往復遅延時間を考慮した無線LANアクセスにおける複数Android端末間の通信制御ミドルウェア

早川 愛[1] 山口 実靖[2] 小口 正人[1]

概要：近年，スマートフォンの高機能化が進むにつれて，多くの機能が実現されている．スマートフォンは小型コンピュータという位置付けではあるものの，膨大なデータ管理が必要なアプリケーションにおいては，リソース不足で処理しきれないため，クラウドサーバ上にそのデータを保持し，大容量通信を行うことで対処している．従ってスマートフォンを利用する上ではこの通信性能が極めて重要であり，本研究では，特に低トラフィックでかつノイズの影響を受けやすい無線通信区間でのクライアント・サーバ間通信において，クライアント側からの発信のパケット転送制御に注目した．本研究が注目する既存研究では，クライアント・サーバ間通信において，クライアント側からのパケット発信の際に，クライアントのアクセスポイント周りで，互いの通信状況を知らせ合うことにより，輻輳を回避し最適な通信環境を実現する制御を行うという方針の通信制御システムの開発が行われている．本研究では，この通信制御システムの高機能化を目指した検討を行う．具体的には各クライアント端末がサーバと通信を行う際の往復遅延時間に着目し，この値を参考にしながらアクセスポイント周りにおける無線通信の最適化を行う手法について，提案と評価を行う．

Transmission-Control Middleware on multiple Android Terminals in a WLAN Environment with consideration of Round Trip Time

Ai Hayakawa[1] Saneyasu Yamaguchi[2] Masato Oguchi[1]

1. はじめに

近年，スマートフォンの高機能化が進んでいる．それに伴い，新機種が次々と市場に出回りスマートフォンが爆発的に増加している．また従来の携帯電話は電話と電子メールに加えて低トラフィックなインターネットアクセスが可能であったが，スマートフォンは小型コンピュータとして機能し，多くの機能が実現された．従来の携帯電話では，OSに組み込まれた唯一のメーラやブラウザしか利用できなかったが，スマートフォンでは，自分の気に入ったメーラやブラウザのアプリケーションをインストールして，利用形態に合うようにカスタマイズすることができる．

ネットワークを利用するアプリケーションは，TwitterやFacebookのようにサーバと連携して情報を受発信するものと，SkypeやLINEのようにクライアント同士で情報を受発信するものに分けることができる．スマートフォンは小型コンピュータという位置付けではあるものの，前者のように膨大なデータ管理が必要なアプリケーションにおいては，リソース不足で処理しきれないため，クラウドサーバ上にそのデータを保持し，大容量通信を行うことで対処している．スマートフォンを利用する上ではこの通信性能が極めて重要であると言えるため，本研究では，クライアント・サーバ間通信におけるクライアント側からの発信のパケット転送制御に注目した．

モバイル端末を用いたクライアント・サーバ間通信は，クライアントのモバイル端末から身近なアクセスポイントまでの無線通信と，アクセスポイントからサーバまでの有線通信で繋がっている．モバイル端末が発信するデータ量のみでは広帯域な有線通信経路上でバッファ溢れを起こす可能性は低いと考えられる．それに対し，無線空間では，ユーザの移動によって，一台のアクセスポイントに繋がっ

[1] お茶の水女子大学
〒112-8610 東京都文京区大塚 2-1-1
[2] 工学院大学
〒163-8677 新宿区西新宿 1-24-2

ている端末数も変わりやすく，トラフィックにも変動がある．すなわち発生するパケットロスの大部分は同じアクセスポイントを共有する端末が多い時や各端末の転送量が多大な時に無線通信区間で起きていると考えることができる．

これまでモバイル端末において通信時に単独で制御 [1]，プロトコルの開発 [2] 等に関する研究は多くなされているが，本研究が注目する既存研究 [3] においては，クライアント・サーバ間通信において，クライアント側からのパケット発信の際に，クライアントのアクセスポイント周りで，互いの通信状況を知らせ合うことにより，輻輳を回避し最適な通信環境を実現する制御を行うという方針の通信制御システムの開発が行われている．これまでのモバイル端末はリソースが最小限であったため，大きな負荷に耐えられず，端末間で高度な制御を行う手法は現実的ではなかったが，現在スマートフォンの需要が高まり，ハードウェアのスペックが格段に向上したため，このような他端末と連携した制御が可能になっている．

このシステムでは，制御パラメータとして同一アクセスポイントを共有する周辺のアクティブな通信端末数が用いられている．本研究ではより通信性能を向上させるための別の制御パラメータとして往復遅延時間 (RTT) に着目し，それが通信性能に及ぼす影響を明らかにした．さらに，その影響を考慮した制御をすることでこのシステムのさらなる高機能化へとつなげる．

2. Android OS

Android は，OS，ミドルウェア，アプリケーション，ユーザインタフェースをセットにしたモバイル端末向けプラットフォームであり，Google 社を中心として開発が行われている．また，2012 年第 4 四半期では全世界のスマートフォン OS の中でも，69.7%とトップシェアを占めている [4]．図 1 に示すように，Android は Linux をベースとし，スマートフォンやタブレット端末をターゲットに，それらに適したコンポーネントが追加されている [5]．Linux OS と大きく異なる部分は，独自に開発された Android の Runtime である Dalvik 仮想マシンを搭載している点である．その上にアプリケーション・フレームワーク，アプリケーションが乗る形態であるため，アプリケーションは Dalvik 仮想マシンに合わせて開発すれば，直感的な操作性に優れた UI を利用することができ，移植性も高い．

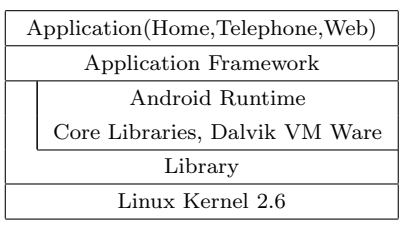

図 1 Android のアーキテクチャ

2.1 Android バージョン

AndroidOS は 2008 年 9 月 23 日に最初のバージョン 1.0 が公開されてから現在まで約 5 年間，次々と最新のバージョンがリリースされている．Android1.5 からは頭文字がバージョンの発表順に C からのアルファベット順で始まるお菓子の名前がつけられている．
本研究では，その様々なバージョンの中でも現在のシェア率が 45.6%と最も高い Android2.3(Gingerbread) と，2012 年 7 月にリリースされた最新バージョンである Android4.1(JellyBean) を対象に取り扱うこととする [6]．

GingerBread と JellyBean では，主に UI の改善・拡張やセキュリティ強化などの様々な仕様変更がされているが，本研究の観点から注目したいのは，TSO(TCP Segmentation Offload) が標準で無効化されていることである．TSO とは，ネットワークカード (NIC) のネットワークインターフェースに内蔵された LSI が送信データを TCP セグメントに分割する処理を行う機能であり，すなわち，NIC が TCP パケットの処理の一部を CPU に代わって実行することにより，CPU のネットワーク関連の処理の負荷が減り、別の処理に集中できるようになる．しかしながら，実際の環境によってはこの機能により通信の劣化やインターフェースに過負荷がかかった際に通信が途切れる，パケットロスが発生するなどの不具合が発生する可能性があるため，TSO を無効化し，より処理能力の高い CPU に仕事を割り当てた方が効率的であるとされている．

JellyBean 以前の Android OS では，この TSO を無効に設定することができなかったため，先に述べた症状が出やすい傾向にあったと思われる．それに対し JellyBean では，TSO が標準で無効になっていることから，輻輳ウィンドウの上限値が撤廃されるなどといった通信面での大きな改善が見られるという特徴がある．

2.2 Android アプリケーション

Android は，無償で提供される開発環境において構築することができ，オープンソースである点からも対応アプリケーションが開発しやすく数も増えるというメリットがある．また Android はキャリア間の制約がないため，アプリケーション開発においても自由度及び汎用性が高いだけでなく，一度マーケットに登録すると，世界中の Android ユーザがインストール可能となる．現在 Android マーケットでは，このような大きなビジネスチャンスを提供されているため，毎年多くのアプリケーションが登録されており，アプリケーション市場は賑わっている．

Android マーケットの存在により，ユーザから見てもアプリケーションの入手は容易である．Dalvik 実行形式のバイトコードの状態で配布されているため，必要なアプリケーションをインストールして，スマートフォンを自由にカスタマイズできる．広告から収益を得ることによりアプ

リケーション自体は無償で提供されているものも多く，気軽にインストールして利用できる．

本研究はこれらのサービスを提供するシステムプラットフォームとしての Android に焦点を当て，通信システムの高速化を目指しているが，このように Android 端末においてアプリケーションの存在を無視することはできない．そこで本研究ではアプリケーションからの無線通信利用を前提として，通信スループットの高速化を目指す．

3. 既存研究

3.1 通信制御ミドルウェア

本研究で改良を加える通信制御システムの概要を説明する [3]．このシステムでは，Android 端末が広帯域有線ネットワーク接続されたクラウドサーバと通信する場合を想定し，輻輳が懸念されるアクセスポイント－ Android 端末間の無線帯域を共有している他端末の通信状況を考慮した制御を目指している．そこで図 2 に示すように，同一アクセスポイントを共有する無線 LAN 空間内において，互いの端末の通信状況，すなわち輻輳ウィンドウを通知し合い，周辺のアクティブな通信端末数を把握することで，全体のトラフィックを予測し，周囲の他端末の通信状況に応じて，輻輳制御アルゴリズムを適切に補正する．それにより，単独に通信するよりも精密な帯域見積りが実現可能となる．

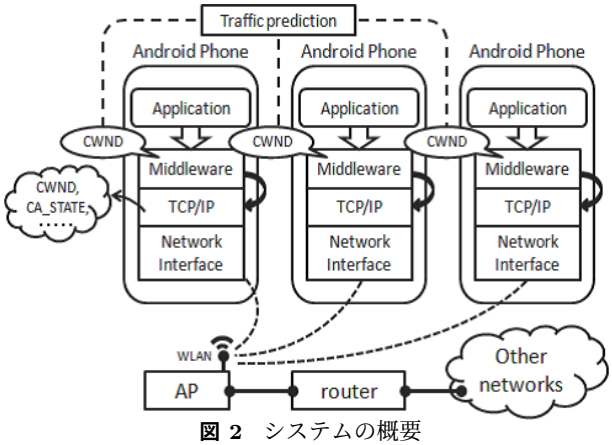

図 2 システムの概要

3.2 カーネルモニタ

前項で述べたシステムのベースとして用いられているツール (図 3) を紹介する．カーネルモニタは，通信時におけるカーネル内部のパラメータ値の変化を記録できる，オリジナルシステムツールである．カーネル内部の TCP ソースにモニタ関数を挿入し再コンパイルすることで，輻輳ウィンドウ値やソケットバッファのキュー長，各種エラーイベントの発生タイミングなどの TCP パラメータを見ることができる [7]．

このツールを Android に組み込み，解析を行う．

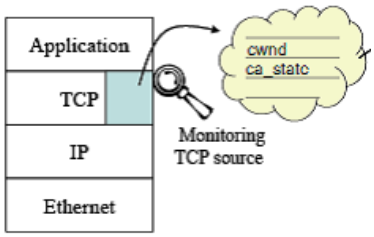

図 3 カーネルモニタ

4. 端末数と通信性能に関する基礎実験

4.1 実験概要

図 4 に示すように，サーバ機と Android 端末の間に実環境を模擬するための人工遅延装置 dummynet を挟み，Android 端末を 1～10 台通信させた時の各 RTT におけるスループットを Iperf[8] を用いて測定した．dummynet で設定した優先部における人工遅延時間は，4ms と 256ms でそれぞれ低遅延，高遅延環境を模擬している．さらに，カーネルモニタにて輻輳ウィンドウ値とエラーイベントを取得し，同時に 1 台の端末に対して ping コマンドを用いて実際にかかる RTT の時間変化を調べた．

Android 端末においては，GingerBread(以下，GB) と Jelly-Bean(以下，JB) でそれぞれ測定し，バージョンによる比較を行った．

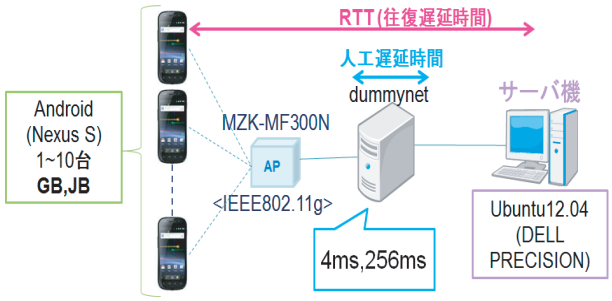

図 4 実験トポロジ

4.2 実験環境

本実験で使用した実験環境を表 1 に示す．また，無線通信方式は IEEE802.11g で行った．

表 1 実験環境

Android	Model number	Nexus S
	Firmware version	2.3.4 , 4.1.1
	Baseband version	I9023XXKD1
	Kernel version	2.6.35.7-hiromi0824 , 3.0.31-ai
	Build number	GRJ22 , JRO03L
server	OS	Ubuntu 12.04 (64bit) / Linux 3.0.1
	CPU	Intel(R) Core 2Quad CPU Q8400
	Main Memory	7.8GiB

4.3 実験結果と考察

図 5，図 6 に GB，JB のそれぞれにおける，通信台数とスループット (平均 (青) と合計 (赤)) の関係を示す．
図 5 より，GB においては，通信端末数を増やすと，合計通信速度が大幅に低下することが分かる．さらに、低下する台数も人工遅延時間によって違うことが分かった．
また，図 6 より，端末数の増加に伴う合計性能の劣化は，GB より JB の方が小さくなっているものの，JB においても端末数を増加させると合計性能が大幅に低下してしまうことが分かる．

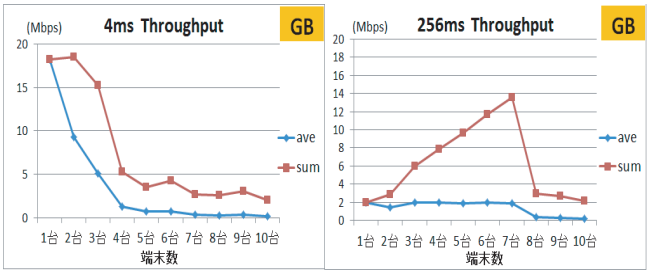

図 5 通信台数の変化によるスループットの平均値，合計値 (GB)

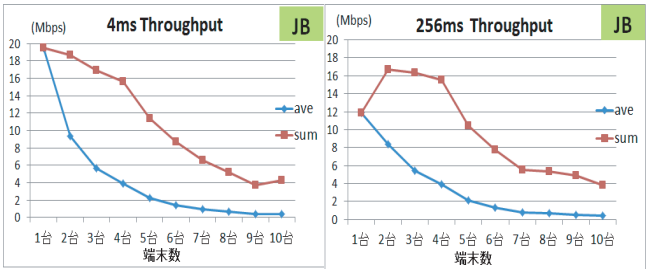

図 6 通信台数の変化によるスループットの平均値，合計値 (JB)

そこでこの原因を調べるために，各遅延環境において輻輳ウィンドウ値と ping により測定した end-to-end の RTT (図 7，図 8，図 9，図 10) をそれぞれ比較した．
図 7，図 8 より，GB においては輻輳ウィンドウは所々落ちているものの，上限値である 60 前後で比較的安定して高い値を保っていることが分かる．それに対して，RTT の遷移を見てみると人工遅延装置で設定した有線部の遅延時間の値 (往復 4ms) よりもはるかに大きな遅延が観察された．
図 9，図 10 の JB では，第 2.1 章で述べたように，輻輳ウィンドウ値の上限が撤廃されたことからその値が大きく成長していることが分かる．図 6 で，図 5 に比べて少数台通信時における性能が向上しているのは，十分な輻輳ウィンドウ値で可用帯域を埋めることができているからだと考えられる．その一方で，JB においても GB と同様に，特に多数台通信において大幅な RTT が発生していることが分かる．
このことから本実験の考察として，有線ネットワーク部の遅延時間や TCP 実装のバージョンによらず，同時に通信する端末数が多い時には，RTT の大幅な増加が通信速度の低下につながるのではないかと予想できる．そこで，カーネルモニタで取得するパラメータに RTT とその最小値を追加した．ここで言う RTT とは，Android OS の TCP 実装内での計測された RTT であり，往復遅延時の実測値である．それに対し RTT の最小値はネットワークの負荷が少ない状態における測定値であり，dummynet で設定した人工遅延時間とほぼ等しくなる．
これらを常時観察することで，現在のトラフィックの混み具合を把握できる．

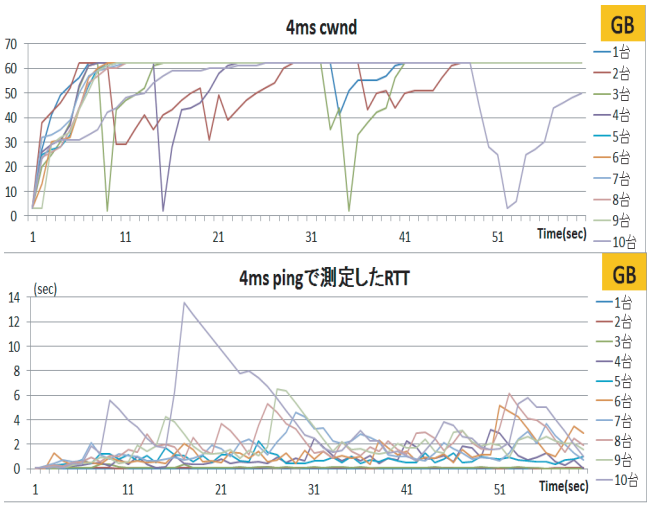

図 7 人口遅延 4ms における輻輳ウィンドウと実際の遅延時間 (RTT) の遷移 (GB)

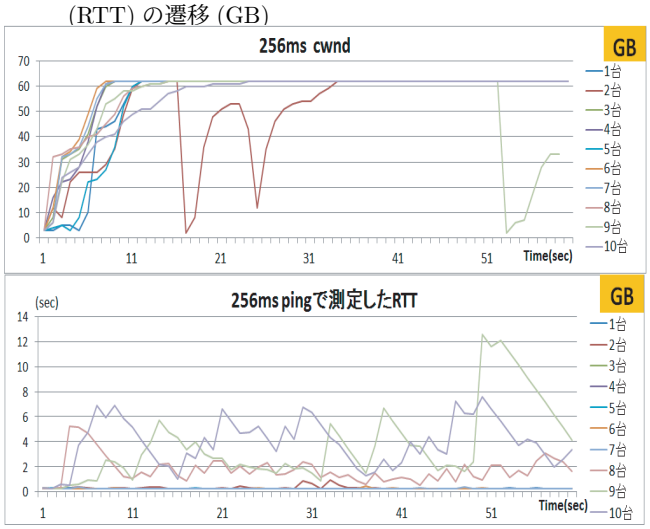

図 8 人口遅延 256ms における輻輳ウィンドウと実際の遅延時間 (RTT) の遷移 (GB)

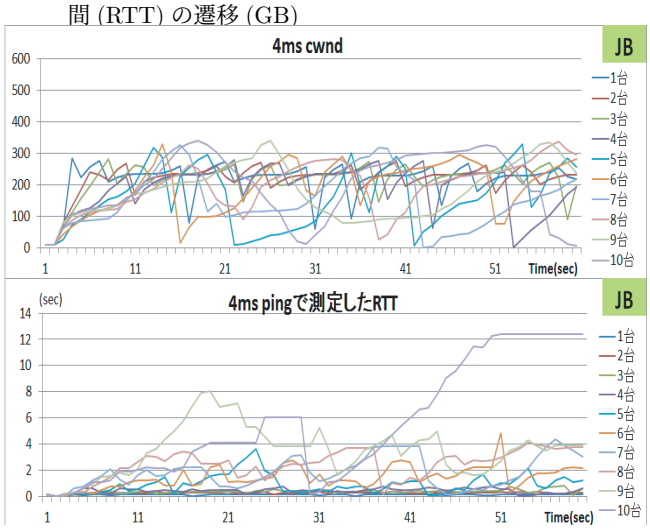

図 9 人口遅延 4ms における輻輳ウィンドウと実際の遅延時間 (RTT) の遷移 (JB)

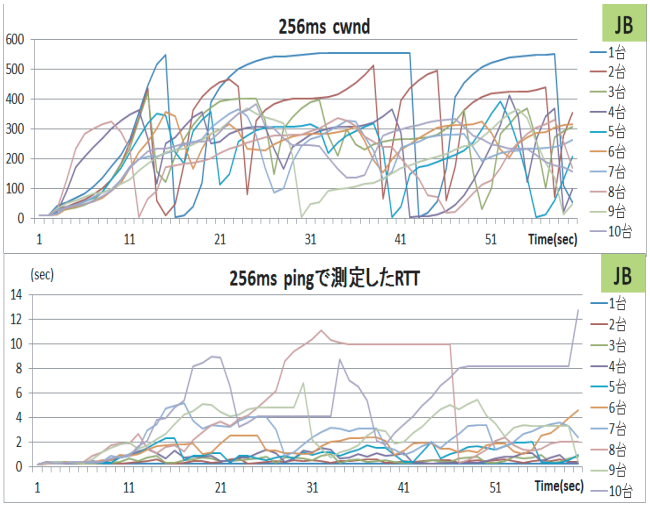

図 10 人口遅延 256ms における輻輳ウィンドウと実際の遅延時間 (RTT) の遷移 (JB)

5. 往復遅延時間と通信性能に関する検証実験

5.1 実験概要

RTT の増加が通信性能に与える影響を明らかにするために，前節と同じ実験環境において改変後のカーネルモニタを導入した Android 端末を用いて, 検証実験を行った.

5.2 実験結果と考察

有線部の人工遅延時間 16ms に設定し，Android 端末を 10 台で通信させた時のスループット，カーネルモニタで取得した RTT の遷移を GB と JB でそれぞれ図 11，図 12 に示す．それぞれのグラフから，時間的遷移を見てみるとスループットが高いところでは遅延時間は小さく，逆にスループットが低いところでは遅延時間は大きいという対比が見られた.

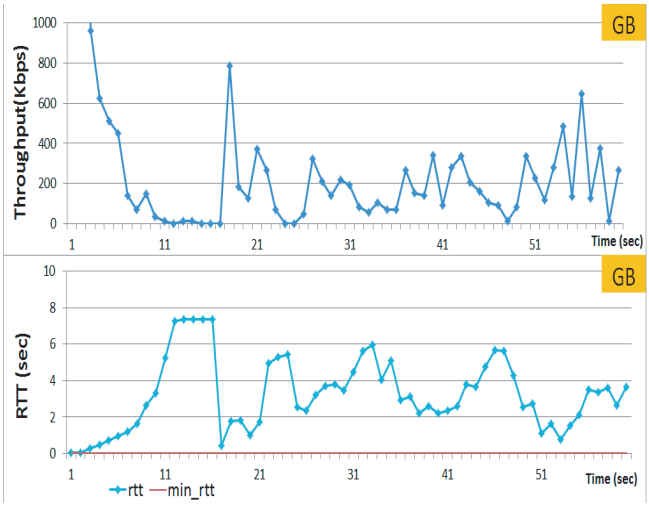

図 11 スループットと往復遅延時間 (RTT) の遷移 (GB)

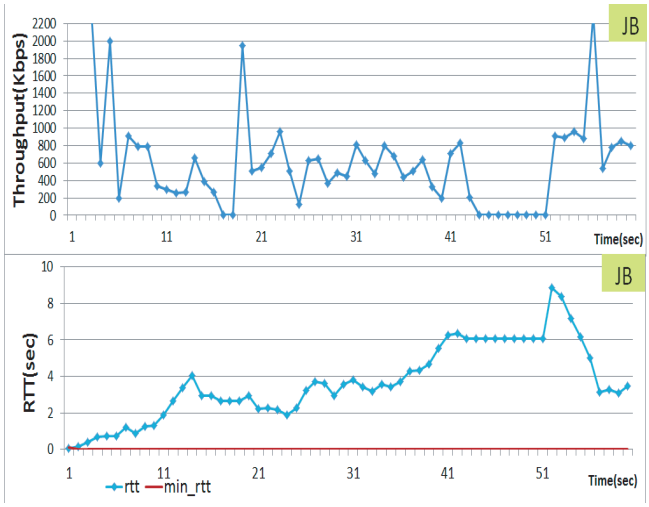

図 12 スループットと往復遅延時間 (RTT) の遷移 (JB)

このことから，前節で予想した通り RTT の大幅な増加が通信性能劣化の大きな要因だと考えられる．よって，通信速度を向上させるためには，RTT の増減を考慮した制御が有効であると言える.

6. 提案ミドルウェア

6.1 ミドルウェアの構成

ここで，本研究で提案する通信制御ミドルウェアの概要を説明する.

改変前のミドルウェアでは，図 13 のように発信部と受信部に分かれて，それぞれで制御を行っていた.

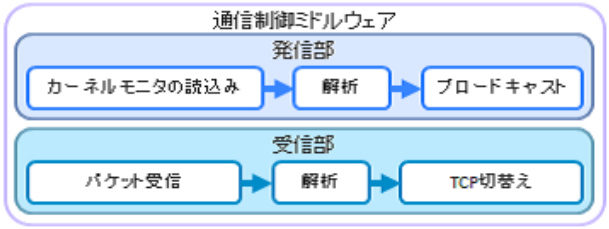

図 13 改変前のミドルウェアの構成

これに対し本研究では，実際に最適化チューニングを行う受信部においてもカーネルモニタの読み込みが必要となったため，この受信部と発信部を一括にまとめることで、導入や制御の簡単化を実現した.

改変後のミドルウェアの構成を図 14 に示す．通信中はカーネルモニタを常時監視し，RTT とその最小値を取得する．取得した値をもとに RTT/最小値で RTT の増減の比率 (ratio-rtt) を求める．また同時に，パケットを受信し，他端末の通信状況を把握してトラフィックを予測する．RTT の比率と通信台数の情報をもとに，外部プロセスから制御可能な proc インタフェースを用いて輻輳ウィンドウの上限値と下限値を設定し，最適化チューニングを行う.

これによって，通信中においても同じアクセスポイント

を共有する他端末が通信を始めたことやそれに伴って急に RTT の値が増加したことをミドルウェアが検知すると，およそ 1 秒毎に輻輳ウィンドウ値を適切な値に設定することでトラフィック発生量を制限し，途中から通信を始めた端末にも均等に帯域を分け合えるよう制御することができる．

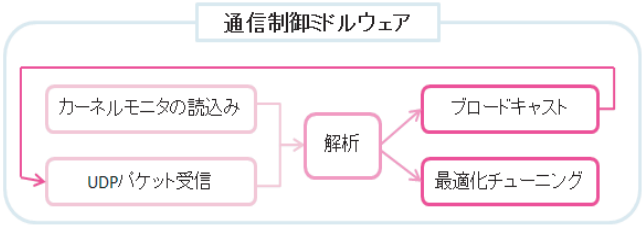

図 14 改変後のミドルウェアの構成

6.2 制御方法

各遅延環境において，以下の式を用いることにより輻輳ウィンドウの上限値，下限値を設定した [9]．

- 帯域幅遅延積 = 帯域幅 [Mbps] * 往復遅延時間 [sec]
- 輻輳ウィンドウの理想値 = 帯域幅遅延積 / 1 セグメントあたりのデータ量 [1.5Kbyte] / 通信端末数
- スループットの理論値 = 輻輳ウィンドウ値 * 1.5[Kbyte] * 8 / 往復遅延時間 [sec]

これらの式より算出した値をもとに設定した，通信端末数における制御と RTT の増減における制御のパラメータ値をそれぞれ表 2，表 3 に示す．また，RTT の最小値 (min-rtt) で以下の 2 段階に場合分けをしている．さらに，GB と JB では輻輳ウィンドウ値の取り得る値が異なるため，それぞれ違う値を設定した．

A ：$0 \leq$ min-rtt <100

B ：min-rtt ≥ 100

表 2 端末数による制御

端末数	GB				JB			
	A		B		A		B	
	max	min	max	min	max	min	max	min
1	21	20	63	62	90	80	555	250
2	16	15	63	62	80	60	300	300
3	11	10	46	45	60	50	400	250
4	9	8	36	35	40	30	200	120
5	7	6	28	27	25	20	150	100
6	5	4	23	22	20	10	100	80
7	4	3	21	20	12	8	80	70
8	3	2	17	16	8	5	70	50
9	2	1	15	14	7	3	50	30
10	2	1	14	13	4	2	35	20

さらに，端末数における制御と RTT の増減における制御をそれぞれ比較して，小さい方の値を採用している．

7. 提案ミドルウェアの評価実験

7.1 実験概要

4 節と同じ実験環境において，本研究で開発したミドルウェアを Android 端末 10 台に導入し，GB と JB のそれぞれにおいて評価実験を行った．

表 3 RTT による制御

GB						JB					
A			B			A			B		
ratio-rtt	max	min	ratio-rtt	max	min	ratio-rtt	max	min	ratio-rtt	max	min
1.0〜	63	62	1.0〜	63	62	1.0〜	100	80	1.0〜	555	300
10.0〜	61	60	2.0〜	62	60	10.0〜	80	60	2.0〜	300	100
15.0〜	58	55	3.0〜	56	53	15.0〜	60	50	3.0〜	200	90
20.0〜	53	50	4.0〜	48	45	20.0〜	50	40	4.0〜	100	80
25.0〜	48	45	5.0〜	42	40	25.0〜	40	30	5.0〜	90	50
30.0〜	41	40	6.0〜	36	34	30.0〜	30	20	6.0〜	50	30
35.0〜	36	35	7.0〜	26	25	35.0〜	20	10	7.0〜	30	20
40.0〜	31	30	8.0〜	21	20	40.0〜	10	5	8.0〜	20	10
45.0〜	26	25	9.0〜	21	20	45.0〜	4	3	9.0〜	8	4
50.0〜	21	20	10.0〜	11	10	50.0〜	3	2	10.0〜	3	2

7.2 実験結果と考察

合計性能を表すトータルスループットの結果を図 15，図 16 に示す．青がミドルウェアなしの状態で，赤が提案手法を用いた制御によるものである．図 15 より，GB では人工遅延 4ms においては端末数 2 台，人工遅延 256ms においては端末数 7 台で大部分の帯域を活用できるようになる．グラフより，提案手法を用いることで人工遅延 4ms では主に少数台通信で性能が向上し，人工遅延 256ms では多数台通信において性能が大きく向上した．図 16 より，JB ではミドルウェアを導入することで設定した人工遅延時間によらず，全体的に性能が向上した．

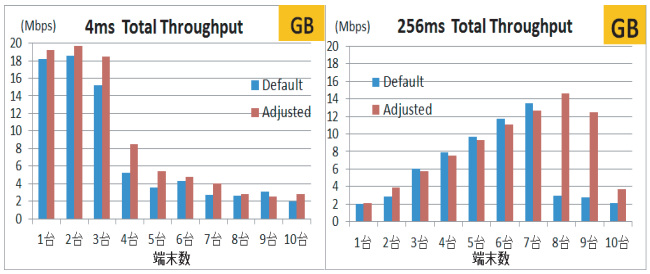

図 15 トータルスループット (GB)

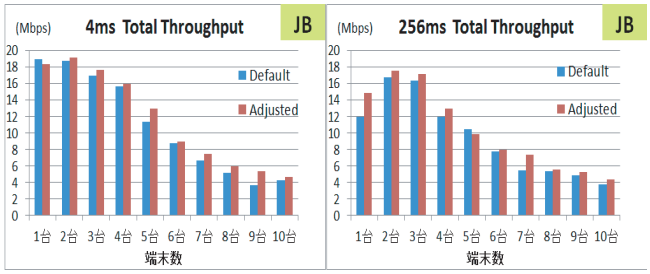

図 16 トータルスループット (JB)

ここで，人工遅延時間 256ms のときの各パラメータ (スループット，輻輳ウィンドウ値，ratio-rtt) の振舞いを，GB では端末数 9 台で，JB では端末数 10 台で通信した場合の時間的遷移によって，それぞれ図 17，図 18 に示す．左軸がスループット，右軸が輻輳ウィンドウと ratio-rtt である．グラフより，RTT が小さいときは設定した輻輳ウィン

ドウの最小値を下回らないこと，また，RTT が大きいときには，設定した輻輳ウィンドウの最大値を上回らないことでその後の大幅な遅延を回避することができた．このことは，ミドルウェア未導入時と提案手法を用いた場合の RTT の時間的遷移を比較することで確認することができる (図 19，図 20)．これらの適切な制御によって，それぞれ通信速度が向上したものと考えられる．

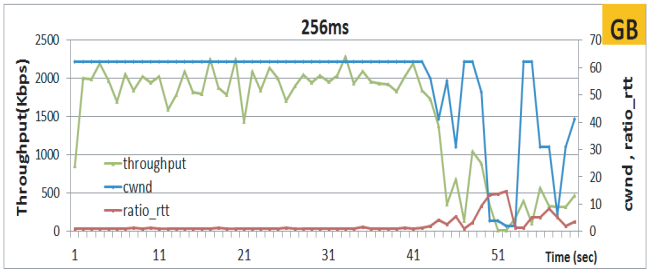

図 17 スループット，輻輳ウィンドウ，ratio-rtt の遷移 (GB)

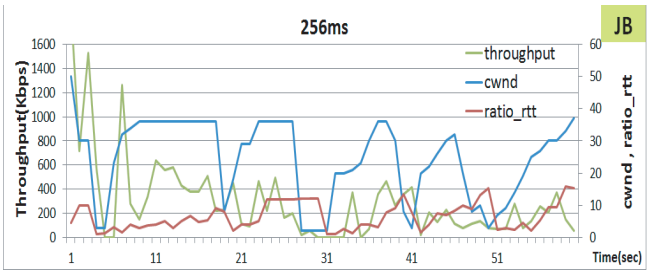

図 18 スループット，輻輳ウィンドウ，ratio-rtt の遷移 (JB)

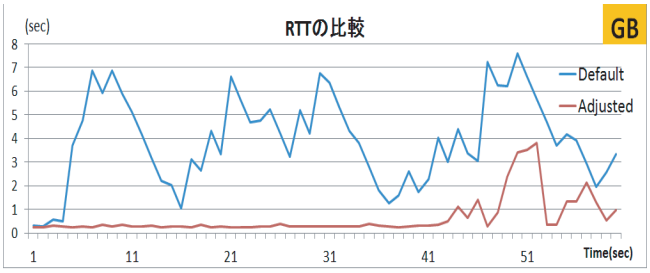

図 19 RTT の比較 (GB)

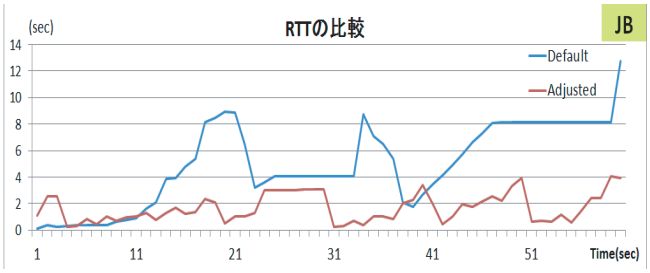

図 20 RTT の比較 (JB)

以上の結果より，GB では特に高遅延環境における多数台通信で，JB では全体的に，本提案手法は効果的であると言える．

8. まとめと今後の課題

本研究では，同時通信端末数が多い環境において合計通信速度が大幅に低下する問題に着目し，同一アクセスポイントにつながる周辺端末数に加える制御パラメータとして RTT を用いる手法について，異なる 2 種類のバージョンにおける Android 端末それぞれにおいて考察を行った．基礎実験の結果から，バージョンの違いによって各パラメータの細かい振舞いや性能は異なることが分かったが，両者に共通して言えることは，同時に通信する端末数が多いときに全体の通信性能が劣化する直接的な原因は RTT の大幅な増加であるということがわかった．そこで，その RTT の増加の比率を制御のパラメータとして取り入れ，より最適な帯域見積りを行う通信制御ミドルウェアを開発した．その評価実験から，本提案ミドルウェアにより，有線部の遅延時間や TCP 実装が異なるそれぞれの環境において，全体の通信速度を向上させることに成功した．

今後の課題としては，さらなる高性能化を目指しより精密な最適化チューニングを行う．さらに，より幅広い環境において性能が向上するための制御手法を考案したい．また，研究目的である連携した通信制御を目指し，今後は各端末の RTT の増減情報を他端末と共有し，その情報に基づく協調的な制御を行う．今回は複数の端末が同じアクセスポイントを介して、遠隔の同じサーバにアクセスするという状況で実験を行ったが，実環境ではそれぞれの端末は同じアクセスポイントを共有していても，その先の接続するサーバは異なることが多い．よってそのような接続先のサーバが複数存在し，さらにそれぞれ遅延環境が異なる場合において，本提案手法が通信性能にどのような影響を及ぼすか調査し，それに適応するための制御手法を考案する．

謝辞

本研究を進めるにあたり，ご指導，ご協力賜りました株式会社 KDDI 研究所の竹森敬祐さん，磯原隆将さんと NTT 研究所の平井弘実さんに深く感謝致します．

参考文献

[1] 塩田 尚基，富森 博幸，奥山 嘉昭，浅井 伸一，佐藤 直樹，”通信ポリシを利用したマルチベアラ利用制御ミドルウェア” 情報処理学会研究報告. MBL，[モバイルコンピューティングとユビキタス通信研究会研究報告]，09196072，一般社団法人情報処理学会，no.44，pp7-12，May 2007

[2] 坪井 祐樹，相田 仁，” 無線環境における複数経路通信プロトコルの検討 ”，情報処理学会研究報告. EIP，[電子化知的財産・社会基盤]，09196072，一般社団法人情報処理学会，no.11，pp1-7，September 2011

[3] 平井弘実，山口実靖，小口正人: モバイルネットワークにおける周辺端末からの情報に基づく協調制御ミドルウェアの提案と実装． DEIM2013，E6-4，2013 年 3 月．

[4] gartner:http://www.gartner.com/it/page.jsp?id=2237315

[5] android:developers:http://developer.android.com

[6] http://japanese.engadget.com/
[7] 三木香央理，山口実靖，小口正人:Android 端末における
カーネルモニタの導入，Comsys2010，2010 年 11 月.
[8] Iperf:http://downloads.sourceforge.net/project/iperf/iperf/2.0.4
[9] W.Richard Stevens 著, 橘康雄, 井上尚司訳, 詳解 TCP/IP
Vol.1 プロトコル，ピアソン・エデュケーション， 2000