AIST

HPCユーザが知っておきたい TCP/IPの話

～クラスタ・グリッド環境の落とし穴～

産業技術総合研究所
情報技術研究部門
高野　了成

2009年5月29日　SACSIS2009チュートリアル

HPCユーザとTCP/IP

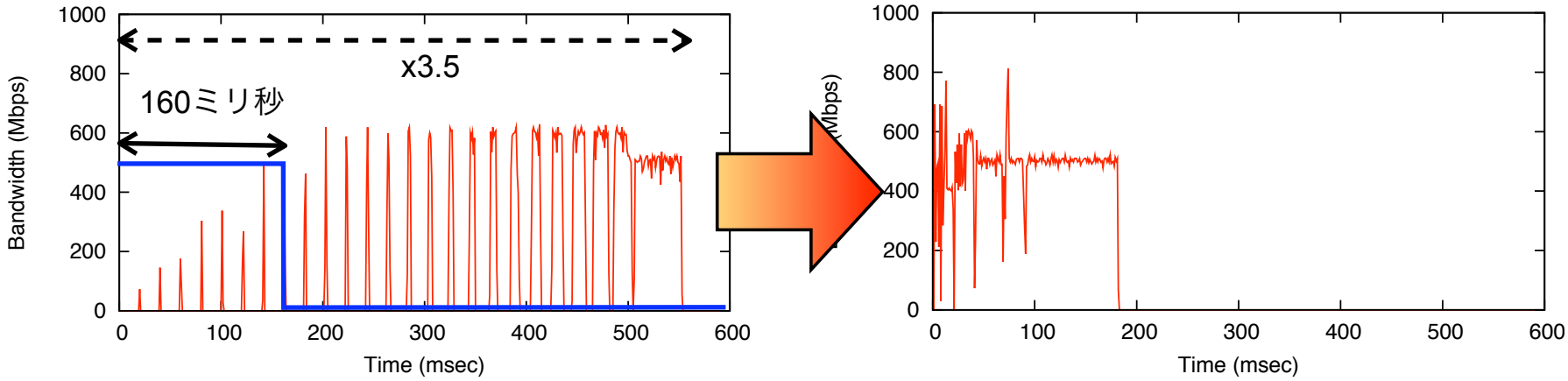
限界までネットワーク性能を出したい！

- グリッド環境で性能が出ないと悩んでませんか？
  - 例）長距離大容量データ転送（帯域 1 Gbps、RTT 100ミリ秒）におけるチューニング
    - デフォルト設定だとスループットは240 Mbps ☹
    - 各種チューニングにより940 Mbps ☺

HPCユーザとTCP/IP

限界までネットワーク性能を出したい！

- Ethernetで安価にPCクラスタを組めたが、性能もそれなりと割り切っていませんか？
  - 例）TCP/IPが苦手とする間欠通信の改善
    （2秒ごとに10MBのデータの連続送信の繰り返し）



チュートリアルの目的

- クラスタ・グリッド環境でTCP/IPの通信性能を引き出すポイントの理解
  - 長距離大容量データ転送
  - MPI並列通信
- TCP/IP、特にTCP輻輳制御の仕組みの理解
  - 標準TCP（Reno [RFC2851]）
  - 最近の高速TCP（CUBIC、Compound TCP）

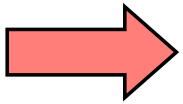
発表の流れ

- 標準TCP/IPの基本
- TCP/IPと長距離大容量データ転送
- 高速TCP輻輳制御アルゴリズム
- TCP/IPとMPI並列通信
- その他のチューニング方法
- TCP/IPをよりよく知るためのツール
- まとめ

標準TCP/IPの基本

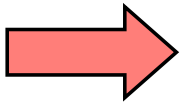
TCP (Transport Control Protocol)

- コネクション指向で、信頼性のあるバイトストリーム型通信の提供



再送制御

- 通信相手やネットワークの混雑状況にあわせた、自律的な送信レート制御



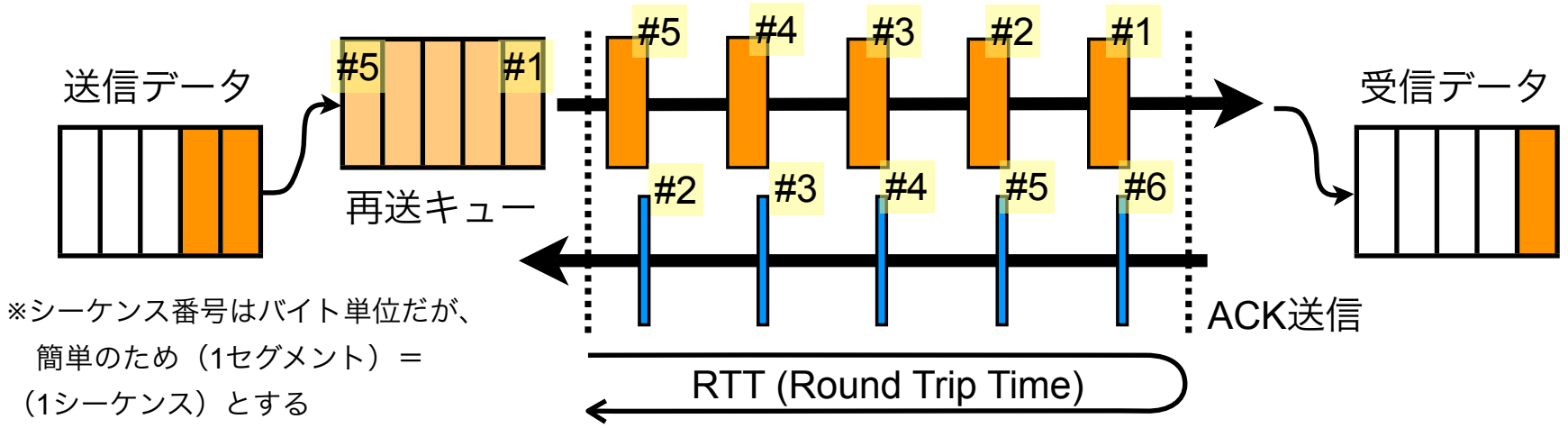
フロー制御、輻輳制御

標準TCPの基本（対象範囲）

- 再送制御
  - 再送タイムアウト
  - 高速再送
  - SACK
- 送信レート制御
  - フロー制御
  - 輻輳制御
    - スロースタート
    - 輻輳回避
    - ACKクロッキング

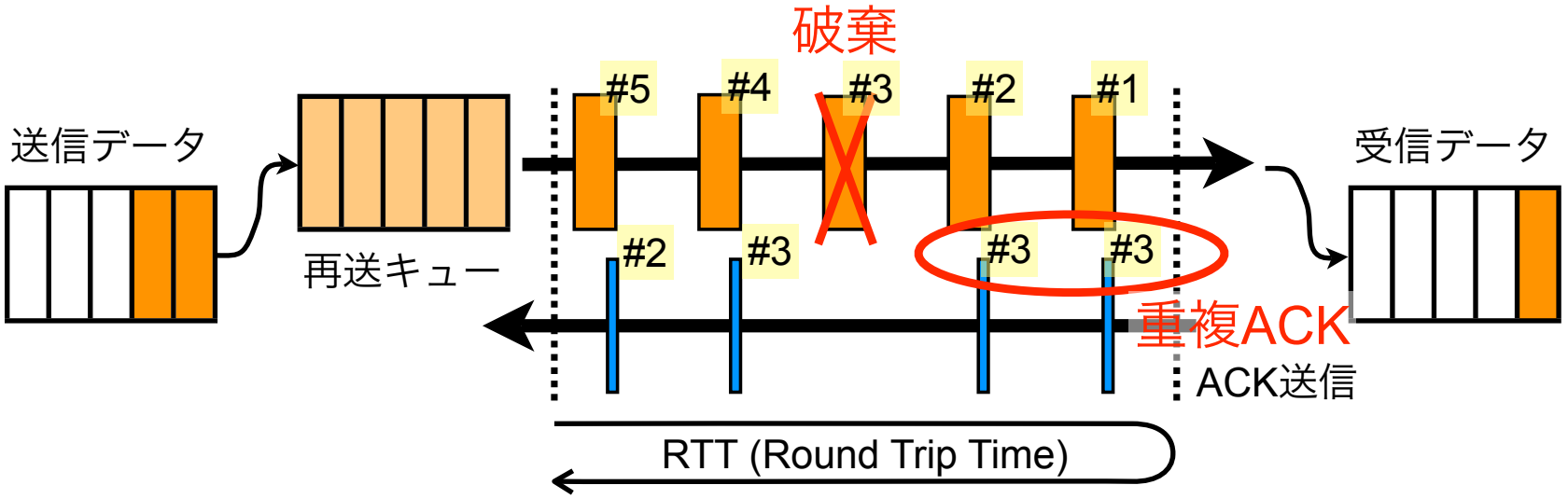
TCPデータ処理の流れ（１）

- データはセグメントに小分けして送受信
  - 送信者：シーケンス番号を付加
  - 受信者：「次に受信したい」シーケンス番号をACK番号に持つACKを返信
- 再送に備えて、セグメントはACKが届くまで再送キューに保持



再送機構

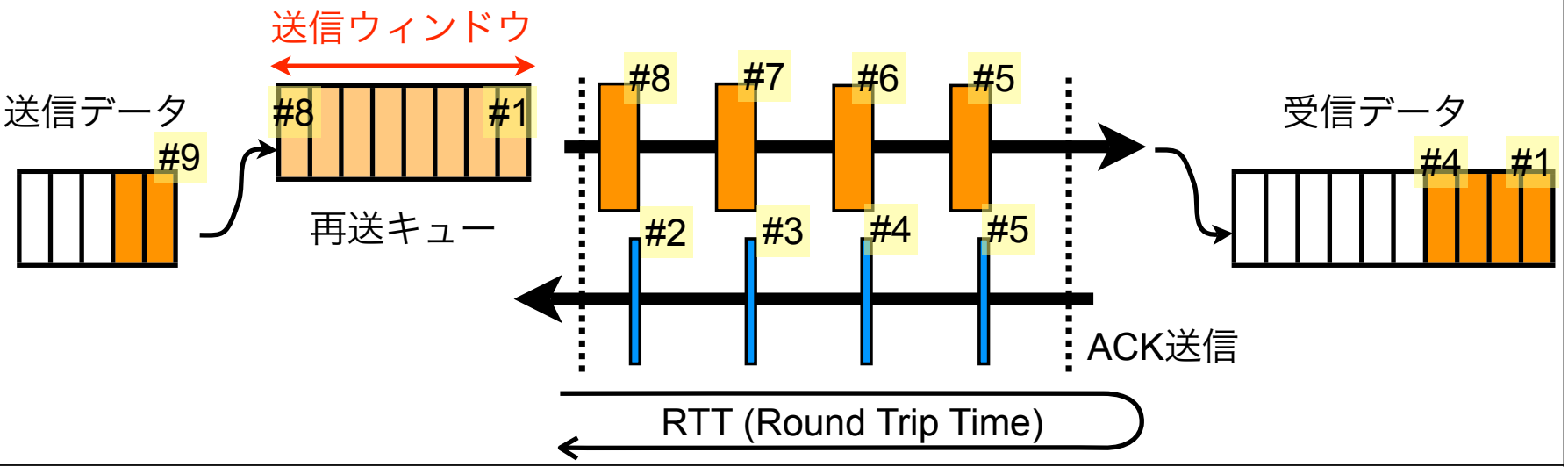
- 再送タイムアウト（RTO）：
  - 一定時間（RTT＋α）でACKが届かなければ再送
- 高速再送：
  - 重複ACKを連続で3回受信すると再送
  - 1 RTTで破棄を検出可能（RTOより「+α」分高速）



TCPデータ処理の流れ（2）

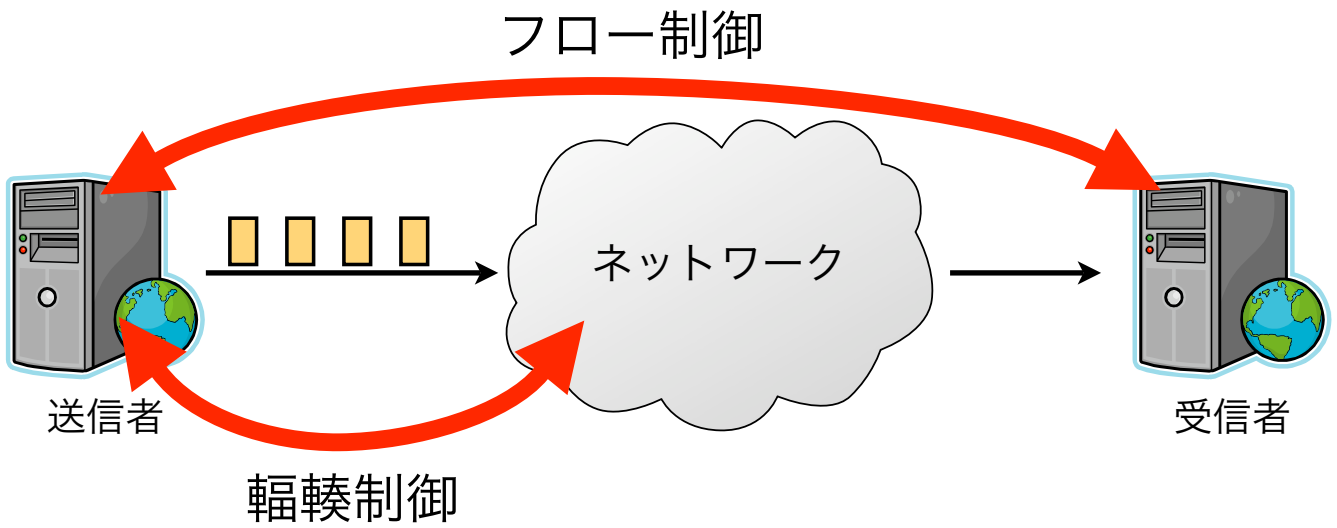
- ウィンドウベースの送信レート制御
- ACKが届くまでに（1 RTT期間内）、送信ウィンドウ分のデータを送信

送信レート＝送信ウィンドウ／RTT



TCPにおける送信レート制御

- フロー制御
  - 「通信相手の処理速度」に合わせた制御
- 輻輳制御
  - 「ネットワークの混雑状態」に合わせた制御



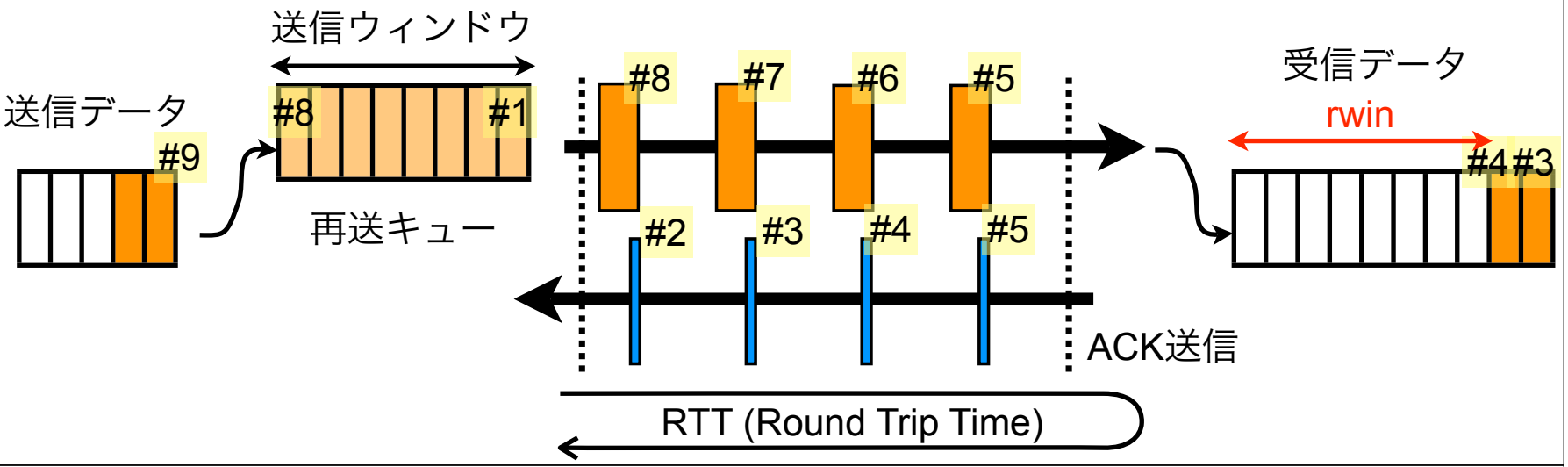
輻輳：過負荷によりネットワーク利用率が低下すること

フロー制御

- 「通信相手の処理速度（受信バッファサイズ）」にあわせて送信レートを制御

➡広告ウィンドウ（rwin）

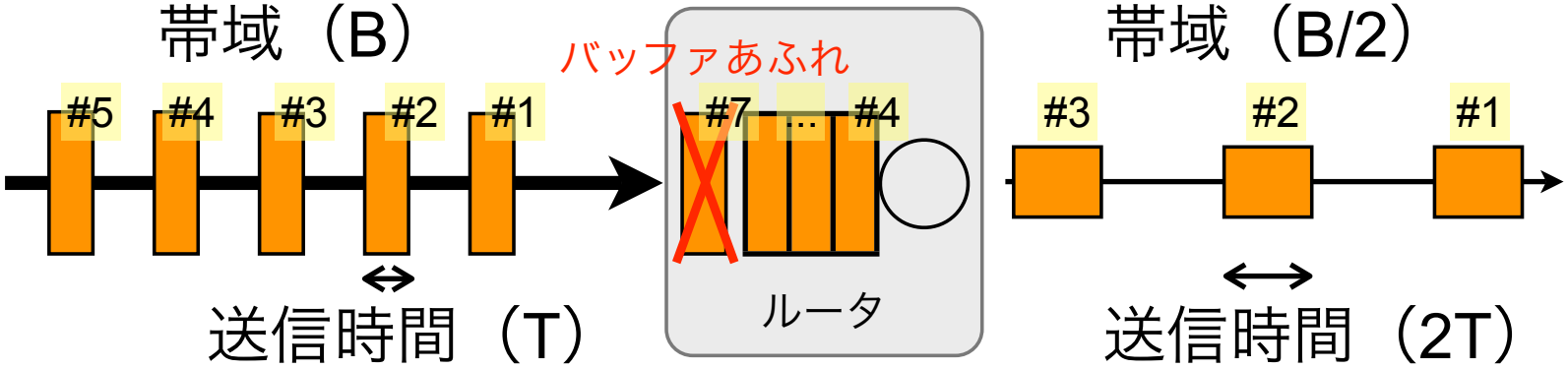
- 受信者がACKと共に空きバッファ量を通知
- 空きバッファ量が減少するとrwinが縮小



輻輳制御の必要性

- 狭帯域回線の存在や複数回線の多重化により、ボトルネックとなる回線の手前のルータで輻輳が発生
- 入出力の速度差はルータのバッファで吸収
- キュー長が一定長を超えると、受信セグメントを破棄（バッファあふれ）

➔ 輻輳崩壊を回避するために輻輳制御が必要

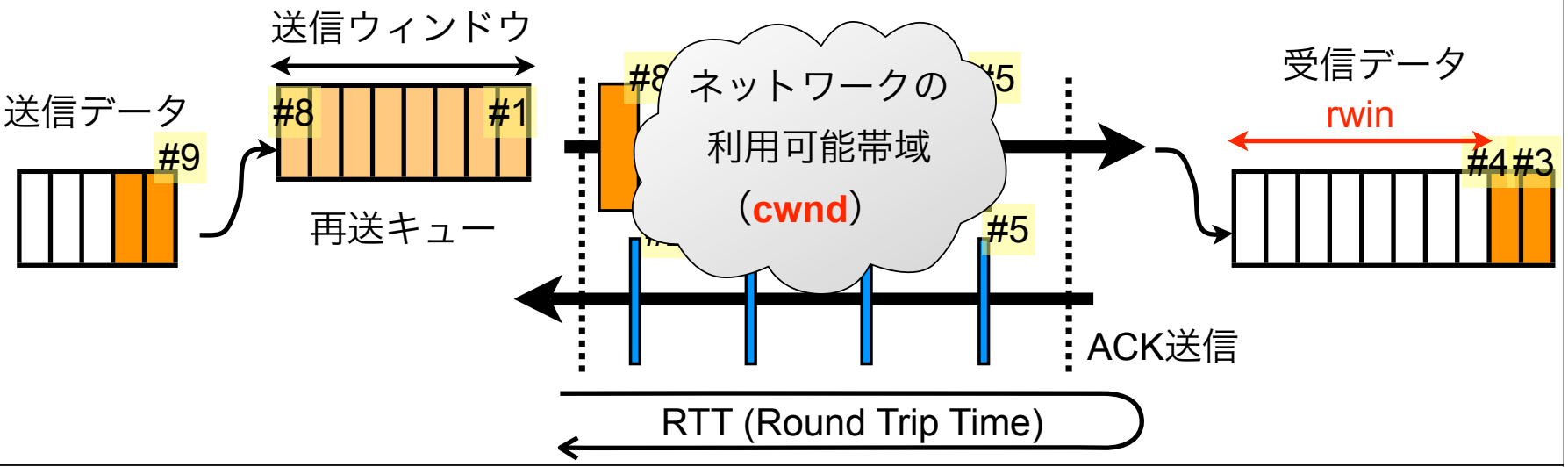


輻輳制御

- 「ネットワークの混雑状況」にあわせて送信レートを制御

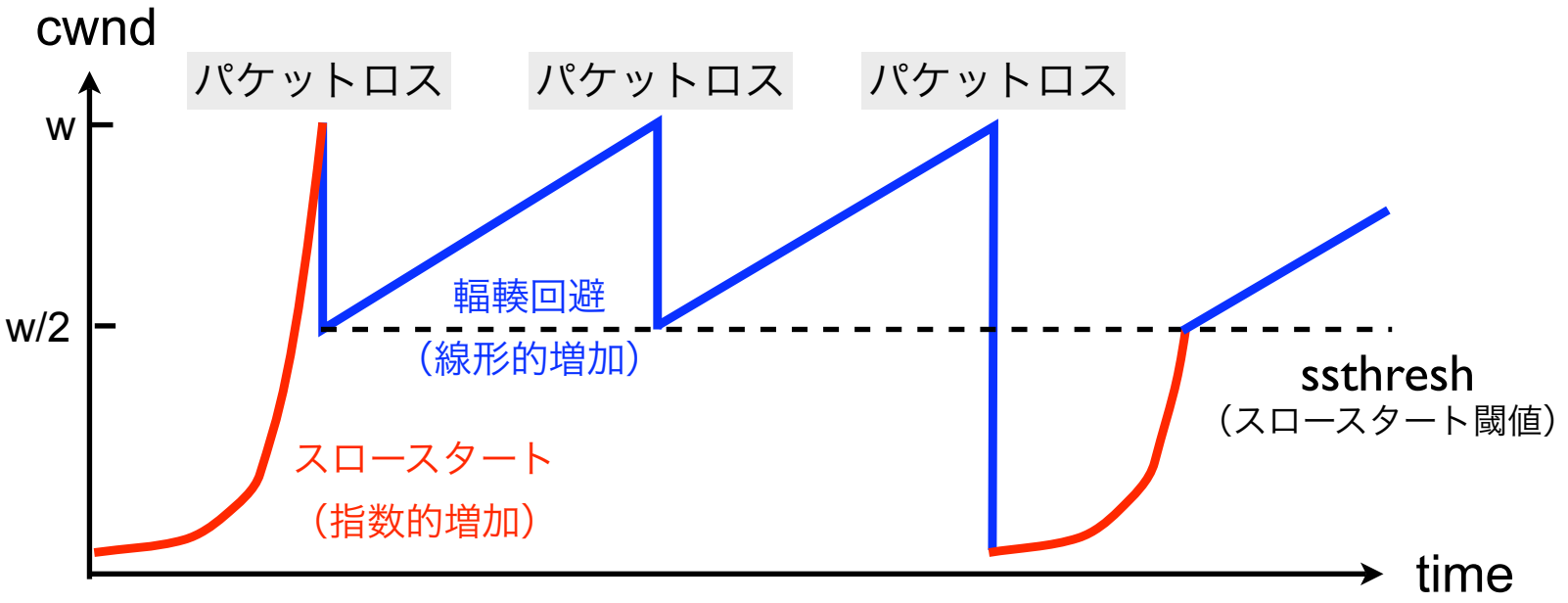
➡輻輳ウィンドウ（cwnd）

- 送信者が帯域見積もり手法を使って推測
- 輻輳発生時にはcwndを小さくして輻輳を抑制



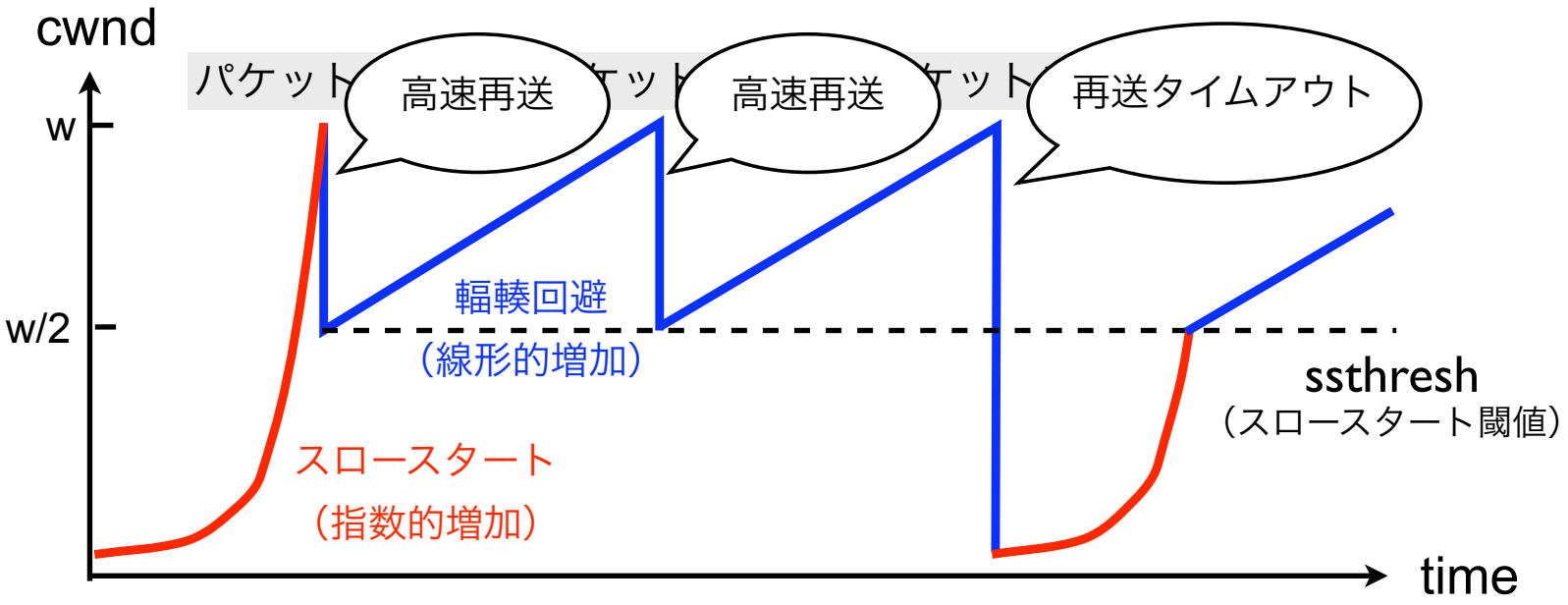
輻輳ウィンドウ制御（１）

- 未知の利用可能帯域をいかに正確かつ迅速に求めるか？
- パケットロスを起こすまで、cwndを拡大
  - 2つのフェーズ：スロースタートで素早くあたりを付け、輻輳回避で徐々に拡大



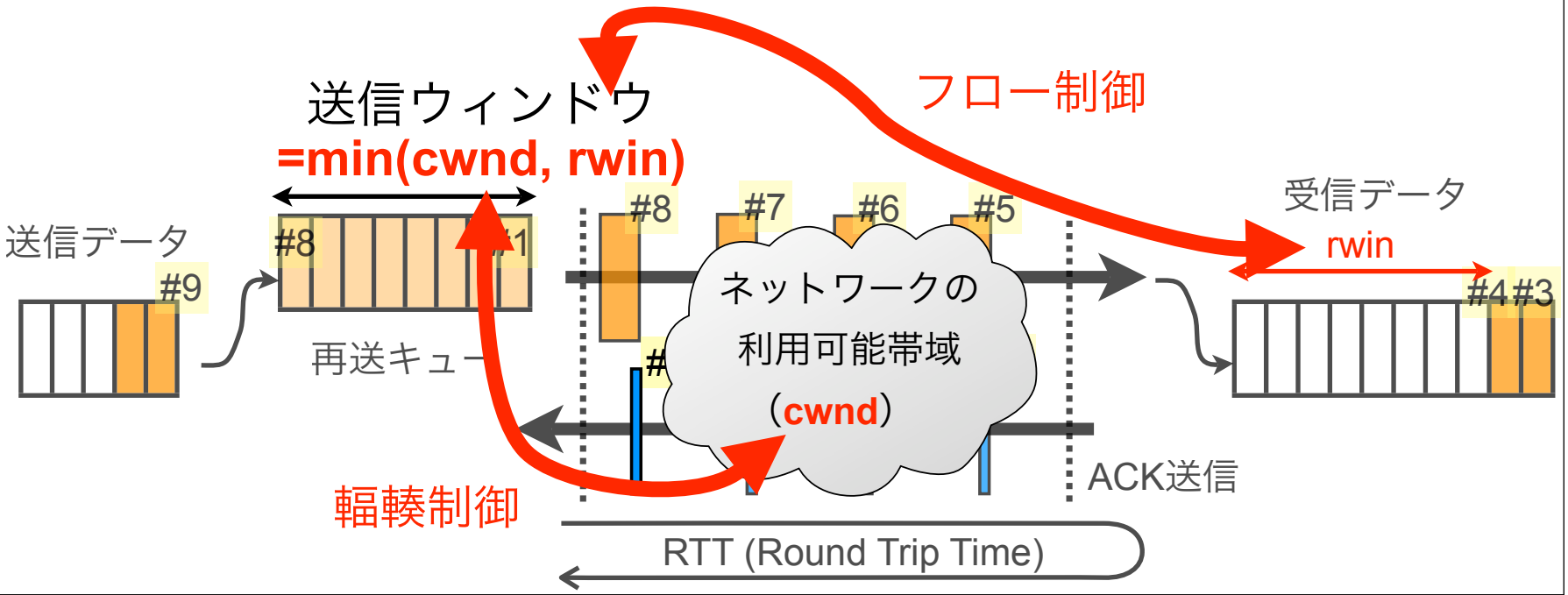
輻輳ウィンドウ制御（２）

- 再送タイムアウト：cwndを最小値まで縮小し、スロースタートから再開
- 高速再送：cwndを半分に縮小
  - 理由：早い段階で輻輳検出→より軽微な輻輳



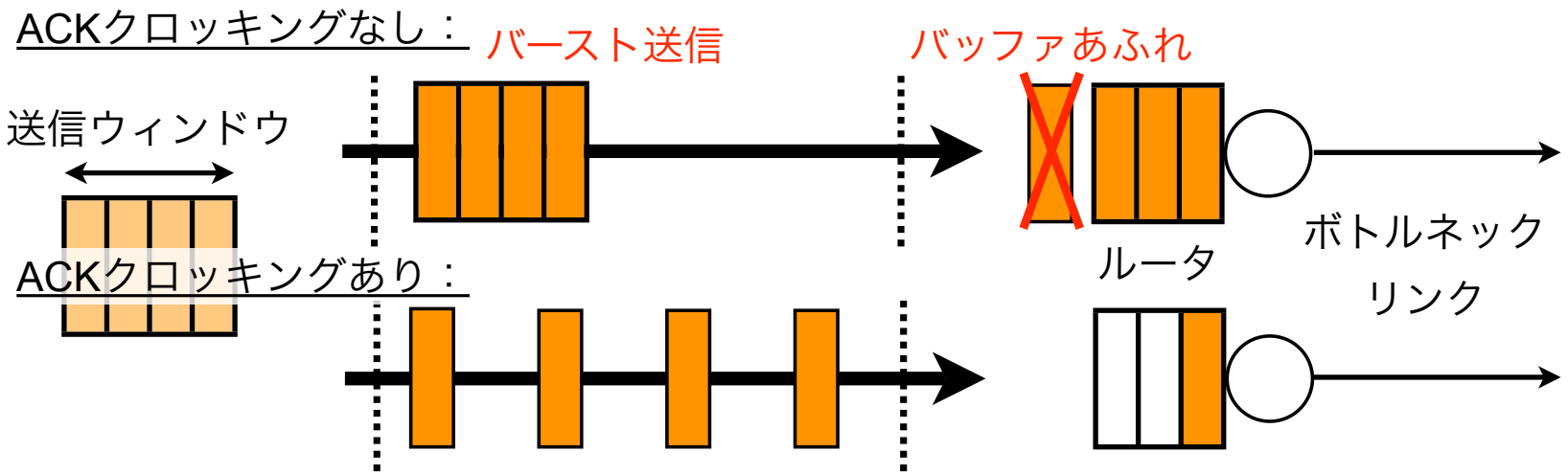
二つのウィンドウ制御

- 広告ウィンドウ（rwin）：ACKのTCPヘッダ
- 輻輳ウィンドウ（cwnd）：内部変数
- 送信ウィンドウとして、両者の小さい方を使用



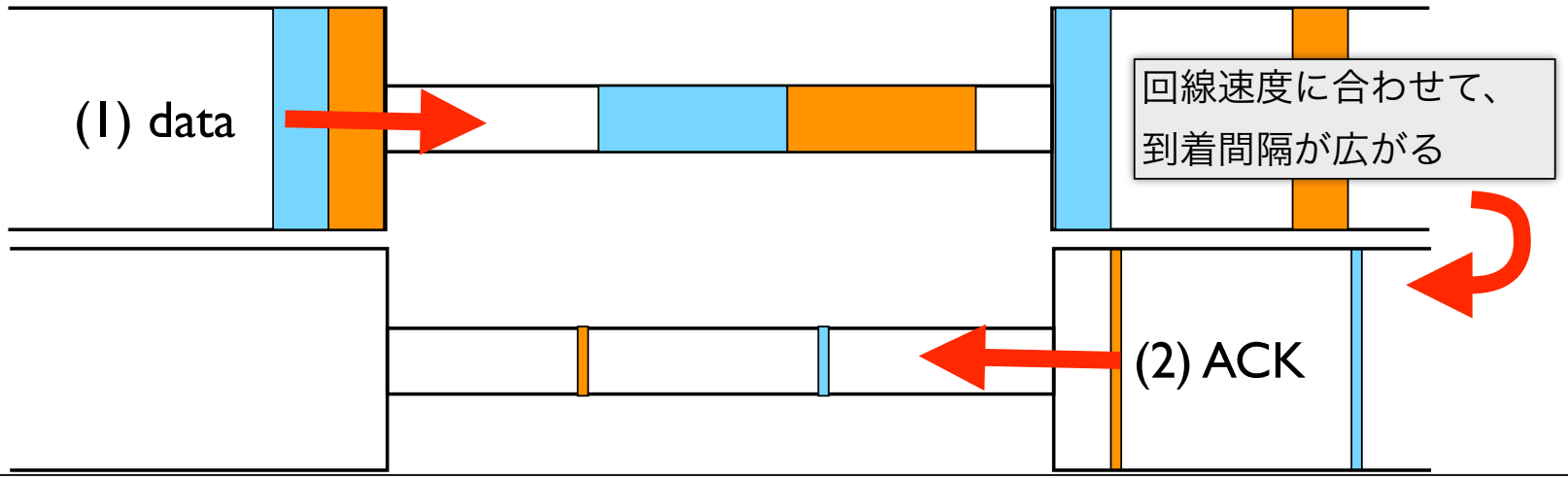
ACKクロッキング（1）

- 送信ウィンドウのセグメントを一度に送信するのではなく、RTT内で均一に送信
  - バースト送信によるバッファあふれの回避



ACKクロッキング（2）

- 送信ウィンドウのセグメントを一度に送信するのではなく、RTT内で均一に送信
  - バースト送信によるバッファあふれの回避
- ACK受信を「クロック」としてセグメントを送信
  - ボトルネックにあわせてセグメント送信間隔が平滑化

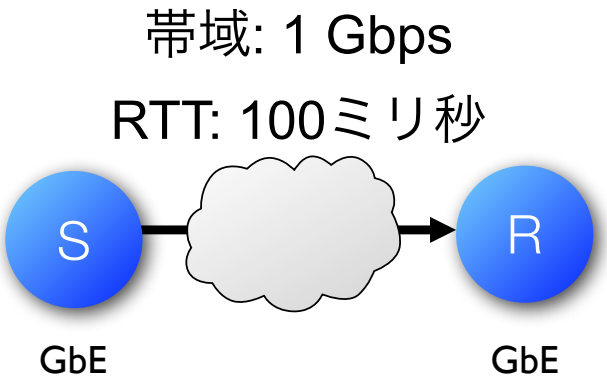


TCP/IPと
長距離大容量データ転送

広域ネットワークのTCP性能

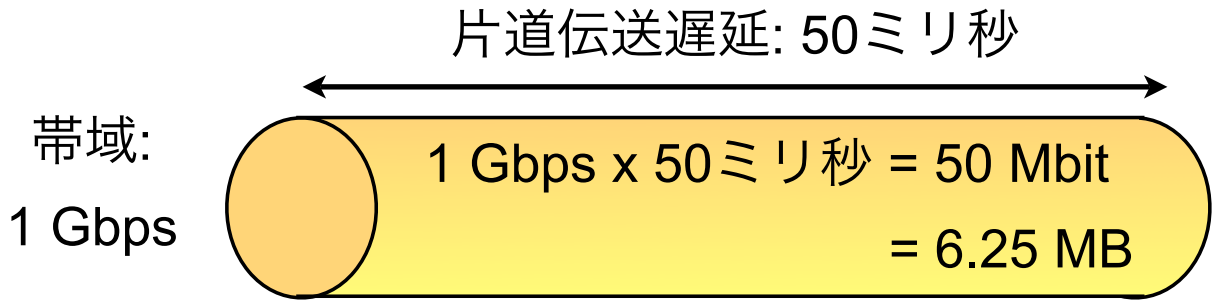
- RTTが100ミリ秒のギガビットネットワークにおいて、輻輳がないにもかかわらず、スループットが**240 Mbps**しか得られなかった

  問題の原因：輻輳ウィンドウサイズ不足



帯域遅延積

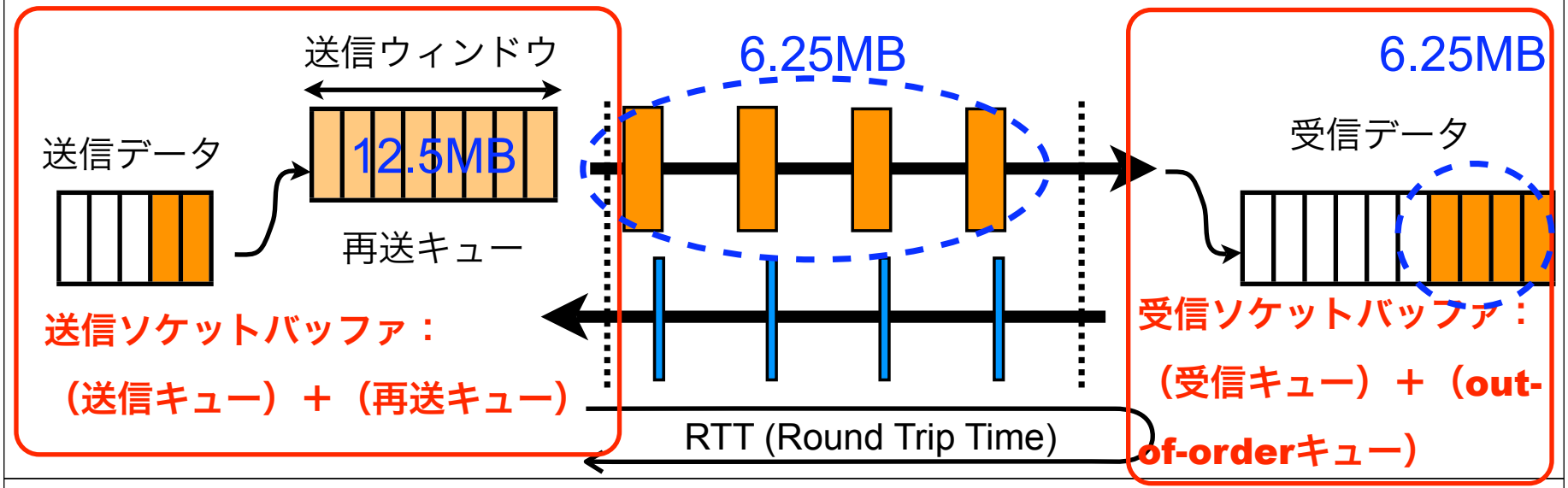
- （帯域）×（遅延）：ネットワークパイプの容量
  - 例）帯域が1Gbps、片道伝送遅延が50ミリ秒（RTTが100ミリ秒）の場合は6.25MB



ネットワーク帯域をすべて使い切るには、送信ウィンドウは帯域遅延積の2倍以上必要である

ソケットバッファチューニング

- 帯域遅延積の２倍（帯域 x RTT）必要
  - 1 Gbps × 50ミリ秒 x 2 = 12.5 MB
  - 送信側と受信側それぞれで設定が必要
- デフォルトの最大ソケットバッファサイズは４MB



TCPパラメータの設定

- システム全体の設定
  - sysctl コマンド
- ソケット（コネクション）ごとの設定
  - ソケットAPI

sysctl パラメータ（net.*の一部）

sysctl	意味	default
net.core.rmem_default	受信バッファサイズのデフォルト値	109568
net.core.rmem_max	受信バッファサイズの最大値	131071
net.core.wmem_default	送信バッファサイズのデフォルト値	109568
net.core.wmem_max	送信バッファサイズの最大値	131071
net.core.netdev_max_backlog	プロセッサごとの受信キューサイズ	1000
net.ipv4.tcp_rmem	TCP受信バッファサイズ [min, default, max]	4096, 87380, 4194304
net.ipv4.tcp_wmem	TCP送信バッファサイズ [min, default, max]	4096, 16384, 4194304
net.ipv4.tcp_mem	利用可能ページ数 [low,pressure, high]	98304, 131072, 196608
net.ipv4.tcp_no_metrics_save	メトリクス保存の無効化	0
net.ipv4.tcp_sack	SACK	1
net.ipv4.tcp_congestion_control	TCP輻輳制御アルゴリズム	cubic

※バッファサイズの初期値は、メモリ搭載量によって変わる（上表は1GBメモリ搭載の場合）

sysctl コマンド

- カーネルの実行時パラメータを変更するコマンド

```
$ sysctl net.core.wmem_max
net.core.wmem_max = 131071

# sysctl -w net.core.wmem_max=1048576
net.core.wmem_max=1048576
```

- （Linux）procfs経由でも同様の操作が可能

```
$ cat /proc/sys/net/core/wmem_max
131071

# echo 1048576 > /proc/sys/net/core/wmem_max
```

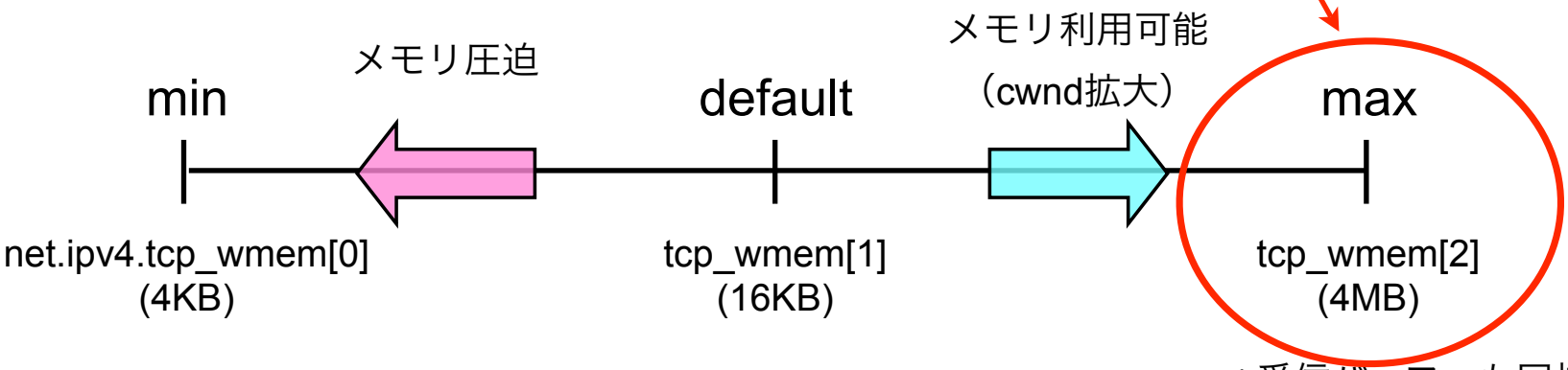
- その他の操作

```
$ sysctl -p      # 設定ファイル（/etc/sysctl.conf）の再読込み
$ sysctl -a | grep tcp   # 現在の設定の確認
```

自動バッファサイズチューニング

- （Linux）メモリ使用状況に応じた、ソケットバッファサイズの動的調整
- 帯域遅延積が大きい場合は、tcp_wmemとtcp_rmemの最大値を大きく設定する必要
  - デフォルト設定ではcwndが4MBで頭打ち

TCP（送信側）の場合：



※受信バッファも同様

ソケットAPI

- コネクション単位の設定
- setsockopt(2)で取得、getsockopt(2)で変更

```
int ssiz; /* send buffer size */
cc = setsockopt(so, SOL_SOCKET, SO_SNDBUF, &ssiz, sizeof(ssiz));

socklen_t optlen = sizeof(csiz);
cc = getsockopt(so, SOL_SOCKET, SO_SNDBUF, &csiz, &optlen);
```

レベル	オプション	意味
SOL_SOCKET	SO_SNDBUF	送信バッファサイズ
SOL_SOCKET	SO_RCVBUF	受信バッファサイズ

- SO_SNDBUFで明示的にバッファサイズが設定された場合、自動チューニングは無効化
- SO_SNDBUFの上限はnet.core.wmem_max

setsockopt関数の注意点

- listen、connect関数よりも前に実行する必要
- （Linux）引数値の2倍に自動的に設定
  - man tcp(7)によると、「TCP はこの余分な空間を、管理目的やカーネル内部の構造体に用い」るため

Iperfベンチマークの警告メッセージ

```
$ iperf -w 1m -c 192.168.0.2
------------------------------------------------------------
Client connecting to 192.168.0.2, TCP port 5001
TCP window size: 2.00 MByte (WARNING: requested 1.00 MByte)
------------------------------------------------------------
```

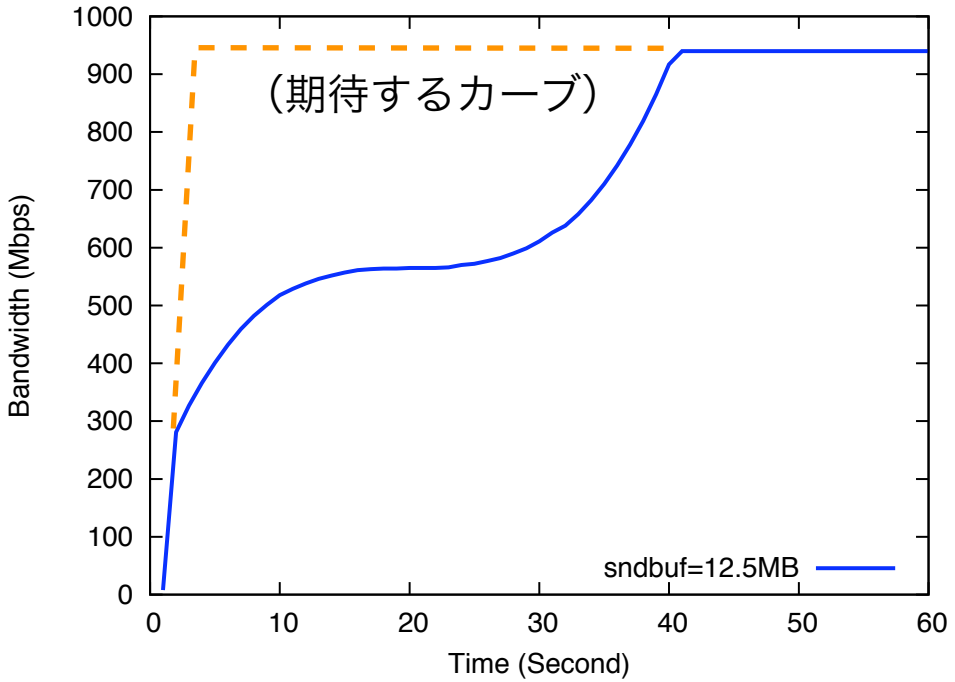
ソケットバッファ設定後

- ソケットバッファを適切に設定しても、性能の立ち上がりが緩慢
- 輻輳が起きないはずなのに、輻輳回避の挙動？

帯域: 1 Gbps

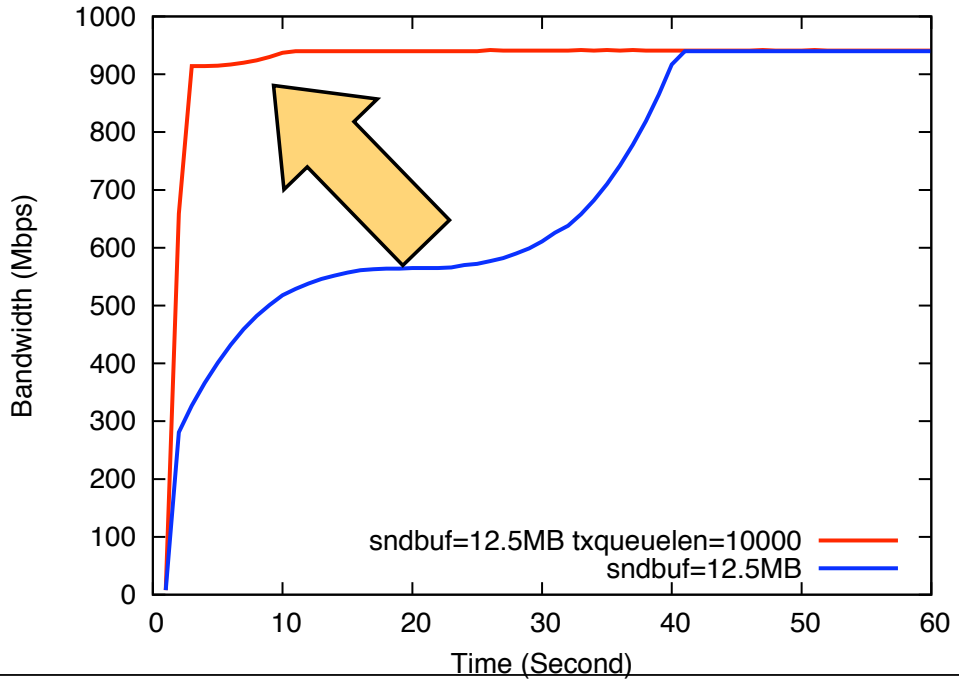
RTT: 100ミリ秒

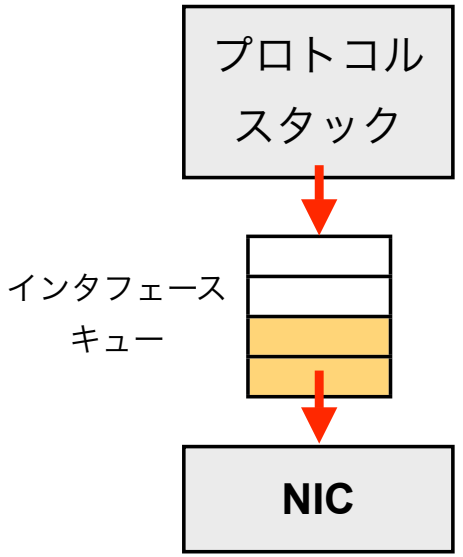
TCP: CUBIC



インタフェースキューあふれ

- インタフェースキュー：QoS制御などに使用される、NICごとの送信キュー
- インタフェースキューあふれは輻輳と判定
- 帯域遅延積が大きい場合、キュー長の拡大が必要





NIC: Network Interface Card

インタフェースキュー長

- インタフェースキューあふれはtcコマンドで確認

```
$ tc -s qdisc show dev eth0
qdisc pfifo_fast 0: root bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 41825726768 bytes 1383455 pkt (dropped 72, overlimits 0 requeues 25164)
 rate 0bit 0pps backlog 0b 0p requeues 25164
```

- ifconfigを用いて、キュー長を設定可能

```
$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:22:64:10:cc:9d
          inet addr:192.168.0.1  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::222:64ff:fe10:cc9d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:795673 errors:0 dropped:0 overruns:0 frame:0
          TX packets:163559 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:72212519 (72.2 MB)  TX bytes:29126498 (29.1 MB)
          Interrupt:17
$ ifconfig eth0 txqueuelen 10000
```

netstat -s

- プロトコルごとの統計情報の取得
- 標準的なLinuxには同梱（net-toolsパッケージ）

```
$ netstat -s
[snip]
Tcp:
    49 active connections openings
    4925 passive connection openings
    56 failed connection attempts
    12 connection resets received
    1 connections established
    226842 segments received
    161944 segments send out
    210 segments retransmited
    1 bad segments received.
    575 resets sent
[snip]
```

セグメント送受信数

再送数

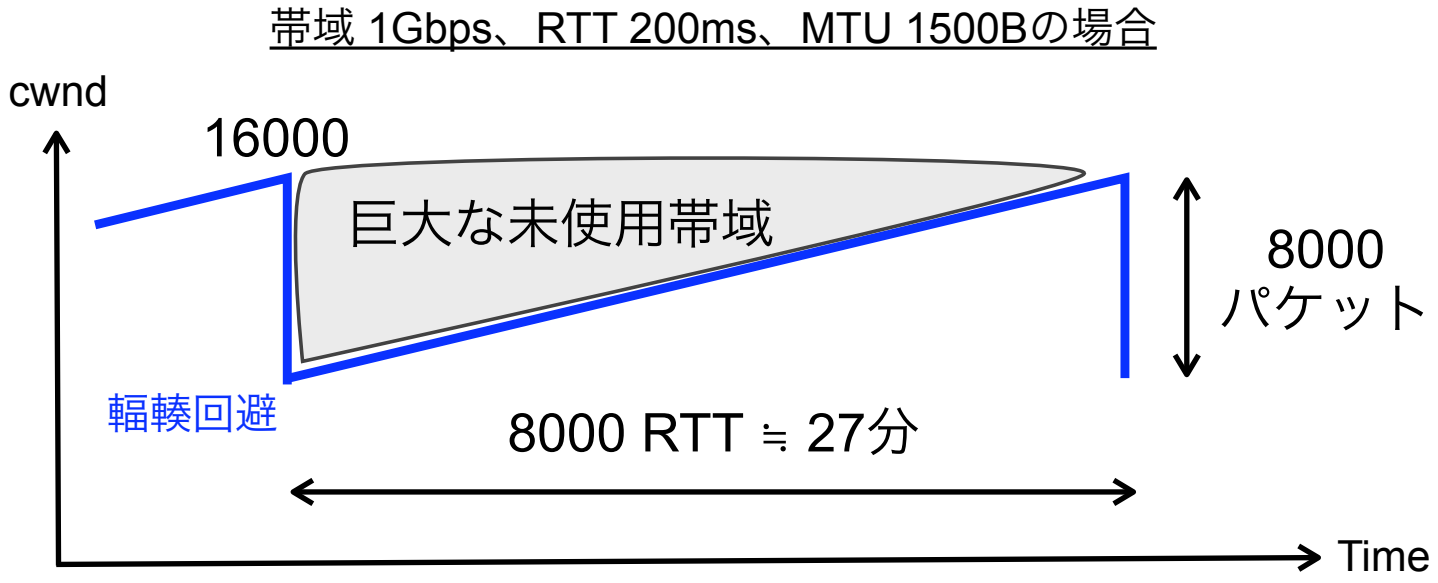
```
TcpExt:
    55 resets received for embryonic SYN_RECV sockets
    108 TCP sockets finished time wait in fast timer
    10969 delayed acks sent
    Quick ack mode was activated 67 times
    7961 packets directly queued to recvmsg prequeue.
    43 bytes directly received in process context from prequeue
    69443 packet headers predicted
    11162 acknowledgments not containing data payload received
    110524 predicted acknowledgments
    4 times recovered from packet loss by selective acknowledgements
    37 congestion windows recovered without slow start after partial ack
    4 TCP data loss events
    6 timeouts after SACK recovery
    1 timeouts in loss state
    5 fast retransmits
    5 retransmits in slow start
    167 other TCP timeouts
[snip]
```

輻輳検出数

高速TCP輻輳制御アルゴリズム

広帯域ネットワークでの問題点

- 標準TCPでは、帯域遅延積に対してcwndが小さすぎて、ネットワーク利用効率が低下
  - 輻輳回避でのcwndの拡大が緩慢
  - 当初の設計時の想定は、せいぜい数十パケット程度



対応策

- 複数コネクションの利用
  - PSockets、GridFTP、(BitTrrent Swarming)
- 輻輳制御アルゴリズムの改良
  - 輻輳回避
    - 帯域遅延積小：既存TCPと公平に（TCP Friendly）
    - 帯域遅延積大：アグレッシブに（スケーラビリティ）
  - スロースタート
    - バースト送信への対応

公平性

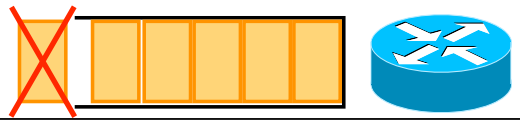
- 公平：コネクション間の送信レートが等しいこと
- 効率の追求だけではなく公平性を保つことも重要

- プロトコル間公平性（Inter-protocol fairness）
  - TCP Friendliness：標準TCPとの公平性
- プロトコル内公平性（Intra-protocol fairness）
  - RTT公平性：RTTが異なるコネクション間の公平性
    - RenoはRTTが短い方が有利

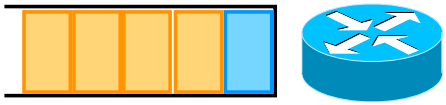
TCP輻輳制御アルゴリズム

アルゴリズム	輻輳検出方法	備考
NewReno	ロスベース（パケット破棄）	RFC 2852
HighSpeed TCP		RFC 3649
Scalable TCP		
BIC		
CUBIC		Linuxのデフォルト
H-TCP		
Vegas	遅延ベース（キューイング遅延の増加）	
Westwood+		
FAST		
Illinois	ハイブリッド	
YeAH		
Compound TCP		Windows Vistaのデフォルト

ロスベース：
パケット破棄≒輻輳

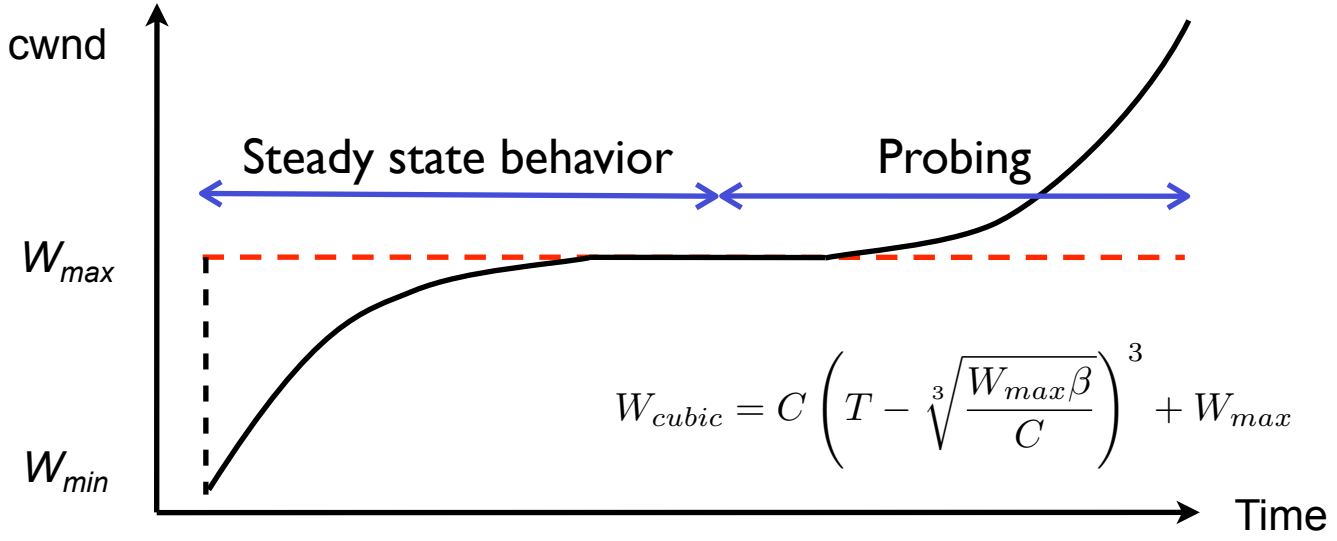


遅延ベース：
キューイング遅延≒輻輳
RTT2 > RTT1



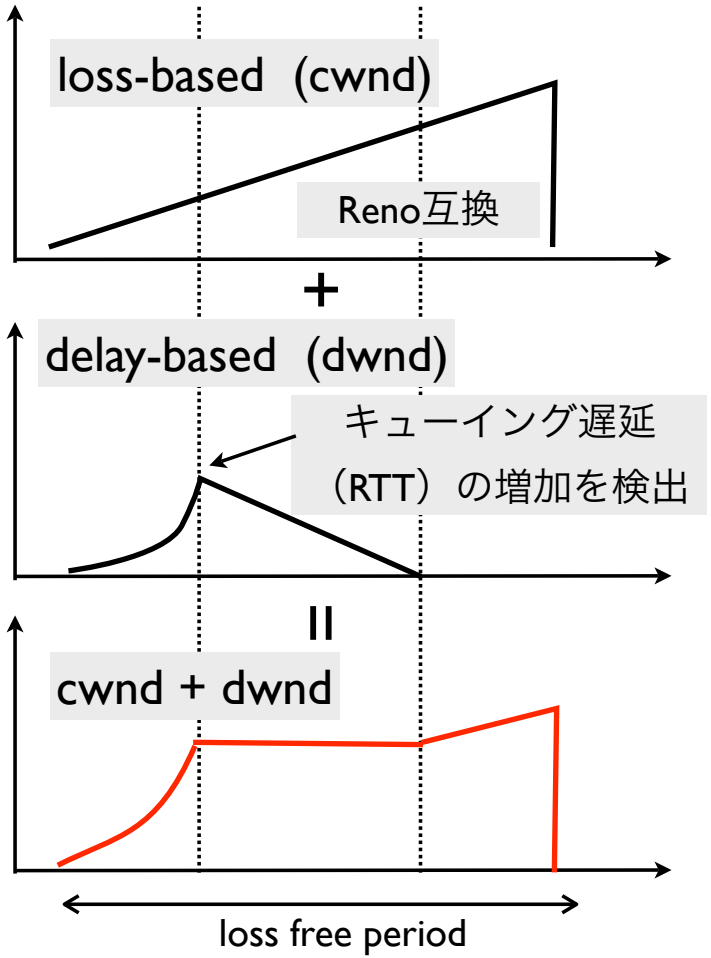
CUBIC

- スケーラビリティと通信の安定性の向上：パケットロス時のcwnd（W_{max}）まで素早く回復後、水平状態をしばらく維持し、さらに上限を探査
- RTT公平性の向上：cubic関数を基に、前回パケットロス時からの経過時間によりcwndを計算



Compound TCP

- Windows Vistaの標準
- ロスベースと遅延ベースのハイブリッド型アルゴリズム
  - 遅延小：アグレッシブに
  - 遅延大：Reno互換
- 公平性に優れるが、他のプロトコルの影響を受けやすい
  - dwndが0になりRenoに後退



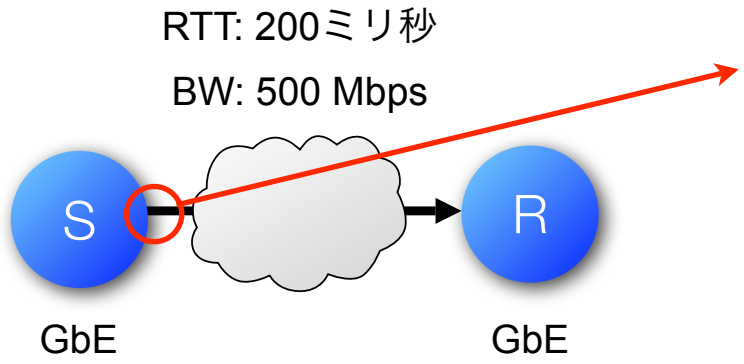
「鈍感力」TCP

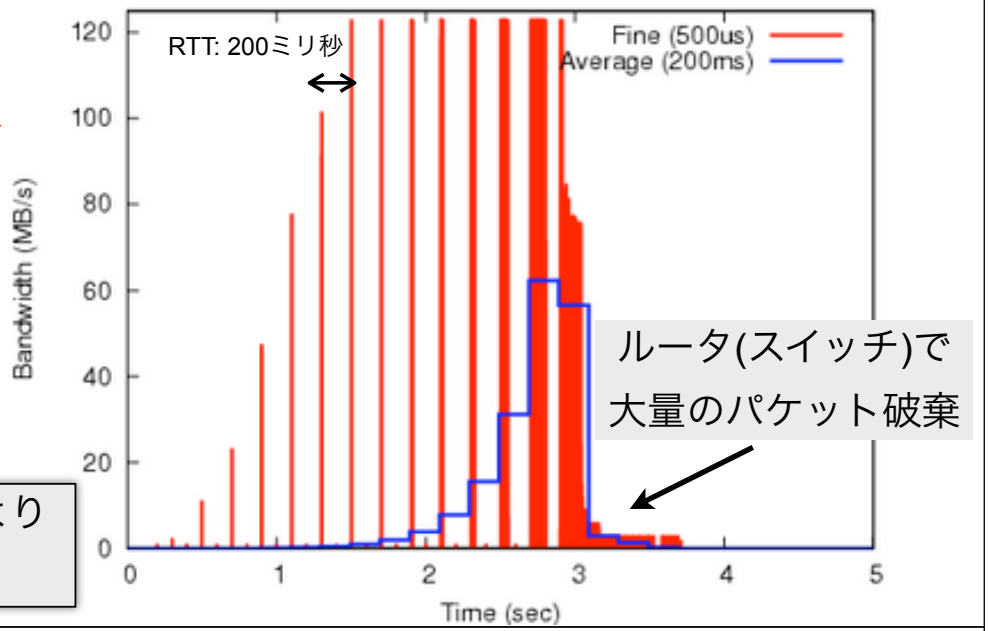
- 輻輳制御しないTCP
  - 常に最大送信レートで通信
  - 輻輳検出に関係なく輻輳ウィンドウを最大値で固定
    - 帯域遅延積を明示的に設定することも可能
  - 広告ウィンドウも最大値で固定
    - 自動バッファサイズチューニングの影響を回避
- 公平性無視で、閉じたネットワークでの利用を想定
- （実はHPCユーザが一番望む形のTCP？）

http://code.google.com/p/pspacer/wiki/DonkanTcp

スロースタートの問題

- RTTごとに送信データ量は倍増
- 帯域遅延積が大きくなると、スロースタートは「スロー」ではなくアグレッシブ
- バースト送信により、大量のパケット破棄が発生
  →帯域利用効率の著しい低下





平均では500Mbps以下だが、バースト送信により
ピークは1Gbpsに達し、バッファがあふれる

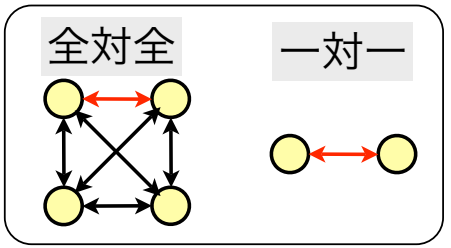
スロースタートの改良

- Limited slow start [RFC 3742]
  - max_ssthresh < cwnd <= ssthreshの場合、RTTごとのcwnd増加をmax_ssthresh/2に抑制
- Swift Start
  - Packet-pair probingによる帯域見積もりに基づいて、ACKクロッキングするまでペーシング
- HyStart (Hybrid slow start)
  - 遅延ベースのアイデアを導入し、RTTの増加が閾値を超えたらスロースタートを終了
  - （Linux 2.6.29）CUBIC使用時にデフォルト有効

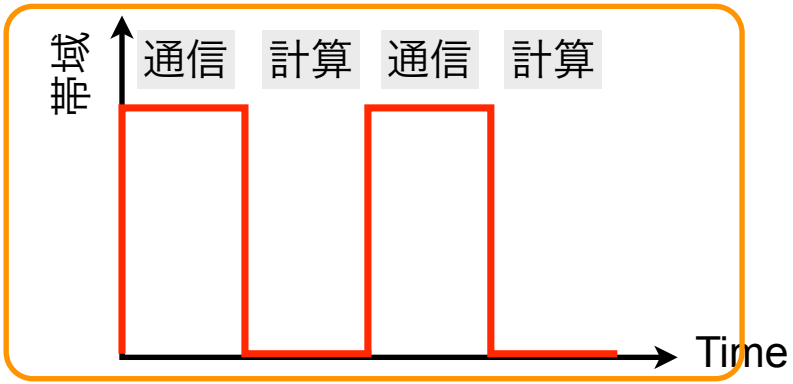
TCP/IPとMPI並列通信

MPI通信におけるTCP性能の問題

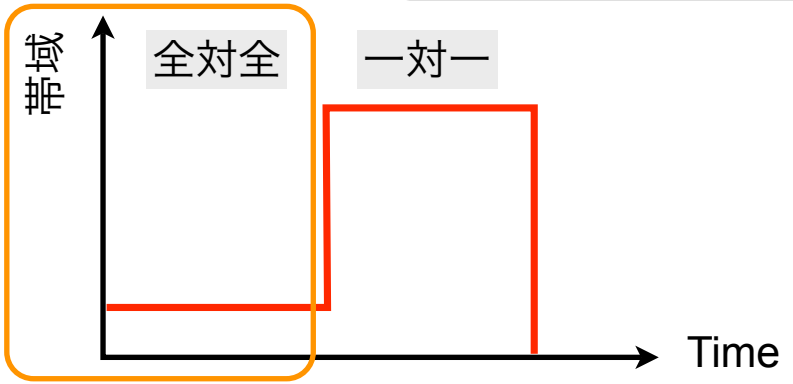
- MPIの典型的アプリは、TCPが仮定するストリーム型のデータ転送（e.g.ファイル転送）とは異なる通信負荷
  - 計算フェーズと通信フェーズの繰り返し
  - 突然、通信パターンが変化
    - 一対一通信と集団通信



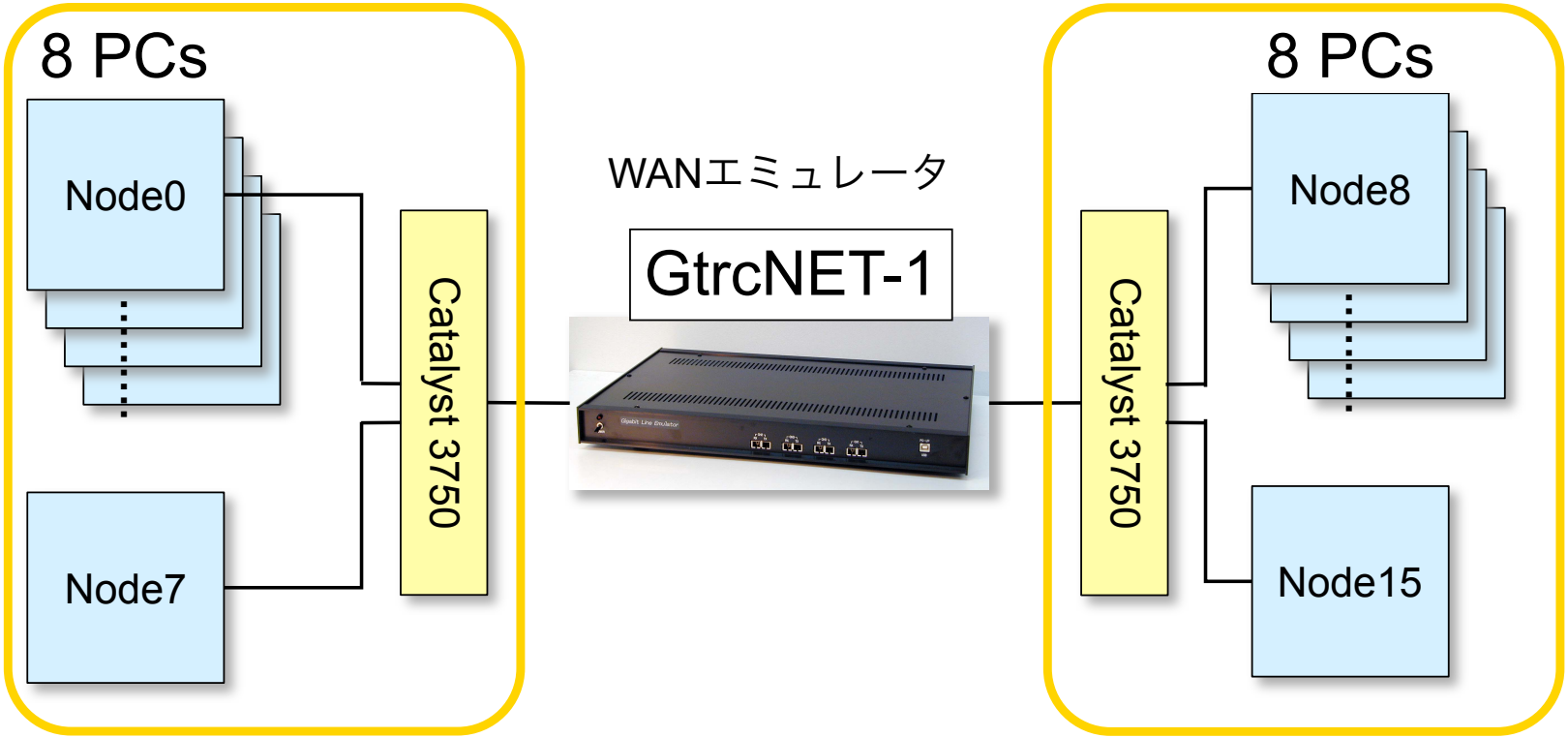
実験１



実験２

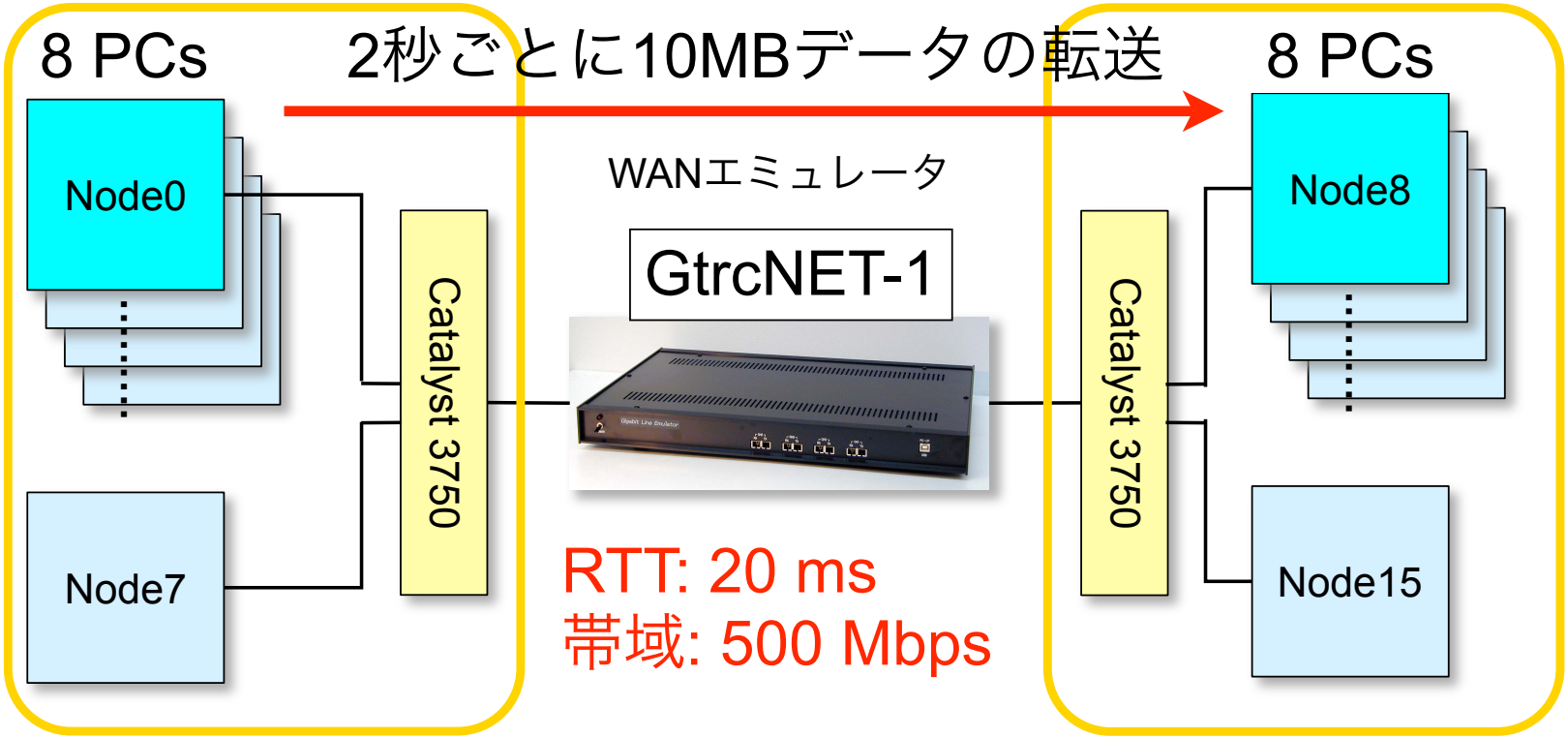


実験に用いたPCクラスタ環境



- CPU: Pentium4/2.4GHz
- Memory: DDR400 512MB
- NIC: Intel PRO/1000 (82547EI)
- OS: Linux-2.6.9-1.6 (Fedora Core 2)
- Socket buffer: 20MB

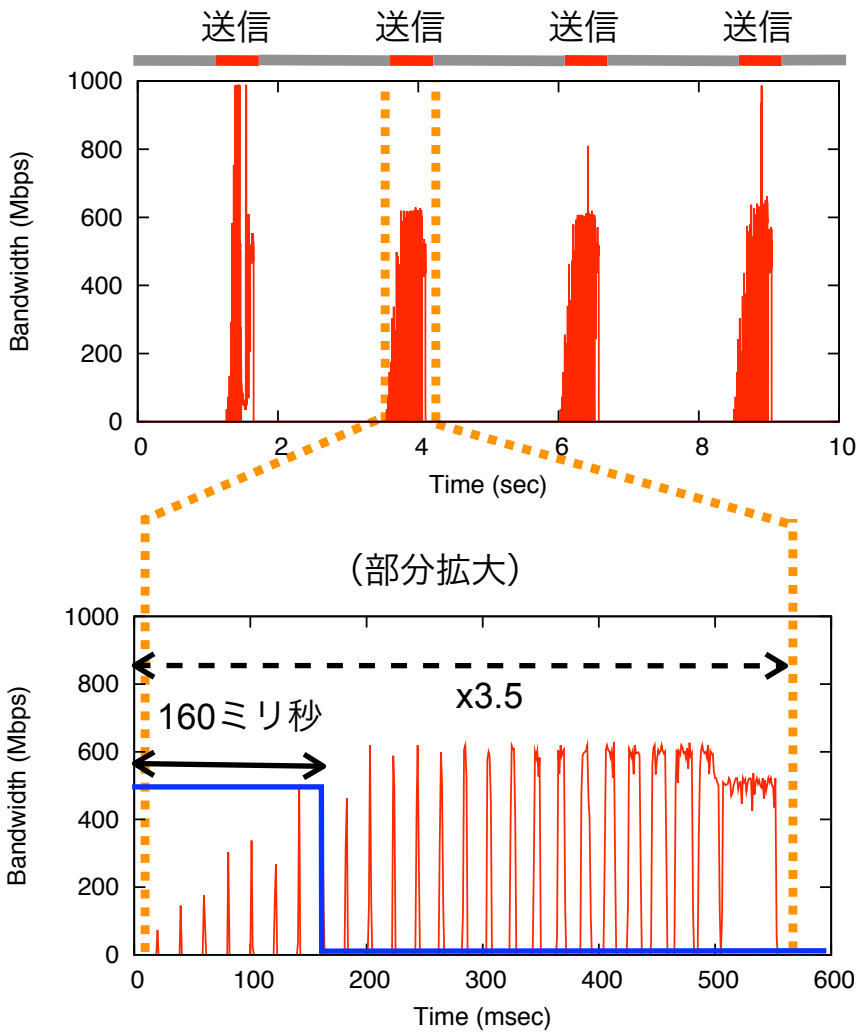
実験１：間欠通信性能



各クラスタから
1ノードずつを使用

- CPU: Pentium4/2.4GHz
- Memory: DDR400 512MB
- NIC: Intel PRO/1000 (82547EI)
- OS: Linux-2.6.9-1.6 (Fedora Core 2)
- Socket buffer: 20MB

間欠通信時のトラフィック

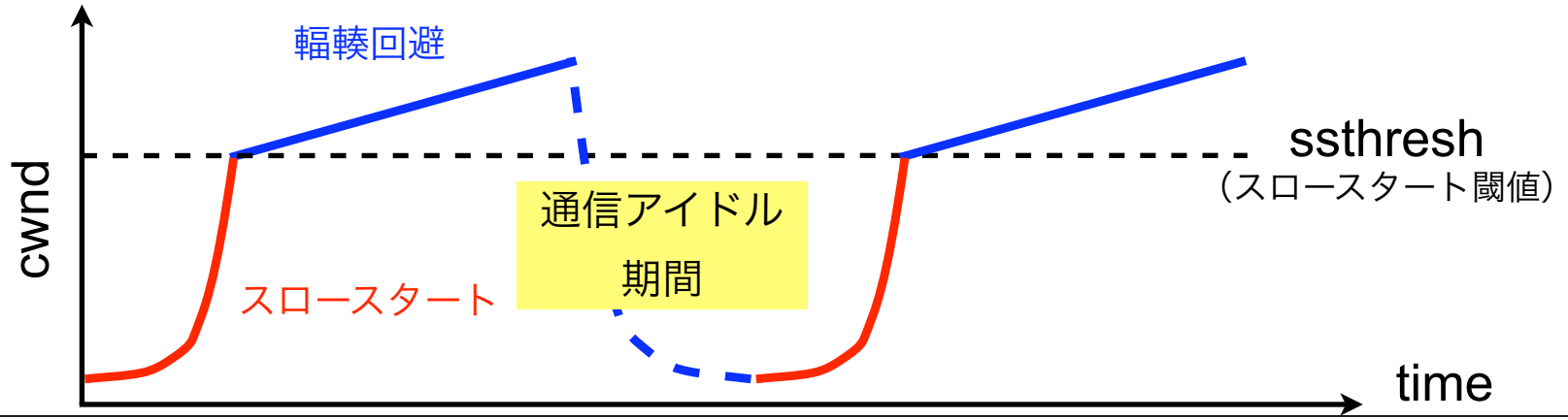


2秒ごとに10MBデータの連続送信の繰り返し

- 理想的は160ミリ秒（10MB／500Mbps）で送信完了
- 通信アイドル後は、毎回スロースタートから再開するため、非効率

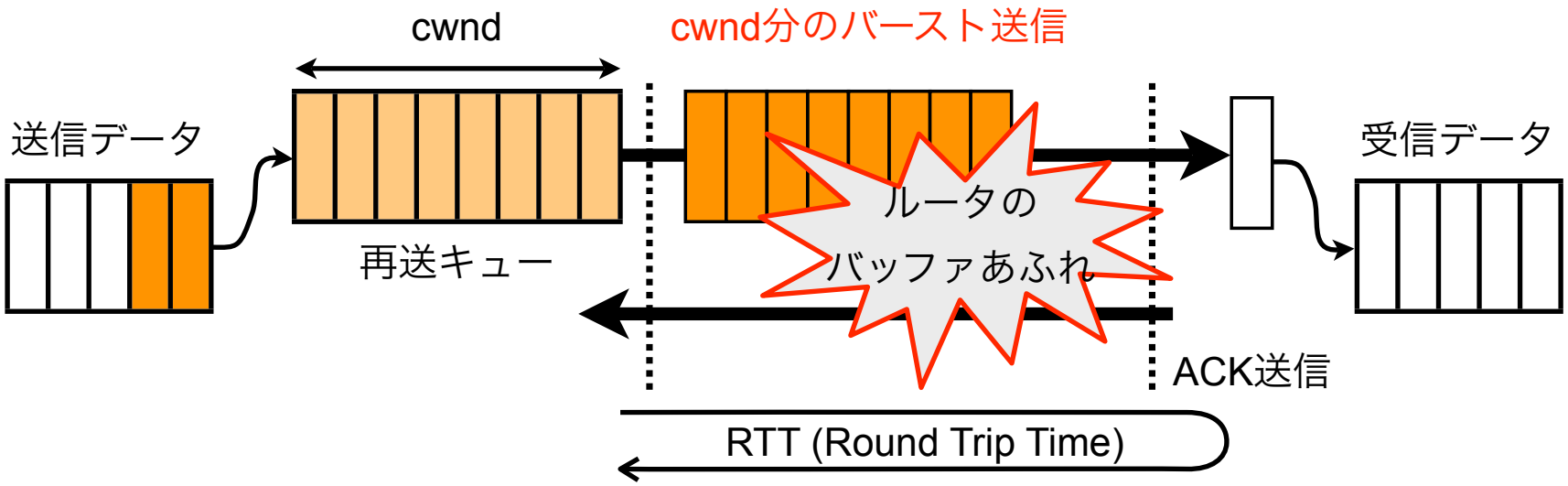
通信アイドル後のスロースタート（１）

- 一定時間通信を行わないと、スロースタートから再開
  - （Linux）RTO時間経過ごとにcwndが半減 [RFC 2861]
- MPIアプリケーションでは高頻度に発生
  - HTTP/1.1のパイプラインでも同様の問題が発生
- （kernel 2.6.18以降）通信アイドル後のスロースタートを省略可能　net.ipv4. tcp_slow_start_after_idle=0



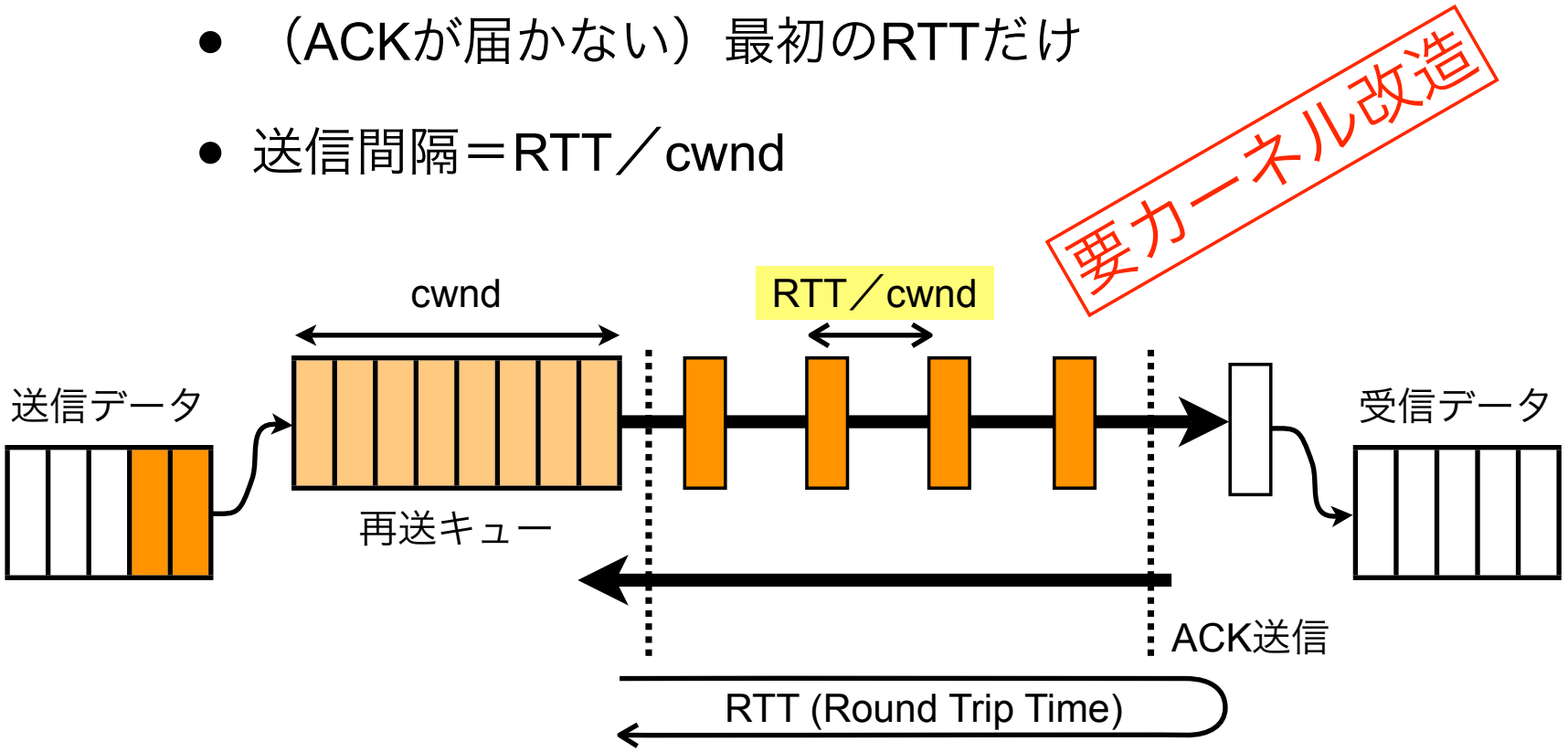
通信アイドル後のスロースタート（2）

- 通信アイドル後にスロースタートする理由：
  - ネットワークの状況が変化する可能性
  - cwnd分のセグメントを一気に送ると、ACKクロッキングが働かずバースト送信が発生→輻輳の原因



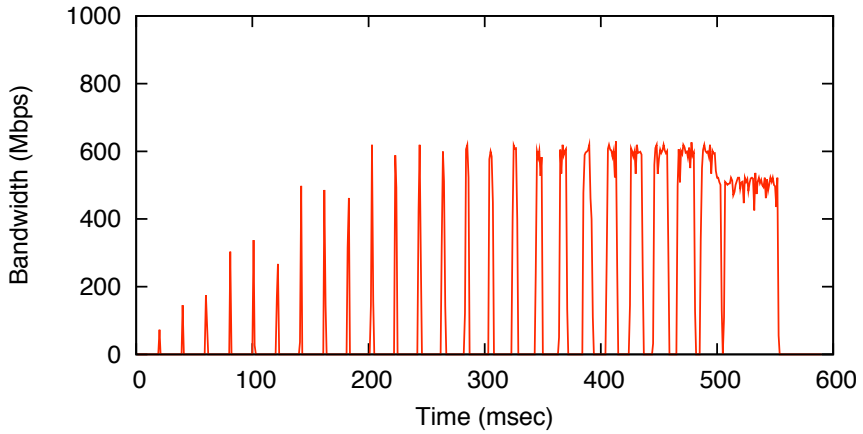
通信再開時ペーシング

- ACKクロッキングの代わりに、高精度タイマによるクロックを基に一定のペースでパケットを送信
  - （ACKが届かない）最初のRTTだけ
  - 送信間隔＝RTT／cwnd

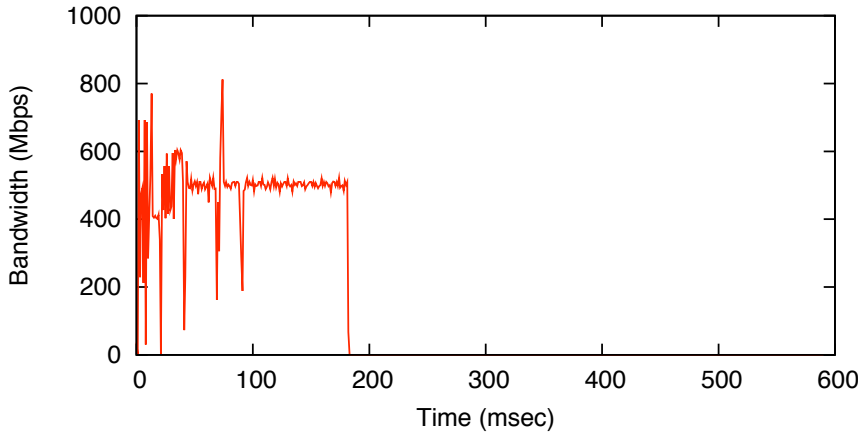


通信再開時ペーシングの効果

通信再開時ペーシングなし



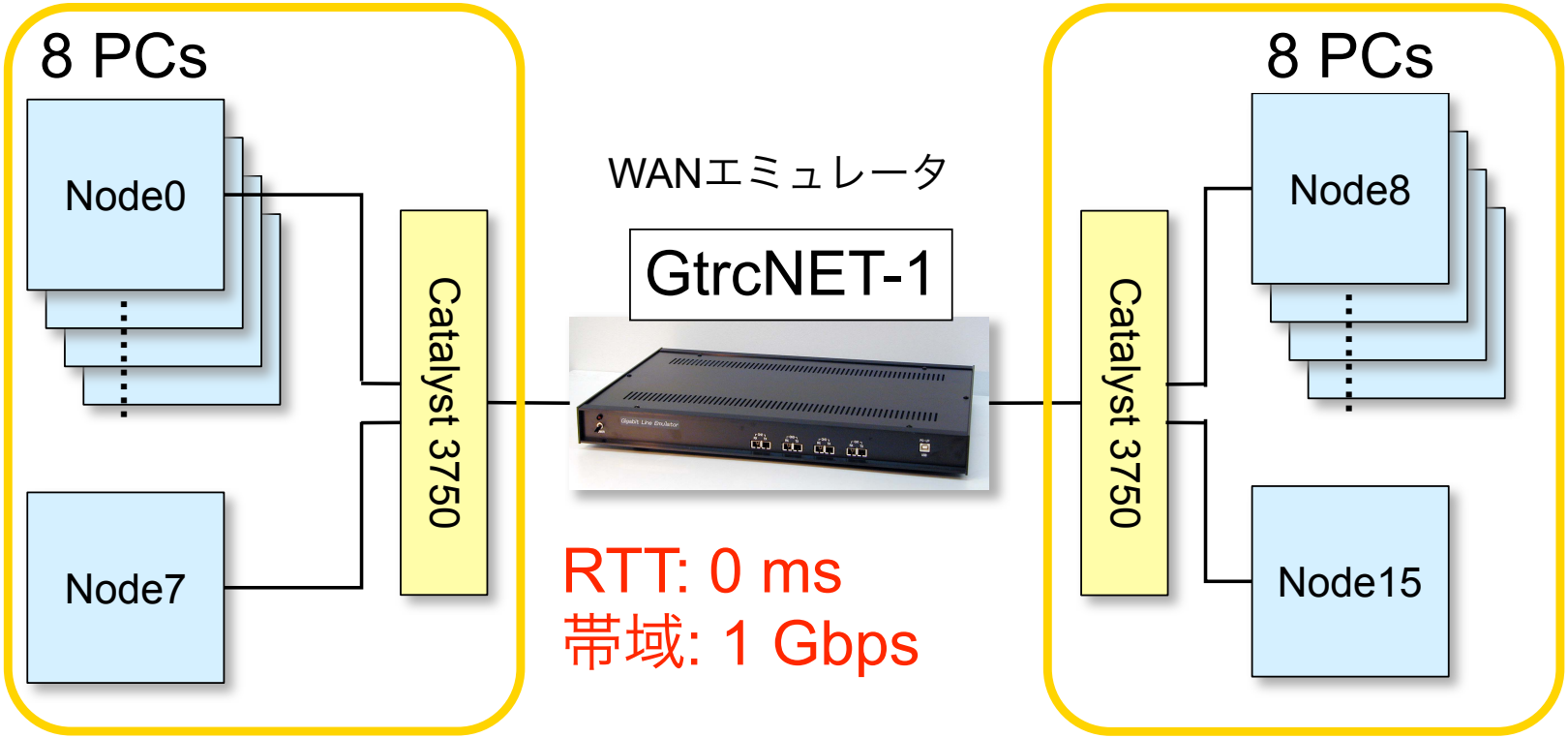
通信再開時ペーシングあり



2秒ごとに10MBデータの連続送信の繰り返し

- スロースタートを省略し、アイドル前のcwndから再開
- さらに、バースト送信回避のため、通信再開時ペーシングを使用

実験２：全対全通信性能

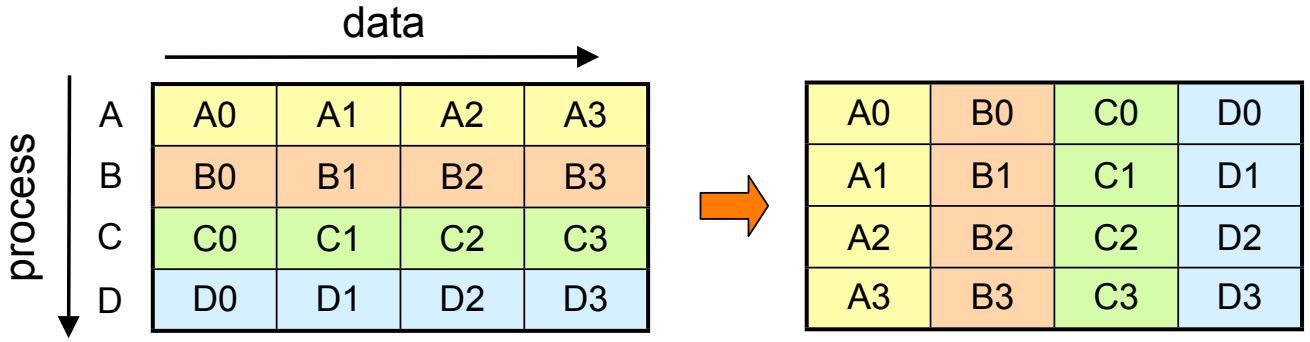


全16ノードを使用

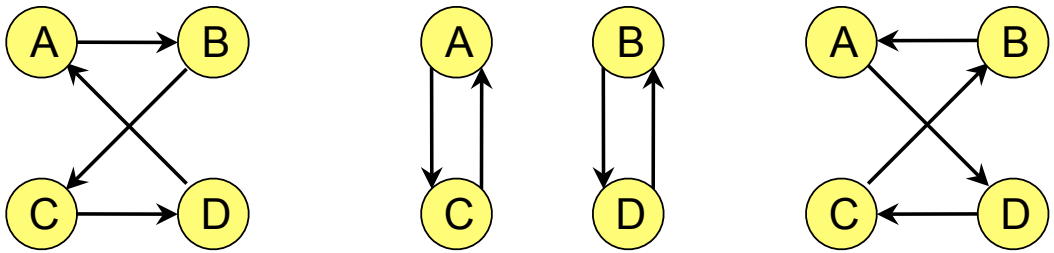
- CPU: Pentium4/2.4GHz
- Memory: DDR400 512MB
- NIC: Intel PRO/1000 (82547EI)
- OS: Linux-2.6.9-1.6 (Fedora Core 2)
- Socket buffer: 20MB

全対全（All-to-all）通信

- すべてのプロセス間でデータを交換する集団通信
- MPIメッセージは複数セグメントに分割して送信
- １プロセス１ノードを仮定



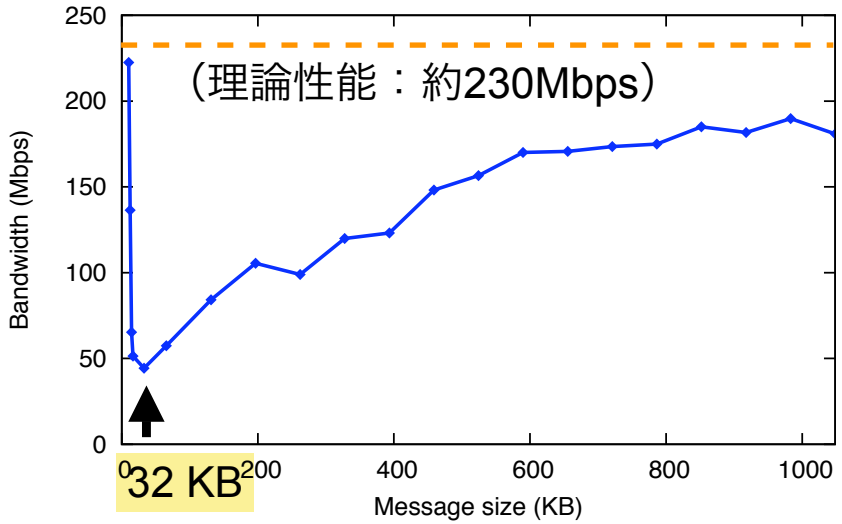
１イテレーション内の通信パターン



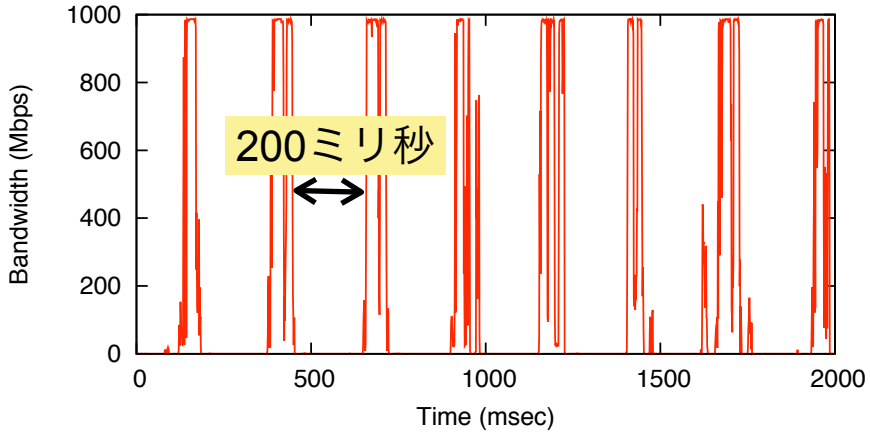
各ノードは全データを受信完了すると、次のイテレーションへ遷移

全対全通信時の輻輳

1ノード当たりの全対全送信帯域



メッセージサイズ32KB時のトラフィック



- 全対全通信の繰り返し
- スイッチでパケット喪失
- 受信待ちのため、後続パケットの送信が中断

- 連続した通信呼出しだが、トラフィックは間欠的
- 中断時間は約200ミリ秒
- 再送タイムアウトが発生

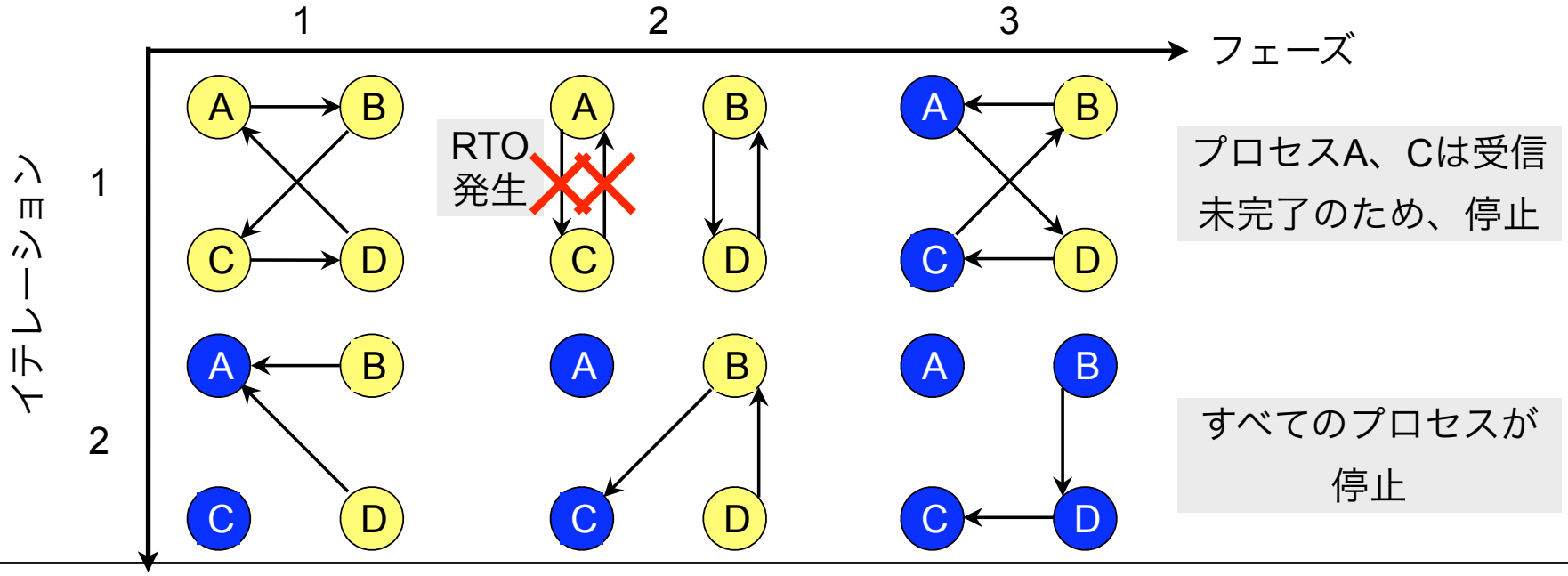
RTO発生の原因

- 次の2つの条件が成立した場合にRTOが発生

  （１）メッセージの末尾パケットが破棄

  （２）（１）がプロセスの組で発生

- TCPとメッセージ通信のセマンティクスギャップ？



RTO時間の計算

- RTO = RTT + α（詳細は下式参照）
  - α過大→再送遅れ
  - α過小→無駄な再送の発生

  } スループットの低下

- RTTが0ミリ秒でも、RTO時間は200ミリ秒以上！

RTO = SRTT + max(G, 4 x RTTVAR)

SRTT = (1 - β) x SRTT + β x RTT

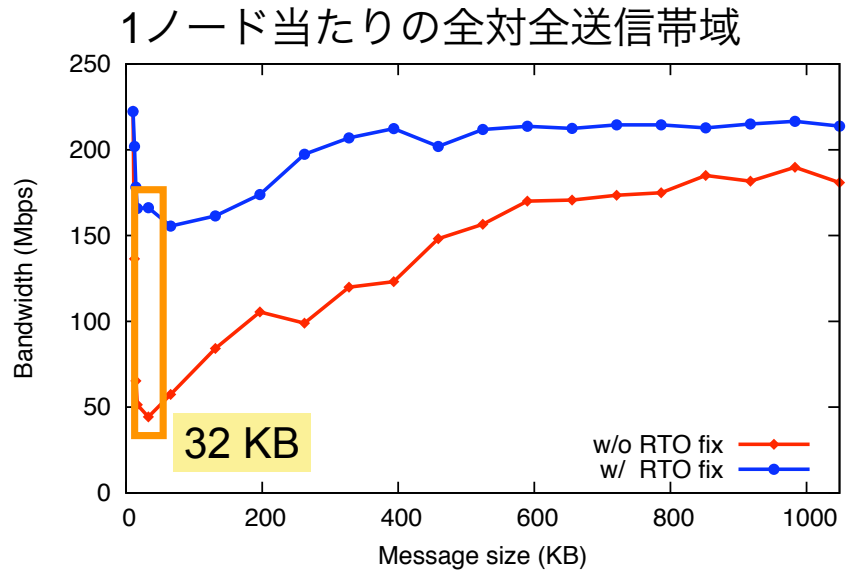
RTTVAR = (1 - γ) x RTTVAR + γ x |SRTT - RTT|

β = 1/8, γ = 1/4, G = 200 msec

SRTT: Smoothed Round Trip Time

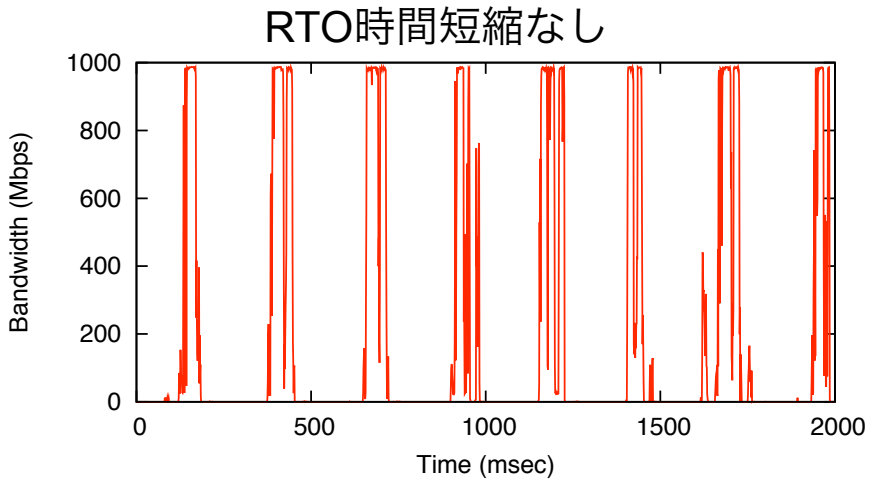
参考：RFC2988では、Gは1秒（RTT計測用のタイマの粒度が500ミリ秒と荒いから）

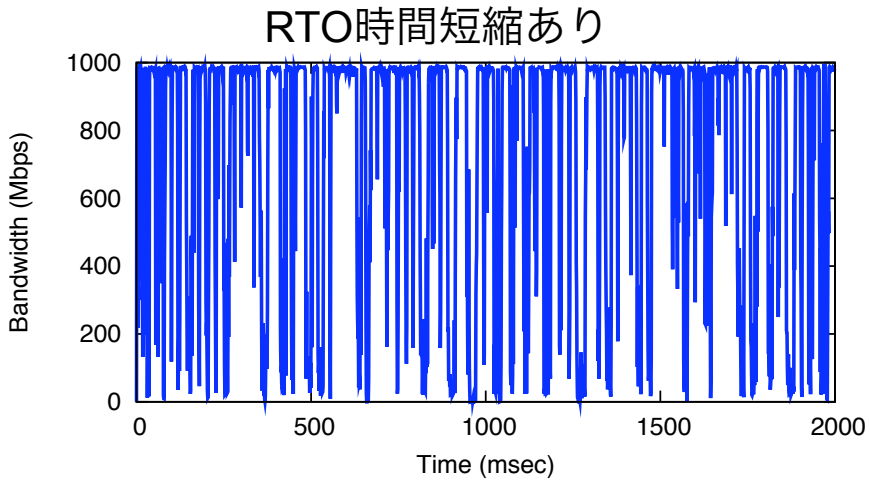
RTO時間短縮の効果



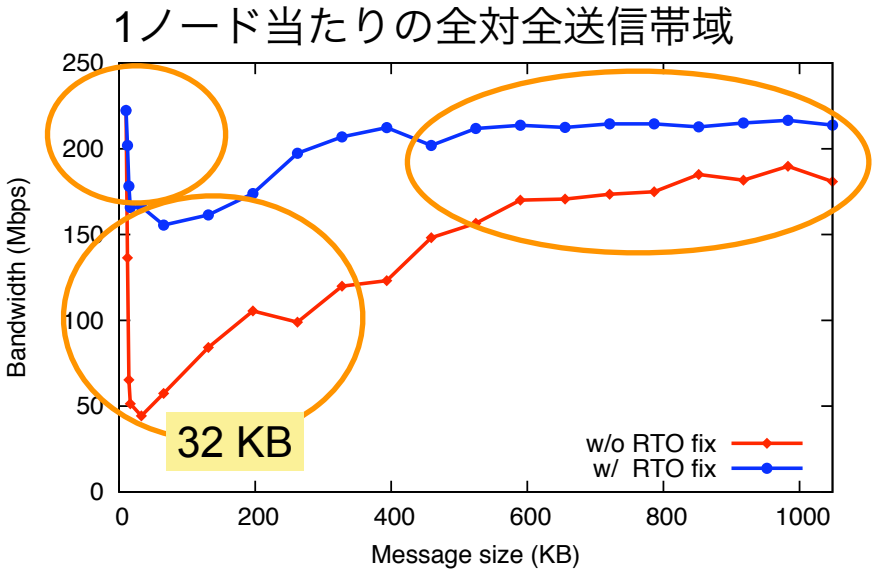
RTO = SRTT + max(G, 4 x RTTVAR)

- 変更前：G = 200ミリ秒
- 変更後：G = 0ミリ秒
  - (kernel 2.6.23以降) ip route コマンドにより設定変更可能





全対全通信の考察



- 小メッセージサイズ
  - バースト送信はスイッチのバッファで吸収
  - ▶ バッファサイズが大きい、ノンブロッキングスイッチ
- 中メッセージサイズ
  - RTOが頻発
  - ▶ RTO時間短縮が効果的
- 大メッセージサイズ
  - 確率的に末尾パケットの欠損が減るため、RTOは減少

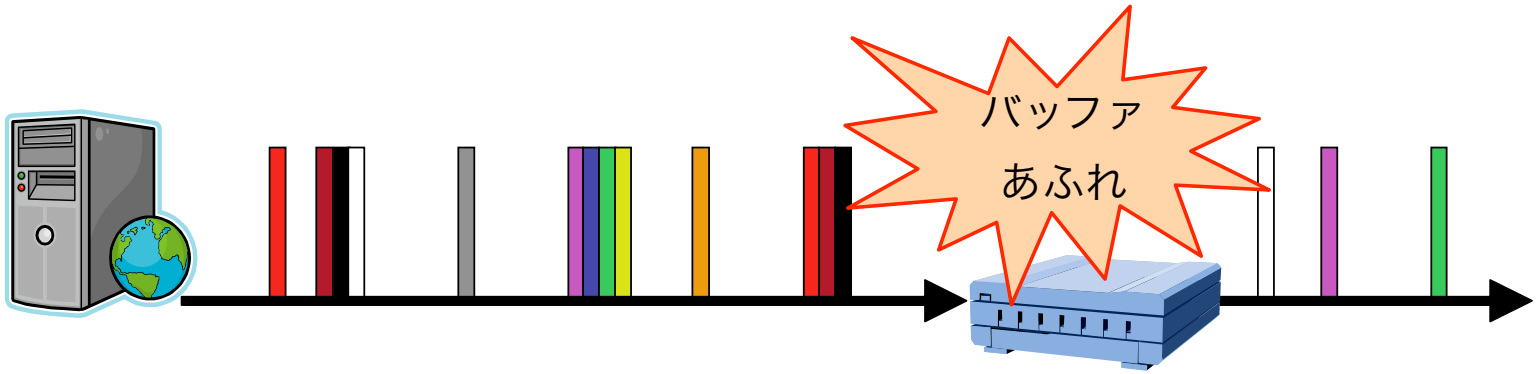
MPI通信向けTCP/IPの改良

３つの変更点	対応する問題点
通信再開時ペーシング	通信開始時における、スロースタートの省略 ACKクロッキングの代用
RTO時間の短縮	再送タイムアウト時の待ち時間の短縮
通信パラメータ保存復元	通信パターンによる、急激なcwnd値の変化に対応

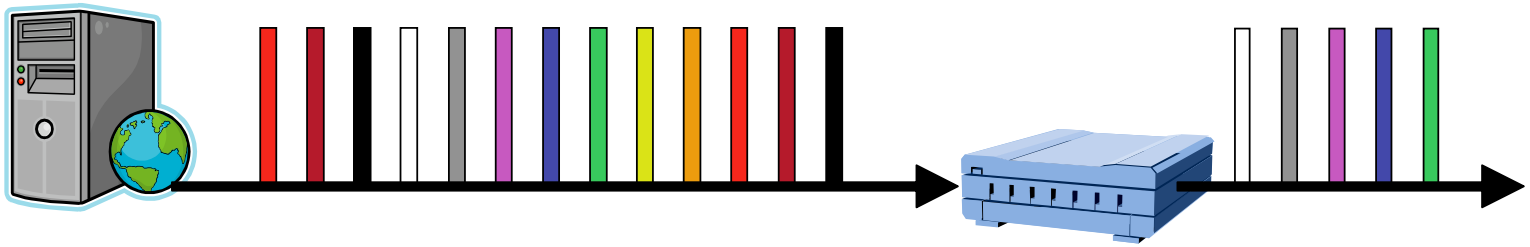
詳細は、松田他「MPIライブラリと協調するTCP通信の実現」,
情報処理学会論文誌コンピューティングシステム（ACS11）, 2005

その他の
チューニング方法

ペーシング技術



バースト性トラフィックはパケットロスによる
著しい性能劣化を引き起こす可能性がある



利用可能帯域に合わせてパケット間隔を平滑化（ペーシング）
することで、パケットロスのない安定した通信を実現できる

PSPacer：ソフトウェアによる精密ペーシング機構

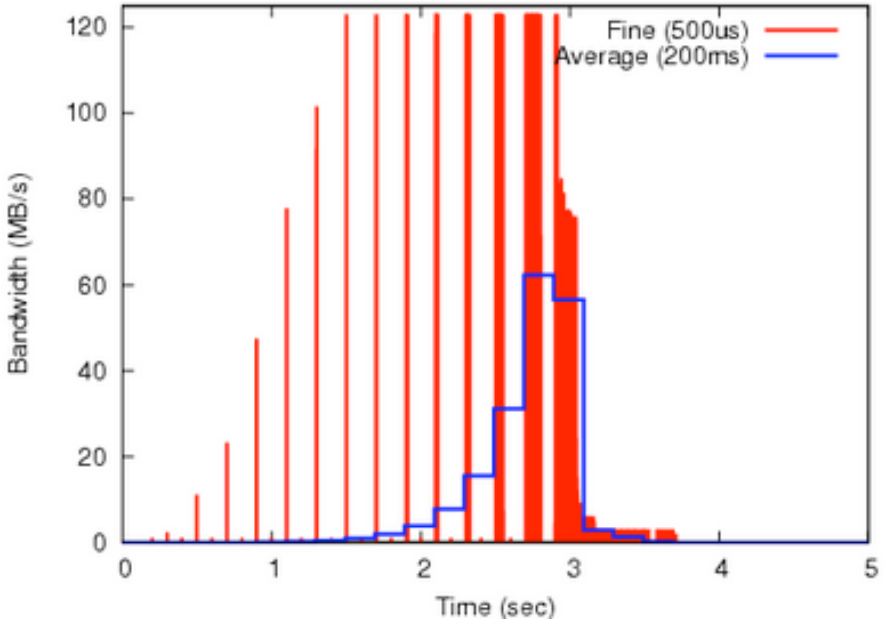
- 特別なハードウェアが不要で、ソフトウェアだけによる精密なペーシングの実現
- Linuxカーネルモジュールとして動作（OSの再インストールが不要で簡単に利用可）
- 産総研で開発し、GPLライセンスによるオープンソースソフトウェアとして公開

http://www.gridmpi.org/pspacer.jsp

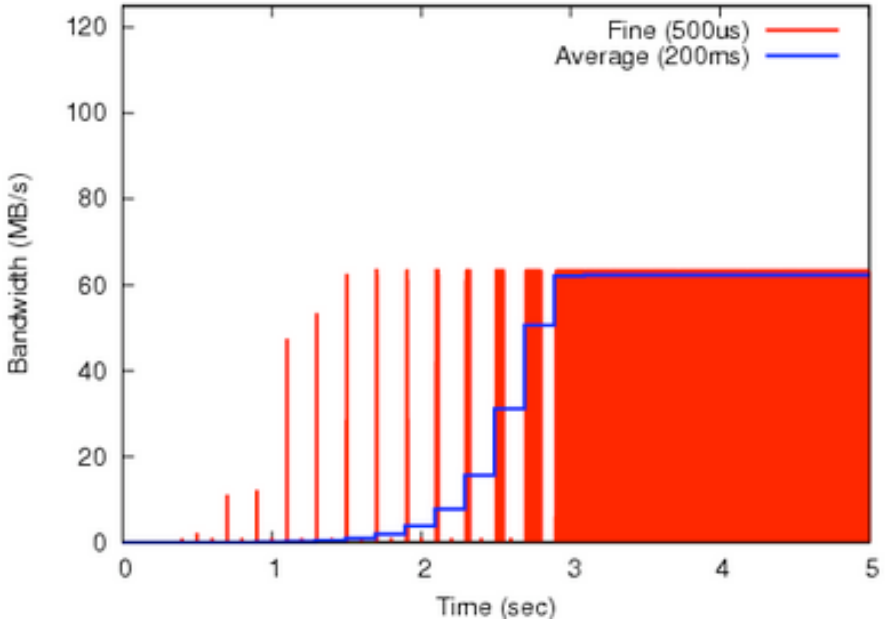
詳細は、高野他「ギャップパケットを用いたソフトウェアによる精密ペーシング方式」， 情報処理学会論文誌コンピューティングシステム（ACS14）, 2006

スロースタート時の効果

- 目標レート（500Mbps）を超過するバースト送信が抑制され、パケットロスを回避
- スロースタートフェーズに限らず、送信レートを制限

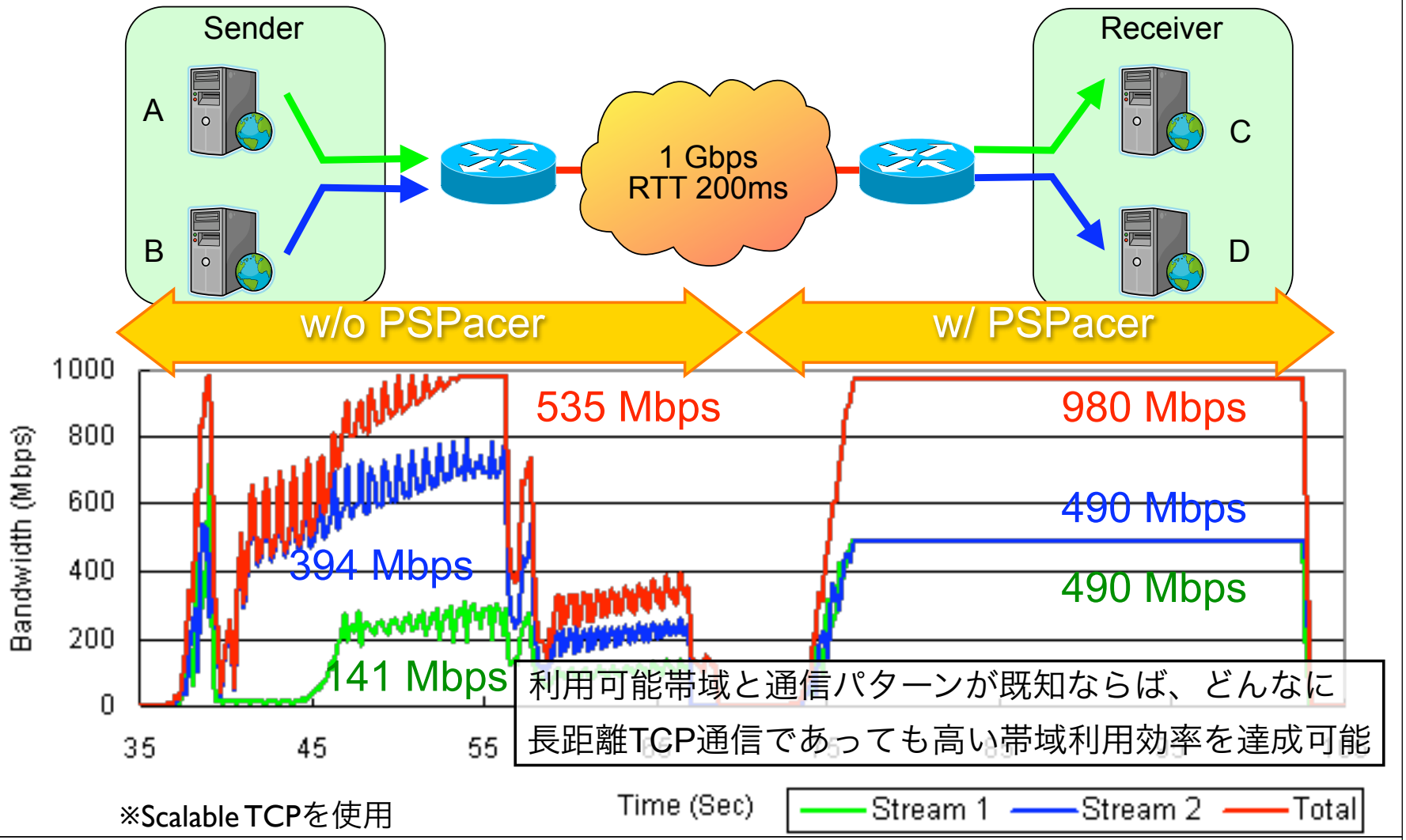


PSPacerなし



PSPacerあり（500Mbps）

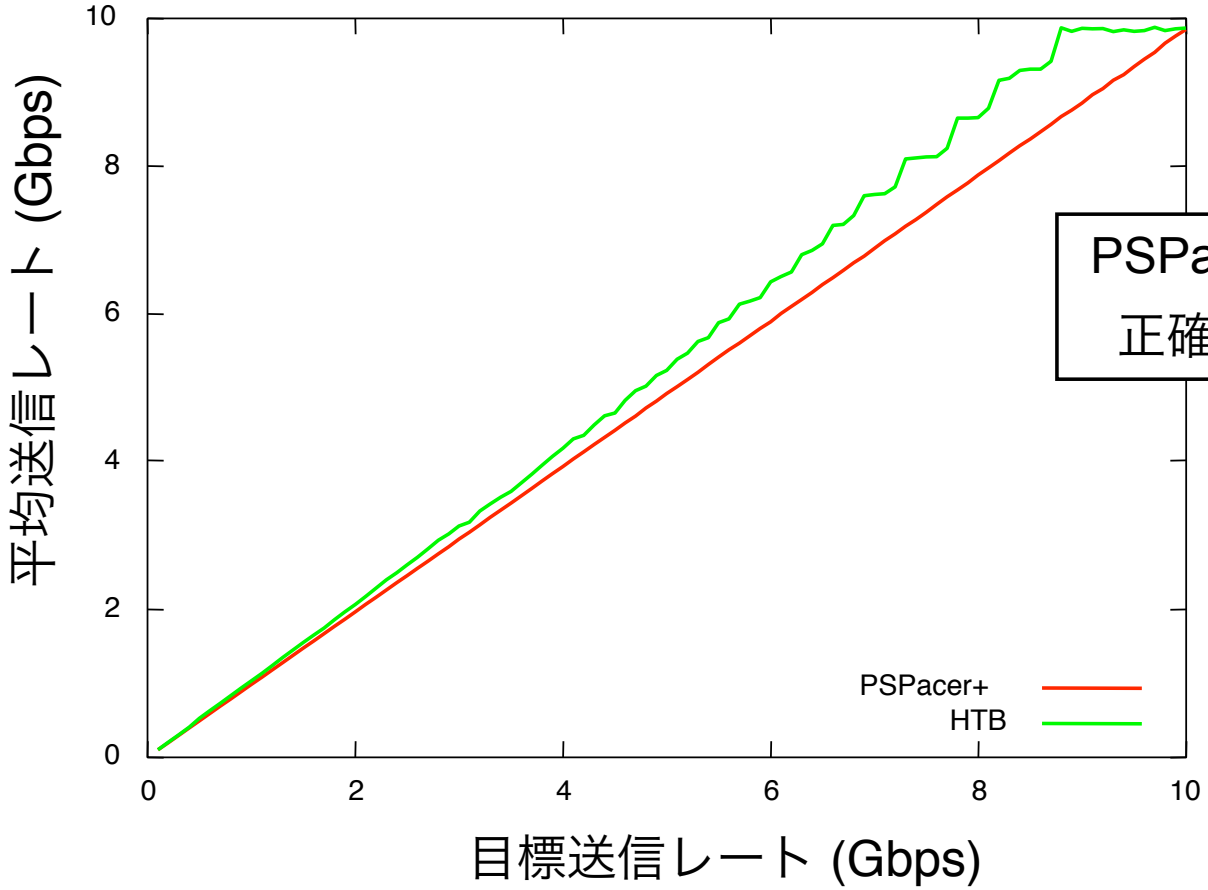
長距離TCP通信への適用



※Scalable TCPを使用

PSPacerの10GbE対応

目標送信レートを100Mbps刻みで設定し、Iperfを5秒間実行した平均送信レートを
GtrcNET-10を用いて観測



PSPacerは10GbEでも、
正確な帯域制御を実現

HTB: Hierarchical Token Bucket
（Linux標準の帯域制御モジュール）

CPU: Quad-core Xeon (E5430) x 2
NIC: Myricom Myri-10G (PCIe x 8)
MTU: 9000 byte
Memory: 4GB DDR2-667

実験の再現性

- 送信先ごとにメトリクス情報をキャッシュするので、実験ごとに初期値がばらばら
  - 何を計測しているのかわからなくなる可能性
- ipコマンドでキャッシュ内容の確認

```
$ ip route show cached to 192.168.0.2
192.168.0.2 from 192.168.0.1 dev eth1
    cache  mtu 9000 rtt 187ms rttvar 175ms ssthresh 1024 cwnd 637
advmss 8960 hoplimit 64
```

- キャッシュの無効化

```
# sysctl -w net.ipv4.tcp_no_metrics_save=1
net.ipv4.tcp_no_metrics_save=1
```

ジャンボフレーム

- ネットワーク帯域を効率的に利用可能
  - ヘッダオーバヘッド削減によるスループットの向上

GbE：MTU 1500B → 949.3 Mbps

MTU 9000B → 991.4 Mbps

- 実装依存なので、“ping -M do”（IPフラグの“Don’t fragment” bitをonという意味）で疎通確認すること

```
$ ping -M do -s 9000 192.168.0.2                                    NG
PING 192.168.0.2 (192.168.0.2) 9000(9028) bytes of data.
From 192.168.0.1 icmp_seq=1 Frag needed and DF set (mtu = 9000)
From 192.168.0.1 icmp_seq=1 Frag needed and DF set (mtu = 9000)
From 192.168.0.1 icmp_seq=1 Frag needed and DF set (mtu = 9000)

$ ping -M do -s 8972 192.168.0.2                                    OK
PING 192.168.0.2 (192.168.0.2) 8972(9000) bytes of data.
8980 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=3.76 ms
8980 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=0.127 ms
8980 bytes from 192.168.0.2: icmp_seq=3 ttl=64 time=0.133 ms
```

イーサネットフロー制御

- InfinibandやMyrinetのクレジットベースフロー制御との違い
  - 一時的に特定の終端ノードに受信が集中した場合、バッファあふれが発生
- イーサネットフロー制御（IEEE 802.3x）
  - 受信バッファが閾値を超えたときに、PAUSEフレームを送信することで、対向の送信を一時中断
  - ホストおよびスイッチの設定確認が必要

```
$ sudo ethtool -A eth0 tx on rx on

$ sudo ethtool -a eth0
Pause parameters for eth1:
Autonegotiate: off
RX:         off
TX:         off
```

［参考］Lossless Ethernetが標準作業中

TCP/IPをよりよく知るためのツール

Rough consensus, running code

- 何が正しい挙動か迷ったらRFC？
  - 標準はReno [RFC 2851]までで、多くは実装依存
- できるだけ新しいカーネルを使おう
  - net/ipv4以下に限ってもメジャーバージョン間で平均180個のパッチがコミット
  - ただし、regressionなどの可能性があるので、できる限り複数バージョンでチェック
    - 例）BICのinit_ssthresh、ABCの性能バグ
- 結局ソースコードを追う羽目になる
  - 静態解析（エディタ）と動態解析の併用

Web100

- 米国Pittsburgh Supercomputing Center (PSC)などで開発
- コネクションごとにカーネル変数値がprocfs経由で取得可能
  - netstat -sよりも詳細な情報が取得可能 [RFC 4898]
- カーネルパッチが必要
- システムの負荷が高いと、データ取得プロセスがスケジューリングされず、欲しいデータが取得できない場合があるので注意

http://www.web100.org/

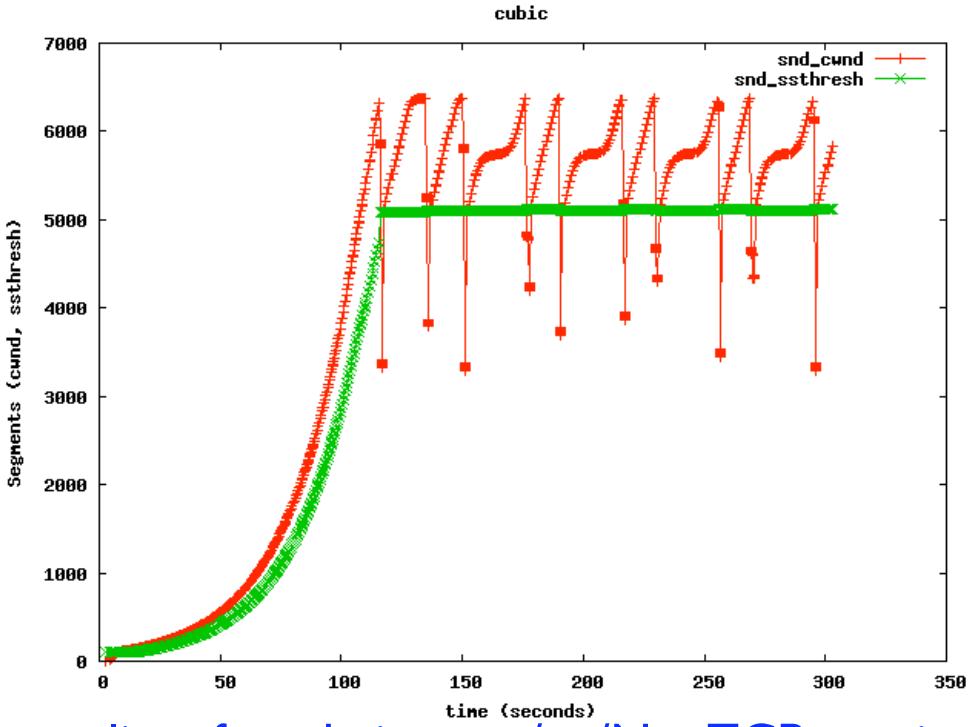
TCP Probe

- KProbesを使って、TCP受信処理関数にプローブを挿入し、変数の値を取得する仕組み
  - データはカーネル内部のバッファに格納され、後からprocfs経由で取得
- カーネルに同梱
- 取得したい場所や変数に合わせて、カスタマイズしたモジュールを用意すると便利

TCP Probeの使い方

```
$ sudo modprobe tcp_probe port=5001
$ sudo chmod 444 /proc/net/tcpprobe
$ cat /proc/net/tcpprobe >/tmp/tcpprobe.out &
$ TCPCAP=$!
    （通信の実行）
$ kill $TCPCAP
```

cwndとssthreshの
プロット結果



http://www.linuxfoundation.org/en/Net:TCP_testing

まとめ（１）

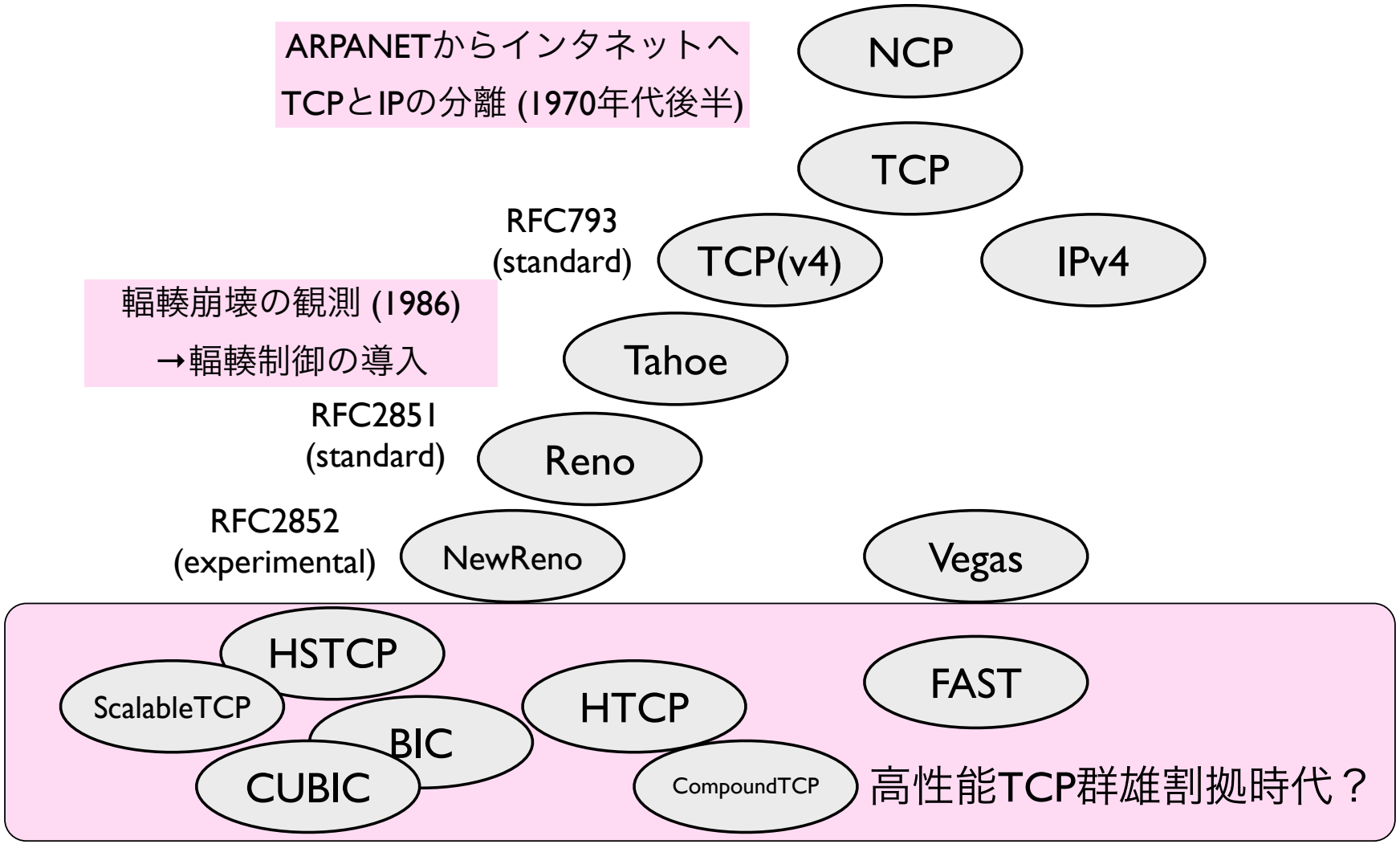
1. ソケットバッファサイズは帯域遅延積の２倍（帯域×RTT）以上必要（デフォルトは4MB）
2. setsockopt関数はlistenやconnect関数の前に実行
3. インタフェースキューあふれは輻輳とみなされるので、帯域遅延積の大きなネットワークでは、キュー長の拡大が必要
4. 高速TCPは公平性を保ちつつ、帯域利用効率の向上を追求。公平性を失った実装の使用はインタネットでは危険
5. 帯域遅延積が大きなネットワークでは、スロースタートはまったく「スロー」ではない。バースト送信の抑制には、ペーシングが効果的

まとめ（2）

1. 間欠通信時の通信効率を上げるには、通信アイドル後のスロースタートの省略が有効（tcp_slow_start_after_idle）
2. さらに通信再開時のバースト送信を回避するには、ペーシングが効果的
3. 集団通信時など、再送タイムアウト（RTO）の頻発による通信性能の低下には、RTO時間の短縮が有効
4. 実験の再現性を高めるために、ルーティングメトリクスのキャッシュを無効化
5. できるだけ新しいカーネルを使用し、できる限り複数バージョンで確認

予備資料

TCP輻輳制御の簡単な歴史



関連RFC

Spec.	sysctl	default
TCP (RFC 793)	-	-
Window scaling (RFC 1323)	tcp_window_scaling	1
TImestamps option (RFC 1323)	tcp_timestamps	1
TIME-WAIT Assassination Hazards (RFC 1337)	tcp_rfc1337	0
SACK (RFC 2018, 3517)	tcp_sack	1
Control block sharing (RFC 2140)	tcp_no_metrics_save	0
MD5 signature option (RFC 2385)	-	-
Initial window size (RFC 2414)	-	-
Congestion control (RFC 2581)	-	-
NewReno (RFC 3782)	-	-
Cwnd validation (RFC 2861)	tcp_slow_start_after_idle	1
D-SACK (RFC 2883)	tcp_dsack	1
Limited transmit (RFC 3042)	-	-
Explicit congestion notification (RFC 3168)	tcp_ecn	0
Appropriate Byte Counting (RFC 3465)	tcp_abc	0
Limited slow start (RFC 3742)	tcp_max_ssthresh	0
FRTO (RFC 4138)	tcp_frto	0
Forward ACK	tcp_fack	1
Path MTU discovery	tcp_mtu_probing	0

参考URL

Project	URL
BIC/CUBIC	http://netsrv.csc.ncsu.edu/twiki/bin/view/Main/BIC
FAST	http://netlab.caltech.edu/FAST/
TCP Westwood+	http://c3lab.poliba.it/index.php/Westwood
Scalable TCP	http://www.deneholme.net/tom/scalable/
HighSpeed TCP	http://www.icir.org/floyd/hstcp.html
H-TCP	http://www.hamilton.ie/net/htcp/index.htm
TCP-Illinois	http://www.ews.uiuc.edu/~shaoliu/tcpillinois/
Compound TCP	http://research.microsoft.com/en-us/projects/ctcp/
TCP Vegas	http://www.cs.arizona.edu/projects/protocols/
TCP Westwood	http://www.cs.ucla.edu/NRL/hpi/tcpw/
Circuit TCP	http://www.ece.virginia.edu/cheetah/
TCP Hybla	https://hybla.deis.unibo.it/
TCP Low Priority	http://www.ece.rice.edu/networks/TCP-LP/
UDT	http://www.cs.uic.edu/~ygu1/
XCP	http://www.ana.lcs.mit.edu/dina/XCP/
Web100	http://www.web100.org/
TCP Probe	http://www.linuxfoundation.org/en/Net:TcpProbe
iproute2	http://www.linuxfoundation.org/en/Net:Iproute2