

画像情報特論 (4)

Hybrid TFRC

情報理工学専攻 甲藤二郎

E-Mail: katto@waseda.jp

NS-2 TCP-Linux

NS-2 TCP-Linux (1)

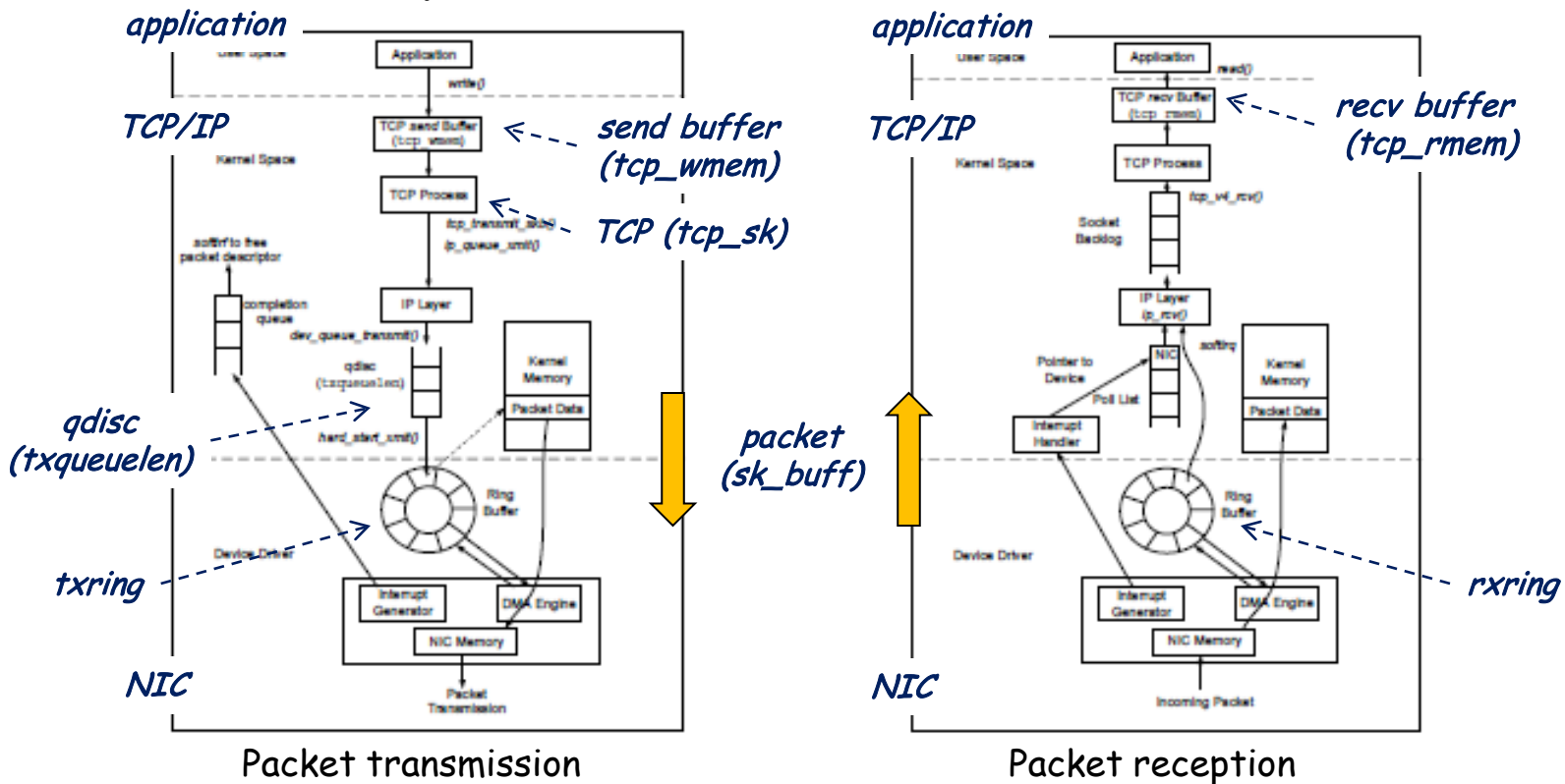
- Linuxカーネル内のTCP実装コードを使ったns-2シミュレーション
 - Linuxカーネル(実装)とns-2(シミュレーション)の橋渡し
 - 実装とシミュレーションの乖離の回避
 - 実装コードの検証
 - 新規アルゴリズムのLinuxカーネルへの容易な組み込み

NS-2 TCP-Linux (2)

- Linux実装されているTCP (2.6.16-3現在)
 - TCP-Reno, TCP-Vegas, HighSpeed-TCP, Scalable-TCP, BIC-TCP, CUBIC-TCP (default), TCP-Westwood, H-TCP, TCP-Hybla
- 追加が検討されているTCP
 - TCP-Veno, TCP-LowPriority, Compound-TCP (Windows)

NS-2 TCP-Linux (3)

- TCP Implementation in Linux



NS-2 TCP-Linux (4)

- Variables in tcp_sk

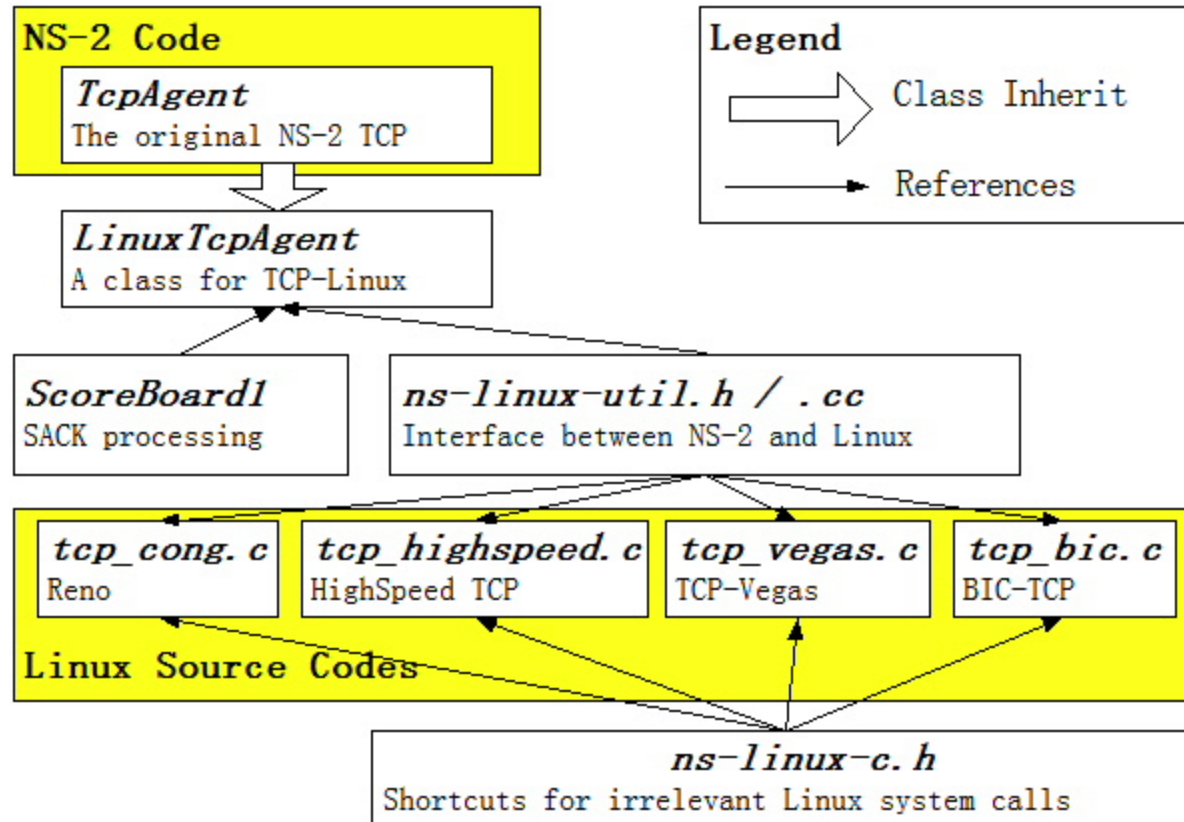
Name	Meaning	Equivalent in NS-2 <i>TCPAgent</i>
snd_ssthresh	slow-start threshold	ssthresh_
snd_cwnd	integer part of the congestion window	trunc(cwnd_)
snd_cwnd_cnt	fraction of congestion window	trunc(cwnd_ ²) %trunc(cwnd_)
icsk_ca_priv	a 512-bit array to hold per-flow state for a congestion control algorithm	n/a
icsk_ca_ops	a pointer to the congestion control algorithm interface	n/a

Congestion control modules:

cong_avoid: slow start & congestion avoidance
ssthresh: loss event handling
min_cwnd: fast retransmission

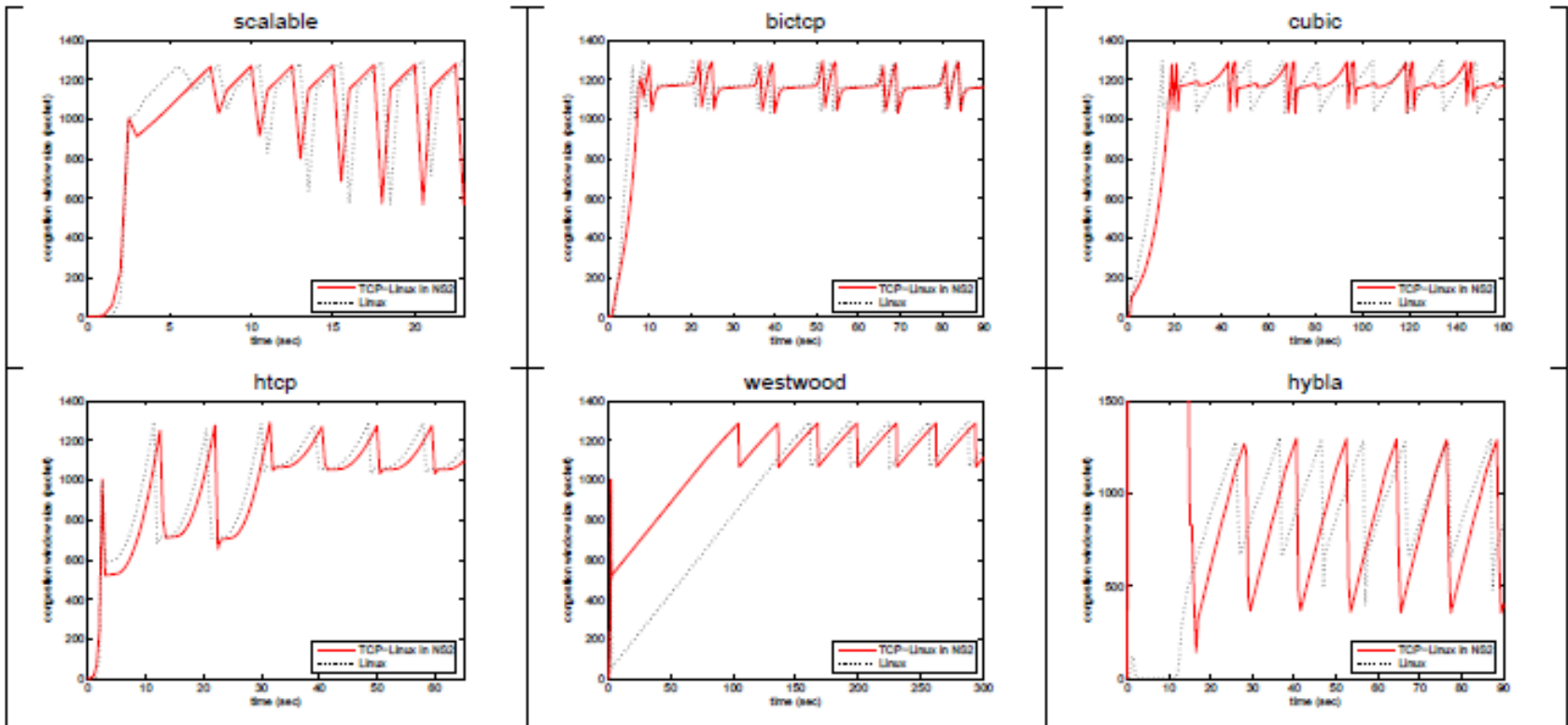
NS-2 TCP-Linux (5)

- Code structure



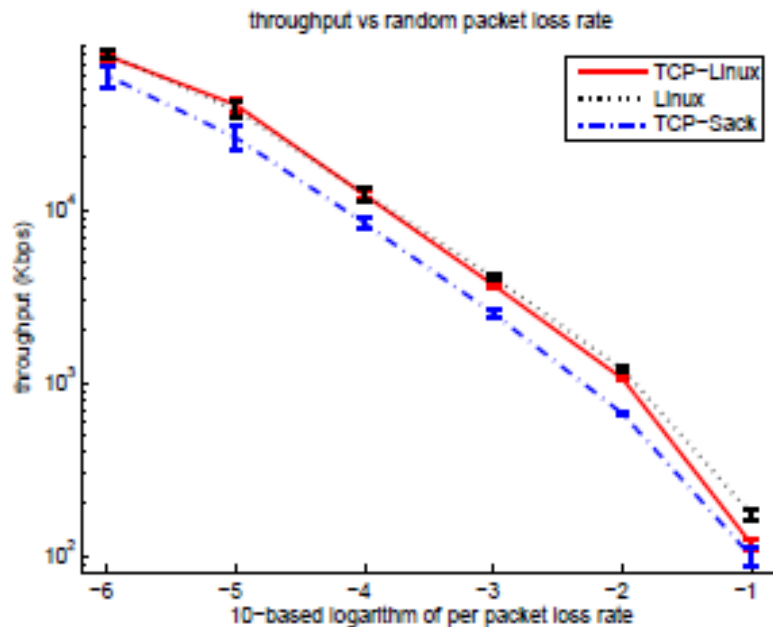
NS-2 TCP-Linux (6)

- Simulation (1) ns-2 & Linux

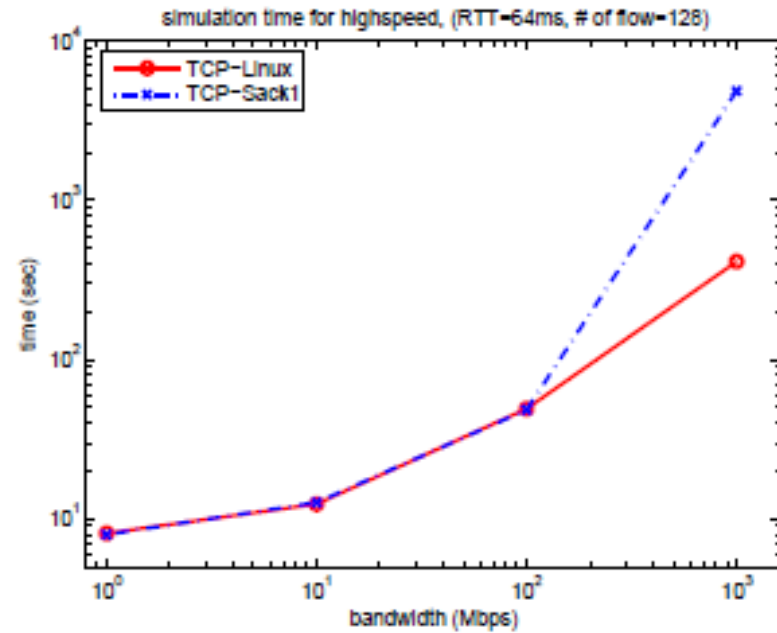


NS-2 TCP-Linux (7)

- Simulation (2) accuracy & speed



Accuracy

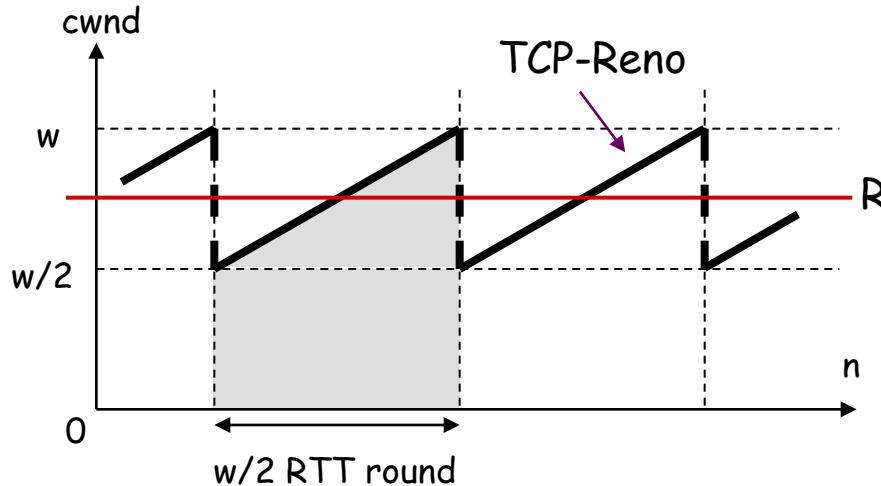


Speed

TCP Equations

TCP Modeling

- TCP-Reno Equivalent Rate



w : パケット廃棄発生時のcwnd
 p : パケット廃棄率
 RTT : ラウンドトリップ遅延
 R : (等価) レート
 b : delayed ACK の個数

\rightarrow

$$\begin{cases} p = \frac{8}{3w^2} & \text{タイムアウト考慮} \\ R = \frac{PS}{RTT} \cdot \sqrt{\frac{3}{2p}} \end{cases}$$
 \rightarrow

$$R_{loss} = \frac{PS}{RTT \sqrt{\frac{2bp}{3}} + t_{RTO,loss} \cdot 3 \sqrt{\frac{3bp}{8}} \cdot p(1+32p^2)}$$

TCP Westwood

- Duplicate ACKs

FSE: Fair Share Estimates

$$ssthresh = FSE * RTT_{\min}$$

$$\text{if } (cwnd > ssthresh) \text{ } cwnd = ssthresh$$

- Timeout

TCP-Reno の場合:

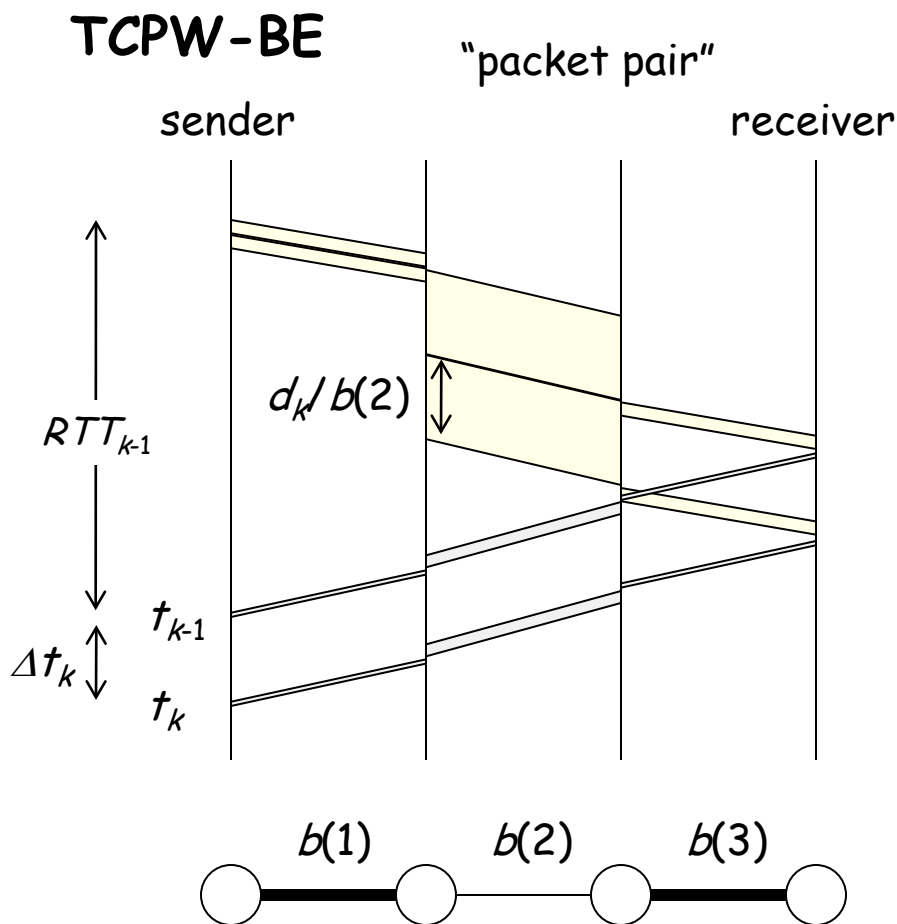
$$ssthresh = cwnd / 2$$

$$ssthresh = FSE * RTT_{\min}$$

$$cwnd = 1$$

- FSEの求め方に応じて複数バージョン

Bandwidth Share Estimation



Bandwidth share: $b = \min_j (b(j))$

t_k : ack arrival time of the k -th packet

d_k : size of the k -th packet



$$\Delta t_k = t_k - t_{k-1} \approx \frac{d_k}{b}$$



$$b_k \approx \frac{d_k}{\Delta t_k}$$



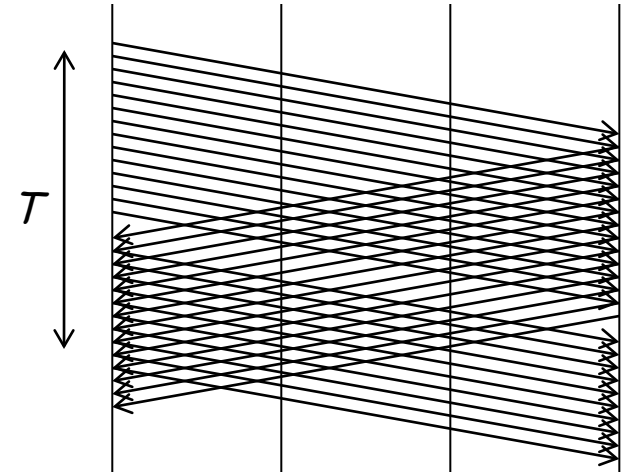
移動平均を行い平滑化: $\hat{b}_k \rightarrow FSE$

Rate Estimation

(参考) TCP-Vegas

$$diff = \left(\frac{cwnd}{RTT_{min}} - \frac{cwnd}{RTT} \right) \cdot RTT_{min}$$

↑ expect rate ↑ actual rate



TCPW-RE:

$$RE_k = \frac{\sum_{t_j > t_k - T} d_j}{T}$$

個数(cwnd) ⇒ バイト数($\sum d_k$)
 RTT ⇒ 観測時間($T = \sum \Delta t_k$)

$$\hat{RE}_k \approx \frac{\sum d_k}{\sum \Delta t_k}$$

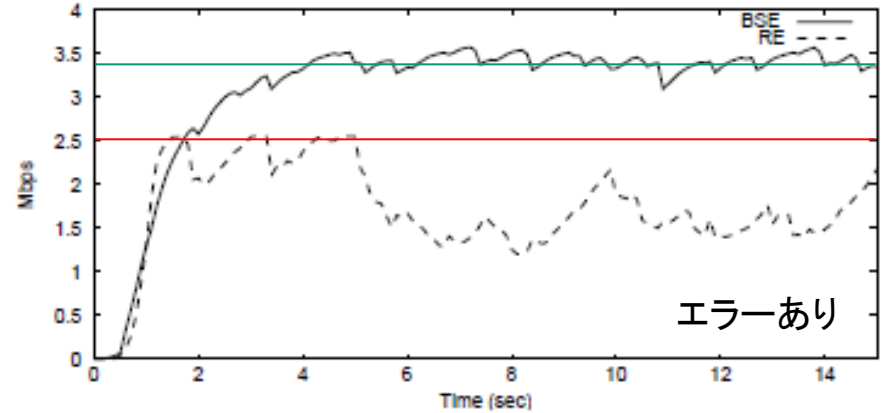
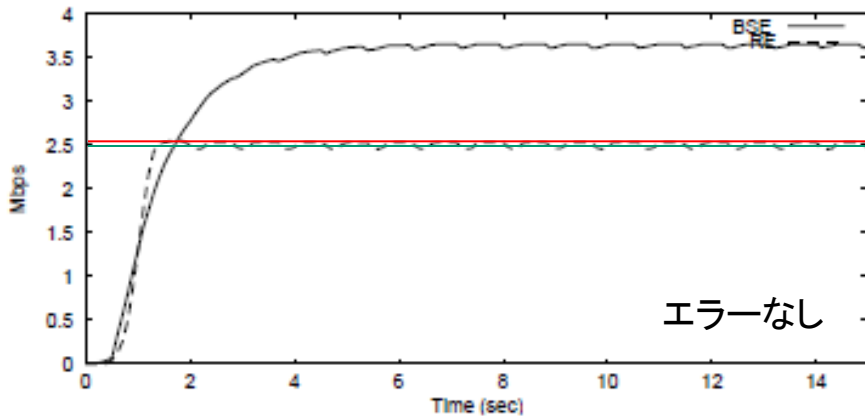


移動平均を行い平滑化: $\hat{RE}_k \rightarrow FSE$

$$T = n \cdot RTT \quad (\text{e.g. } n=4)$$

BSEとREの比較

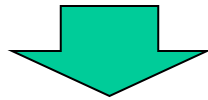
実線: BSE、破線: RE、赤線: fair share、緑線: キャパシティー競合フロー



- BSEは値が高めになりやすい(バースト性のため)
- REはロスがあると値が低めになりやすい

Adaptive Bandwidth Share Estimation

- BSEは高め、REは保守的(低め)
- BSEとREの違い: サンプル間隔Tの大小


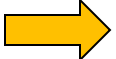


- 輻輳時はTを大きく、非輻輳時はTを小さく

TCPW-ABSE:

$$T_k = \max \left(T_{\min}, RTT \cdot \left(1 - \frac{\hat{T}h \cdot RTT_{\min}}{cwnd} \right) \right)$$

T_{\min} : ACK到着間隔

$\hat{T}h \cdot RTT_{\min} < cwnd$  輻輳状態  T_k を大きく(保守的なREへ)

多数のパケットを送っても実レートが上がらない

HYbrid TFRC

TFRC (RFC 3448)

- TCP-Friendly Rate Control

$$R_{TFRC} = \frac{PS}{RTT \sqrt{\frac{2bp}{3}} + 4 \cdot RTT \cdot 3 \sqrt{\frac{3bp}{8}} \cdot p(1 + 32p^2)}$$

b=1: delayed ACK (推奨)

t_{RTO} TCP再送タイムアウト

- RTTとパケットロス率pを観測して“TCP-Renoに等価”なレートを計算
- RTP/UDP や DCCP によるリアルタイム系メディア (音声、音楽、映像、ゲーム) 応用を想定

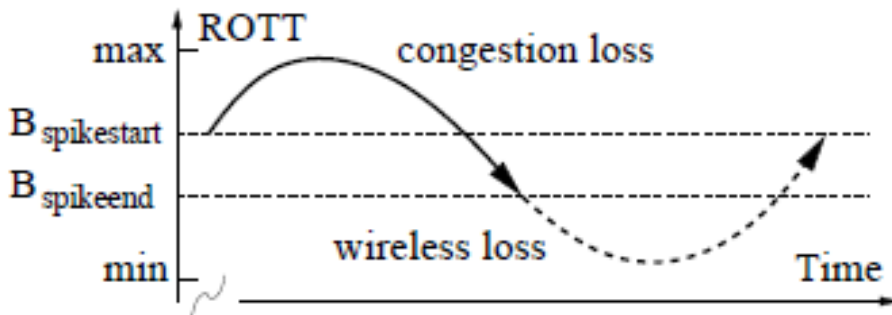
TFRC の弱点

- TCP-Renoの弱点を踏襲
 - バッファが BDP よりも小さい場合、頻繁に空き帯域が生じる
 - エラーが発生しやすい環境 (無線環境等) で、不必要にレートを下げる ⇒ LDA

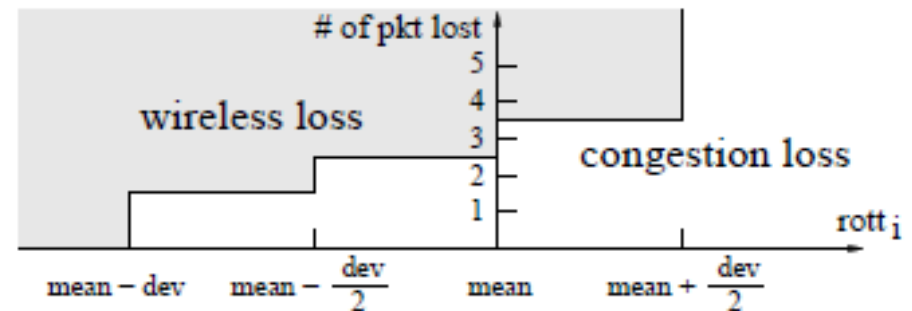
LDA (1)

"TFRC Wireless"

- Loss Differentiation Algorithm
 - 輻輳ロスとランダムエラーロスの区別



Spike algorithm



ZigZag algorithm

$$B_{spikestart} = rott_{min} + \alpha \cdot (rott_{max} - rott_{min})$$

$$B_{spikeend} = rott_{min} + \beta \cdot (rott_{max} - rott_{min})$$

$$\alpha = 1/2, \beta = 1/3$$

ROTT: Relative One-way Trip Time

LDA (2)

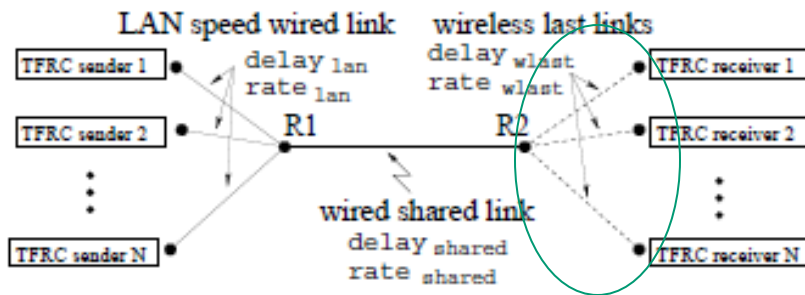
"TFRC Wireless"

シミュレーション評価

表のメトリック(上から)

- スループット
- 輻輳ロス
- ランダムロスと判断された輻輳ロス
- 輻輳ロスと判断されたランダムロス

Wireless last hop

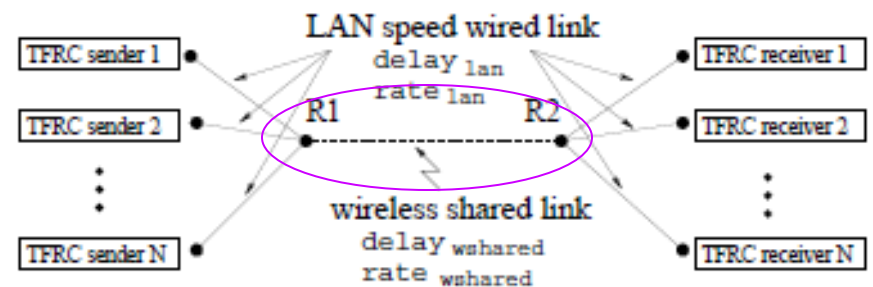


PERFORMANCE FOR WIRELESS LAST HOP, 1 FLOW

	TCP	TFRC	Omni	Biaz	mBiaz	Spike	ZigZag
thput	55	84	99	99	99	99	98
cong.	0.8	0.2	2.3	2.3	2.3	0.4	0.3
M_c	0	0	0	0.0	0.0	0.0	0.0
M_w	100	100	0	6.3	6.6	58	66

LDA

Wireless backbone



PERFORMANCE FOR WIRELESS BACKBONE, 1 FLOW

	TCP	TFRC	Omni	Biaz	mBiaz	Spike	ZigZag
thput	23	37	99	97	91	99	53
cong.	0.1	0.0	0.4	0.4	0.4	0.0	0.0
M_c	0	0	0	0.0	0.0	0.0	0.0
M_w	100	100	0	2.4	7.0	29	60

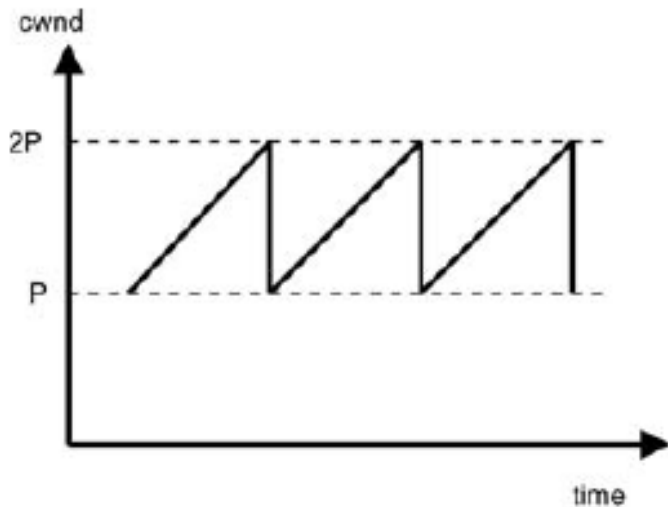
LDA

VTP (1)

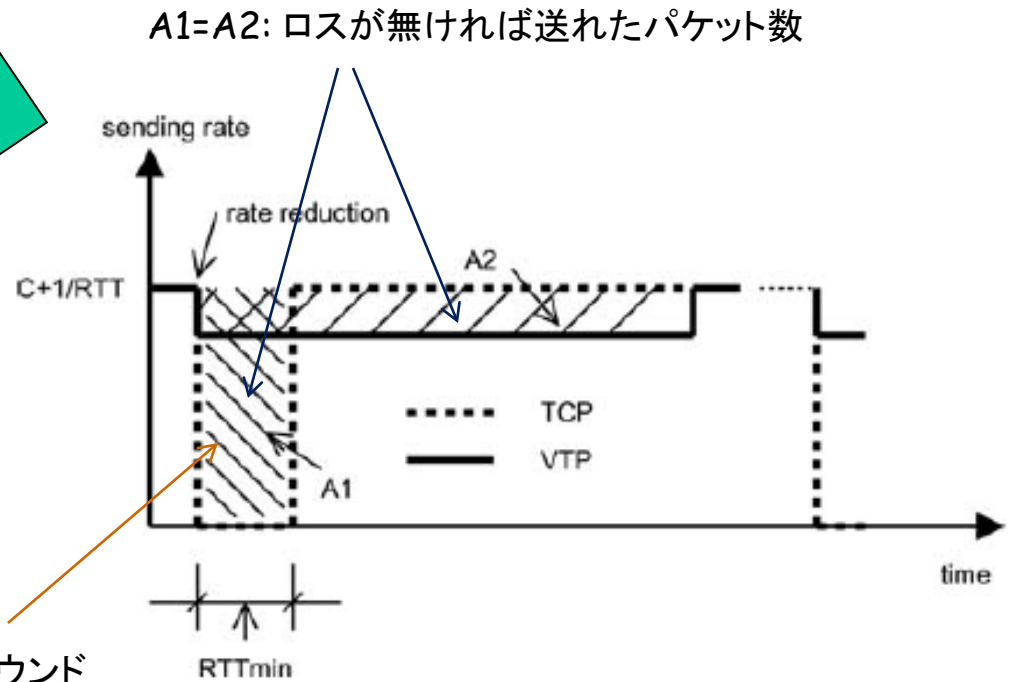
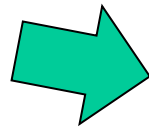
- Video Transport Protocol
 - LDA (輻輳ロスとランダムロスの区別～TFRC Wireless)
 - TCPW-REと同様のレート推定 (Achieved Rateの活用)
 - TCP-Renoのウィンドウ制御のエミュレート (レガシーTCPとの親和性)

VTP (2)

• VTPの概略



Buffer size = BDP を仮定



パケットロス再送ラウンド

VTP (3)

- VTPのウィンドウ制御
 - 初期値: TCPW-REで求まるAchieved Rate
 - 更新: RTTごとに1パケット追加

$$R_0 = \text{Achieved Rate}$$

RTTが増加しなければ $R_{k+1} = R_k + \frac{1}{RTT_k}$

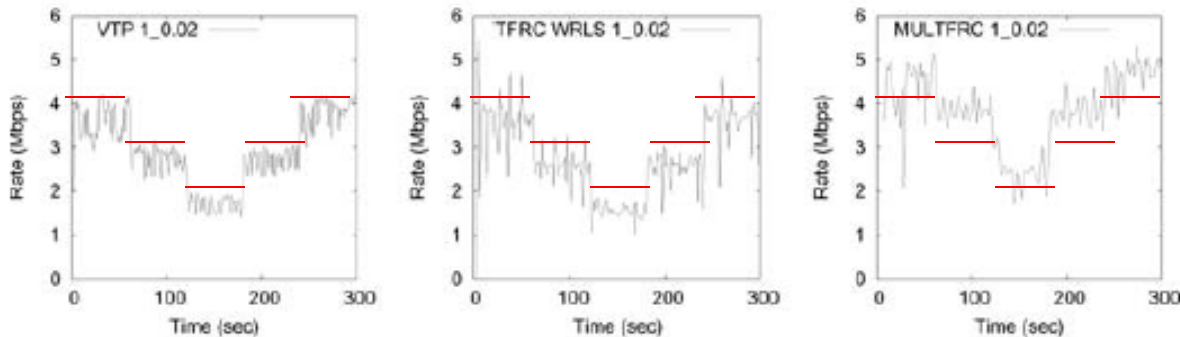
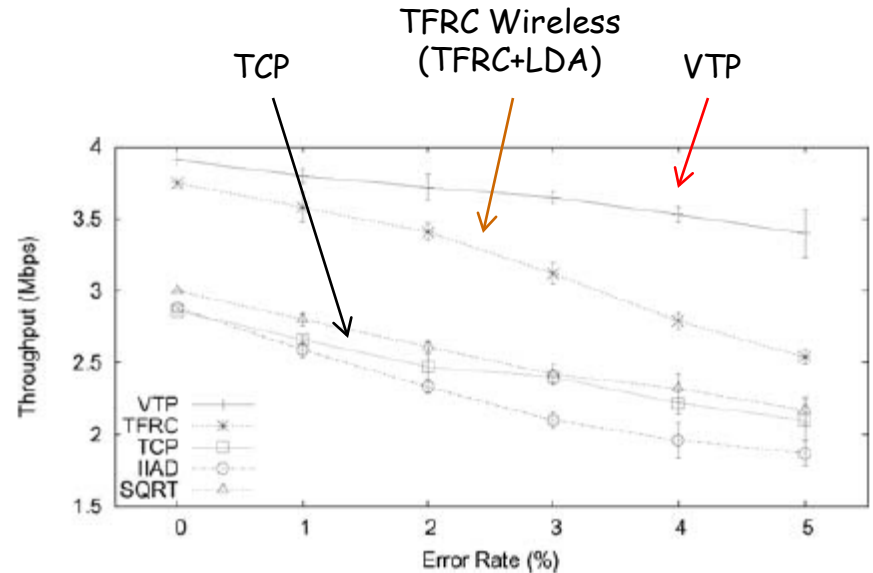
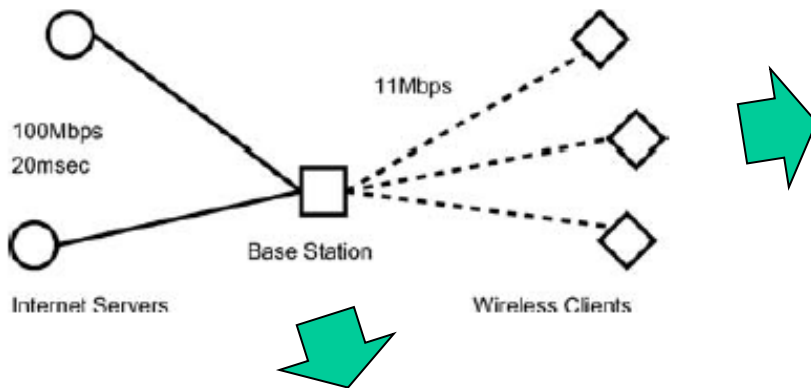
$$ewnd_k = R_k \times RTT_k$$



$$R_{k+1} = \frac{ewnd_{k+1}}{RTT_{k+1}} = \frac{ewnd_{k+1}}{RTT_k + \Delta RTT_k} = \frac{R_k \times RTT_k + 1}{RTT_k + (RTT_k - RTT_{k-1})}$$

VTP (4)

• シミュレーション評価



(b) 2% errors (avg. time in good state = 1 sec, avg. time in bad state = 0.02 sec).

MULTFRC: 複数のTFRC
コネクションを張り、帯域利
用効率の改善を図る

VTP (5)

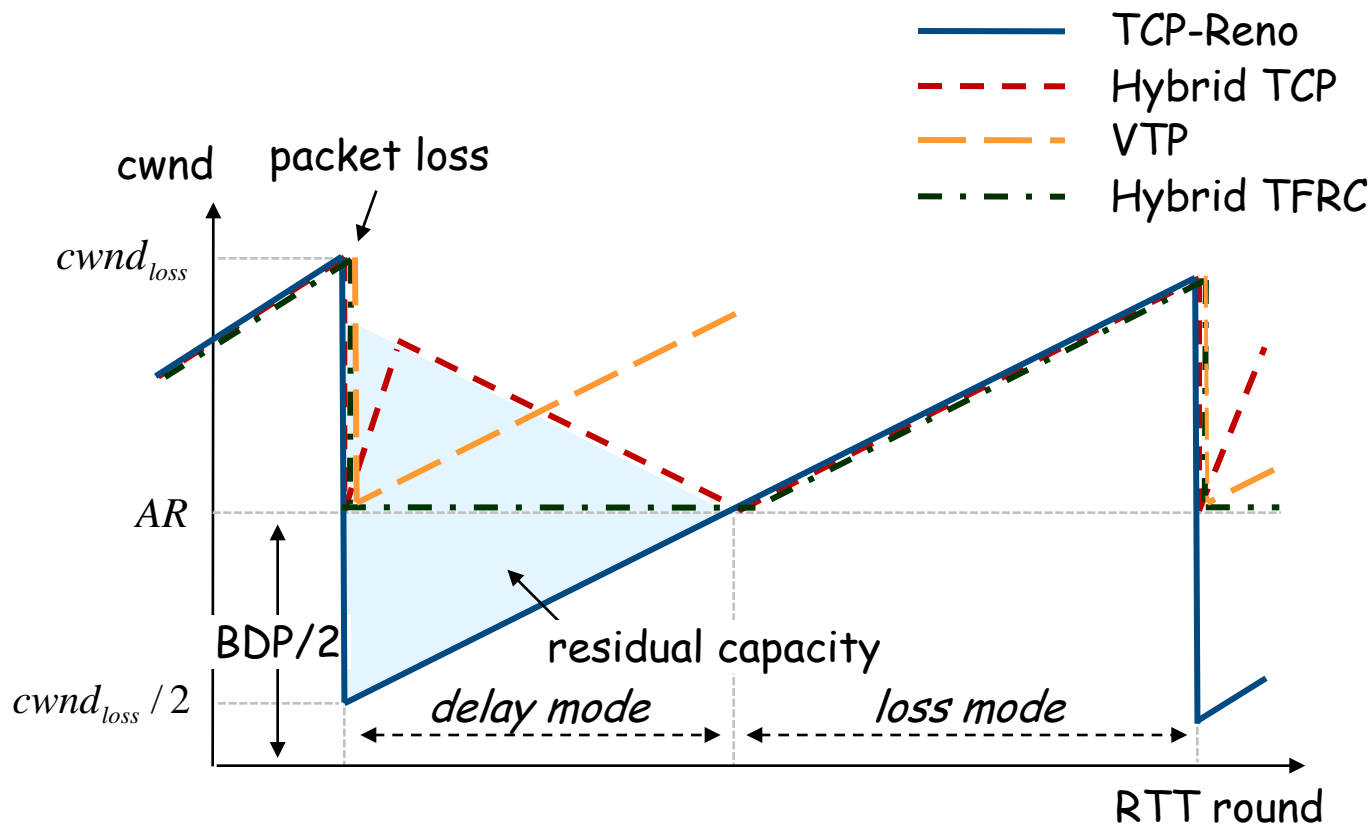
- VTPの弱点
 - Buffer size = BDP のみ想定
 - Buffer size < BDP の場合に生じる空き帯域有効利用の課題は想定外

ハイブリッドTFRC (1)

- ハイブリッドTCPのTFRC版
 - RTTが増加しない場合はAchieved Rateで送信(効率性の改善)
 - RTTが増加したらVTPと同様のウィンドウ制御で送信(親和性の確保)
 - 小バッファサイズへの対応 \Rightarrow RTTの低減

ハイブリッドTFRC (2)

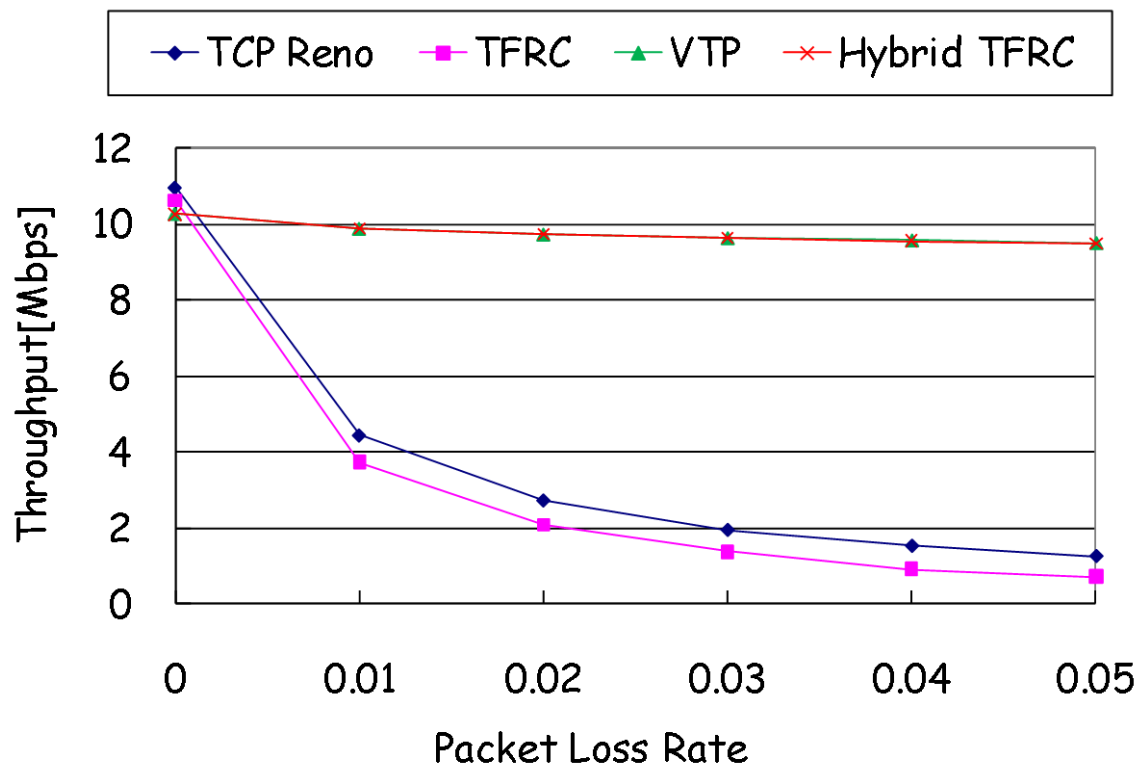
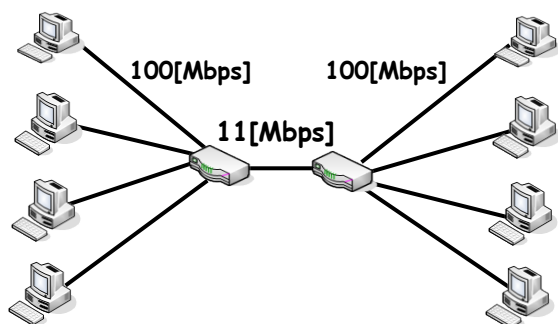
- ハイブリッドTFRCの動作 (Reno競合時)



ハイブリッドTFRC (3)

- シミュレーション評価(1)

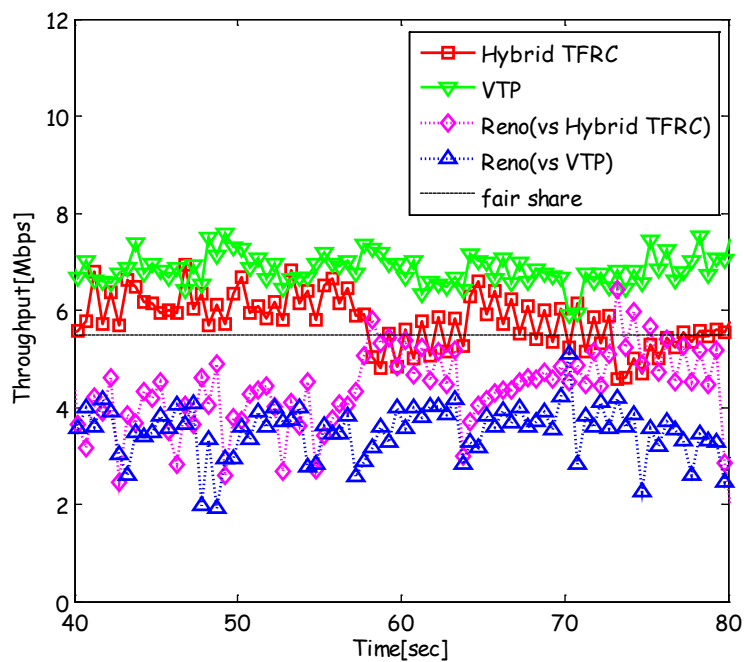
Buffer size = BDP



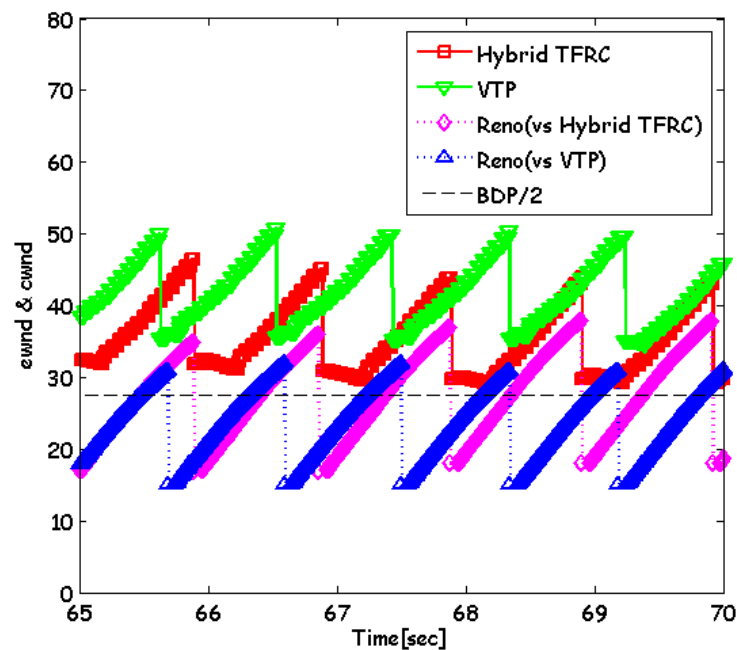
ハイブリッドTFRC (4)

- シミュレーション評価(2)

Buffer size = BDP/2



Throughput



ewnd & cwnd

TFWC & Relentless CC

TCP-Friendly Window-based Congestion Control (1)

- 従来のTFRCの弱点
 - Friendliness: 小バッファ時に不公平が生じる (競合TCPを追い出す)
 - Smoothness: 送信レートが大きく振動することがある (RTTが安定しないため)
 - Smoothness: RTTが小さ過ぎるとレート計算が安定しない
 - Responsiveness: フローのON/OFFに対する応答性は？
- Rate-based \Rightarrow Window-based

TCP-Friendly Window-based Congestion Control (2)

- Ack Vector:

- 受信者は、受信したパケットのシーケンス番号の並び(Ack Vector)を送信者に返す
- 送信者は、Ack Vectorからパケットロスの平均発生間隔を計算し($1/p$)、cwndを更新する

$$W_{TWRC} = \frac{1}{\sqrt{\frac{2p}{3}} + 12\sqrt{\frac{3p}{8}} \cdot p(1+32p^2)} \iff R_{TFRC} = \frac{PS}{RTT\sqrt{\frac{2p}{3}} + 12 \cdot RTT\sqrt{\frac{3p}{8}} \cdot p(1+32p^2)}$$

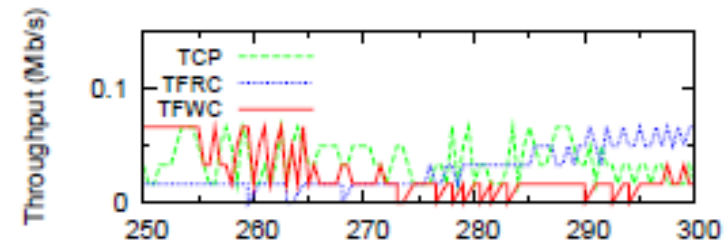
- RTTが含まれないので数値的に安定

TCP-Friendly Window-based Congestion Control (3)

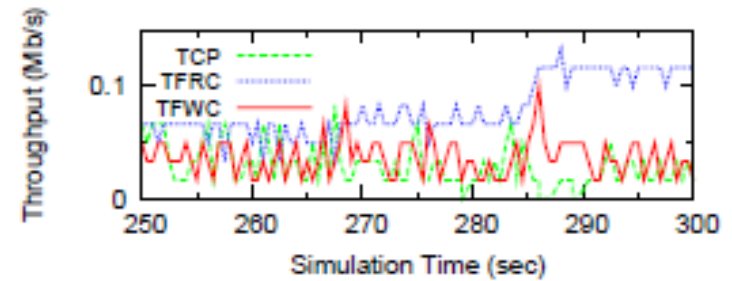
- Hybrid Window & Rate
 - cwndが小さくなり過ぎるとタイムアウトが発生する



- cwndが小さくなり過ぎた場合はTFRCを動作させる



(a) operated by window-based only



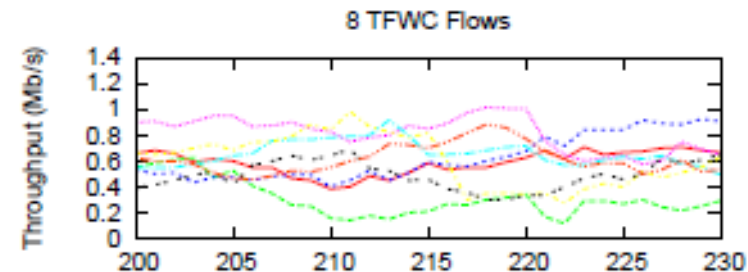
(b) operated by window/rate-based

TCP-Friendly Window-based Congestion Control (4)

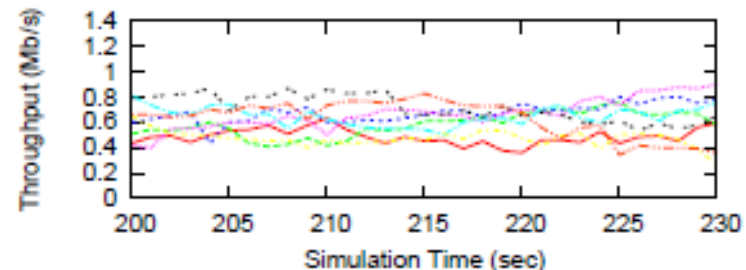
- Phase effect対策
 - drop tailでは競合フローのロス率が大きく異なることがある (特定フローのロスが“同期”)
 - 対策: RED (Random Early Drop) によるランダム廃棄



- 類推: cwndに微小な“乱数”を加える



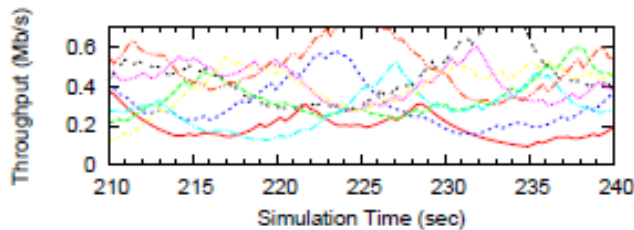
(a) Without Jitter



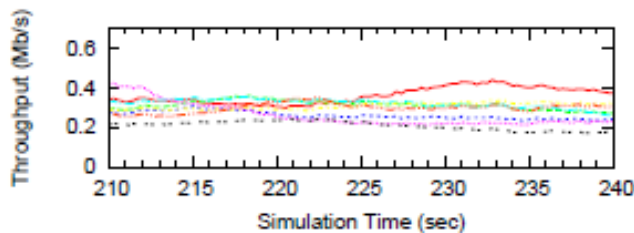
(b) With Jitter

TCP-Friendly Window-based Congestion Control (5)

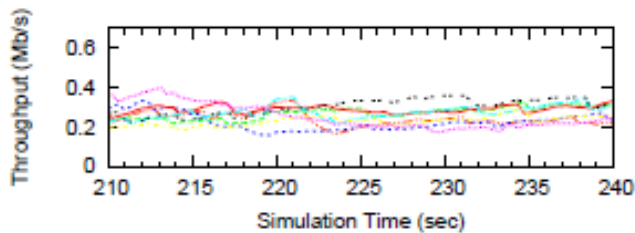
- Smoothness



(a) TCP's Abrupt Sending Rate Change

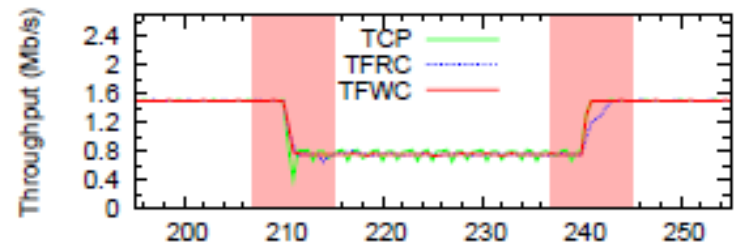


(b) TFRC Smoothness

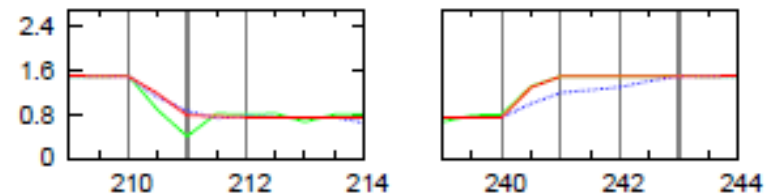


(c) TFWC Smoothness

- Responsiveness



(a) Impulse Response



(b) Zoom-In (high-lighted area)

Relentless Congestion Control (1)

- Packet conservation

- パケットが取り出されたら、新しいパケットを送信する (SIGCOMM 1988)



- パケットがロスしたら、そのロス数分だけ *cwnd* を減少 (1/2は絶対にしない)

$$cwnd = cwnd - \text{loss counts}$$

$$ssthresh = cwnd - \text{loss counts}$$

- 定常状態までは Slow Start で増加

Relentless Congestion Control (2)

- Scalability

- 従来のAIMDは、パケットロスの時間間隔が帯域幅に応じて変化する
- Relentless CCは、帯域幅に依存せず、パケットロス間隔を一定値にする (e.g. $3 \cdot RTT$)
 - CUBIC-TCPの踏襲
- End-to-end \Rightarrow Network assist (baseline AQM)

Relentless Congestion Control (3)

- Relentless CC + Baseline AQM (Active Queue Management)
 - ロスがなければ $cwnd = cwnd + 1$
 - $3 * RTT$ 毎にパケットを廃棄 (by AQM)
 - $cwnd$ からロス数を引いて更新
 - 以上の繰り返し
- TCP-unfriendly paradigm