「マルチメディア，分散，協調とモバイル
(DICOMO2013) シンポジウム」 平成25年7月

3G トラヒック削減とプッシュ通知を共存させるアプリ単位の通信制御手法

高木 雅[1] 川原 圭博[1,2] 浅見 徹[1]

概要：
本研究では，スマートフォンにインストールされている様々なアプリに対して，アプリ単位で通信制御を行うことで，スマートフォンの 3G トラヒックと消費電力を効果的に削減する手法を提案する．スマートフォンのトラヒックには，バックグラウンド待機中のアプリが行う通信など，直接的にはユーザに必要とされていない通信が含まれている．このような通信を，ユーザに不自由を感じさせることなく削減するためには，アプリ単位での通信制御を行うことが望ましい．しかしながら，Android SDK にはアプリ単位での通信制御を行う API が存在しない．そこで我々は，Android が Linux カーネルを採用していることに着目し，iptables コマンドを用いて通信制御を行うことを提案する．また，制御対象とするアプリは，ユーザの手を煩わせることなく自動的に選択されることが望ましいが，トラヒック量や通信頻度だけを頼りにユーザへの影響が少ないアプリだけを抽出することは容易ではない．そこで我々は，アプリの「比スリープ通信率」という指標を定義し，スマートフォンの画面点灯中／スリープ中の通信量の比率を基準として，制御対象の自動選択を行う手法を提案する．また，本研究では，SMS（Short Message Service）や GCM（Google Cloud Messaging）によるプッシュ通知を重要視し，プッシュ通知を妨げることのないよう制御機構の実装を行った．

An App-by-App Communication Control to Balance 3G Traffic Reduction with Push Notification

Takagi Masaru[1] Kawahara Yoshihiro[1,2] Asami Tohru[1]

1. はじめに

1.1 背景

ここ数年，スマートフォンの普及が急速に進行したことに伴い，3G ネットワークへの負荷が指数関数的に増大している [2]．スマートフォンのユーザは，フィーチャーフォンのユーザと比較して，平均で 10 倍以上のデータトラヒックを発生させ，また，常時のインターネット接続を前提とするスマートフォンは，フィーチャーフォンより多くの制御信号を発する．携帯電話事業者による 3G ネットワークの設備増強を上回るペースで，3G トラヒックが急増した結果として，都心部では著しい通信速度の低下が見られ，サービスエリア圏内であっても通話や通信ができない輻輳状態が発生するようになった．また．キャリアメールのシステムなどを含めた全国規模の通信障害も頻繁に発生するようになった．2012 年には，制御信号の量がコアネットワークの処理能力を上回ったことによる通信障害が，国内最大手の通信事業者である NTT docomo のネットワークで発生するに至っている [3]．ひとたび通信障害が発生すると，数百万人単位で影響を及ぼす場合もあるため，早急な対策が求められている．

このような事態に至った背景として，3G 通信機器の動作を定めた 3GPP の Fast Dormancy ポリシーの設計思想と，スマートフォン向け OS の通信周りの設計思想とが，大きく食い違っていることが挙げられる．Fast Dormancy ポリシーでは，端末と基地局の間で通信が完了すると，数

[1] 東京大学大学院 情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
[2] School of Architecture + Planning, Massachusetts Institute of Technology

秒から数十秒という短い時間でコネクションを切断し，無線アクセス部分のチャネルを解放することになっている．これにより，チャネルが長時間専有されることを防ぎ，また，端末側では 3G インタフェースを短時間でスリープ状態へ遷移させることで，端末のバッテリ浪費を防ぐ仕組みになっている．Fast Dormancy ポリシーは，フィーチャーフォンのトラヒックパターンを想定しており，端末はインターネットに常時接続しないこと，端末が行う通信は短いこと，通信の間隔は長いことが暗黙の前提となっている．

一方，Android や iOS などのスマートフォン向け OS は，インターネットへの常時接続を前提として設計されており，端末上のアプリがバックグラウンド通信を行うことを原則として認めている．それゆえ，ユーザが操作中のアプリに限らず，端末にインストールされている様々なアプリからトラヒックが発生し，ユーザが端末を操作していない時でもトラヒックが発生する可能性がある．また，端末上のアプリは各々がバラバラのタイミングで通信を行うため，その都度 3G コネクションの確立と切断が行われることとなり，制御信号と消費電力の増加に繋がる．2011 年 10 月にリリースされた Android 4.0 では，Fast Dormancy ポリシー部分が改善され，通信完了後にコネクションを保持する時間が以前より長く設定されたため，制御信号の数は減少しているが，Android 2.3 以前のスマートフォンと Android 3.2 以前のタブレットを合わせたシェアは，2013 年 4 月現在でも 45.7%と高い水準を維持しており，これらの端末への対策が必要である．

ところで，スマートフォンのトラヒックには，バックグラウンド待機中のアプリが行う通信など，直接的にはユーザに必要とされていない通信が含まれている．このような通信の例として，インスタントメッセンジャアプリがログイン状態を維持するための Keep Alive 信号や，メールアプリが新着メールを確認するポーリング信号などが挙げられる [4]．このような通信を選択的に制御することで，ユーザが不自由を感じることなく通信機能を利用できるようにしつつ，3G トラヒックと端末の消費電力を同時に削減すること可能である．

1.2 目的

本研究では，Android スマートフォンにインストールされている様々なアプリに対して，各アプリのトラヒック量に基いて個別に通信制御を行うことで，ユーザに不便を感じさせることなく，3G トラヒックと 3G 制御信号，端末の消費電力を同時に削減することを目標としている．また，通信の発生をリアルタイムでユーザに通知することにより，従来はユーザが気づかなかったトラヒックを可視化し，通信制御の対象に加えることができるようにする．さらに，SMS （Short Message Service）や GCM（Google Cloud Messaging）によるプッシュ通知が，ユーザビリティの維持に必要不可欠であることから，プッシュ通知を妨げることのないよう，制御機構の設計を行った．

本稿の構成は以下のようになっている．まず第 2 節で実際の 3G トラヒックと消費電力について述べる．第 3 節で本研究の提案手法であるアプリ単位の通信制御について詳しく説明し，第 4 節で提案手法の評価とユーザへの影響について触れる．第 5 節では，関連研究について紹介する．最後に，第 6 節で本稿の内容をまとめる．

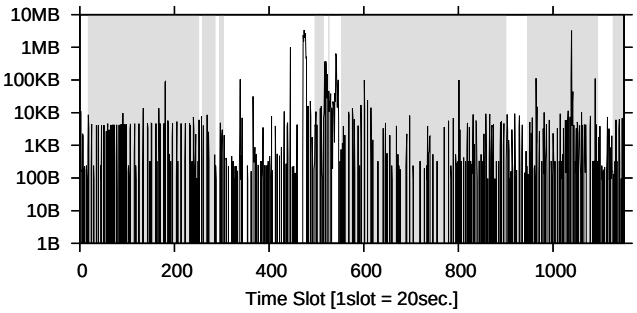

図 1 Android 端末での実際のトラヒック

表 1 トラヒック計測の統計情報

タイムスロット数	1150	
（通信あり）	543	(47.2%)
（通信なし）	507	(52.8%)
計測時間	6 時間 23 分	
（画面点灯中）	1 時間 56 分	(30.3%)
（スリープ中）	4 時間 27 分	(69.7%)
通信量	56,197,702Byte	
（受信）	37,793,241Byte	(67.3%)
（送信）	18,404,461Byte	(32.7%)
（画面点灯中）	36,251,893Byte	(64.5%)
（スリープ中）	19,945,809Byte	(35.5%)

2. 実際の 3G トラヒックと消費電力

2.1 全体の 3G トラヒック

まず，スマートフォンにおける通信の実態を明らかにするため，Android 端末を用いて計測した実際のトラヒックを図 1 に示す（網掛け部分はスリープ時）．ここでは，1 タイムスロットを 20 秒として，各タイムスロットで発生した端末全体のトラヒック量と，各アプリ毎のトラヒック量を，1150 タイムスロット分（6.5 時間）に渡って記録した．Android 端末は GALAXY Nexus SC-04D（Android 4.1.1）を使用し，3G 回線は docomo の FOMA 回線を利用した．なお，都心部での混雑の影響を避けるため，インターネットサービスプロバイダ（ISP）は，sp モードではなく mopera を採用した．

図 1 からは，1〜2 分に 1 回という高い頻度で，3G 通信が行われていることが分かる．6.5 時間の計測において，合計 56.2MB のトラヒックが発生し，コネクション確立と切断がそれぞれ 235 回行われたと推定される．また，合

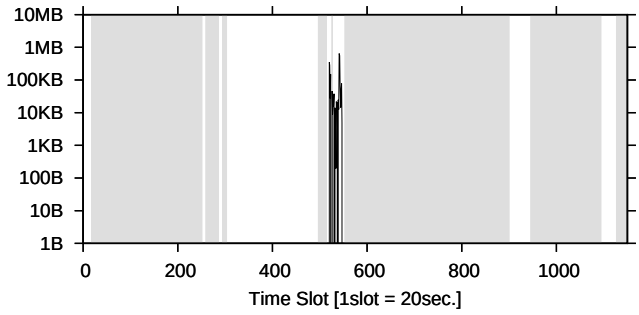

図 2 標準ブラウザのトラヒック

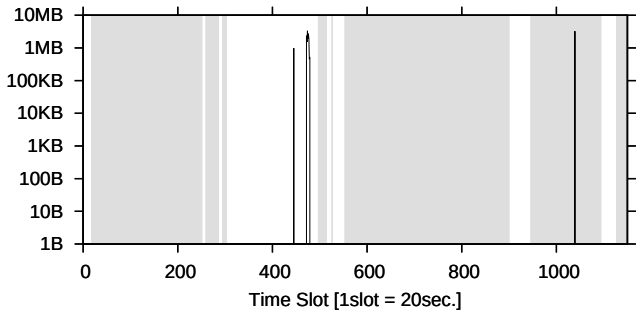

図 3 ダウンロードマネージャのトラヒック

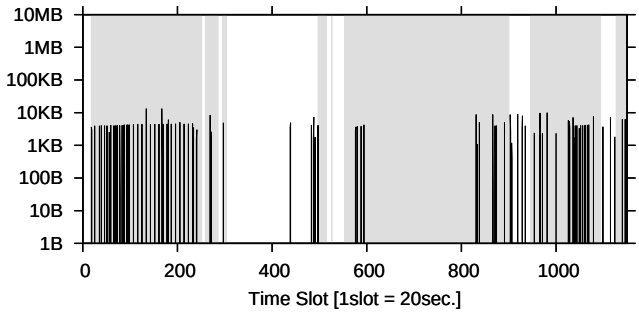

図 4 Gmail のトラヒック

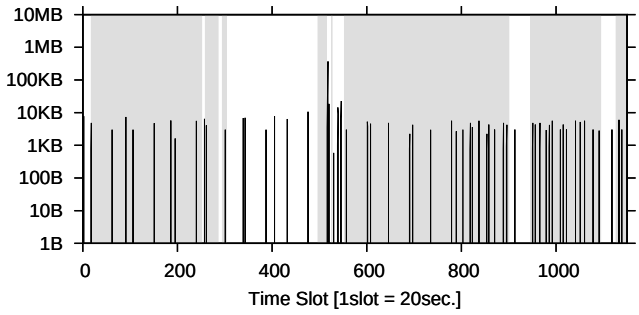

図 5 Google 検索のトラヒック

計 4.5 時間は端末の画面が消灯したスリープ状態であったが，GALAXY Nexus のコネクション持続時間が 5 秒程度であること（後述）を踏まえると，3G インタフェースは合計 2.5 時間しかスリープしていなかったことになる．このように，スマートフォンはユーザの操作中に限らず，トラヒックを生み出すことが分かる．

2.2 アプリ別の 3G トラヒック

今回，トラヒック計測を行った GALAXY Nexus には，システムアプリを含めて約 140 個のアプリがインストールされていたが，6.5 時間の計測中に通信を行なっていたのは，18 個のアプリのみであった．中でも通信量が多かったアプリとして，「標準ブラウザ」「ダウンロードマネージャ」「Gmail」「Google 検索」が挙げられる．これら 4 アプリのトラヒックを図 2 〜図 5 に示す（網掛け部分はスリープ時）．

「標準ブラウザ」や「ダウンロードマネージャ」のトラヒックは，通信頻度が低く，1 回あたりの通信量が数百 KB〜数 MB と大きいのに対し，「Gmail」や「Google 検索」のトラヒックは，通信頻度が高く，1 回あたりの通信量が数 KB 〜数十 KB と小さい．また，前者では通信が画面点灯中に集中しているのに対して，後者では端末の画面点灯中，スリープ中に関わらず，通信が発生している．本研究では，ユーザが直接操作するタイプのアプリで，前者のような特徴を示すものを「ユーザ操作型アプリ」と呼ぶ．また，主にバックグラウンドで動作するタイプのアプリで，後者のような特徴を示すものを「バックグラウンド（BG）通信型アプリ」と呼ぶ．

これら 2 種類のうち，BG 通信型アプリは通信頻度が高く，3G 制御信号数や端末の消費電力に大きく影響を及ぼすと考えられる．BG 通信型アプリには，メーラやインスタントメッセンジャ，天気予報ウィジェットなどが含まれる．これらのアプリは，サーバから最新情報を取得したり，データを同期したりするため，定期的にサーバにポーリングを掛けており，スリープ中でも通信頻度が高い状態が続く．しかしながら，スリープ中の端末はポケットやカバンの中にあることが多く，結局ユーザには新着情報が伝わらないことが多いため，このような通信は必ずしも必要性が高くない．そこで，本提案手法では，BG 通信型アプリのスリープ中の通信のみを制御することで，3G トラヒックの削減と端末の消費電力の削減を実現する．なお，BG 通信型アプリの判定基準については 3 節で詳しく述べる．

2.3 消費電力

図 6 は GALAXY Nexus に約 30 秒周期で数 KB 程度の通信を行わせた場合の消費電力の推移である．ここでは，1 秒ごとの平均の消費電力を，マルチメータと 0.1 Ω抵抗を用いて測定している．通信開始直後（11 秒，43 秒，75 秒）に消費電力が 0.8W 前後まで急上昇し，通信終了後の数秒間（19 〜24 秒，51〜56 秒，80〜86 秒）は 0.5W 前後で安定した後，アイドル状態の 0.05W 前後まで急落している．このことから，GALAXY Nexus における，Fast Dormancy ポリシーのコネクション持続時間は 5 秒程度であり，少量の通信であれば，全体が 20 秒以内に収まることが分かる．これを踏まえ，前述のトラヒックの計測では，1 タイムスロットを 20 秒と設定した．

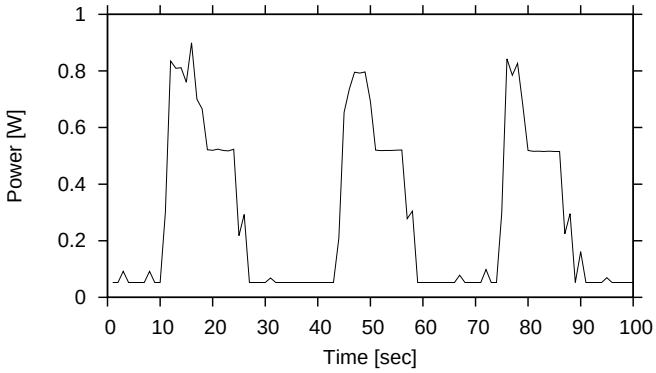

図 6 通信時の GALAXY Nexus の消費電力

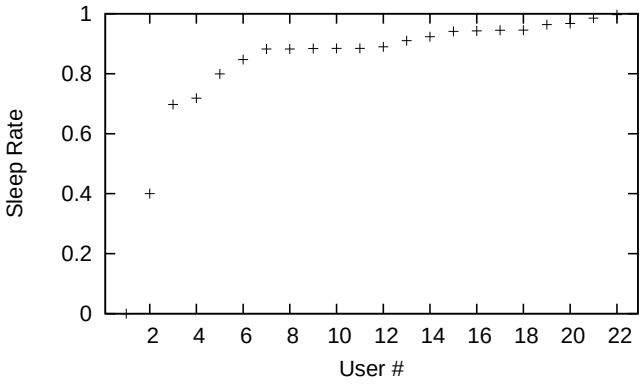

図 7 実際の端末のスリープ率の分布

3. アプリ単位の通信制御

3.1 概要

アプリ単位の通信制御を実現する方法としては，まず Android SDK の隠し API が考えられる．この隠し API は，Android 4.0 から追加されたもので，アプリ毎にバックグラウンド通信の可否を設定するものである．しかしながら，この隠し API はパーミッションの関係でシステムアプリからしか実行できず，ユーザアプリからは実行できない．そこで我々は，Android が Linux カーネルを採用していることに着目し，iptables コマンドの uid-owner オプションを用いてアプリ単位の通信制御を実現した．

3.2 iptables と UID

iptables は，パケット処理を司るフレームワークである netfilter を制御することで，通信制御の設定を行う Linux カーネルのコマンドである．iptables の filter テーブルには INPUT ／ FORWARD ／ OUTPUT の 3 つのルールチェインがあり，それぞれ，その端末に到着／通過／送出されるパケットに対する処理ルールを管理している。各ルールは，送信元 IP アドレスや宛先のポート番号などの条件によって対象とするパケットを規定し，通過／拒否／破棄などの処理を指定できる．本提案手法では，このうち OUTPUT チェインのみを使用する．これは，INPUT チェインで受信パケットをブロックしたとしても，3G ネットワーク側からの受信データによって，3G トラヒックが発生し，3G インタフェースのスリープ状態が解除されることを阻止できないためである．なお，OUTPUT チェインでは，uid-owner オプションを用いて，送信元プロセスの UID を条件として対象とするパケットを指定できる．

ところで，Android OS は，複数の Dalvik 仮想マシンを同時に実行できるように設計されており，アプリは各々に割り当てられた Dalvik 仮想マシン上で実行される．それぞれの Dalvik 仮想マシンは，アプリに割り当てられた固有の UID で実行されており，ファイル IO やネットワーク通信などのカーネルレベルの命令を実行する場合は，その UID と Linux カーネルのユーザ管理機能を用いて，アプリ間の独立性を確保している．それゆえ，どのアプリが特定のファイルを読み書きしたか，あるいは，どのアプリがどれだけネットワーク通信を行ったかを，Android OS では簡単に調べることができる．

そこで，本提案手法では，UID を元に各アプリの通信量を調査し，3.3 節で述べる基準に従って BG 通信型アプリのみを抽出した後，それらの UID を iptables の uid-owner オプションで指定して，BG 通信型アプリのスリープ中の通信のみを遮断する．ここで，システムアプリに割り当てられた UID は制御対象から除外し，OS アップデートや GCM によるプッシュ通知など，Android OS のシステム機能が阻害されることがないようにする．GCM のサポートについては，3.5 節で詳しく述べる．

ちなみに，Android 4.0 の新機能である「バックグラウンドデータの制限」を設定アプリで有効にした場合，iptables に reject ルールが出現することから，この機能は iptables を用いて実現されていると推測される．同様に，「モバイルデータの (使用量) 制限」も，iptables の quota オプションを用いて実現されていると考えられる．

3.3 対象アプリの自動判定

本提案手法において，ユーザの手を煩わせることなく制御対象とすべきアプリを自動判定するため，我々は端末の「スリープ率」，アプリの「スリープ通信率」，アプリの「比スリープ通信率」という 3 つの数値を定義した．

まず，端末の「スリープ率」を「端末がスリープ状態であった時間の割合」と定義する．ここで，端末のスリープ状態とは，Android 端末の画面が完全に消灯し，タッチパネル操作に反応しない状態を指す．例えば，1 日（24 時間）のうち 18 時間だけ端末がスリープ状態になっていたとすると，スリープ率は，18/24=0.75 となる．

後述する「通信量お知らせアプリ」を用いて調査した実際のスリープ率の分布を図 7 に示す．22 ユーザのスリープ率の平均は 0.83 であり，約半数のユーザで 0.9 を上回って

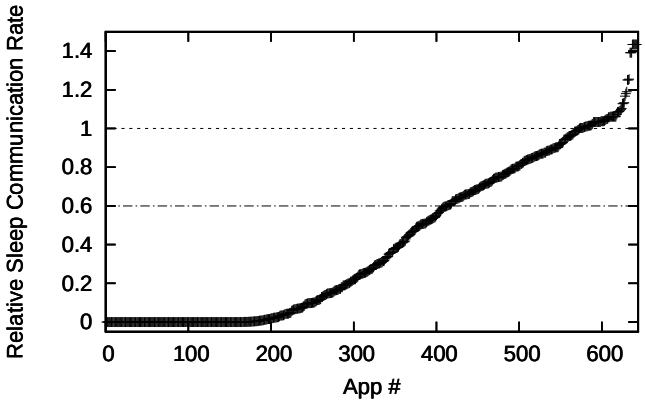

図 8 実際の比スリープ通信率の分布

表 2 実際の比スリープ通信率

アプリ名	実際の値	アプリ名	実際の値
Chrome	0.000	Google 検索	1.008
ブラウザ	0.060	Gmail	1.190
YouTube	0.359	Google+	1.255
Google Play Music	0.717	カレンダー	1.434

いた．つまり，平均的なユーザの場合，全体の 9 割の時間は端末がスリープ状態になっている．

次に，アプリの「スリープ通信率」を「当該アプリが端末のスリープ中に行った通信が，そのアプリが行った全通信に占める割合」と定義する．例えば，あるアプリが端末のスリープ中に 30MB，画面点灯中に 70MB 通信したとすると，スリープ通信率は 30/(70+30)=0.3 となる．

さらに，アプリの「比スリープ通信率」を「スリープ通信率を，端末のスリープ率で割ったもの」と定義する．前述の例の場合，比スリープ通信率は 0.3/0.75=0.4 となる．

本研究では，比スリープ通信率を用いてアプリの自動判定を行う．理想的なユーザ操作型アプリでは，スリープ中には通信が発生しないため，スリープ通信率が 0 となり，比スリープ通信率も 0 となる．一方，理想的な BG 通信型アプリでは，スリープ中であっても画面点灯中であっても関係なく均等に通信が発生するため，スリープ通信率は端末のスリープ率に収束し，比スリープ通信率は 1 となる．

ここで，ユーザ操作型アプリと BG 通信型アプリを区別する比スリープ通信率の適切な閾値を設定するため，実際の比スリープ通信率の分布を調査した．我々は，端末上にインストールされている全アプリの比スリープ通信率を調査するアプリ「通信量お知らせアプリ」を製作し，Google PLAY で公開してテスタを募った [5]．図 8 は，延べ 643 アプリについて実際の比スリープ通信率を左から昇順にプロットしたものである．

図 8 を見ると，完全なユーザ操作型の性質を示し，比スリープ通信率 0 のアプリが，全体の 1/3 を占めている．一方，BG 通信型の性質を示すアプリは，比スリープ通信率 1 に集中することなく，広範囲に渡って分布している．また，端末のスリープ中にばかり通信する性質を持ち，比スリープ通信率 1 以上のアプリが全体の 10%ほどを占めている．ここでは，比スリープ通信率 0.6 付近で分布の傾きが緩やかになることから，0.6 を閾値として採用する．比スリープ通信率が 0.6 未満のアプリをユーザ操作型アプリ，0.6 以上のアプリを BG 通信型アプリと判定し，本提案手法では後者のみを制御対象とする．実際の比スリープ通信率の例を表 2 に示す．

3.4 通信量の計測

本提案手法では，比スリープ通信率を算出するため，端末上の各アプリの通信量を定期的に監視している．アプリ毎の累計データ送受信量は，Linux カーネル内で管理されており，/proc/uid_stats/[uid]/ディレクトリ内のファイルに記録されている．この値は，Android SDK の TrafficStats API からも取得可能であるが，通信しないアプリも含めて全てのアプリを調べる実装となってしまい，負荷が大きくなりすぎるため，ここでは直接ファイルにアクセスして数値を取得している．本研究では，スリープ中の通信と画面点灯中の通信とを区別できれば十分であるため，Service を用いた常駐監視は行わず，画面が点灯／消灯した瞬間にのみ通信量を取得する．ところで，Android OS には，画面の点灯や消灯，画面ロック解除といった端末全体に影響するイベントの発生を，端末上の全アプリに通知する Broadcast Intent という仕組みがある．そこで，本研究では，画面の消灯と画面ロック解除の Broadcast Intent をトリガーとして通信量を取得することで，端末への負荷を低減し消費電力が増加しないようにしている．

3.2 節で述べた通り，本提案手法ではスリープ中にのみ通信制御を行うため，通信制御中アプリについては，比スリープ通信率を正しく計測することができない．というのも，通信制御中のアプリは，画面点灯中にしか通信できない状態となるため，画面点灯中の累計通信量ばかりが増加し，実際よりも比スリープ通信率が小さくなってしまうためである．本提案手法では，これを逆手に取り，あえて通信制御中のアプリについても普段通りに通信量の計測を行うことで，一度，通信制御の対象となったアプリでも，スリープ中に通信できる機会を定期的に得られる仕組みを確立した．この仕組みは，実装がシンプルで，端末への負荷が低いのが特徴である．

3.5 プッシュ通知（GCM）のサポート

本研究では，プッシュ通知を重要視し，通信制御中であってもプッシュ通知が利用可能となるよう制御機能の実装を行った．ここでは，通信制御中でも GCM によるプッシュ通知を利用可能とするために必要な要件について論じる．

まず，GCM の仕組みについて簡単に説明する．

端末側では，GCM のプッシュ通知を利用するアプリが

起動されると，OS は GCM サーバにアクセスしてユーザトークンを登録し，GCM サーバとの間に TCP コネクションを確立する．その後は，定期的にパケットをやり取りすることでコネクションを維持し，何らかの理由でコネクションが切断された場合にはコネクションを張り直す．

一方，GCM サーバにメッセージの送信依頼が届くと，宛先のユーザトークンと登録 DB から端末を特定し，その端末との間のコネクションを調べる．もし，コネクションが維持されていれば，そのコネクションを通じて端末にメッセージを送信する．受信した端末では，Android OS が GCM サーバに ACK を返すとともに，Broadcast Intent で当該アプリにメッセージを伝える．もし，コネクションが切断されていた場合には，端末がオフライン状態にあると判断し，次にコネクションが確立されるまで待つ．

端末側では，GCM の通信部分は全て Android OS によって管理されており，アプリはユーザトークン登録の API を 1 度コールすれば，後は Broadcast Intent を待つだけで良い仕組みになっている．

それゆえ，本提案手法のように，UID を指定して特定のアプリの通信を遮断した場合でも，当該アプリは正常にプッシュ通知を受信することができる．ただし，プッシュ通知をトリガーとして，当該アプリがメールの取得やデータの同期などの通信を行う場合には，通信制御の対象となる．

一方，root，system，shell など，Android OS のシステムを司る UID を指定して通信を遮断した場合や，UID を指定せずに全通信を遮断した場合には，プッシュ通知の受信ができなくなる．その場合，GCM サーバとの間のコネクションを維持できなくなるため，サーバは端末がオフライン状態になったと判断し，以降のメッセージを送信しなくなる．

ここで，root 権限を取得した GALAXY Nexus を用意し，GCM のサンプルアプリをインストールして実際の挙動を確認した．ここでは，端末を USB ケーブルで PC に接続して，Android Debug Brigde（ADB）越しに端末にリモートログインし，直接 iptables コマンドを操作した時の挙動を調べた．なお，実験中は 3G ネットワークに端末を接続していた．

想定と異なる挙動を示したのは，OUTPUT チェインで UID を指定せずに全ての送信パケットを遮断した場合で，遮断後 1 回目のプッシュ通知だけを受信でき，それ以降のプッシュ通知は受信できなかった．これは，INPUT チェインを操作しなかったため，1 回目のプッシュ通知の受信はできたものの，サーバに ACK を返すことができず，GCM サーバは端末がオフラインになったと判断して，以後のプッシュ通知の配信を停止したためと考えられる．

図 9 「通信量お知らせアプリ」の通信量一覧画面

3.6 トラヒックの可視化

ユーザに気づかれずに通信しているアプリを発見し，制御対象に加えることを補助するため，3.4 節の通信量の計測機構を活用してトラヒックの可視化を行った．ここでは，端末のスリープ中に行われる通信と，ユーザが端末を操作している間に行われる通信の両方を可視化する．この機能も，前述の「通信量お知らせアプリ」に実装し，公開している．

まず，スリープ中の通信量については，前節の機構をそのまま流用し，端末の画面ロック解除時にユーザに通知を行う．ここでは，Android の Notification API を利用し，通信量の上位 3 アプリのアプリ名，アイコン，送信量，受信量と，端末のスリープ時間を通知バーに表示する．

次に，ユーザが端末を操作している間の通信量については，通信量の取得機構を 1 秒間隔で実行し，通信があった場合には通知バーに表示する．これにより，ユーザは通信の発生をリアルタイムで確認できるようになり，どのアプリがどのようなタイミングで通信しているのか，を把握できるようになる．もし，ユーザがその通信を不要だと感じた場合には，通知バーをタップして後述するアプリ一覧画面を開き，そのアプリを制御対象に追加することができる．

さらに，各アプリの累計のデータ送受信量を，上位から順に一覧表示する機能も用意した．図 9 のように，各アプリの通信量をスリープ中／画面点灯中に分けて 1 本の棒グラフで表示することで，スリープ通信率を直感的に確認できるようにした．同様に，端末のスリープ時間と画面点灯時間も棒グラフで表示し，端末のスリープ率と比較しやすいようにした．また，一覧からアプリを起動，情報を表示，アンインストール，制御対象に追加，制御対象から削除できる機能も搭載した．なお，通信制御には root 権限が必要

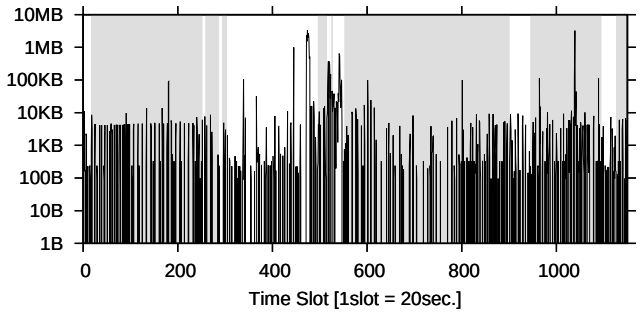

図 10 本提案手法を適用しない場合のトラヒック

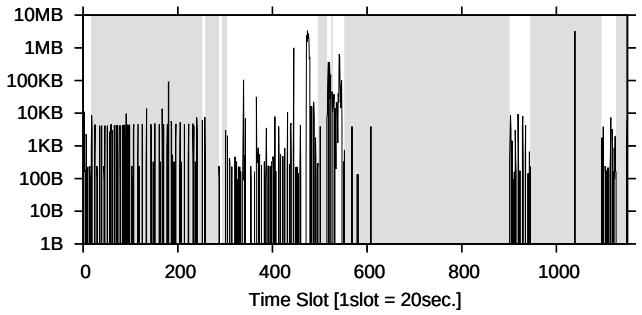

図 11 本提案手法を適用した場合のトラヒック

となるため，公開中の「通信量お知らせアプリ」では機能を削除している．

4. 通信制御の評価と影響

4.1 3G トラヒックと制御信号の削減

我々は，2 節のトラヒック記録に対して本提案手法を適用し，比スリープ通信率 0.6 以上のアプリのスリープ中の通信を遮断した場合のトラヒックをシミュレーションした．ここでは，3G コネクション確立時と切断時の制御信号について，以下の 2 つの仮定を置いて制御信号の最低量を見積もった．

- 連続したタイムスロットの通信では，3G コネクションが維持される
- 通信のないタイムスロットがあれば，3G コネクションは必ず切断される
- 1 回の 3G コネクション確立と切断で，合計 30 個の制御信号が発生する

本提案手法を適用しない場合のトラヒックを図 10 に，本提案手法を適用した場合のトラヒックを図 11 示す（網掛け部分はスリープ時）．両者を比較すると，スリープ中のトラヒックが大きく削減されており，スリープ中の通信総量は 23.7%削減されている．スリープ中の総通信時間は 97 分から 27 分へ減少し，スリープ中の 3G コネクションの確立回数も 168 回から 52 回へ減少している．これに伴い，3G 制御信号の発生数も 5040 個から 1560 個へ 69.0%削減されている．言い換えると，スリープ中の制御信号数はもともとの 1/3 以下まで大幅に減少している．

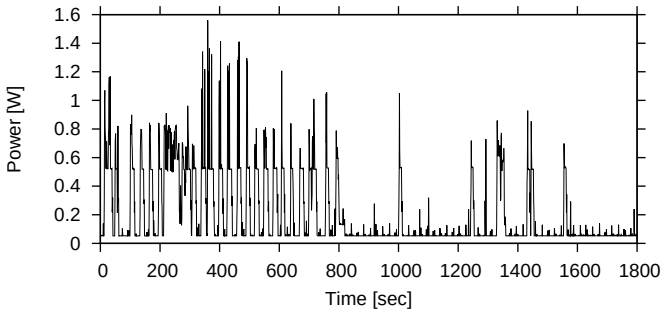

図 12 本提案手法を適用しない場合の消費電力

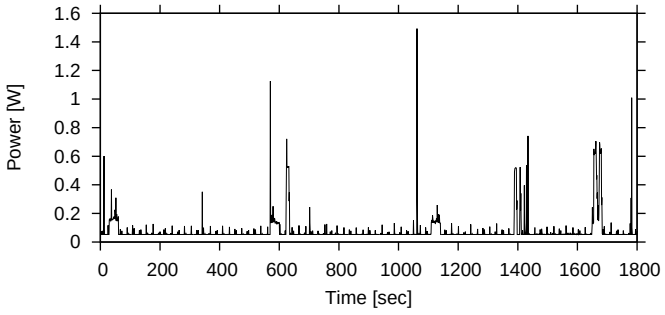

図 13 本提案手法を適用した場合の消費電力

4.2 消費電力の削減

我々は，GALAXY Nexus（Android 4.0.4）を用いて，本提案手法を適用した場合，適用しない場合のそれぞれについて，端末の消費電力を 30 分ずつ計測した．この計測では，端末のバッテリスロットの+端子とバッテリの+端子の間に 0.1 Ωの抵抗を挟み，Agilent 社の 34410A デジタル・マルチメータを用いて，抵抗での電圧降下を計測した．マルチメータでは 1 秒ごとの平均電圧を計測し，バッテリの出力電圧が 3.8V で一定であると仮定して，端末の消費電力を算出した．なお，端末上のアプリの構成は 2 節のトラヒック計測と同様で，計測中は端末をスリープ状態とした．また，スリープ状態にした直後は消費電力の変動が大きかったため，スリープ状態にして 10 秒程度の時間をおいて計測を開始した．2 回の計測は同じ端末を用いて，立て続けに行った．

図 12 と図 13 はそれぞれ，本提案手法を適用しない場合，適用した場合の 30 分間の消費電力の推移である．図 12 では，0 秒〜800 秒の時間帯に，図 6 と同様の通信による消費電力の波形が見られるが，図 13 では，通信による波形が大幅に間引かれている．30 分間の平均消費電力はそれぞれ，0.212W と 0.077W であり，スリープ時の消費電力を 63.5%削減することに成功した．つまり，スリープ状態でのバッテリ持続時間は 2.74 倍に伸びることになる．

5. 関連研究

スマートフォンの 3G トラヒック削減については，様々な研究が盛んに行われてきた．しかし，多くの研究では，Wi-Fi など他の通信手段へのオフロードを目標としてお

り，トラヒックそれ自体を削減しようと試みる研究は少ない．Keller らは，スマートフォンでの動画配信において，動画コンテンツを複数台のスマートフォンで協力してダウンロードし，Wi-Fi で共有することで，3G ネットワークを有効活用する手法を提案している [6]．類似の手法として，Vingelmann らは，複数のスマートフォンで協調して 3G 基地局からの信号を受信し，Wi-Fi で情報交換することで，無線アクセス部分でのパケットロスを補完し，基地局からの再送回数を減らす手法を提案している [7]．また，Sicard らは，スマートフォン同士で Wi-Fi アドホックネットワークを構築し，3G トラヒックを固定網にオフロードする手法を提案している [8]．

スマートフォンの消費電力削減についても，盛んに研究が行われてきた．しかし，ほとんどの研究では，スマートフォンに搭載されている Wi-Fi モジュールや GPS モジュールといったデバイスの消費電力に焦点を当てている．Zhang らは，スマートフォンに計測機器を接続することなく，機種ごとに異なる各種のデバイスがどれくらいの電力を消費しているかをモデリングし，ユーザやデベロッパを支援するソフトを作成している [9]．Shye らは，ユーザが実際にスマートフォンを利用している時の各デバイスの消費電力について調査し，画面輝度や CPU の動作周波数を徐々に低下させることで，ユーザの満足度を保ちつつ，消費電力を低減する手法を提案している [10]．一方，ソフトウェア面では，Pathak らが，Android OS においてバッテリドレインの原因となりやすい API を列挙し，Google PLAY で配布されている 86 個のアプリについて，どの API が使用されているか調査を行っている [11], [12]．また，Han らはスマートフォンの Wi-Fi テザリング機能の利用時に，3G 接続が不安定または利用不能な場所において，Wi-Fi インタフェースをスリープ状態とすることで，消費電力を削減する手法を提案している．また，センシングの分野では，山本らが上りトラヒックの発生に合わせてセンシングデータを送信することで，センシング時の消費電力を削減する手法を提案している [14]．

6. 結論

本稿では，スマートフォン上の様々なアプリに対して，個別に通信制御を行うことで，ユーザビリティを維持しつつ，3G トラヒックと消費電力を効果的に削減する手法を提案した．

評価実験では，実際の Android 端末のトラヒック記録に対して，本提案手法を適用した場合の 3G トラヒックのシミュレーションを行い，また，スリープ時における消費電力の削減量を計測した．結果として，スリープ中の 3G トラヒックを 23.7%，スリープ中の制御信号を 69.0%，スリープ中の消費電力を 63.5% 削減できることを示した．

現在，GCM のプッシュ通知を検知して当該アプリの通信を一時的に許可する機構について検討しており，実装方法の調査を進めている．

参考文献

[1] 高木雅, 川原圭博, 浅見徹, “アプリ単位の動的な通信制御を用いた Android 端末の 3G 制御信号と消費電力の削減手法,” 信学総大, No.B-15-8, March 2013.

[2] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2010 - 2015.

[3] 2012 年 1 月 25 日の FOMA 音声・パケット通信サービスがご利用しづらい状況について - NTT docomo http://www.nttdocomo.co.jp/binary/pdf/info/news_release/2012/01/26_00.pdf

[4] Haverinen, H. Siren, J. Eronen, P. Energy Consumption of Always-On Applications in WCDMA Networks. Vehicular Technology Conference, 2007. VTC2007-Spring. IEEE 65th, pp.964-968, April 2007.

[5] 通信量お知らせアプリ - Google PLAY https://play.google.com/store/apps/details?id=akg.traffic.notifier

[6] L. Keller, A. Le, B. Cici, H. Seferoglu, C. Fragouli and A. Markopoulou. MicroCast: Cooperative Video Streaming on Smartphones. MobiSys, 2012.

[7] P. Vingelmann, M. V. Pedersen, F. H. P. Fitzek, and J. Heide. On-the-fly packet error recovery in a cooperative cluster of mobile devices. in Proc. of Globecom, Houston, TX, Dec. 2011.

[8] Leo Sicard, Matyas Markovics and Giannakis Manthios. An Ad-hoc Network of Android Phones Using B.A.T.M.A.N. Project Report SPVC-E2010, IT-University Copenhagen

[9] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones, Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, October 24-29, 2010, Scottsdale, Arizona, USA.

[10] Alex Shye, Benjamin Scholbrock, Gokhan Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures, Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, December 12-16, 2009, New York, New York.

[11] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, Samuel P. Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys), pp.267-280, June 2012.

[12] Abhinav Pathak, Y. Charlie Hu, Ming Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices, Proceedings of the 10th ACM Workshop on Hot Topics in Networks, p.1-6, November 14-15, 2011, Cambridge, Massachusetts.

[13] Hao Han, Yunxin Liu, Guobin Shen, Yongguang Zhang and Qun Li. DozyAP: Power-Efficient Wi-Fi Tethering. MobiSys, 2012

[14] 山本 享弘, 猿渡 俊介, 南 正輝, 森川 博之. Piggyback Transport Protocol : Participatory Sensing における低消費電力なアップロードエンジン情報処理学会論文誌, Vol.53, No.1, pp.274-285, Jan. 2012.