

Android 端末の無線 LAN 通信時の トランスポート層の振舞に関する一検討

三木 香央理^{†1} 山口 実 靖^{†2} 小 口 正 人^{†1}

近年, スマートフォン市場の成長に伴い, 携帯端末で動作する組み込み機器のソフトウェアプラットフォームとして Google 社開発の Android が注目されている. アプリケーション開発や柔軟な拡張性において注目度の高い Android 携帯に対し, 本研究ではそのサービス提供を可能にしたシステムプラットフォームとしての Android に焦点を当て, 特にそのネットワーク能力およびネットワークコンピューティング能力について評価する. 本論文では Android のソースコードを x86 系 CPU 搭載 PC 上で動かし, その無線ネットワークにおける通信能力について解析し, そのトランスポート層を最適化することで, より高性能な通信を目指す.

A Study about Behavior of Transport Layer on Android Terminals Connected with a Wireless LAN

KAORI MIKI,^{†1} SANEYASU YAMAGUCHI^{†2}
and MASATO OGUCHI^{†1}

In recent years, with the rapid growth of smart phone market, Android is drawing an attention as software platform of embedded system on personal digital assistant developed by Google. While Android is taken notice for flexible development of application software and expansion of the system, we are interested in Android as a system platform, and in particular, we aim to optimize and evaluate performance of network computing ability. We have evaluated detailed behavior of Android on x86CPU personal computer in the wireless LAN environment, and we have optimized Transport layer and we aim to achieve higher performance of network communications.

^{†1} お茶の水女子大学 Ochanomizu University

^{†2} 工学院大学 Kogakuin University

1. はじめに

近年, 1 人 1 台の携帯電話を所有することが当たり前となってきた. また 1 人で複数台所有することも稀ではなく, サービスや用途によって使い分けているユーザが増加している. 以前は音声通話とメールが主な使われ方であったが, 最近では通信速度が向上し, インターネットや音楽, 動画, ラジオ, テレビ, 非接触型 IC など多機能化されている. 従って, キャリアごとに仕様が違う OS を用いると開発に膨大なコストが掛かるため, 独自の OS では対応しきれなくなっている.

そこで携帯電話向けの汎用 OS が求められてきた. この場合, 基本部分はプラットフォームとして共有化し, 独自の機能やサービスは個別に開発する. これによって開発の効率化が実現できる. またプラットフォームを公開し, オープンソフトウェアにすることで, 対応アプリケーションが作りやすくなり数も増えるというメリットがある. これを実現したものが Google 社により開発された, 携帯端末で動作する組み込み機器のソフトウェアプラットフォームである Android¹⁾ である.

Android はこれまでの携帯端末用ソフトウェアとは異なり, オープンソースであるためアプリケーション開発における制約がない. また Android 搭載の携帯上でならキャリア, 端末を問わずアプリケーションを実行でき, カスタマイズの自由度が高い. これらの要因から多くのユーザにシェアが広まってきている. この様にアプリケーション開発や柔軟な拡張性において注目度の高い Android 携帯に対し, 本研究ではそのサービスを提供することを可能にしたシステムプラットフォームとしての Android に焦点を当て, 特にそのネットワーク能力およびネットワークコンピューティング能力について掘り下げていく. Android については, CPU 負荷量の解析などが行われているが²⁾ ネットワークの能力については, まだあまり知られていない. そこで本論文では Android を x86 系 CPU 搭載 PC 上で動かし, 無線 LAN の通信能力について解析し, そのトランスポート層の仕様を変更することで, より高性能な通信を目指す.

2. 研究背景

2.1 Android の概要

Android のアーキテクチャを図 1 に示す. Android は Linux2.6 カーネルを用いて構築されており, この OS に各種コンポーネントを追加し Android というプラットフォームを構成している. Linux のうち必要最低限の動作を担うカーネル部分だけが採用されているた

め、市場に複数ある Linux パッケージでも Android が構成できるように設計されている。

また、Linux カーネルの上に Android 独自のアプリケーション実行環境である Android Runtime を実装し、Dalvik と呼ばれる独自の仮想マシンを搭載している。これは Java の仮想マシン (JVM) に相当する。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 にあわせて開発すればよく、ポータビリティが高い。

このように、これまでの携帯端末用開発ツールとは異なり、オープンソースでありキャリア間の制約がない。そのためユーザのカスタマイズ自由度が高く、アプリケーション開発の負荷が軽減され、他キャリアおよび他機種への柔軟な拡張性があるといえる。

一方、通信については Linux カーネルの中のプロトコルスタックを用いて行われているため、この TCP 実装部分などで性能が決まってくると考えられる。そのため、本研究ではカーネル中のトランスポート層実装に焦点を当て評価を行う。



図 1 Android のアーキテクチャ

2.2 クロス開発

携帯端末はディスプレイも小さく、CPU 性能やメモリ容量が高くないため、開発時と実行時に異なるコンピュータ環境を用いるクロス開発の形が取られることが一般的である。Android においても一般にクロス開発が行われ、主に開発を行うコンピュータがホスト環境で、Android 実機がターゲット環境となる。ホスト環境に通常無いカメラ機器等はホスト内にあるエミュレータを動かすことで実行できるようになっている。Android は組込み機器であるため、実機で実行できるコマンドにも制限があることから開発環境としては適していない。クロス開発を行うことで開発の効率を上げることができる。

3. 実験システム

本章では本実験で使用した測定ツール、実験環境および実験手順を示す。表 1 および図 2 ～ 5 に本研究の実験環境を示す。

3.1 実験環境

本研究では Android を x86CPU 搭載 PC 用にビルドし、x86CPU 搭載 PC の上で動かしスループット測定を行っている。スループット測定は iperf-2.0.4³⁾ をクロスコンパイルし、Android-x86 を動かしている x86CPU 搭載 PC に送り込んで実行した。クロスコンパイラとしては Android-x86 用 gcc(i686-unknown-linux-gnu-4.2.1) を使用した。

3.2 カーネルモニタツール

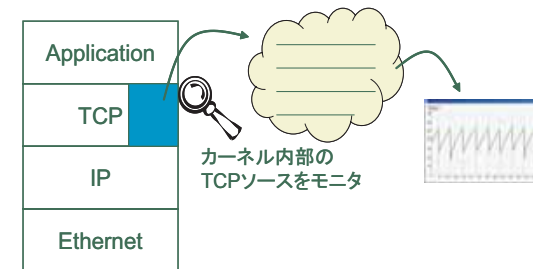


図 2 カーネルモニタツール

本実験では次に、TCP カーネルの振舞をモニタするツールを構築した。図 2 に示すように、カーネル内部の TCP ソースにモニタ関数を挿入しカーネルを再コンパイルした。これによりモニタできるようになった値には、輻輳ウィンドウ、ソケットバッファのキュー長その他、各種エラーイベント (Local device congestion, 重複 ACK, SACK 受信, タイムアウト検出) の発生タイミングなどがある。

3.3 測定環境

3.3.1 基本性能測定 1

まず、図 3 に示すように Android-x86PC 間を IEEE802.11g 無線 LAN 機能を用いて AP を経由で通信した場合の基本性能を測定した。

3.3.2 基本性能測定 2

次に、図 4 に示すように Android-x86 をビルドしたものと同性能の PC (Ubuntu 9.10) 間



図 3 Android を x86CPU 搭載 PC 上で動かしたものの同士の通信



図 4 x86CPU 搭載 PC 間通信 (Ubuntu9.10 間通信)

の通信性能を測定した。こちらも IEEE802.11g 無線 LAN 機能を用いて AP を経由で通信した場合の基本性能を測定した。

3.3.3 高遅延環境における通信性能の測定 1

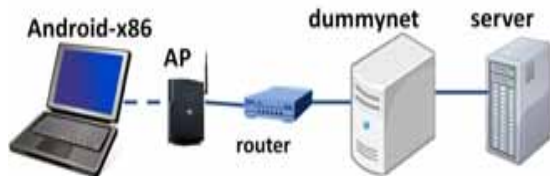


図 5 高遅延環境におけるサーバと Android-x86PC 間通信

図 5 に示すように、間に人工的に遅延を発生させる装置である dummynet を用いて高遅延環境における Android 端末とサーバ間の通信性能を測定した。これは遠隔地に存在するモバイルクラウドを提供するサーバへアクセスする通信を想定している。

3.3.4 高遅延環境における通信性能の測定 2

図 6 に示すように Android-x86 をビルドしたものと同性能の PC (Ubuntu9.10) でも同様に高遅延環境における通信の基本性能を測定した。

4. 性能測定結果

本研究では iperf を用いてソケット通信の性能を測定した。その結果を以下に示す。

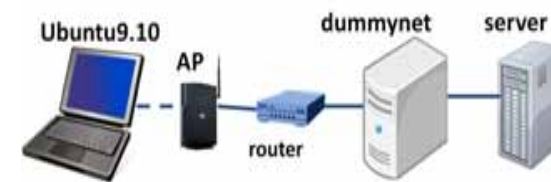


図 6 高遅延環境におけるサーバと Ubuntu 間通信

Ubuntu9.10	OS	Linux-2.6.28-18-generic
	CPU	Intel(R) Pentium(R) M processor 1.73GHz
	Main Memory	512MB
Android-x86	OS	Linux-2.6.29-android-x86
	CPU	Intel(R) Pentium(R) M processor 1.73GHz
	Main Memory	512MB
server	OS	Fedora release 10 (Cambridge)
	CPU	CPU : Intel(R) Pentium(R) 4 CPU 3.00GHz
	Main Memory	1GB

表 1 実験環境

4.1 基本性能測定

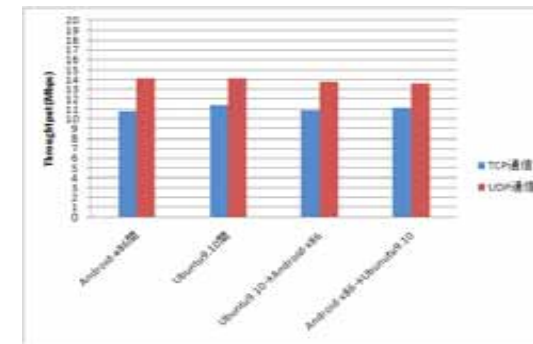


図 7 Android-x86 と Ubuntu9.10 の AP 経由の通信

図 7 に示すように Android-x86 間 TCP 通信の平均スループットは 10.8(Mbps), UDP 通信においては 14.1(Mbps) となった。Ubuntu9.10 間 TCP 通信の平均スループットは

11.4(Mbps),UDP 通信においては 14.1(Mbps) となった。この結果から Android-x86 間の TCP 通信の性能がやや Ubuntu 間の通信性能に劣ることがわかった。

UDP 通信においては双方ともほぼ同等の結果が得られた。既存研究⁵⁾においては,Android 実機の場合,TCP 通信よりも UDP 通信の方が性能が劣ることが確認されているが,本研究では UDP 通信のスループットの方が良い結果となった。

また,参考までに Android-x86 と Ubuntu9.10 間の通信性能も測定したが,特に大きな差は見られなかった。

4.2 高遅延環境における通信性能の測定

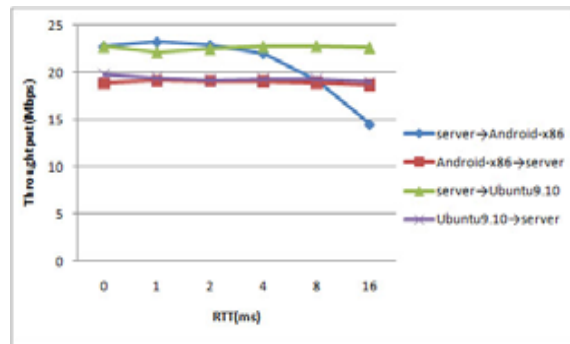


図 8 Android-x86 と Ubuntu9.10 の高遅延環境における TCP 通信

図 8 は横軸に dummynet により設定した往復遅延時間 (RTT) を取った時のスループットのグラフである。Android-x86 からサーバへの TCP 通信と Ubuntu9.10 からサーバへの TCP 通信のスループットは高遅延環境においてもほぼ同じ結果になった。しかし,逆方向の通信において,サーバから Ubuntu9.10 へのスループットは高遅延環境になってもスループットが低下しないのに対し,サーバから Android-x86 への通信は高遅延環境になるとスループットが極端に低下していることが確認できる。

UDP 通信においては図 9 に示すようにサーバから送信するときはスループットがほぼ同じなのに対し,逆方向の通信は Ubuntu9.10 から送信するときの方が Android-x86 から送信するときよりも全体的にスループットが高いことがわかった。

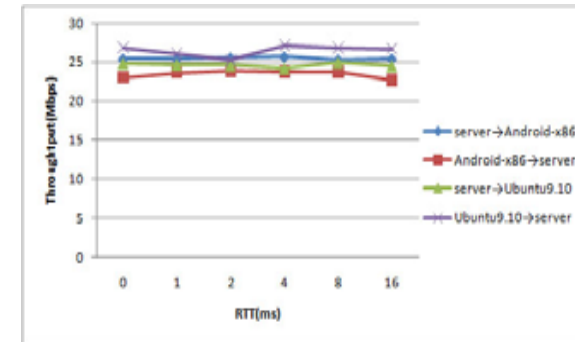


図 9 Android-x86 と Ubuntu9.10 の高遅延環境における UDP 通信

5. TCP チューニング

次に高遅延環境でのサーバとの通信において,TCP の輻輳制御アルゴリズムを変え,その性能の違いを測定した。Android-x86,Ubuntu9.10 で用いた輻輳制御アルゴリズムは,reno,cubic(default)である。Android-x86,Ubuntu9.10 共に,対応している輻輳制御アルゴリズムは 2 つのみであった。

5.1 輻輳制御アルゴリズム

reno は輻輳が発生したときの輻輳ウィンドウサイズの 2 分の 1 の値を ssthresh に設定し,次に輻輳が発生した場合は輻輳ウィンドウサイズを ssthresh から再開して輻輳回避段階に入るアルゴリズムである。これにより過剰な輻輳ウィンドウサイズの減少を避けられるため高速な転送を可能としている。cubic は RTT 公平性に着目し積極的にウィンドウサイズを増加させるアルゴリズムである bic の改良版でウィンドウサイズの増加が緩やかである。

5.2 Android-x86 TCP チューニング

図 10 に Android-x86 に各アルゴリズムを適用した測定結果を示す。データ送信は Android-x86 からサーバ方向である。パケットロスがない場合は 2 つのアルゴリズムにおいてほぼ同等の安定した性能を得ることができた。

パケットロスが加わった場合、高遅延環境になるにつれて性能低下が見られるが,2 つのアルゴリズムに目立った差は見られなかった。

5.3 Ubuntu9.10 TCP チューニング

図 11 に Ubuntu9.10 に各アルゴリズムを適用した測定結果を示す。データ送信は

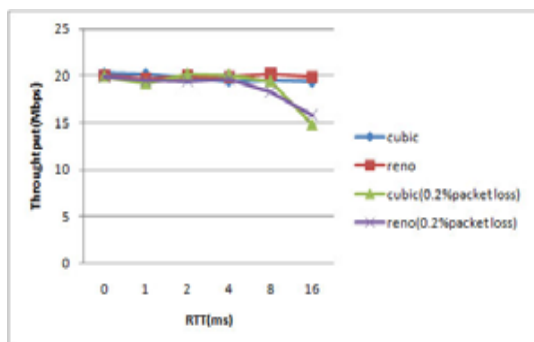


図 10 高遅延環境におけるサーバと Android-x86 の TCP 通信

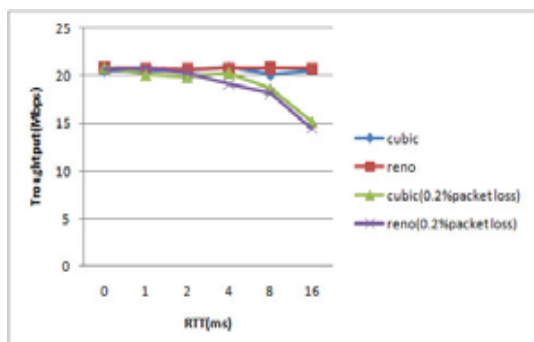


図 11 高遅延環境におけるサーバと Ubuntu9.10 の TCP 通信

Ubuntu9.10 からサーバ方向である。結果は Android-x86 とほぼ同様になった。これにより本実験の範囲において Android-x86 のカーネルの通信性能は輻輳制御アルゴリズム reno,cubic の違いに依存しないことがわかった。

6. パケット解析

4.2 節の基本性能測定において高遅延環境におけるサーバと Android-x86 間の TCP 通信のスループットが低下する原因について調べるため、この時の通信のパケットを解析した。解析には LAN アナライザである Wireshark⁶⁾ を用いた。

6.1 パケット送出量

図 12 ~ 図 13 に RTT=16ms におけるサーバから Android-x86, Ubuntu9.10 への TCP 通信パケット送出総量を示す。Wireshark をサーバ上で動かし、サーバにおけるパケットの送

受信を調べた。これらの図より, Android-x86 が受信する場合の方がパケット総量が少ないことがわかる。このことが、図 8 の RTT=16ms においてサーバと Android-x86 間の TCP 通信のスループットが落ちている直接的な理由と考えられる。

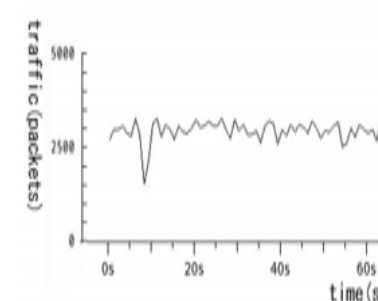
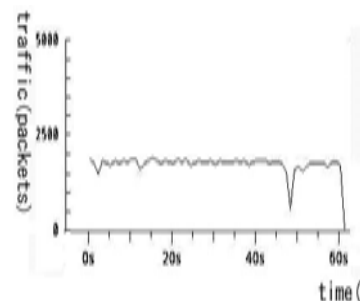


図 12 RTT16ms, サーバから Android-x86 への TCP 通信 図 13 RTT16ms, サーバから Ubuntu9.10 への TCP 通信

6.2 確認応答受信時間

図 14 ~ 図 15 は横軸にシーケンス番号、縦軸に TCP セグメントに対する確認応答セグメントが返ってくるまでの往復遅延時間 (RTT) を示したグラフである。

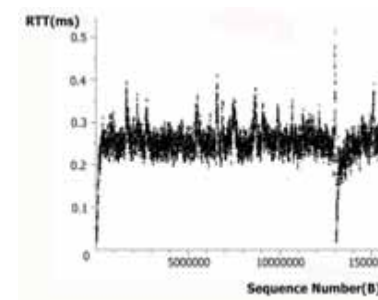
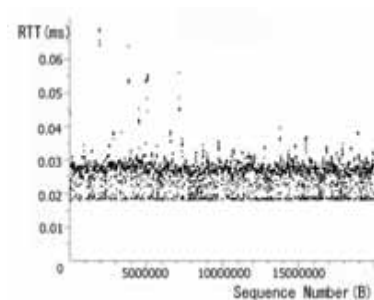


図 14 RTT16ms, サーバから Android-x86 への TCP 通信 図 15 RTT16ms, サーバから Ubuntu9.10 への TCP 通信

図 14 より、サーバから Android-x86 への通信の方が RTT 時間が少ないことがわかる。これは Android-x86 がすぐに確認応答を返しているのに対し、Ubuntu9.10 側が効率よくパ

ケットを受信するために受信バッファに一旦データを溜めているため、確認応答も複数セグメント分まとめて送っているためと考えられる。

以上の結果から、高遅延環境において Android-x86 は一度に大量の通信をするのではなく、小さな塊でパケットを送出しているため、スループットが伸びないということが考えられる。

7. カーネルパラメータチューニング

Ubuntu9.10 よりも Android-x86 で受信バッファの最大値の値が 1 桁小さかったため、Android-x86 の受信バッファの値を変更して性能評価を行った。この 1 桁小さい値は実際に発売されている Android 携帯電話実機⁴⁾ に設定されている受信バッファ量と同じである。変更前、変更後の受信バッファメモリの大きさを表 2 に示す。

	min	default	max
Android-x86(default)	4095	87380	110208
Android 実機	4095	87380	110208
Ubuntu9.10	4095	87380	1966080
Android-x86(変更後)	4095	87380	1966080

表 2 rmem

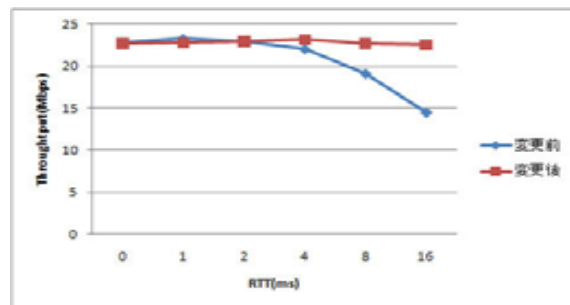


図 16 受信バッファチューニング

受信バッファの値をチューニングした結果、図 16 に示すように、高遅延環境においてもス

ループットが減少しないことが確認された。また、パケット送出総量、確認応答グラフを比べてみても Ubuntu9.10 とほぼ同等の結果が確認された。

8. カーネルモニタツールによる解析

次にカーネルモニタツールを使って Android で TCP 通信によるパケットの流れの詳細を解析する。カーネルモニタは OS のカーネル内にモニタコードを埋め込むことにより、カーネル内部のパラメータ等の実行時における動的な値を観測できるようにしたツールである⁷⁾。今回はその一例として、高遅延環境におけるサーバとの通信における輻輳ウィンドウサイズとスループットの関係について解析する。送信方向は Android-x86 からサーバで、カーネルモニタツールは Android-x86 のカーネル内部に埋め込み、再コンパイルすることにより実装した。

8.1 高遅延環境における性能評価

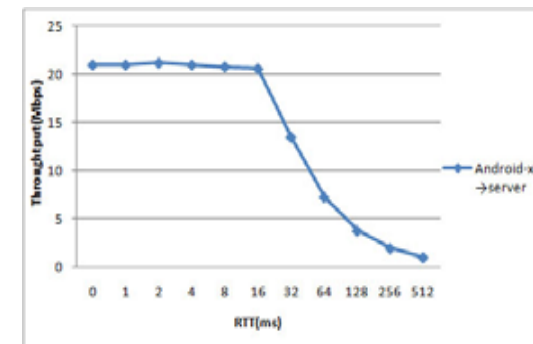


図 17 Android-x86 とサーバ間の高遅延環境における通信

図 17 に示すように、Android-x86 からサーバへの TCP 通信において RTT=32ms から更に高遅延環境になるにつれてスループットが低下されることが確認された。

8.1.1 広告ウィンドウサイズ確認

高遅延環境においてスループットが低下する原因として、ウィンドウ切れがあげられるが、Wireshark を用いて、広告ウィンドウサイズを確認したところ、RTT=512ms の高遅延環境になっても十分な値が保たれていることが確認された。

8.1.2 輻輳ウィンドウサイズ確認

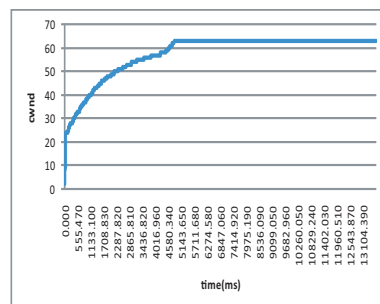


図 18 RTT0ms における cwnd の値

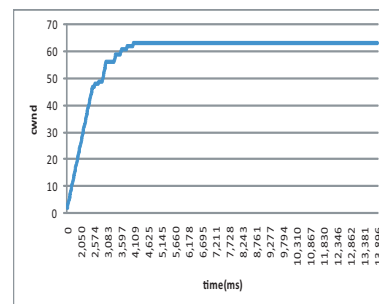


図 19 RTT512ms における cwnd の値

図 18, 図 19 は共にパケットロスの無い状況の輻輳ウィンドウ cwnd の値である。図 19 より RTT=512ms と遅延が大きくなっているにもかかわらず、cwnd の値が 63 で頭打ちになっている。cwnd が 63 の場合、各パケットの大きさを約 1.5KB とすると、ACK なしで送信できるデータ量の最大値は $1.5\text{KB} \times 63 = 94.5\text{KB}$ となる。通信速度を約 20Mbps とすると、RTT=16ms の場合は ACK が返るまで送り続けるべきデータ量は $20\text{Mbps} \times 10^3/8 \times 0.016 = 40\text{KB}$ となってウィンドウサイズは足りているが、RTT=512ms になると、 $20\text{Mbps} \times 10^3/8 \times 0.512 = 1280\text{KB}$ となり、ウィンドウサイズが不足する。すなわち、遅延が短い環境では輻輳ウィンドウ切れを起さないが、遅延が大きくなると輻輳ウィンドウ切れであることが確認された。このことによりスループットが低下しているといえる。

8.2 TCP チューニング時の評価

TCP チューニングを行った際の cwnd の違いを評価した。図 20, 図 21 に RTT=0ms, 0.2% のパケットロス環境での輻輳ウィンドウサイズを示す。両図を比較すると、reno アルゴリズムは輻輳が発生したときの輻輳ウィンドウサイズの 2 分の 1 の値を ssthresh に設定し、次に輻輳が発生した場合は輻輳ウィンドウサイズを ssthresh から再開して輻輳回避段階に入るため cwnd 値を下げる時、cubic アルゴリズムよりも減少が小さいことが解る。

パケットロスが起こった場合 2 つのアルゴリズムによって cwnd 値の違いが確認された。

また、パケットロスの無い環境では 2 つのアルゴリズムに目立った違いは見られなかった。

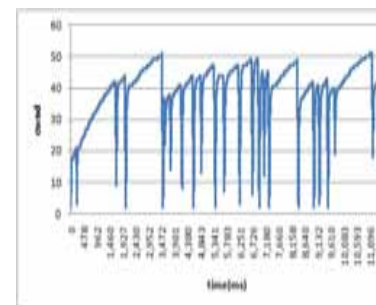


図 20 RTT0ms における cubic の cwnd の値

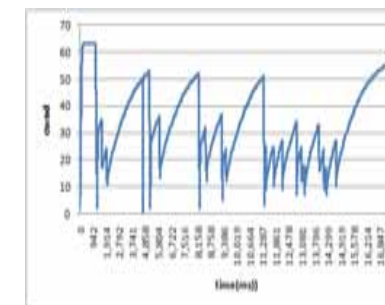


図 21 RTT0ms における reno の cwnd の値

9. まとめと今後の課題

Android-x86 と同性能の CPU を搭載する Ubuntu9.10 間で性能を比較してみたが、双方の間に目立った性能の違いは殆んど見つけられなかった。これは Android のコアの部分には Linux2.6 カーネルが使用されているからであろう。1 つだけ、性能の違いが見られたのは高遅延環境において Android-x86 が受信側となった場合の性能が低下してしまうことであつた。これは Android-x86 受信バッファ量のデフォルトにおける最大値の設定が低かったためであることが確認された。これは Android 実機においてリソースの制限より最大値が低く設定されており、Android-x86 においてもそれが引き継がれて最大値が低く設定されているものと考えられる。また、カーネルモニタツールを導入し、カーネル内部の動きを解析できるようにした。

今後はカーネルモニタツールを用いて様々なパラメータを解析し、Android の詳細な振舞いについて研究を進めていきたい。

参考文献

- 1) Android: <http://www.google.co.jp/mobile/android>
- 2) 間島 崇、横山 哲郎、曾 剛、神山 剛、富山 宏之、高田 宏章: "Android プラットフォームにおける Dalvik バイトコードの CPU 負荷解析", 情報処理学会研究報告, 2010 年 3 月
- 3) Iperf: <http://downloads.sourceforge.net/project/iperf/iperf/2.0.4>
- 4) docomo HT-03A : <http://www.nttdocomo.co.jp/product/foma/pro/ht03a/index.html>
- 5) 三木香央理、小口正人: "Android 端末の様々な無線 LAN 環境における通信性能の考

察”, DEIM2010, 2010 年 3 月

6) Wireshark : <http://www.wireshark.org/>

7) 山口実靖, 小口正人, 喜連川優: ”高遅延広帯域ネットワーク環境下における iSCSI プロトコルを用いたシーケンシャルストレージアクセスの性能評価ならびにその性能向上手法に関する考察”, 電子情報通信学会論文誌 Vol.J87-D-I, No.2, pp.216-231, 2004 年 2 月
