INTERVAL_HOUR,
            pendingIntent);
    }
}
```

図 8 アクティビティクラス

```
/**
 * アラームから呼び出されるサービス。
 */
public class SampleService extends Service {
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
    }

    @Override
    public void onStart(Intent intent, int startId) {
        super.onStart(intent, startId);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        /*
         * 通信スレッドを開始する。
         */
        new CommunicateThread(getApplicationContext()).start();
        return super.onStartCommand(intent, flags, startId);
    }
}
```

図 9 サービスクラス

```
/**
 * 通信を行うスレッド。
 * 通信に失敗した場合は、Exponential Backoff 制御を利用してリトライする。
 */
public class CommunicateThread extends Thread {
    /** スロット時間[msec]。*/
    private static final int SLOT = 1000;

    /** 指数上限。*/
    private static final int EXP_MAX = 10;

    /** リトライ上限回数。*/
    private static final int RETRY_MAX = 15;

    /** リトライを行う最大時間(=30 分)。*/
    private static final long MAX_RETRY_PERIOD = 30 * 60 * 1000;

    private Context context;

    public CommunicateThread(Context context) {
        this.context = context;
    }

    /**
     * 通信処理を含む何らかの処理を行います。
     * 始めに端末のネットワーク状態や充電状態を確認し、
     * 場合によっては通信処理は行わずにリターンします。
     * リトライ上限回数かリトライ最大時間のいずれかに達するまで
     * Exponential Backoff 制御しながらリトライする。
     */
    public void run() {
        /*
         * ネットワーク圏外ならば処理を行わない。
         */
        if (!Utils.isConnected(context)) {
            return;
        }

        /*
         * 充電状態によって処理を行わないなどの制御を行う。
         *
         * ここでは非充電中の場合は、
         * 1/3 の確率で処理を行わない実装にしています。
         */
        if (!Utils.isCharging(context)) {
            int value = new Random().nextInt(3);
            if(value==0) return;
        }

        /*
         * 最大 5 分間、処理開始を遅延させる。
         */
        int offset = new Random().nextInt(5 * 60 * 1000);
        try {
            Thread.sleep(offset);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        int retry = 0; // リトライ回数
        long total = 0; // 経過時間。
        do {
            boolean ret = doCommunicate();
            if (ret) {
```

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

```
                // 通信に成功したらループを抜ける。
                break;
            }

            // ★待ち時間を取得する。
            double wait = calcWaitTime(retry);
            total += (long) wait;

            try {
                Thread.sleep((long) wait);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            retry ++;

        } while ((retry <= RETRY_MAX) && (total <= MAX_RETRY_PERIOD));

        /*
         * このスレッドを起動したサービスを停止する。
         */
        context.stopService(new Intent(context, SampleService.class));
    }

    /**
     * Exponential Backoff 制御のための待ち時間を計算する。
     * @param count リトライ回数。
     * @return 待ち時間。
     */
    private double calcWaitTime(int count) {
        return SLOT * Math.pow(2.0,
                ((count < EXP_MAX) ? (double) count : (double) EXP_MAX));
    }

    /**
     * 通信処理を行う。
     * @return 通信が正常終了したら true。エラーが発生したら false。
     */
    private boolean doCommunicate() {
        Log.i("sample", "doCommunicate()!!!");
        return true;
    }
}
```

図 10 通信スレッドクラス

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

```
/**
 * ユーティリティクラス。
 */
public class Utils {

    /**
     * ネットワークが接続状態かどうか確認する。
     * @param context
     * @return 接続状態なら true、それ以外なら false。
     */
    public static boolean isConnected(Context context) {
        ConnectivityManager cm
            = (ConnectivityManager) context
                .getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo info = cm.getActiveNetworkInfo();
        if (info == null || !info.isConnected()) {
            return false;
        }
        return true;
    }

    /**
     * 現在充電中かどうか確認する。
     * @param context
     * @return 充電中なら true、それ以外なら false。
     */
    public static boolean isCharging(Context context) {
        Intent intent = context.registerReceiver(null,
                            new IntentFilter(Intent.ACTION_BATTERY_CHANGED));

        // バッテリ状態を取得する。
        int status = intent.getIntExtra(BatteryManager.EXTRA_STATUS, 0);
        // 充電中なら true を返す。
        if (status == BatteryManager.BATTERY_STATUS_CHARGING) {
            return true;
        }
        // それ以外なら false を返す。
        return false;
    }



    /**
     * GZIP 圧縮を実行するメソッド
     * @param is
     * @param os
     * @return 圧縮成功なら true、失敗なら false
     */
    public static boolean compressToGZip(InputStream is, OutputStream os) {
        // GZIP 出力ストリーム
        GZIPOutputStream gos = null;

        try {
            gos = new GZIPOutputStream(os);
        } catch (IOException e) {
            // GZIPOutputStream の初期化失敗時の処理
            return false;
        }

        // 出力バッファ
        byte[] buf = new byte[1024];

        int len = 0;
        // GZIP ストリームへ書き出す
        try {
            while ((len = is.read(buf)) != -1) {
                gos.write(buf, 0, len);
```

```
            }
        } catch (IOException e) {
            // GZIP ストリームへの書き出し失敗時の処理
            return false;
        } finally {
            // ストリームを close する
            try {
                is.close();
                gos.close();
            } catch (IOException e) {
                // stream の close 失敗時の処理
                return false;
            }
        }

        // GZIP 圧縮に成功したので true を返す
        return true;
    }

    /**
     * ホームアプリが表示中であるかどうか判定する。
     * @param context
     * @return ホームアプリ表示中なら true、それ以外なら false
     */
    public static boolean homeAppIsShowing(Context context) {
        Intent intent = new Intent(Intent.ACTION_MAIN);
        intent.addCategory(Intent.CATEGORY_HOME);
        ArrayList<String> homeList = new ArrayList<String>();

        //カテゴリに CATEGORY_HOME が指定されたインストール済みアプリの package 名のリストを作成
        for (ResolveInfo info :
                context.getPackageManager().queryIntentActivities(intent, 0)) {
            homeList.add(info.activityInfo.packageName);
            Log.v("aaa",info.activityInfo.packageName.toString());
        }

        ActivityManager am =
        (ActivityManager) context.getSystemService(Context.ACTIVITY_SERVICE);
        //現在表示中のアプリのパッケージ名を取得し、homeList に合致するものがあればホームアプリ
表示中として true を返す
        for (RunningTaskInfo t : am.getRunningTasks(1)) {
            if (t != null && t.numRunning > 0) {
                ComponentName cn = t.baseActivity;
                if (cn == null)
                    continue;
                else if (homeList.contains(cn.getPackageName())) //
                    Log.v("bbb",cn.getPackageName());
                    return true;
            }
        }
        return false;
    }

}
```

図 11 ユーティリティクラス

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

```
/**
 * HTTP GET 時に GZIP 要求を行うクラス
 *
 */
public class AsyncHttpGet2 extends AsyncTask<URL, Void, String> {

    //gzip を要求するときにヘッダに指定する Key と Value
    private static final String HEADER_KEY = "Accept-Encoding";
    private static final String HEADER_VALUE = "gzip,deflate";

    // コールバック Interface を定義
    public interface HttpGetCallback {
        void postExecute(String result);
    }

    private HttpGetCallback callback = null;

    public AsyncHttpGet2(HttpGetCallback _callback) {
        this.callback = _callback;
    }

    String result = null;

    @Override
    protected String doInBackground(URL... url) {
        BufferedReader br = null;
        try {
            HttpClient httpClient = new DefaultHttpClient();
            HttpGet httpRequest = new HttpGet(url[0].toString());

            // ヘッダに GZIP 圧縮要求を指定
            httpRequest.addHeader(HEADER_KEY, HEADER_VALUE);

            HttpResponse response = httpClient.execute(httpRequest);

            /** レスポンスが GZIP 圧縮されている場合は GZIPInputStream で受け取る*/
            if (isGzipEnabled(response)) {
                br = new BufferedReader(
                   new InputStreamReader(
                       new GZIPInputStream(response.getEntity().getContent())));

            } else {
                br = new BufferedReader(
                        new InputStreamReader(
                                response.getEntity().getContent()));
            }
            StringBuffer bf = new StringBuffer("");
            String line = "";
            String NL = System.getProperty("line.separator");

            while ((line = br.readLine()) != null) {
                bf.append(line + NL);
            }

            result = bf.toString();

        } catch (ClientProtocolException e) {
            // http 通信のプロトコルエラー時の処理
        } catch (IOException e) {
            // http 通信のコネクション切断時の処理
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    // バッファの close 時の例外処理
```

```
                }
            }
        }
        return result;
    }

    /**
     * コールバックで呼び元に結果を返す
     */
    @Override
    protected void onPostExecute(String result)
    {
        super.onPostExecute(result);
        callback.postExecute(result);
    }

    /**
     * サーバからのレスポンスが GZIP 圧縮されているかどうかを判定するメソッド
     *
     * @param response
     * @return GZIP 圧縮されていれば true、されていなければ false
     */
    private Boolean isGzipEnabled(HttpResponse response) {
        Header header = response.getEntity().getContentEncoding();
        if (header == null) {
            return false;
        }
        String value = header.getValue();
        return (!TextUtils.isEmpty(value) && value.contains("gzip"));
    }

}
```

```
/**
 * 【使い方】
 * インターフェース HttpGetCallback を implements し、
 * コールバックメソッド postExecute()で HTTP のレスポンスを受け取ります。
 * ここではサンプルとして、TextView にレスポンスを解析したものをそのまま表示しています。
 */
public class DefaultHttpClientActivity extends Activity
          implements HttpGetCallback {
    TextView tv = null;
    URL url = null;
    AsyncHttpGet mAsyncHttpGet = null;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_default_http_client);
        tv = (TextView) this.findViewById(R.id.textview);
        mAsyncHttpGet = new AsyncHttpGet(this);

        url = null;
        try {
            url = new URL("http://www.example.com/");
        } catch (MalformedURLException e) {
            // 不正な URL であるときの処理
        }

        mAsyncHttpGet.execute(url);
    }

    @Override
    public void postExecute(String result) {
        tv.setText(result);
    }
```

Copyright © 2012-2014 NTT DOCOMO, INC. All Rights Reserved.

図 12 HTTP 通信で GZIP 要求を行うクラスと使用例