Linux の TCP/IP 通信における高帯域高遅延ネットワーク上で性能低下を引き起こす通信中断の原因解析と改良

児玉祐悦[†]　高野了成[†]
岡崎史裕[†]　工藤知宏[†]

高帯域高遅延ネットワーク上で Linux の TCP/IP を用いた通信中に大量のパケット破棄が生じると、数秒間にわたり通信が停まってしまい、グッドプットをボトルネック帯域で割った通信効率が15%程度に低下してしまう現象が確認された。種々のツールを用いてこの状況について詳細に調べることにより、Linux カーネル 2.6.17 までの TCP/IP 実装に問題があり、再送パケットの破棄を正しく検出できないことが原因であると分かった。この問題を解決する方法を提案し、提案方式が有効であること、および、一部共通の改良を適用している Linux カーネル 2.6.22 よりもさらに平均 28%の性能向上を実現し、通信効率が 96%に向上することを確認した。

Analysis and Improvement of Communication Stops using TCP/IP of Linux on a Fast Long Distance Network

Kodama Yuetsu ,[†] Takano Ryousei ,[†] Fumihiro Okazaki [†] and Kudoh Tomohiro [†]

When many packets are lost using TCP/IP of Linux on a fast long distance network, the communication sometimes stops during several seconds, and the communication efficiency, which is goodput divided by bottleneck bandwidth, is decreased to 15%. In this paper we examined the detail situation of the communication stop by using several tools, and found that the communication stop was caused by missing loss of retransmitted packets because of problems on TCP/IP implementation of Linux kernel 2.6.17 or former version. We fixed the problems, and improved the communication efficiency to 96% on the situation. The performance is about 28% faster than that of the Linux kernel 2.6.22, which fixed some of the problems.

1. はじめに

ネットワークの通信性能の向上は著しく、現在国内外のバックボーンは 10Gbps から 40Gbps の高帯域ネットワークが利用されている。また、日米間や日欧間などの高遅延環境下での大容量データ転送等も実証実験から実運用に移行しつつある。ネットワーク上の通信は TCP/IP による通信が一般的であるが、このような高帯域高遅延ネットワーク上での TCP/IP による通信性能には、いくつか問題点が指摘されている。特に、パケット破棄により通信性能が低下した際に、通信性能の回復に時間がかかりすぎる問題に関しては、種々の TCP/IP プロトコルの改良が行われている [1)~4)]。

我々も高帯域高遅延ネットワーク上での種々の通信性能の評価を行ってきたが、いくつかの実験環境で数秒から数十秒間にわたり通信がほとんど停まってしまう現象を観測した。その一つは、ネットワークルータの QoS 機能により 500Mbps に帯域制御を行ってい

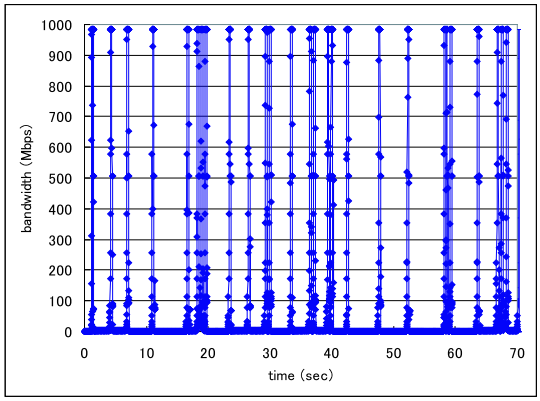

図 1　数秒間通信が中断する例

† 産業技術総合研究所グリッド研究センター (National Institute of Advanced Industrial Science and Techinology (AIST) Grid Technology Research Center (GTRC))

る往復遅延 10 ミリ秒のネットワーク上で、1 ギガバイトのデータを 1 ストリームで転送した際に観測された。このとき、グッドプット (TCP/IP ヘッダなどを除いたアプリケーションレベルで有効なデータのスループット) をボトルネック帯域で割った通信効率は 15%に低下していた。10 ミリ秒ごとの転送バンド幅を調べると、図 1 に示すように、通信が数秒間停まっている区間がいくつもあることを確認した [8]。

我々は、高帯域遅延積ネットワーク上で通信がほとんど停まってしまうこの問題について、その状況を詳細に観測することにより原因を探った。その結果、Linux カーネル 2.6.17 までの TCP/IP 実装に問題があり、再送パケットの破棄を正しく検出できないことが原因であることが分かった。本稿では、その実装の問題点を明らかにするとともに、これを解決する方法を提案し、評価により本改良が有効であること、および、一部共通の改良を適用している Linux カーネル 2.6.22 よりも平均性能で約 28%の向上を実現していることを示す。まず 2 節で Linux における TCP/IP の再送制御について、特に SACK 処理であるスコアボード管理と FACK 輻輳制御についてまとめる。次に 3 節で種々の観測手法を用いて、この問題の状況を詳細に把握する。4 節で、原因について考察すると共に、問題の解決法を提案する。5 節で提案した手法について評価を行い、その有効性を確認する。6 節でまとめを行う。

2. Linux における TCP/IP の再送制御

Linux における TCP/IP の再送制御の概要についてまとめる。

TCP/IP による通信では、ACK を用いてパケットが相手に届いたことを確認している。また、通信遅延が大きな環境で性能を向上させるために、ACK を待たずに複数のパケットを転送するウインドウ制御を行っている。TCP/IP ではパケット破棄を検出するとウインドウサイズを動的に変更する輻輳制御を行っている。通信開始時には輻輳ウインドウサイズ (cwnd) を 2(packet) として、その後 ACK が到着するにしたがい cwnd を+1 するスロースタート処理を行う。このスロースタート処理時には、TCP/IP の通信は 1 RTT (Round Trip Time) 毎にパケット数が 2 倍となるバースト転送が行われる。

TCP/IP では、重複 ACK を受け取ることによりパケット破棄を検出している。この輻輳を状態として管理しており、初期状態の Open(順調にパケットを送信できている状態)、Disorder(パケットの入れ替えを検出した状態)、Recovery(パケット破棄を推定し再送を行っている状態)、Loss(再送タイムアウトが起きた状態) という状態遷移により輻輳制御を行っている。

また、TCP/IP では SACK(Selective ACK) として不連続な ACK 情報を送信側に送ることにより、途中のパケット破棄を効率的に再送できるようになっている。Linux では SACK を受け取ったときの再送処理として、パケット毎のスコアボード管理 [5] と、FACK (Forward ACK) 輻輳制御 [6] を採用している。FACK 輻輳制御では、ネットワーク中のパケット数 (in_flight) を以下の公式により正確に見積もっており、SACK ブロック間のパケットを破棄されたとみなして再送処理を行う。

$$\text{in\_flight} = \text{packets\_out} - \text{lost\_out} - \text{sacked\_out} + \text{retrans\_out}$$

ここで packets_out は snd_nxt(次に送信する新しいパケットのシーケンス番号) から snd_una(未だ ACK を受け取っていないパケットのシーケンス番号) までのパケット数、lost_out は SACK によりパケット破棄と判断したパケット数、sacked_out は SACK により受信通知を受け取ったパケット数、retrans_out は再送を行い ACK 待ちのパケット数である。

一方、Recovery 状態において、輻輳ウインドウサイズ (cwnd) の更新が rate-halving 処理 [6] で行われる。TCP/IP プロトコル処理の基本となる Reno では、再送が開始されると cwnd を直ちに 1/2 に減らす。それと共に、再送を行う前にネットワーク中に存在するパケット数 (in_flight) が cwnd を越えないことをチェックしているため、およそ 1/2 RTT (Round Trip Time) の間パケット送信が停まることになる。この通信停止期間を抑制し、ACK クロッキングを維持しつつ送信レートを 1/2 にするため、rate-halving 処理では cwnd を 2ACK に 1 づつ徐々に下げていく。Linux 2.6.17 カーネルでデフォルトで使用される BIC TCP[4] では cwnd を再送開始時の 8/10 になるまで、この rate-halving 処理により、2ACK に 1 づつ cwnd を下げていく。さらに、rate-halving 処理では、その値と in_flight + 1 とのどちらか小さい方を cwnd として設定している。このため、パケット破棄が検出されると、cwnd は徐々に小さくなり、再送開始時に定めた閾値を越えて小さくなることがある。

図 2 で、SACK スコアボード管理および FACK 輻輳制御について具体例により説明する。ただし簡単のためパケット順序の入れ替えはないものとする。また、シーケンス番号の代りにパケット番号を用いる。以下で、SACK(n,m) とはスタートシーケンスが n でエンドシーケンスが m である SACK を示すものとする。正確には、SACK スコアボード管理により sacked_out, lost_out 等が更新されたあと、rate-halving 処理により cwnd が更新され、その後送信が行われるが、簡単のため以下では cwnd は常に in_flight + 1 に更新されるものとした。このため、1ACK につき 1 パケットの送信が可能となる。以下では SACK スコアボード管理から送信までを 1 つの処理単位として説明する。

- 今、P(0) から P(9) の 10 個のパケットを送信して ACK を待っているとする。このとき in_flight =

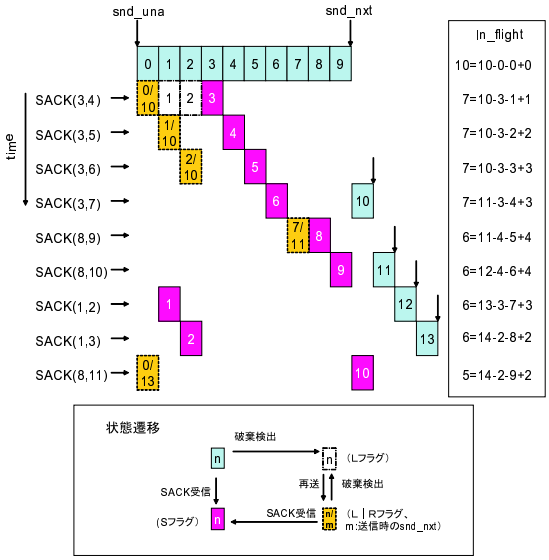

図 2 Linux の SACK スコアボード管理と FACK 輻輳制御

packets_out = 10 である。

- そこに、ACK(0) が来れば P(0) が受信されたことを示すが、SACK(3,4) が来ると、P(0), P(1), P(2) が破棄されたものと判断し、L (Lost) フラグを付け再送の対象とする。P(3) には S (Sacked) フラグを付け、再送の対象からはずす。P(0) を再送する時に、R (Retrans) フラグを追加するとともに、この時点の snd_nxt(ここでは 10) を high_seq として記録する。したがって、lost_out = 3, sacked_out = 1, retrans_out = 1 となるので、in_flight = 7 となる。
- SACK(3,5) が来ると、P(4) に S フラグを付け、再送すべきパケットがあればそれを優先して送信を行う。ここでは P(1) に L フラグがあるので、P(1) を送信して、R フラグ、high_seq を追加する。sacked_out = 2, retrans_out = 2 となるが、in_flight = 7 で変らない。
- SACK(3,6) も同様。
- SACK(3,7) が来ると、P(6) に S フラグをつけ、再送すべきパケットがないので、新しいパケット P(10) を送信する。packets_out = 11, sacked_out = 4 となるが、in_flight = 7 で変らない。
- SACK(8,9) が来ると、P(7) に L フラグをつけ、P(8) に S フラグをつけ、P(7) を再送し、P(7) に R フラグと high_seq(11) を追加する。in_flight = 6 となる。
- SACK(8,10) は SACK(3,7) と同様。このように、SACK スコアボード管理では、SACK 情報をパケット毎に管理し、SACK ブロックの個数にかかわらず、SACK を受け取っていないパケットのみを再送できる。
- その後、P(0) の再送に対する ACK(0) が来れば再送パケットが受信されたということで問題ないが、SACK(1,2) が来た場合には、P(0) の再送パケットが再度破棄されたことを示す。しかし、FACK 輻輳制御ではこのとき P(0) が破棄したとは判断しない。これは SACK(1,2) が P(1) の再送に対する SACK であるか、最初に送った P(1) に対する SACK であるかをパケットの送信順序によっては判断できないためである。P(1) のフラグを $L \mid R$ から S に変更し、再送すべきパケットがないので、新しいパケット P(12) を送信する。in_flight = 6 で変らない。
- SACK(1,3) も同様。
- P(0) の再送の破棄を検出するのは、P(0) の再送のあとで送ったことが確実なパケットに対する SACK を受け取った時である。このために、P(0) の再送時に high_seq を記録している。すなわち、P(0) の high_seq 以降の SACK である SACK(8,11) が来れば、P(0) の再送が破棄されたと判断して、P(0) の R フラグをクリアして再再送を開始する。P(7) の high_seq は 11 なので再送中であると判断しフラグはそのままとする。lost_out = 2, sacked_out = 9, retrans_out = 2 となり、in_flight = 5 となる。

3. 通信中断の詳細な状況

3.1 評価環境

次に、1 節で述べた通信がほとんど停まってしまう現象が確認された評価環境について詳細に述べる。図 3 に示すように、2 台のクラスタをネットワークルータ (Cisco GSR12404) と遅延エミュレータ (GtrcNET-1[9),10)], 片道 5 ミリ秒) を経由して接続し、ノード A からノード B に対して 1 ギガバイトのデータを連続転送する評価を行った。各ノードおよびルータの詳細は表 1 に示す。

GtrcNET-1 は産総研で開発したネットワークテストベッドで、ネットワーク I/F(GbE GBIC) と高速メモリ (SSRAM 16 メガバイト) を各 4 ポートづつ、大規模 FPGA(Xilinx XC2V6000) に接続した構成をとっている。FPGA の回路を変更することにより機能の追加や変更が可能なハードウェア装置である。本稿で利用している主な機能は以下の通りである。

遅延のエミュレーション 2^{-24}秒 $\simeq$ 59.6 ナノ秒 単位で 0 から最大 1 秒まで遅延を挿入する。134 ミリ秒以下の遅延に関してはワイヤレートのトラフィックに対してもパケット破棄を起こさない。

詳細なバンド幅測定 指定した時間間隔でポートの入出力バンド幅を測定する。連続測定できる最少時間間隔は制御 PC の USB ドライバに依存するが、

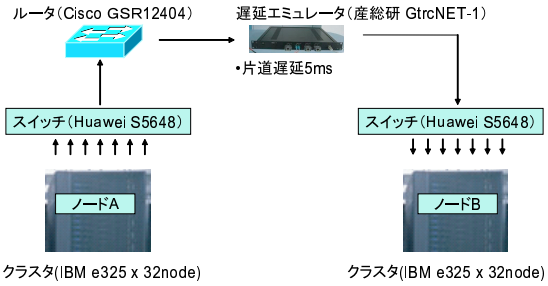

図 3 評価環境

表 1 ノード/ルータの構成

Router (Cisco GSR 12404)	
IOS GS Software	Version 12.0(32)S6
Route Processor Card	PRP-2, 1.2GHz, 1GB mem
Edge	Engine3 (GbE x 4)
Node PC (IBM eServer 325)	
CPU	Opteron/2.0 GHz dual
Ethernet	Broadcom BCM5704
OS	SuSE Enterprise Server 9 (Linux-2.6.17-web100)
system parameters	
net.core.rmem_max	3000000
net.core.wmem_max	3000000
net.ipv4.tcp_rmem	3000000 3000000 3000000
net.ipv4.tcp_wmem	3000000 3000000 3000000
net.ipv4.tcp_no_metrics_save	1
txqueue	10000
TCP variant	BIC (initial_ssthresh=1000)

通常 1 ミリ秒以下の間隔での測定が可能。

パケットキャプチャ 各入力ポート毎に最大 16 メガバイトのパケットデータをキャプチャする。パケットデータとしては、パケット全体、あるいは、パケットの先頭 128 バイトの任意の 4 バイトブロックを指定することができる。

このような評価環境で、ノード A からノード B に対して 1 ギガバイトのデータを連続転送する評価を行った。ネットワークルータでポリシングにより出力を 500Mbps に帯域制御し、許容バーストを 256 キロバイトとした場合には 25 秒程度で終了するのに対し、許容バーストをデフォルトの 16 メガバイトとした場合には 100 秒を越える時間を要することがあった。このとき、グッドプットをボトルネック帯域で割った通信効率で比較すると前者が 64%程度であるのに対し、後者では 15%と 1/4 以下に低下していた。

ルータの帯域制御では通常、トークンバケットによるパケット制御が行われる。ポリシングの制御では、トークンバケットにパケット長分のトークンがあれば、そのパケットを転送し、その分のトークンをバケットから減らす。もし、十分なトークンがなければ、そのパケットを破棄する。トークンバケットの最大値は許容バーストの指定により制御され、トークンは指定した帯域にしたがって定期的にバケットに追加される。通信開始時には十分なトークンがバケットにあり、設定した帯域を越えてパケットが転送されることになる。そして、通信の途中でトークンがなくなると、パケット破棄が起き、その後、設定した帯域にしたがってパケット転送とパケット破棄が繰り返される。

このポリシングの挙動と TCP/IP のスロースタート時のバースト転送の挙動を組み合わせて考えると、ルータで帯域制御を行っているにもかかわらず、バースト長がポリシングの許容バースト長を越えるまで、輻輳ウインドウは増大する。そして、バースト長がルータの許容バースト長を越えたところで、パケット破棄が起きる。例えば、上の実験の例では、RTT が 10 ミリ秒なので、許容バーストが 16 メガバイトの時には、帯域制御が 500Mbps であるにもかかわらず、輻輳ウインドウが GbE の帯域遅延積 (1.25 メガバイト) 以上まで広がってからパケット破棄が起き、輻輳ウインドウサイズの約半分のパケットが破棄されることになる。

3.2 詳細なバンド幅情報

このように、輻輳ウインドウサイズが大きい状態で、大量のパケット破棄が起きるので、通信性能が低下することは理解できるが、帯域制御した帯域の 1/5 以下まで性能が低下することは説明がつかない。そこで、GtrcNET-1 を用いて、10 ミリ秒毎のバンド幅を測定した。その結果が 1 節で示した図 1 である。10 回の試行で実行時間は 26.3 秒から 103.3 秒と大きくばらついたが、性能低下の原因を探るため、図では平均的な時間 (69.4 秒) のかかった場合について示している。また、図のバンド幅はルータからの出力段で測定した値である。この図は 1 ギガバイトのデータを連続的に送ろうとしている時のバンド幅であるが、数秒間通信がほとんど行われていない区間がいくつもあることが分かる。ただし、この区間でも全く通信が行われていないわけではなく、1.2Mbps ほど、すなわち 10 ミリ秒 (1 RTT) で 1 パケット程度の通信が行われていることが確認できている。

3.3 Web100 によるカーネル変数情報

通信が数秒間に渡ってほぼ 0 になってしまう理由を詳細に検討するために、Web100[11] パッチを利用して、カーネル変数を 10 ミリ秒毎に観測した。Web100 は高性能ネットワーク環境における TCP バッファチューニングと性能解析ツールの提供を目的としたプロジェクトである。Web100 カーネルパッチはプロトコルスタックにフックを挿入することで、TCP コネクション単位での各種統計情報を取得し、proc ファイルシステムを介してユーザアプリケーションに提供する。本評価では TCP プロトコルスタックの内部動作を調査するため、Web100 変数である Duration(実行開始からの時間)、CurCwnd(輻輳ウインドウサイズ), Timeouts (再送タイムアウト数)、CurSsthresh(スロースタート閾値) を取得した。それを図示したものが図 4 であ

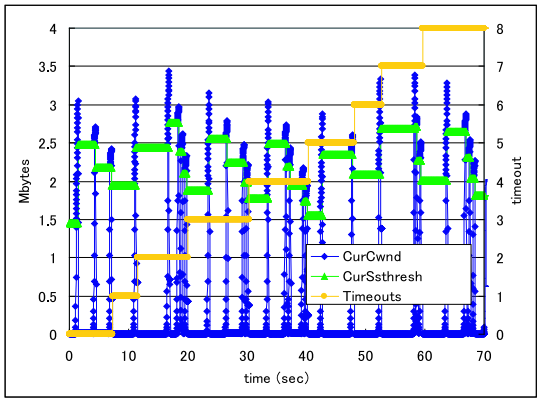

図 4 10 ミリ秒ごとの輻輳ウインドウサイズ等

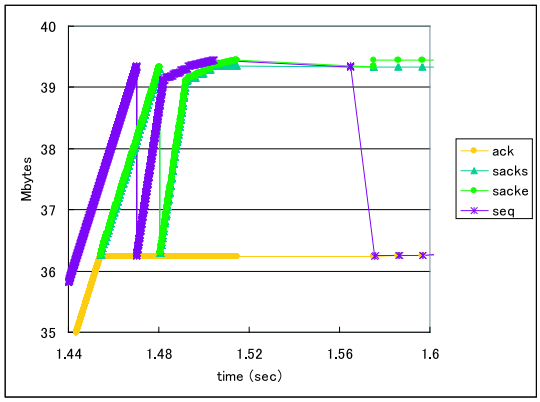

図 5 パケットシーケンス

る。図によると、再送タイムアウトが起きたあとで通信が停まったように見えるパターン（例えば 8 秒付近）と、再送タイムアウトが起きずに通信が停まったように見えるパターン（例えば 1 秒付近）があることが分かる。いずれの場合も、cwnd は 1448 バイト（1 パケット分）あるいはそれに近い値に低下しており、通信中断の間一定である。また、輻輳ウインドウサイズが低下しているところを拡大すると、10 ミリ秒毎に変数の値を読み出しているはずが 100 ミリ秒ほど値を読み出せていない区間が存在していた。これはカーネルがビジーのため Web100 の変数読み出しプロセスがスケジュールされなかったためと思われる。

3.4 パケットキャプチャによるデータおよび ACK のシーケンス情報

10 ミリ秒間隔のデータでは不十分のため、次に、入出力パケットのヘッダをキャプチャして、データパケットのシーケンス番号、および ACK パケットの ACK シーケンス番号や SACK 情報を取得した。パケットキャプチャには遅延エミュレーション用とは別の GtrcNET-1 をノード A とスイッチの間に挿入し、パケットの先頭 128 バイトをキャプチャし tcpdump 形式で保存した。

キャプチャしたデータを解析し、図示したのが図 5 である。図には、データパケットのシーケンス番号（seq）、ACK パケットの ACK シーケンス番号（ack）、SACK ブロックスタートシーケンス番号（sacks）、および SACK ブロックエンドシーケンス番号（sacke）を示している。ACK パケットには最大 3 つの SACK ブロックが含まれるが、最新の情報である先頭のブロックのみを示している。図は最初に輻輳ウインドウが低下した付近を拡大したものである。図によると、1.44 秒から 1.47 秒の間、データシーケンスは 36Mbyte から 39Mbyte に延びている。これは、その部分のデータを送信していることを示している。その送信データに対し、1.45 秒付近まではほぼ 10 ミリ秒の遅延の後に順調に ACK を受け取っているが、その後に 36Mbyte 付近の SACK を受け取ると 1.48 秒まで SACK を受け取り、ACK は更新されていない。このグラフでは SACK は連続しているように見えるが、SACK ブロックを解析すると、SACK ブロックは 20 パケットほど連続したあと、20 パケット飛んで次の SACK ブロックを受信している。これは、ルータではほぼ 20 パケット破棄、20 パケット転送を繰り返して 500Mbps に帯域制御していることを示している。1.47 秒ほどから再度 36Mbyte から 39Mbyte のデータを送信している。このときのシーケンス番号をみると、SACK を受け取らなかった（すなわちパケット破棄と認識された）パケットのみを再送していることが確認できる。その後、1.48 秒付近で再送パケットに対する SACK を受け取り、SACK の受信は 1.49 秒まで増加している。この SACK パケットは、再送パケットの一部が再度破棄されたことを示している。しかし、図によるとこの破棄されたパケットの再再送は 1.57 秒過ぎまで起きておらず、その後 10 ミリ秒毎に 1 パケットづつ送信されていることが分かる。このとき Web100 変数によると、再送タイムアウトは起きていない。また、1.59 秒で輻輳ウインドウサイズは 11584 バイト（8 パケット）となって以降 4.3 秒にスロースタートが始まるまで一定であることを確認できたが、その直前は 100 ミリ秒ほど変数の値を読み出せていない。

これより、再送パケットが破棄された際の再再送処理に不具合があると予想されるが、パケットヘッダの解析や 10 ミリ秒ごとの Web100 の変数読み出しでは情報が不十分で原因究明には及ばなかった。

3.5 カーネル変数の関数呼び出しごとのダンプ

この SACK 処理をより詳細に検討するため、Web100 を拡張し、カーネル変数の値をタイムスタンプ付きでコネクションごとのメモリ配列に格納しておき、あとから proc ファイルシステム経由で値をダンプする機能 epool を実装した。ダンプをとるメモリサイズは Web100 変数として動的に変更可能で、確保したメモリを使いきるとダンプを中断する。

epool により取得したデータを図 6 に示す。カーネル

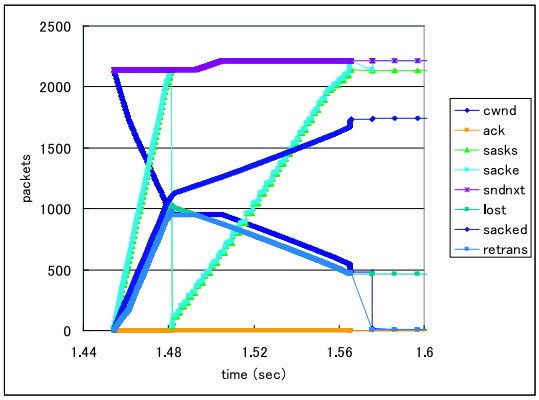

図 6 ACK パケット毎の輻輳ウインドウサイズ等

変数を読み出した場所は、SACK ブロックのフラグの管理を行う tcp_sacktag_write_queue 関数の終了時である。読み出したカーネル変数は、cwnd(輻輳ウインドウサイズ)、ack(ACK シーケンス番号)、sacks(SACK ブロックスタートシーケンス番号)、sacke(SACK ブロックエンドシーケンス番号)、snd_nxt(次送信パケットシーケンス番号) 等のパケットキャプチャで用いたデータの他、2 節で説明した in_flight(ネットワーク中のパケット数) の計算に用いる lost_out(パケット破棄と判断しているパケット数)、sacked_out(SACK を受け取ったパケット数)、retrans_out(再送中のパケット数) である。また、cwnd, ack 等のシーケンス番号は in_flight に合わせて、ダンプ開始時の ack を 0 とした相対値として MSS (Maximum Segment Size) 単位で示している。このため、ack が 0 である間は packets_out は snd_nxt と等しい。また、cwnd = in_flight + 1 である。

送信したパケットに対する SACK ブロックを受け取ったときは lost_out、sacked_out、retrans_out が増加し、cwnd が減少している。次に再送パケットに対する SACK を受け取ったときは sacked_out が増加し、lost_out が減少する。retrans_out は、R フラグのパケットが SACK を受け取り -1 されるが、パケットを再送し +1 されるため最初一定であるが、その後再送するパケットがなくなり減少する。cwnd は、再送および新しいパケットが送信されている間は一定であるが、新しいパケットが送信されなくなると、減少している。ここまでは 2 節で説明した通りである。しかし、1.56 秒付近で high_seq より大きな SACK を受け取っても retrans_out は変化しておらず、再送パケットの破棄の検出がうまく働いていないと考えられる。その後、再度 SACK スタートシーケンスが変化する SACK を受け取ったあとで、retrans_out および cwnd が 1 となっている。この状況で再送パケットの破棄を検出したとすると、各パケットのフラグはすべて S か L となってしまい、2 節で示した公式より in_flight は 0 となり cwnd が 1 となり、観測結果と一致する。したがって、問題は、最初に high_seq より大きな SACK を受け取ったときに、再送パケットの破棄を検出していない点であると考えられる。

4. 通信中断の原因と改良方法の提案

前節の考察により、通信中断の原因は以下の 2 つであると考えられる。

- パケットの再再送が開始されず、Recovery 状態で cwnd が 1 に低下してしまうため。cwnd が 1 に低下してしまうと、Recovery 中は cwnd を増加することはないため、cwnd が 1 のままで残りの再送が行われる。cwnd が 1 の時には RTT あたり 1 パケットしか転送できないため、例えば RTT 10 ミリ秒 で 500 パケット再送し終わるには 5 秒間かかる計算になる。
- パケットの再再送が開始されず、再送タイムアウトが発生してしまうため。再送タイムアウトが発生すると、Recovery 状態から Loss 状態に移行し、スロースタートが開始されるはずであるが、スロースタートが開始されていない。

これらの原因を念頭に、Linux カーネル 2.6.17 の実装を確認すると、以下の 3 つの不具合が確認された。

- ABC (Appropriate Byte Count)[7] の処理に問題があり、再送タイムアウトが起きたあとで Loss 状態に移行したときに cwnd が増加しない不具合があることが判明した。
- ACK パケット内には SACK ブロックは最大 3 ブロック格納されている。ACK パケット内ではデータ受信ノードが最近検出した SACK ブロックが先頭になるように並んでいる。一方、ACK を受信した送信ノードでは、SACK ブロックを昇順に並び替えている。しかし、この SACK ブロックを昇順に並び替える処理のソートに問題があり、先頭の SACK ブロックの情報が後ろの SACK ブロックの情報で上書きされてしまう不具合があることが判明した。これにより high_seq を越える SACK を含む SACK ブロックの情報が消えてしまっていて、再送パケットの破棄を検出できなくなっていた。
- 上で述べたように、Linux では複数の SACK ブロックを持つ ACK パケットの場合、SACK ブロックをいったん昇順に並び替え、最初に低位のシーケンスを含む SACK ブロックの処理を行い、次の SACK ブロックの処理はその直後から続けて行う。これにより、ACK パケットあたり再送キューを一回スキャンするだけで良い。しかし、この最適化が不完全であることが判明した。例えば、図 7 に示す再送バッファの状態の時を考える。これは図 2 で SACK(1,3) を受信したあとの状態である。その

表 2　500Mbps に帯域制御されたネットワーク上の 1GB 転送時の通信性能

	通信時間 (s) 平均/最大/最小/標準偏差	グッドプット 平均 (Mbps)	再送タイムアウト 平均 (回)	通信効率 最小 (%)
2.6.17	52.4/103.3/26.3/27.0	187	5.6	15
2.6.22	21.3/22.2/20.7/0.5	376	28.3	72
提案方式	16.6/16.8/16.5/0.1	483	0	96

後 SACK(8,11) を単独で受信した場合には、図 2 で述べたように P(0) の high_seq がエンドシーケンス (11) より小さいので、P(0) が再再送される。しかし、SACK(8,11) と SACK(1,3)☆の 2 つの SACK ブロックを含む ACK パケットを受信した場合には、SACK ブロックがソートされ、まず SACK(1,3) の処理が行われ、P(0) から P(2) がチェックされる。P(0) の high_seq はエンドシーケンス (3) より大きいので、再再送は行われない。次に SACK(8,11) の処理が、P(3) から P(10) に対して行われ、P(10) に SACK フラグが立つだけで再送処理は行われない。このように複数の SACK ブロックを持つ SACK 処理において、high_seq を越える SACK ブロックが届いているにもかかわらず、再送パケットの破棄が検出できない、という場合があることが分かった。

最初の 2 つの問題は単なるバグであり修正は自明である。また、この 2 つについては、最新の Linux カーネル 2.6.22 では修正されていることが分かった。3 つ目の問題は、最適化のアルゴリズムが不完全であることに起因しており、これを解決するためには、以下の 2 つの方法が考えられる。この問題は Linux カーネル 2.6.22 でも修正されていない。

- 再送キューをスキャンする際、受信した SACK ブロック内の最大エンドシーケンスをあらかじめ調べ、これと再送パケットの high_seq を比較することで、破棄をチェックするように変更する。例えば、図 7 では SACK ブロックの最大エンドシーケンスは 11 であり、SACK(1,3) の処理時にも各パケットの high_seq をその値と比較することにより P(0) の再再送を検出できる。
- SACK のスコアボード管理では、SACK ブロックの管理をパケットごとに行っているため、一度処理した SACK ブロックを再度処理する必要はない。前回処理した SACK ブロックを覚えておき、同じ SACK ブロックの処理を省略することにすると、新しい SACK ブロックに対して再送キューの先頭からスキャンすることになり、その SACK が high_seq を越えるブロックを含んでいる場合には、正しく再送パケットの破棄を検出できるようになる。

☆ SACK(1,3) は直前の SACK ブロックと同じ情報である。複数 SACK ブロックが送信される場合、先頭 SACK ブロックのみ新しい情報で、残りの SACK ブロックは直前の SACK ブロックがシフトされていることが多い。

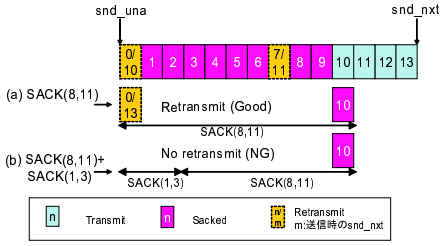

図 7　再再送すべきパケットを検出できない例

例えば図 7 では、SACK(1,3) が直前の SACK ブロックに含まれていることをチェックすることにより、SACK(8,11) の処理のみを行うこととなり、図 7(a) と同様に正しく再再送を行える。

どちらの改良方法でも、正しく再送パケットの破棄を検出できるが、後者の方法では余分な SACK 処理を省略することができ、SACK 処理オーバヘッドを軽減する効果も期待できると考え、後者の方法を実装した。

5. 性能評価

これらの改良の効果を評価するため、Linux カーネル 2.6.17 (改良を行う前)、Linux カーネル 2.6.22 (2.6.17 に最初の 2 つの改良を適用することと同等)、提案方式 (2.6.22 に 3 つめの改良を適用) の 3 種類について、10 回の試行を行った時の結果を、表 2 にまとめた。

Linux カーネル 2.6.17 では、通信時間が 26.3 秒から 103.3 秒とばらつきが非常に大きく、グッドプットの平均も 187Mbps と低い。グッドプットをボトルネック帯域で割った通信効率の最小値は 15%と非常に低い。

Linux カーネル 2.6.22 では、通信時間のばらつきが小さくなり、標準偏差が 0.5 と Linux カーネル 2.6.17 に比べて 1/50 以下になった。また、平均グッドプットも 376Mbps と約 2 倍に向上した。さらに、通信効率の最小値は 72% と大きく改善された。Web100 変数により 10 ミリ秒ごとにチェックし、再送タイムアウト後に直ちにスロースタートが開始されない問題、および、再送タイムアウトを起こさない状態で cwnd が 1 となり破棄を検出したパケットをすべて再送するのに時間がかかる問題は解決されたことを確認した。しかし、再送タイムアウトは依然として起きており、その回数は平均 28 回と Linux カーネル 2.6.17 よ

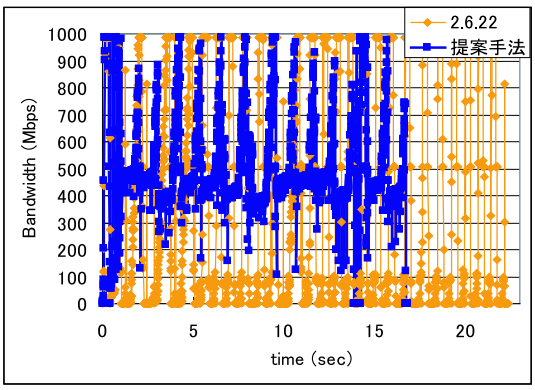

図 8 提案手法適用時の 10 ミリ秒毎の転送バンド幅

りも増えていた。これは、数秒間の通信途絶が解消された結果、再送タイムアウトの起きる状況が起きやすくなったためと考えられる。

提案方式では、通信時間のばらつきはさらに減り、標準偏差は 0.1 と Linux カーネル 2.6.22 に比べて約 1/5 になった。また、平均グッドプットは 483Mbps と、Linux カーネル 2.6.22 に比べて約 28%性能が向上した。さらに、通信効率の最小値は 96%となった。再送タイムアウトは起きていない。図 8 に通信時間が最大の場合における Linux カーネル 2.6.22 と提案手法の 10 ミリ秒単位の転送バンド幅を示す。図 8 によると、Linux カーネル 2.6.22 ではバンド幅が 0 から 1Gbps まで大きく変動しており、特に RTO を起こしてからスロースタートが始まる付近ではバンド幅が 100Mbps 以下に集中している。一方、提案手法ではパケットの破棄によるバンド幅の低下は定期的に起っているが、バンド幅が低下したときでも 400Mbps 程度を保ち、直ちにバンド幅が立ち上がっていることが分かる。

6. おわりに

本稿では、Linux 上で TCP/IP を用いて通信している時に、大量のパケット破棄が生じると、数秒間にわたり通信性能がほぼ 0 になってしまう現象について、詳細なバンド幅測定、Web100 パッチによるカーネル変数のチェック、パケットキャプチャによる SACK 情報の解析、カーネル変数メモリダンプツール、等種々のツールを用い、その状況を詳細に調べた。これにより、Linux カーネル 2.6.17 までの TCP/IP 実装に問題があり、再送パケットの破棄を正しく検出できないことが原因であると分かった。そして、この問題を解決する方法を提案/実装し、通信が数秒間にわたって停まってしまうことがなくなり、通信効率の最小値が 15%から 96%に大きく向上することを確認した。また、本改良は、一部同様の改良を行っている最新 Linux カーネル 2.6.22 に対しても平均で約 28%の性能向上を確認した。本改良点をまとめ、Linux カーネル開発者へ報告をする予定である。

本改良は Linux の TCP/IP の実装に対する改良であるが、Linux の TCP/IP の実装は新しい提案/機能を積極的に採用しており、それらの提案の参照実装としての意味合いもある。そのため、本改良は Linux での改良のみならず、それらの提案が実装のバグにより不適切に評価されることを防ぐと共に、Linux を参照して実装される他の OS にも効果があると考える。

謝辞

本研究の一部は、文部科学省科学技術振興調整費「グリッド技術による光パス網提供方式の開発」および文部科学省「経済活性化のための重点技術開発プロジェクト」の一環として実施している超高速コンピュータ網形成プロジェクト（NAREGI：National Research Grid Initiative）による。

参 考 文 献

1) T. Kelly: “Scalable TCP: Improving Performance in Highspeed Wide Area Networks”, ACM SIGCOMM Computer Communication Review, vol.33, no.2, pp.83-91, April 2003.
2) S. Floyd: “HighSpeed TCP for Large Congestion Windows”, RFC 3649, December 2003.
3) C. Jin, D.X. Wei and S.H. Low, “FAST TCP: motivation, architecture, algorithms, performance,” Proc. of IEEE INFOCOM, March 2004.
4) L. Xu, K. Harfoush, and I. Rhee, “Binary Increase Congestion Control for Fast Long-Distance Networks,” INFOCOM 2004.
5) E.Blanton, M. Allman, K. Fall, L. Wang, ”A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP,” RFC 3517, 2003.
6) M. Mathis, “TCP Rate-Halving with Bounding Parameters,” http://www.psc.edu/networking/papers/FACKnotes/current/, 1997.
7) M. Allman, ”TCP Congestion Control with Appropriate Byte Counting (ABC),” RFC 3465, 2003.
8) 児玉, 岡崎, 工藤, 高野, “ネットワーク装置の帯域制御機構の評価,” 第 8 回インターネットテクノロジーワークショップ, 日本ソフトウェア科学会インターネットテクノロジ研究会, 2007.
9) Y. Kodama, T. Kudoh, R. Takano, H. Sato, O. Tatebe and S. Sekiguchi, “GNET-1: Gigabit Ethernet Network Testbed,” Proc. of 2004 IEEE International Conference on Cluster Computing (Cluster2004), pp.185-192, 2004.
10) http://www.gtrc.aist.go.jp/gnet.
11) The Web100 Project, http://www.web100.org/.