Vol.2013-GN-86 No.20
Vol.2013-CDS-6 No.20
2013/1/17

Android OSの状態変化通知機構における通信集中回避制御手法の検討

川崎 仁嗣[1]　神山 剛[1]　小西 哲平[1]　大久保 信三[1]　稲村 浩[1]

概要：Android OS では，ユーザ操作やタスクの動作により発生した端末状態変化を他のタスクへ通知する仕組みが提供されている．あるタスクに因る端末状態変化に伴って他のタスクが動作するため，タスクの挙動によっては開発者の意図していなかったタイミングで動作してしまうことがある．結果として，短期間に集中した通信を発生させてしまうことがあり輻輳の原因となる．本研究では，端末状態変化の通知タイミングを遅延させることで通信集中を回避する制御手法を提案する．実端末を用いた評価において，意図しないタイミングでの動作が抑制されていることを確認した．

A congestion avoidance method for event delivery mechanism on the Android OS

Satoshi Kawasaki[1]　Takeshi Kamiyama[1]　Teppei Konishi[1]　Shinzo Ohkubo[1]　Hiroshi Inamura[1]

Abstract: An event delivery mechanism, that deliver an event indicating a change in states caused by user operation or other tasks, is provided on the Android OS. Due to invocation of tasks when a task changed a device states, tasks may be invoked at developer's unintended timing. Thus a burst traffic may occur in the short term and be appeared as a network congestion. In this paper, we propose a congestion avoidance method by delaying a timing of event delivery. We confirmed that our method suppresses unintended invocations in our evaluation.

1. はじめに

昨今のモバイルインターネット利用環境は今までのフィーチャーフォンから Android OS[1] や iOS[2]，Windows Phone[3] を搭載したスマートフォンに移行しつつある．既存のフィーチャーフォンが比較的軽量なコンテンツを扱っていたのに対し，スマートフォンは動画や音楽などのデータ量が大きいコンテンツの扱いを得意としており，ネットワーク上で通信されるデータ量が急速に増加している．これに伴い通信事業者は通信設備を増強することで対応してきている．

しかしながら，想定を大幅に上回る速度でスマートフォンの普及が進み，多くの人が集まるような場所ではネットワークの輻輳が大きな問題となっている．輻輳が発生する原因としては，データ量の大きいコンテンツが扱われるようになったことで転送量が増加したことだけでなく，バックグラウンドで行われる通信に伴う通信頻度増加に因る部分も大きい．スマートフォン向けの OS ではユーザーが操作を行っていない画面オフの状態においても，アプリケーション（以降ではアプリと呼ぶ）がデータをサーバーと同期するために定期的な通信が発生する．通信タイミングを OS 側で制御している iOS や Windows Phone と比べると，Android OS は同期通信のタイミングをアプリ開発者が自由に指定できるため，注意深く実装を行わないと通信集中の原因となりかねない．定期的な通信処理を行うアプリが発生させるトラフィックについての分析 [4] も行われている．

さらに，この問題を複雑化している要因として，Android OS では，あるアプリの挙動に影響を受けて他のアプリが動作を開始するような動作（以降ではカスケード起動と呼

[1] NTT ドコモ
NTT DOCOMO, INC.

ぶ）が頻繁に発生しており，アプリの動作タイミングが集中しないよう実装を行っていても，意図していないカスケード起動が発生しうるということが挙げられる．これにより，アプリ開発者が本来意図したタイミング以外でも動作することになり，想定していなかったタイミングで通信を発生させてしまう可能性がある．例えば，目覚ましアプリが AM6:00 に鳴動しディスプレイが画面オン状態へと遷移することで通知が配送され，カスケード起動したアプリの通信により通信集中が発生するような事例が考えられる．

本稿では，端末状態変化の通知によって発生するカスケード起動において通信集中が発生する要因の分析を行い，アプリ開発者が輻輳回避のための処理を行わずとも通信集中が発生しないよう，OS 側で動作タイミングを制御する手法について検討する．

2. 状態変化通知機構

端末状態変化の通知による通信集中の発生過程を理解するにあたって，状態変化通知機構が関わってくるため，本章では動作の概要を述べる．

スマートフォンにおいては，端末の状態などが変化したタイミングで動作するアプリが多い．例えば，電話アプリは音声通話の着信が発生した際に，画面を点灯させて着信中の画面を表示するとともに着信音を再生する．このようなアプリを実装するために，端末状態変化が発生したことをアプリ側に通知する機能が OS 側で提供されていることが多い．この通知機能を実現する仕組みを状態変化通知機構と呼ぶ．

Android OS においては，受けとりたい通知を事前にシステム側へ登録する必要がある．このように，システム側に登録を行ったアプリのみが通知対象アプリとなる．例えば，電話アプリは電話の着信に関する通知は受取るがメールの着信については扱わないため，電話着信の通知のみを受取るよう指定する．

2.1 状態変化の種類

状態変化は，その発生要因に応じて以下の通りに大別できる．

(1) 内部要因状態変化
　バッテリー残量の変化や，他アプリの実行による画面の点灯，および音楽再生の開始などの，端末内での要因により発生する状態変化

(2) 外部要因状態変化
　ユーザー操作による画面の点灯や，ネットワーク側からの電話やメール着信，および電波状態の変化などの，外部からの要因により発生する状態変化

2.2 通知方法

Android OS においては，システム側からの通知方法として以下の 2 種類があり，通知の種類に応じて通知方法が決定される．

(1) 一斉通知
　全ての通知対象アプリに対して同時に通知を行う方法

(2) 逐次通知
　通知対象アプリに対し優先順位の高いものから順に通知を行う方法

3. 同期干渉による大域同期起動問題

複数の端末間でアプリの動作タイミングがほぼ同一になってしまっている状態，つまり大域同期して起動する状態を大域同期起動と呼ぶ．大域同期起動状態となっているアプリが通信を発生する場合，複数の端末から一斉に通信が発生することとなり，ネットワーク側での通信開始のための制御信号が集中し，輻輳の発生要因となる．したがって，大域同期起動してしまうアプリでは通信を伴わない処理を行うか，通信処理のタイミングを端末間で分散するような実装とすべきである．一方で，特に同期した動作を意図していないアプリを開発する際には，動作タイミングが端末間で分散していることから，通信処理のタイミングを分散化する必要性は無いように思える．

しかし，端末上で大域同期起動したアプリが存在する状態で同期干渉が発生することにより，アプリ開発者の意図に反して大域同期したタイミングで動作してしまう場合がある．同期干渉を受ける側のアプリの開発者が本事象を予見することは難しく，OS 側で本事象を回避するような手法が必要である．

3.1 同期干渉

特に同期した動作を意図してはいないアプリであっても，大域同期起動している他のアプリからカスケード起動されることにより，複数の端末間で同期して起動してしまう事象を，我々は同期干渉と呼ぶ．

ある端末内において，特定時刻に動作するアプリ（定時アプリ）と，特定時刻には動作せずに他アプリの動作タイミングに依存して動作するアプリ（連動アプリ）の双方が動作している場合，カスケード起動により定時アプリの動作するタイミングにおいて連動アプリも動作する．定時アプリが大域同期起動しやすい挙動，例えば AM7:00 に動作するよう実装されている場合，連動アプリは本来，他の端末と同期した動作を意図していないにもかかわらず，実際には他の端末と同期したタイミング，この場合は AM7:00 に動作することとなる．

アプリ開発者は定時アプリの実装において通信集中を避けるよう意識するため，特定時刻に起動した際は通信を発生させないような処理とするが，連動アプリはそもそも動作タイミングが端末間で同期してしまうことに気付かない開発者が多く，動作を開始した時点で通信を行う処理とす

ることも多い．結果として，定時アプリが大域同期起動する場合，連動アプリが同一端末上で動作している定時アプリからの同期干渉を受け，連動アプリが大域同期起動状態となるため，輻輳の発生要因となる．

同期干渉の発生要因としては，大きく分けて 2 つ存在する．1 つ目は，アプリをあらかじめ設定した時刻において起動させる定時実行動作によるものであり，Android OS においては Alarm Manager[5] と呼ばれる機構が定時実行動作を実現している．2 つ目は，2 章で述べたような状態変化通知動作によるものであり，Android OS においては端末状態変化を検知すると Broadcast Intent[6] を配送する事で通知を行う仕組みが提供されている．

3.1.1 定時実行動作による大域同期起動

定時実行動作は，アプリが指定した時刻において処理を実行するものであるが，スマートフォン端末では一般的にバッテリー容量が限られいることから，スマートフォン向け OS では電力消費を抑えるために指定した時刻で直ちに処理を実行するのではなく，電力効率の良くなるようなタイミングで処理を行うように実装されている場合が多い．スマートフォン端末では常に CPU などのデバイスが動作しているのではなく，例えば画面オフ状態においては必要がなければ各デバイスをスリープ状態へと遷移させ，消費電力を削減するようになっている．したがって，電力効率を考慮すると，出来るだけスリープ状態を維持し，稼働状態となる頻度や期間をおさえることが重要となる．

Android OS においては，アプリの実行時刻指定として，指定時刻を経過して次に端末が稼働状態となった時点で動作を開始するよう指定する方法（以降では non-wakeup 型指定と呼ぶ）があり，実行タイミングが厳密に特定の時刻でなくても良いような処理は non-wakeup 型指定で実装することにより，他の処理と重畳されて実行され結果として稼働状態となる頻度を抑えることができる．一方で目覚し時計のようなアプリは特定の時刻で動作する必要があるため，指定した時刻において端末がスリープ状態であっても強制的に稼働状態へ遷移させてから処理を開始するよう指定する方法（以降では wakeup 型指定と呼ぶ）を用いている．

non-wakeup 型指定の処理は，ユーザーによる画面オンにより稼働状態となったタイミングや，wakeup 型指定を用いているアプリにより稼働状態へ遷移したタイミングにおいて実行される．この挙動により，連動アプリの動作タイミングが定時アプリの動作タイミングに同期してしまい同期干渉が発生しうる．また，定時アプリの動作タイミングが複数の端末間で同一である場合，同期干渉を受けた連動アプリも大域同期起動する状態となってしまう．

3.1.2 状態変化通知動作による大域同期起動

状態変化通知動作は，2 章で述べたように端末状態変化をアプリに対して通知するものであるが，端末状態変化のうち内部要因状態変化は他のアプリの実行に因っても引き起こされることがある．これにより，端末状態変化を引き起こす，あるアプリの動作タイミングにおいて，該状態変化の通知が行われて他のアプリが動作する．つまり，カスケード起動が発生する．これは本来想定される動作であり，これだけで問題となるものではないが，図 1 に示すように，端末状態変化を引き起こすアプリが大域同期起動する状態となっている場合は同期干渉が発生し，通信集中の原因となる．

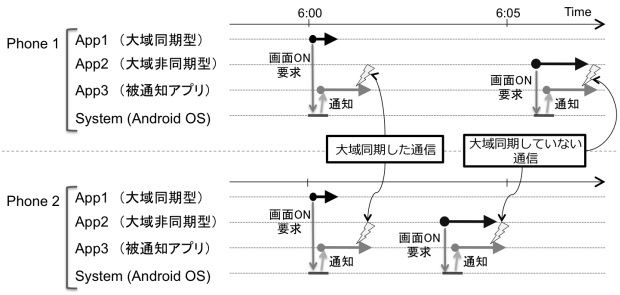

図 1 状態変化通知動作による大域同期起動
Fig. 1 Global synchronization caused by an event delivery mechanism.

例えば，複数人でメッセージを送受信するようなメッセージングアプリの場合，ユーザー操作による画面オンのタイミングで新着のメッセージが無いかをサーバに確認する処理を実装することが想定される．これはユーザーが端末を操作した際に新着のメッセージを通知することが利便性に繋がるものであり，ユーザー操作契機により動作を開始すること自体に問題は無いと考えられる．しかしながら，画面オンとなる契機はユーザー操作起因だけではなく，実際には他のアプリ，例えば目覚し時計アプリなどにより画面オンとなることがある．目覚し時計アプリによる画面オンは，メッセージングアプリ開発者が意図したユーザー操作契機による動作タイミングではないにも関わらず，通信が発生してしまう．

このように，端末状態に変化を引き起こすアプリ（以降では干渉源アプリと呼ぶ）の動作タイミングが複数の端末間で同一となりやすい場合，同期干渉を受けたアプリも大域同期起動する状態となってしまう．大域同期起動を引き起こす干渉源アプリは以下の 2 つに大別できる．

(1) 特定時刻型
アプリの動作タイミングが端末間で同一，または同一となる可能性が高いもの
（例：目覚し時計アプリ）

(2) 一斉同報型
ネットワーク側から複数の端末に対して同報されたメッセージを受け取るもの
（例：メッセージングアプリ）

4. 関連研究

ネットワーク上での輻輳制御手法は古くから研究されてきており，Jacobson の研究 [7] では，TCP 通信における輻輳を回避するために，BSD の TCP スタックに対して再送信のバックオフタイマーの改善やスロースタートなどの修正を行っている．

ネットワーク機器等において，転送すべきパケット量が増加して処理可能な量を超えそうになった際，単純に後から来たパケットを破棄するテールドロップ方式では TCP における再送コネクションの大域同期が発生しやすい．RED(Random Early Detection)[8] では現在の処理量を監視し，確率に基づいてパケットを破棄することにより TCP コネクションの大域同期を回避可能な輻輳制御手法を提案している．

4.1 定時実行動作による大域同期起動

3.1.1 節で述べた定時実行動作による大域同期起動について，小西らの研究 [9] では，non-wakeup 型指定タスクの起動タイミングを，特定時刻に実行される wakeup 型指定タスクの起動タイミングとは重ならないように制御している．これにより，特定時刻に実行される wakeup 型指定タスクからカスケード起動されなくなるため，定時実行動作による同期干渉が発生しない．

4.2 状態変化通知動作による大域同期起動

状態変化通知動作による同期干渉という概念はあまり一般的ではなく，意図しないカスケード起動により発生する輻輳について扱っている研究は多くない．

Kashibuchi らの研究 [10] では，クライアント端末からサーバーに対し Handoff の通知を行った際，以降の処理をクライアント端末ごとに分散させるために応答のパケットを送出するタイミングを一定時間内で乱数的に決定していることが述べられている．

5. 提案手法

本提案手法においては，端末状態変化を引き起こしたアプリが大域同期起動しているか判別し，大域同期起動していればカスケード起動による同期干渉が発生しないようタスクの動作タイミングを制御する手法をとる．

状態変化通知動作による意図しない大域同期起動を防ぐためには，その発生要因となる，大域同期起動アプリを排除するか，カスケード起動による同期干渉を発生させないかのどちらかとなる．しかし，大域同期起動アプリを排除することは，アプリの提供する価値を考慮すると非現実的である．

5.1 大域同期性の分類

意図しない大域同期起動が発生するのは，端末状態変化を引き起こしたアプリが大域同期起動している場合に限られる．そこで，端末状態変化を引き起こしたアプリについて大域同期起動の発生有無をあらかじめ分類しておき，大域同期起動の発生していないアプリについては既存の挙動を保つこととした．また，同様にユーザー操作による端末状態変化の場合も，既存の挙動を保ち即座にアプリへ通知を行う．

分類においては，複数の端末間で同一の時刻に動作するよう実装されており，大域同期起動が発生しやすいアプリ（以降では大域同期型と呼ぶ）と，複数の端末間で特に同期した時刻で動作するようには実装されておらず，大域同期起動が発生しづらいアプリ（以降では大域非同期型と呼ぶ）とに分類し，大域同期型アプリにより引き起こされた端末状態変化の通知については同期干渉が発生しないようタスクの動作タイミングを制御する．

大域同期性の有無を判断するにはアプリ挙動の意図まで理解する必要があり，本質的にはアプリ開発者でなければ分からないため，干渉源になり得るとアプリ開発者が考えた場合には API 呼び出しやマニフェストファイルなどで明示的に示すことが必要となる．しかし，3.1.2 節で述べたように，干渉源アプリとなるパターンは特定時刻型と一斉同報型であるため，アプリの振る舞いを解析することで完璧では無いものの，ある程度の精度で自動的に判別することも可能と考えられる．

前者の特定時刻型であれば，定時動作を行うときの時刻指定方法が，厳密にある時刻での動作を指定している場合が該当する．また，複数の端末からあるアプリがシステムに登録している定時動作の指定時刻を取得することで，複数端末間で同一の指定時刻となっている時刻を指定しているアプリを特定することも可能である．

後者の一斉同報型であれば，ネットワークから受信したデータの送信元やプロトコル，ポート番号などと，そのデータを受信した際に起動したアプリとの組み合わせを記録し，複数の端末で記録したデータを突き合わせることで一斉に同一のデータを受信してアプリが起動していることを検知することが可能である．

5.2 大域同期性に基づく通知の制御

大域同期性を持つアプリによる端末状態変化を通知する場合，通知先アプリがカスケード起動することになるので，同期干渉が発生しないようタスクの動作タイミングを制御する必要がある．制御方法としては，以下の 3 つが考えられる．

- 通知の遮断
  通知先アプリへの通知を破棄する．通知先アプリがカスケード起動しないため，同期干渉は発生しない．

Vol.2013-GN-86 No.20
Vol.2013-CDS-6 No.20
2013/1/17

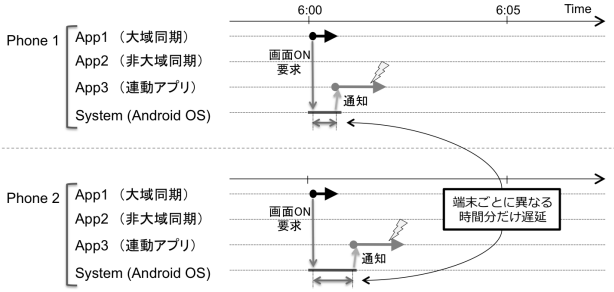

図 2　通知の遅延制御
Fig. 2　Control to delay of notifications

表 1　通知制御方法の比較
Table 1　Comparison of controlling notifications.

	アプリ挙動への影響	実装コスト
通知の遮断	×（大きい）	○
通知の遅延	△（小さい）	○
通信のみ遅延	○（無い）	×（カーネル改変あり）

- 通知の遅延
  通知先アプリへの通知を遅延させる．図 2 のように，遅延時間を端末間で異なる値とすることで，カスケード起動するタイミングが通知元アプリの動作タイミングとずれるため，同期干渉は発生しない．
- 通信のみ遅延
  通知先アプリへの通知は既存の機構と同様に即時通知を行う．通知先アプリでは同期干渉が発生することになるが，通信処理の開始タイミングを遅延させることで通信集中を避ける．ただし，通知先アプリの起動タイミングは大域同期した状態となるため，アプリが次回の起動タイミングを特定間隔（例えば 30 分後）に設定してしまう場合，これを検知および回避することは出来ない．

それぞれの制御方法について，アプリ挙動へ与える影響，および実装コストを比較した結果を表 1 に示す．

通知の遮断を行った場合，アプリは通知を受け取れることを前提に実装されているため，挙動に影響を与える可能性が高い．例えば，電話の着信通知を受け取り，指定時間経過後に留守録機能を起動するようなアプリを考えた場合，そもそも通知を受け取れないため正常に動作しない．

通知の遅延を行った場合，遅延は発生するがアプリは通知を受け取れるため全く動作しなくなってしまう状況は回避できる．また，Android OS の場合，そもそも逐次通知はより優先度の高いアプリが処理を完了しないと通知が行われないため，既存の機構においても通知が遅延する可能性がある．しかし，低遅延で通知が受け取れることを前提にしたアプリが存在する可能性はあり，アプリ挙動への影響もあると考えられる．

通信のみ遅延を行った場合，通知自体は即時に受け取れるため，既存の機構と何ら変わらないため，アプリ挙動への影響は軽微である．一方で，通知先アプリによる通信を遅延させるための処理が必要となり，Android OS の広範にわたって修正が必要となる．特に，NDK[11] を利用して作成されたアプリは Android OS のフレームワークを経由せずに直接カーネル側のソケットを利用できてしまうため，通信の遅延処理を行うためにはカーネル側の改変も必要となる．

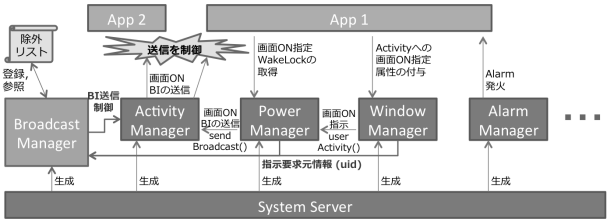

図 3　実装構成
Fig. 3　Architecture of implementation

以上より，我々は，実装コスト，およびアプリ挙動への影響を抑えることが可能な，通知の遅延による制御を用いることとした．

6. 実装

端末状態変化の発生事由を取得し，大域同期性を持つアプリかどうか判別を行い，Broadcast Intent による状態変化の発生を通知する動作に対して遅延制御を実施する機能を Android OS 上へ実装した．なお，ベースとして Android プロジェクトによりオープンソースとして開示されている AOSP 4.0.3 r1 を用い，フレームワーク層の JAVA 実装箇所に対し修正を行っている．実装の構成を図 3 に示す．なお，図中では通知されるデータ，つまり BroadcastIntent の事を BI として記載している．

状態変化通知機構で通知が行われる状態変化は多岐に渡るが，同期干渉による大域同期起動が発生する事例においては，ある特定の端末状態変化に対する通知動作で発生していることがユーザー調査から判明した．散見される事例として，以下の 2 つがある．

- 目覚ましアプリによる画面オン
  目覚ましのようにユーザーが任意の時刻を設定する場合，よほどの理由がない限り正時丁度の時刻を設定することが多いため，アプリは大域同期起動してしまう状態となる．目覚ましアプリは設定時刻に画面オン状態へ遷移するためディスプレイの状態変化を発生させる．
- メッセージングアプリによる画面オン
  1 対多のコミュニケーションが可能なメッセージングアプリでは多数の端末に対して一斉にメッセージ送信が行われる事があり，アプリは大域同期起動してしまう状態となる．メッセージングアプリも同様に，受信

したメッセージを表示するために画面オン状態へ遷移するためディスプレイの状態変化を発生させる．

以上の事例で示した端末状態変化の発生時に Android OS 上で生成される通知は以下の通りである．

(1) `Intent.ACTION_SCREEN_ON`
端末のディスプレイがオフ状態からオン状態へ遷移したことを示す通知．
ユーザー操作によるものか，アプリからの指示によるものかに関わらず通知が行われる．

(2) `Intent.ACTION_SCREEN_OFF`
端末のディスプレイがオン状態からオフ状態へ遷移したことを示す通知．
ユーザー操作によるものか，画面表示のタイムアウトに伴うシステムによるものかに関わらず通知が行われる．

(3) `Intent.ACTION_USER_PRESENT`
ディスプレイがオンになった直後，ロック画面を表示する設定となっている場合にユーザーがロックを解除したことを示す通知．
ただし，ロック画面を表示する設定となっていない場合は通知されない．また，ロック画面を表示する設定になっており，かつ，アプリの挙動によりロック画面の表示が抑制されている場合においても，通知は行われる．この場合，該アプリの画面が表示された時点で通知が発生する．

これらの状態変化通知により同期干渉が発生する状況は，アプリによりディスプレイをオン状態へと遷移させた場合，または，アプリによりディスプレイをオン状態へと遷移させてロック画面を抑制しアプリの画面を表示した場合，の 2 つとなる．したがって，提案手法の実装においてはディスプレイをオン状態へ変化させたアプリを識別する必要がある．アプリからディスプレイをオン状態に遷移させる方法は以下の 2 つがある．

(1) `WakeLock` の取得
ACQUIRE_CAUSES_WAKEUP フラグを付与した WakeLock を取得することにより，画面オフ状態から画面オン状態へ遷移する．
PowerManager において，WakeLock 取得操作が為されると該フラグを参照し，必要に応じて画面オン状態への遷移が行われる．

(2) `Activity` の表示
FLAG_TURN_SCREEN_ON フラグを付与した Activity を生成して表示することにより画面オフ状態から画面オン状態へ遷移する．
WindowManager において，ディスプレイ上に該フラグを有する Activity が表示されることを認識すると，PowerManager に対し画面オン状態へ遷移するよう指示が行われる．

6.1 Broadcast Manager

画面オン状態への遷移を指示したアプリを識別するためには，PowerManager 内での画面オン状態への遷移処理においてアプリを識別すれば良いように思えるが，実際には Activity の表示による画面オン状態への遷移について PowerManager 側でアプリの識別を行うことができない．PowerManager に対し指示を行っているのは ActivityManager であり，どのアプリの指示によるものか区別できないためである．したがって，PowerManager 内，および ActivityManager 内のそれぞれでアプリの識別を行い，その結果を新たに設けた BroadcastManager にて保持する方法とした．

BroadcastManager は，受取ったアプリの識別子から大域同期性を持つアプリであるかどうかを判定し，大域同期性を持つアプリであれば，ActivityManager に対して該当の通知を行うタイミングを制御するよう指示を行っている．遅延時間は 30 秒〜60 秒の範囲で，端末起動時刻を基に算出することで，端末毎に異なる値としている．

7. 評価

5 章では，状態変化通知動作における同期干渉により意図していなかった大域同期起動が発生してしまうことを回避するため，端末状態変化の通知方法を制御する手法を述べた．本章では，実際に干渉源となるアプリにより端末状態変化を発生させ，意図しない大域同期起動が発生しないことを検証する．一方で，ユーザー操作や大域非同期型アプリによる端末状態変化に同期してアプリが動作する挙動は，アプリ開発者の意図した同期動作であるため，これらの動作については既存機構と同様の通知が行われていることも併せて検証する．

評価実験の結果，干渉源アプリにより画面オン状態となっても同期干渉は発生せず，意図しない大域同期起動も起こらなかった．このため，画面オン状態への遷移時には通信が発生していない．また，ユーザー操作や大域非同期型アプリにより画面オン状態になると既存機構と同様に即座に通知が行われたため，アプリ挙動への影響は無かった．

同期干渉の発生有無は端末上で動作するアプリの種類などに依存するため，今回は 380 名程度のモニターユーザーの協力を得てインストールアプリ数やインストールアプリ名を収集した．インストール数上位アプリの挙動を調べ，端末状態変化の通知を受取った際の動作や，定時実行動作を模倣したアプリを作成し，ユーザー調査での平均的なアプリインストール状態とほぼ同等となるアプリ環境を再現した．3.1.1 節で述べたような定時実行動作による同期干渉を排除するため，端末が常に稼働状態となるようにした．

表 2　アラームアプリによる画面オン時の起動回数
Table 2　Number of invocations when alarm app turned screen ON.

	画面オン時起動回数	画面オン時通信有無	通知配送時起動回数
既存機構	5 回	有	—
提案手法	0 回	無	5 回

また，通知に依らず，元々該当の時刻に動作予定であった AlarmManager を利用するアプリの動作は除外した．

詳細な実験環境は以下のとおりである．

実験環境

- 端末仕様
  プロセッサ： ARM v7 デュアルコア 1.2GHz
  RAM 容量： 1GB
  ROM 容量： 16GB
  通信方式：UMTS(3G), GSM, IEEE802.11a/b/g/n(無線 LAN), Bluetooth 3.0+HS
- OS バージョン
  Android AOSP 版 4.0.3r1
- インストールアプリ
  インストール数上位アプリの動作模倣アプリ： 271 個
- スリープ状況
  常に稼働状態（スリープしない）
- 通信状況
  3G による通信のみ

7.1　同期干渉の回避

3.1.2 節で述べた干渉源アプリの 2 種類それぞれで，画面オフ状態から画面オン状態への端末状態変化を 5 回発生させ，その際に他アプリが起動した回数，および，通信の有無を確認することで意図しない大域同期起動が発生していないかを検証した．また，遅延させた通知の配送時においても，他アプリが起動した回数を確認した．

7.1.1　特定時刻型のアプリによる画面オン

特定時刻型のアプリとしてアラームアプリを干渉源アプリに用いた．画面オン時の起動回数および通信有無を表 2 に示す．既存機構では画面オンのタイミングに同期してアプリが動作することで通信が発生しており，同一時刻にアラームを設定する端末数が増えることで問題となる可能性がある．一方で提案手法では，画面オン時に同期した動作をしていないため通信が発生していない．遅延して画面オンの通知が配送されると，その時点でアプリが動作して通信が発生した．なお，本手法では通知を行う時間は乱数的に決定されており，通知のタイミングによってはアプリ側での通信が発生しないことがあった．

表 3　メッセージングアプリによる画面オン時の起動回数
Table 3　Number of invocations when messenger app turned screen ON.

	画面オン時起動回数	画面オン時通信有無	通知配送時起動回数
既存機構	5 回	有	—
提案手法	0 回	無	5 回

表 4　大域非同期型アプリによる画面オン時の起動回数
Table 4　Number of invocations when non-synched app turned screen ON.

	画面オン時起動回数	画面オン時通信有無
既存機構	5 回	有
提案手法	5 回	有

7.1.2　一斉同報型のアプリによる画面オン

一斉同報型のアプリとしてメッセンジャーアプリを干渉源アプリに用いた．画面オン時の起動回数および通信有無を表 3 に示す．特定時刻型と同様に，既存機構では画面オンのタイミングに同期してアプリが動作することで通信が発生している一方で，提案手法では，画面オン時における同期動作をしていないため通信が発生していない．また，遅延して画面オンの通知が配送されると，その時点でアプリが動作して通信が発生した．

7.2　既存機構動作の維持

ここまでで述べてきたとおり，ユーザー操作や大域非同期型アプリによる端末状態変化に同期してアプリが動作する挙動は，アプリ開発者の意図した同期動作である．画面オフ状態から画面オン状態への端末状態変化を 5 回発生させ，その際に他アプリが起動した回数，および，通信の有無を確認することで，既存機構と同様の通知動作が維持されていることを検証した．

7.2.1　大域非同期型アプリによる画面オン

大域非同期型のアプリとして，大域同期していないタイミングにて画面オンへの遷移を行うアプリを作成した．画面オン時の起動回数および通信有無を表 4 に示す．既存機構と同様に，提案手法においても画面オン時のタイミングに同期してアプリが動作し通信を発生している．

大域非同期型のアプリによる端末状態変化は，発生タイミングが各端末によって異なると考えられるため，画面オン時に通信が発生しても問題は無い．

7.2.2　ユーザー操作による画面オン

画面オン時の起動回数および通信有無を表 5 に示す．既存機構と同様に，提案手法においても画面オン時のタイミングに同期してアプリが動作し通信を発生している．

ユーザー操作による端末状態変化は，発生タイミングが各端末によって異なると考えられるため，画面オン時に通

表 5 ユーザー操作による画面オン時の起動回数
Table 5 Number of invocations when user turned screen ON.

	画面オン時起動回数	画面オン時通信有無
既存機構	5 回	有
提案手法	5 回	有

信が発生しても問題は無い.

8. おわりに

Android OS における通信集中の発生要因として，端末状態変化の通知動作による同期干渉の発生に伴い大域同期起動状態となってしまったアプリが通信を発生することが挙げられる．同期干渉により大域同期起動してしまうことはアプリ開発者の意図していない動作であり，この挙動を意識した実装を全ての開発者に求めることは難しい.

本研究では，意図しない大域同期起動を引き起こす同期干渉を抑制するために，端末状態変化を引き起こしたアプリが大域同期性を持っている場合に，通知の配送を遅延するよう制御した．提案手法を AOSP 版の Android OS に実装し，目覚ましアプリやメッセージングアプリなどの大域同期性を持つアプリの動作時においても，任意のアプリに大域同期起動が起こらないことを実端末を用いた実験により確認した.

参考文献

[1] Google Inc. : Android - Discover Android, <http://www.android.com/about/>, (last access:2012.12.5)
[2] Apple Inc. : アップル - iOS 6, <http://www.apple.com/jp/ios/>, (last access:2012.12.5).
[3] Microsoft Corporation : Windows Phone, <http://www.microsoft.com/ja-jp/windowsphone/>, (last access:2012.12.5).
[4] F Qian, et al. : Periodic Transfers in Mobile Applications: Network-wide Origin, Impact, and Optimization, International World Wide Web Conference, 2012.
[5] Google Inc. : Alarm Manager, <http://developer.android.com/reference/android/app/AlarmManager.html>, (last access:2012.12.5).
[6] Google Inc. : Broadcast Intent, <http://developer.android.com/reference/android/content/Context.html#sendBroadcast(android.content.Intent)>, (last access:2012.12.5).
[7] V Jacobson, M. J. Karels : Congestion Avoidance and Control, ACM SIGCOMM Computer Communication Review, 1988.
[8] S Floyd, V Jacobson : Random Early Detection Gateways for Congestion Avoidance, IEEE/ACM Transactions on Networking, 1993.
[9] 小西，神山，川崎，稲村 : Android 端末のための画面オフ状態におけるバックググラウンドタスク実行タイミング制御手法の検討, DPS Workshop, 2012.
[10] K Kashibuchi, et al. : A new smooth handoff scheme for mobile multimedia streaming using RTP dummy packets and RTCP explicit handoff notification, Wireless Communications and Networking Conference, 2006.
[11] Google Inc. : Android NDK, <http://developer.android.com/tools/sdk/ndk/index.html>, (last access:2012.12.5).