Extending TFWC for Multimedia Applications

Midori Kato

Faculty of Environmental Information
Keio University
5322 Endo, Fujisawa Kanagawa 252-8520 JAPAN

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science

Advisors:
Hideyuki Tokuda
Jun Murai
Osamu Nakamura
Kazunori Takashio
Jin Nakazawa
Rodney D. Van Meter III
Keisuke Uehara
Jin Mitsugi
Hiroyuki Kusumoto
Keiji Takeda

Abstract of Bachelor's Thesis

Extending TFWC for Multimedia Applications

Multimedia streaming over TCP is common in the Internet. Under condition of scarce network resource, the rate oscillation due to congestion control degrades the user's experience. In addition, the data transmisson of multimedia services under the congested network needs to consider the priority of services depending on the application type. While TFWC supports smooth congestion control for real-time multimedia streaming, it cannot provide a high-enough sending rate to the services that require high throughput rather than smoothness. In this thesis, I propose Tunable TFWC (TTFWC), an extension to TFWC to achieve variable throughput behavior with regard to trade-off between smoothness and the sending rate based on application or user requirements. Unlike many existing works, this proposal is suitable for both throughput-sensive applications and smoothness-sensitive application by using a tuning parameter. I observe that TTFWC with large tuning parameter improves around 10% higher throughput than the original congestion control algorithm in a low-speed network. In a high-speed network, TTFWC throughput with large value of the tuning parameter over 4% compared with the original.

keywords

1. congestion control, 2. responsiveness, 3. high throughput, 4. smoothness, 5. tunability

Midori Kato

Faculty of Environmental Information

Keio University

Abstract of Bachelor's Thesis

Extending TFWC for Multimedia Applications

マルチメディアアプリケーションのための TFWC の拡張

近年のインターネット技術の発展に伴い，TCP を利用したマルチメディアストリーミングが普及しつつある．しかし，ネットワーク資源が乏しいモバイルネットワーク環境などでは，TCP による輻輳制御が転送レートの変動を大きくしてしまい，コンテンツ再生が不安定になる．またネットワークの輻輳が激しい環境では，サービスの品質を考慮した上でサービスタイプ別にデータ通信を行うことが求められる．リアルタイムマルチメディアストリーミングサービス向けに提案された TFWC という輻輳制御アルゴリズムがあるが，転送レートの滑らかさよりも大きさを重視するサービスには利用できない．本研究では， TFWC の拡張を行い，アプリケーション，ユーザの需要にあわせて転送レートの大きさと滑らかさのトレードオフを調節可能にした Tunable TFWC (TTFWC) を提案する．多くの既存研究とは異なり，本研究で提案する輻輳制御アルゴリズムはチューニングパラメータを設定することで，転送レートの大きさと滑らかさのどちらを優先するアプリケーションにも調節できる．実験の結果，既存の輻輳制御アルゴリズムと比較して，低速なネットワークでは 10%以上のスループットの向上，高速なネットワークでは 4%程度のスループットの向上が観測できた．

キーワード

1. 輻輳制御, 2. 応答性, 3. 高スループット, 4. 滑らかさ, 5. 同調性

慶應義塾大学　環境情報学部

加藤　碧

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter describes the background on this research topic. Also, I describe the problems I are concerned with, and the approach to attacking the issues.

1.1 Motivation

The proliferation of high-speed, broadband networks and mobile networks has made multimedia streaming popular. The growth of multimedia streaming increases the demand to distribute high-quality video and audio data in real-time streaming mode. As the usage of multimedia streaming requires more bandwidth than normal web browsing or email services, the design of congestion control multimedia streaming is an important issue in the network, even atop non-congestion-controlled protocols such as UDP.

Congestion control for multimedia streaming is designed to meet two goals; first, it must be TCP-Friendly, sharing the equally bandwidth with competing TCP flows at the bottleneck. TCP transmits the majority of bytes on the Internet, thus compatibility with TCP congestion control is adopted as a reasonable metric to evaluate congestion control algorithms [9]. Second, its throughput must be smooth so that the application achieves constant sending rate, whereas TCP throughput behavior is sawtooth-shaped. [16] shows that burst transmission does not provide a good user experience. Smoothness is the new functionality that overcomes the drawbacks of TCP congestion control.

TCP-Friendly Rate Control (TFRC) [10] is the Internet standard congestion control for streaming media applications. TFRC achieves a smooth sending rate without compromising TCP-friendliness. However, TFRC transmits excessively high sending rate at short Round Trip Time (RTT), depending on timer granularity of the end system. TCP-Friendly Window Control (TFWC) [6] is a window-based variant of TFRC, thus it does not require accurate RTT calculation. TFWC also achieves a smoother sending rate, and is more TCP-Friendly than TFRC. In addition, TFWC can be implemented in various transport protocols, such as TCP and SCTP [21], because they also perform window-based congestion control.

1.2 Challenges and Contribution

Smoothness and high sending rate are trade-off, because the congestion control algorithm for smooth throughput pressures the sending rate not to suddenly increase, even if more bandwidth is available. Indeed, in interactive applications, such as VoIP and video conferencing, data have to be delivered smoothly, because they cannot buffer large amounts of sending and receiving data. However, applications such as on-demand video and audio streaming, and non-interactive live streaming can buffer some data. They require a sending rate as high as possible and a certain level of smoothness, rather than the perfect smoothness that TFRC, TFWC and the other TFRC extensions strive to achieve. In other words, sending rate is more important than smoothness in such applications.

In this thesis, I propose Tunable TFWC (TTFWC), a TCP-Friendly congestion control algorithm that can tune the balance between the sending rate and smoothness. TTFWC supports a wide range of multimedia applications, including those that desire as high sending rate as possible and certain level of smoothness. I define one tuning parameter to control the balance between the sending rate and smoothness. In my experiments, TTFWC achieves variable throughput behavior by changing the tuning parameter value. For example, TTFWC achieves around 10% higher throughput than TFWC in the low-speed network and over 4% higher in the high-speed network. I also demonstrate TTFWC is TCP-Friendly over a wide range of the tuning parameter value.

1.3 Structure of Thesis

The remainder of this thesis is organized as follows: Chapter 2 describe related work and TFWC overview. I describe the design of TTFWC window calculation in Chapter 3. Chapter 4 explains my implementation. I present the results of implementation, over dummynet and the simulation results in Chapter 5. I evaluate TTFWC in terms of responsiveness, throughput, TCP-Friendliness, fairness and smoothness. In this Chapter, I include the discussion for the TTFWC required network. Then, the thesis concludes in Chapter 6.

Chapter 2

Problem Definition

This chapter describes existing proposals which target smooth congestion control and compare these works with my approach.

2.1 Related work

Congestion control for multimedia streaming falls into three categories: Additive Increase Multiple Decrease (AIMD)-based [23, 3], formula-based [20, 10, 6, 13] and Constant Bit Rate (CBR)-based [8]. At the end of this section, I show you another approach as well.

2.1.1 AIMD-based Congestion Control Algorithm

AIMD-based congestion control is RENO extension. In AIMD-based congestion control algorithms, flows achieve lower throughput than competing TCP flows when congestion frequently occurs. In congested network, as the sender decrease little window size after a packet loss, it achieves lower throughput comparing to other types of congestion control. On the other hand, TTFWC provides high sending rate without the amplitude of saw-toothed bandwidth change and decrease.

2.1.2 Formula-based Congestion Control Algorithm

Formula-based congestion control uses TCP throughput equation [14] to estimate `cwnd`. SIMD [20] and TEAR [13] are representatives of the second category. They maintain a high sending rate against frequent congestion. However, SIMD has a slightly higher packet loss rate than other congestion control algorithm, thus the overall performance is poor. Although TEAR achieves a smooth sending rate by using sliding window with the receiver control, its response to congestion is quite slow. TTFWC overcomes these problems, and as shown it in Section 5.1.

2.1.3 CBR-based Congestion Control Algorithm

The CBR-based approach [8] proposes a congestion control algorithm that achieves a smooth sending rate with a new definition of TCP-Friendliness. Since multimedia streaming historically uses UDP, CBR-based congestion control extends existing works to smooth throughput behavior. Flows employing this algorithm are stochastically TCP-Friendly, that is, these flows are fair to aggregates of competing TCP flows. However, these flows cause the throughput of one or some of the competing TCP flows to drastically reduce. In addition, although TTFWC is designed to meet the current definition of TCP-Friendliness, this approach has not been throughly vetted.

2.1.4 New Metric-based Congestion Control Algorithm

STCP [22] also aims at smooth throughput behavior by using some counters such as timeout, fast retransmit and so on. However, its TCP-friendliness is not evaluated and analyzed. In addition, this uses extra ICMP packets to detect congestion, which will be affected by middleboxes or routers [17].

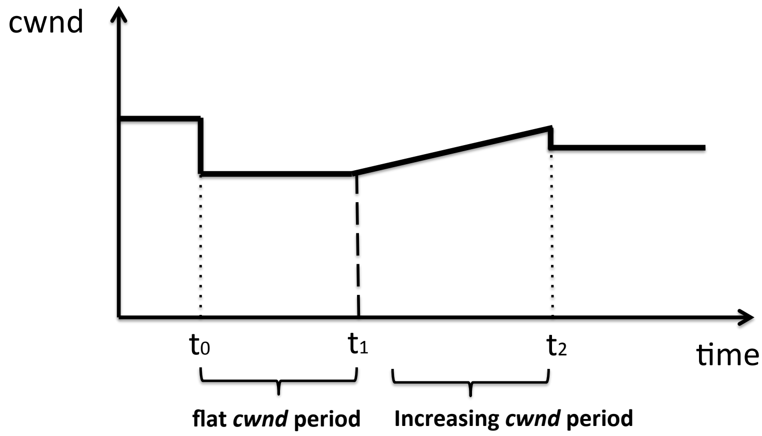


Figure 2.1: Window behavior in TFWC

2.2 TCP-Friendly Window-based Congestion Control

The TFWC sender detects packet loss from ACK packets sent from the receiver. The TFWC sender calculates the congestion window size `cwnd` from the packet loss event rate p based on the TCP throughput equation (Equation 2.1) [14].

$$\texttt{cwnd} = \frac{1}{\sqrt{\frac{2p}{3}} + \left(12\frac{3p}{8}\right) p \left(1 + 32p^2\right)} \tag{2.1}$$

When the sender detects a packet loss event (*i.e.*, loss of one or more packets in the window size) from ACK packets, the sender calculates the loss-event interval, which is the number of packets successfully arriving at the receiver between two loss events. The track of loss-event intervals is called the "loss-event history". From the loss-event history, the sender calculates the Average Loss Interval (ALI), converted to the calculated loss event rate by Equation 2.2.

$$Loss\ Event\ Rate\ (p) = \frac{1}{Average\ Loss\ Interval} \tag{2.2}$$

The calculation of an ALI is important because it impacts the next `cwnd`. When the sender updates its loss event history, two kinds of ALIs are calculated [5]. The first one is calculated with Exponentially Weighted Moving Average (EWMA), which uses the most recent sixteen loss intervals, the last one of which is the number of transmitted packets after the last loss event. The other one is calculated with EWMA which uses sixteen loss intervals, not counting the number of transmitted packets after the last loss event in (*i.e.*, use an older loss interval than the first one). The sender adopts the larger one as the estimated ALI to derive `cwnd`. We call the interval that the first one is chosen a "**flat `cwnd` period**", and the interval that the other one is chosen an "**increasing `cwnd` period**".

Fig. 2.1 illustrates TFWC `cwnd` behavior. A loss event happens at time t_0 and t_2. Between time t_0 and t_1, the flat `cwnd` interval is applied. Between time t_1 and t_2, the increasing `cwnd` interval is applied, thus the `cwnd` increases little by little.

Chapter 3

Tunable TCP-Friendly Congestion Control

This chapter describes my proposal congestion control algorithm for multimedia streaming named TTFWC, which is composed of three algorithms.

3.1 TTFWC Overview

In this section, I explain the design of TTFWC and the effect of the tuning parameter *tp* that varies between 0.0 and 1.0. I call the range of *tp* less than 0.5 the "smooth mode" and more than or equal to 0.5 the "responsive mode". This is because, the former gives relatively high priority to smoothness, whereas the latter gives that to throughput. More importantly, TFWC uses two different window calculations, including different numbers of loss intervals. I make three modifications in TFWC `cwnd` calculation to set the tuning parameter: First is that the sender increases its `cwnd` more aggressively at the increasing `cwnd` period, second is that the sender changes the number of loss interval in the ALI calculation from this value, and third is `cwnd` calculation in low bandwidth.

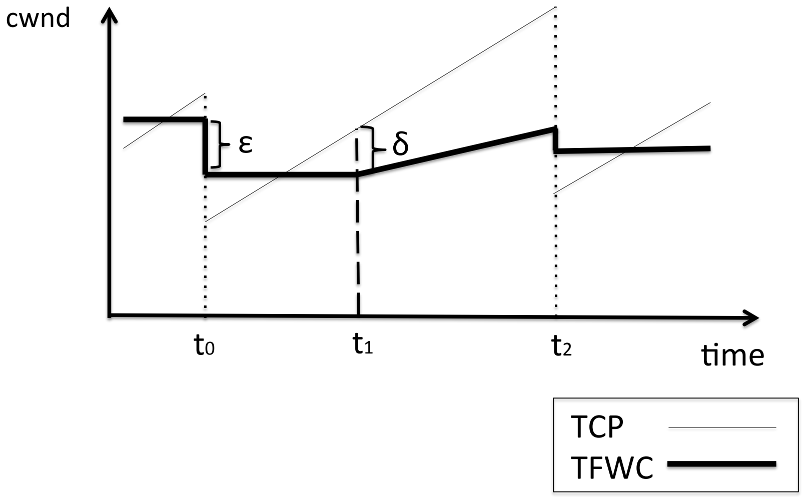


Figure 3.1: TCP and TFWC `cwnd` behavior

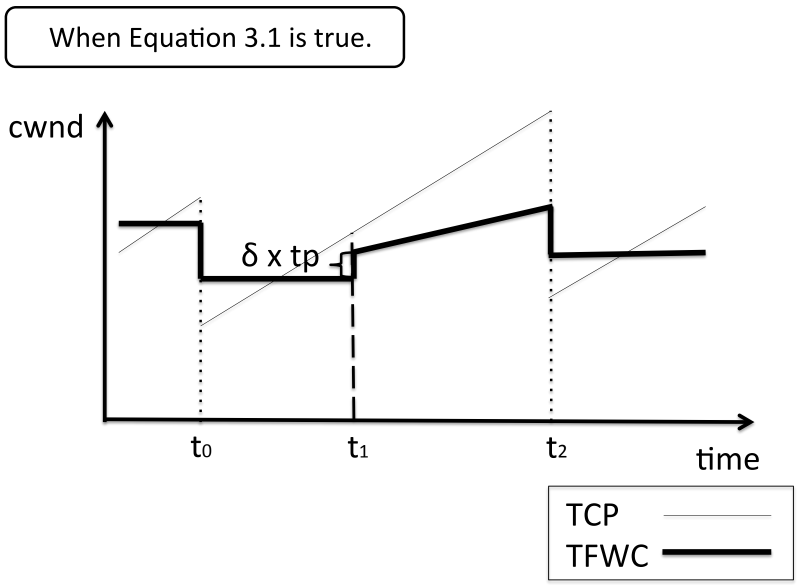


Figure 3.2: TTFWC `cwnd` behavior (When Eq.3.1 is true)

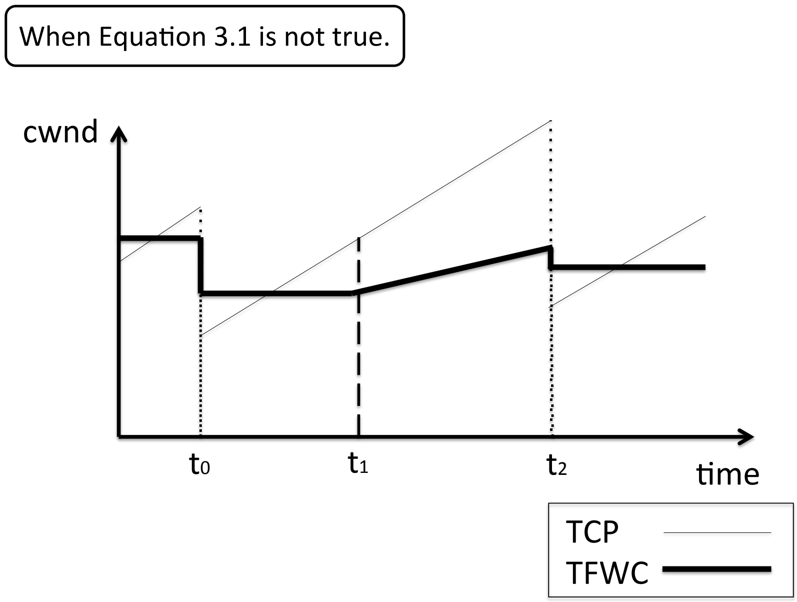


Figure 3.3: TTFWC `cwnd` behavior when (When Eq.3.1 is false)

3.2 Base `cwnd` Update at the Beginning of the Increasing `cwnd` Period

In order to achieve higher throughput than the original TFWC, TTFWC increases the base `cwnd` at the beginning of the increasing `cwnd` period. The rationale is that TFWC does not increase the congestion window quickly when the calculated TCP-Friendly congestion window is larger to keep the smooth sending rate as shown as δ in Fig. 3.1. This is also the reason why TFWC achieves lower throughput than TCP (see Sec. 5.1 with Fig. 5.6 for simulation results).

In the responsive mode (*i.e.*, $0.5 \leq tp < 1.0$), TTFWC increases the base `cwnd` by $tp \times \delta$, where tp is the tuning parameter and δ is the difference between the current TCP `cwnd` and the calculated TCP-Friendly `cwnd` (Fig. 3.2 illustrates δ at time t_1). Hence, the responsive mode achieves the higher throughput based on the tuning parameter.

In the smooth mode (*i.e.*, $0 \leq tp < 0.5$), TTFWC increases the base `cwnd` by $tp^2 \times \delta$. Since the smooth mode is sensitive to throughput oscillation caused by the transient congestion, I conservatively increase the base `cwnd`.

In Fig. 3.2, the integral of TCP `cwnd` and TFWC `cwnd` are equal through the time between t_0 and t_1. When a packet is lost before the time t_1, the TFWC sender receives higher throughput than TCP. Hence, I limit the opportunity to increase the base `cwnd` to prevent unfair utilization of the bottleneck link. I increase the base `cwnd` only when Equation 3.1 is true on the ALI calculation:

$$\epsilon < \frac{1}{\sum_{i=1}^{n} weight_i} \tag{3.1}$$

Let ϵ be the `cwnd` decrease fraction after the packet loss event, as illustrated in Fig. 3.1, and $weight_i$ be the i-th EWMA weight. Fig. 3.2 and 3.3 show TTFWC `cwnd` behavior when Equation 3.1 is true and not true, respectively. If the value ϵ of Equation 3.1 is not true after a packet loss, the sender uses its window size as is the case with TFWC `cwnd` behavior.

3.3 Change of the Number of Loss Intervals in the ALI Calculation

I change the number of loss intervals from sixteen to eight in the responsive mode. The larger number of loss intervals makes `cwnd` decrease more slowly. Hence, if I use the same number of loss intervals as the original TFWC, the TTFWC flow becomes overly aggressive, because it increases the base `cwnd` at the beginning of the increasing `cwnd` period. When the number of loss intervals is halved, the sender receives the convergence to the ideal window size up to twice as fast. Since the total EWMA weight in an ALI calculation is halved, the sender decreases its `cwnd` by up to half after the sender experiences an extremely short loss interval. After the sender experiences an extremely long loss interval, it increases its `cwnd` up to twice. I note that throughput in responsive mode TTFWC using 8 loss intervals in its ALI calculation is still much smoother than TCP (See Fig. 5.9 in Sec. 5.1).

I adopt the same number of loss intervals as the original TFWC in the smooth mode. If I apply the tuning parameter in the same way as in the responsive mode, the smooth mode TTFWC flow is obviously overly aggressive as the tuning parameter is larger. Hence, I have to conservatively apply the tuning parameter to the base `cwnd`. My design choice is applying the tuning parameter to the base `cwnd` exponentially.

3.4 RENO Operation when `cwnd` is Less than two

TTFWC performs RENO operation [7, 11] when `cwnd` falls below two. While the original TFWC uses rate control, I adopt RENO operation to prevent from high CPU load as a result of `cwnd` calculation. I will describe the CPU processing time for the main functions in the evaluation.

Chapter 4

Implementation

This chapter describes the TTFWC implementation.

I implemented TTFWC as a TCP congestion control algorithm in the Linux 3.0.0 kernel. The implementation consists of around 900 total lines of code, structured as a loadable kernel module. Thus, the application and the system administrator can enable TTFWC with ease. As TTFWC is a sender-only algorithm, I do not need modification to the receiver so long as the receiver behaves as well as regular TCP. We enable TTFWC by a `sysctl` parameter. The subsequent sections describe how to perform TTFWC to match the congestion avoidance, slow start, and loss recovery behavior.

Fig. 4.1 shows the logic of a TCP socket after the connection is established on a sender. The TCP option structure must be initialised at this time and a call to TCP register congestion control is made. Then, every time the machine receives ACK, it checks the state of the socket in the function `tcp_ack()`. When the function `tcp_ack_is_dubious()` returns its state is normal, the machine enters the congestion avoidance phase. At this time, the soceket calls congestion control function `cong_avoid()`. In the Linux kernel, when the machine encounters a loss event, three types of states are prepared for loss recoveries. TCP_CA_Recovery is set for fast transmission. TCP_CA_CWR means that the machine raises congestion inside NIC, or ECN and etc. The machine handles loss recovery depending on the loss event

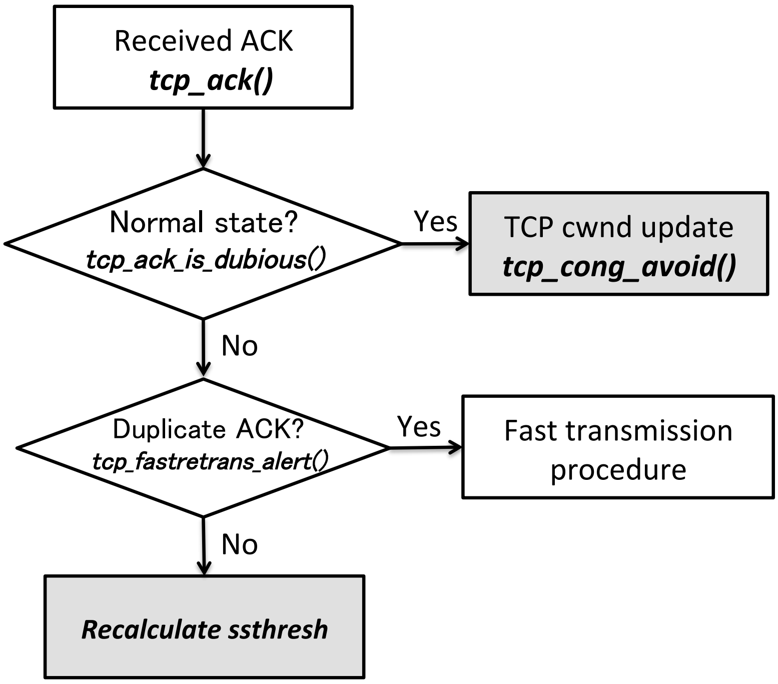


Figure 4.1: The TCP procedure diagram

type. In the Linux implementation, the function `tcp_fastretrans_alert()` divides into fast retransmission and other procedure.

Fig. 4.2 shows the variables in the TTFWC structure with the definitions and Fig. 4.3 shows the TTFWC procedures. The TTFWC structure and TTFWC procedures' definition are employed only if the kernel is compiled with the TTFWC support enabled.

```
struct ttfwc {
    u32 tcp_cwnd;     // TCP cwnd calculated by TTFWC procedures
    u32 is_reno;      // flag when it is in RENO phase
    u32 reno_cwnd;   // RENO cwnd estimation
    u32 alpha_cwnd;   // additional cwnd depending on a parameter
    u32 li_dif;       // the number of packets until the ALI is over
    u32 li_counter;   // the counter of loss intervals
    u32 li_mean;      // the weighted ALI
    struct ttfwc_loss_hist *lh; // array to stock loss intervals
};
```

Figure 4.2: *ttfwc_structure*

```
static struct tcp_congestion_ops tcp_ttfwc __read_mostly = {
    .init       = ttfwc_init,        // variables initiate
    .ssthresh   = ttfwc_recalc_ssthresh,    // ssthresh calculation
    .cong_avoid = ttfwc_cong_avoid,  // cwnd update
    .undo_cwnd  = ttfwc_undo_cwnd,   // a false loss detected
    .owner      = THIS_MODULE,
    .name       = "ttfwc",
};
```

Figure 4.3: *the TTFWC procedures' definitions*

4.1 Slow Start and Congestion Avoidance

`ttfwc_cong_avoid()` updates the `cwnd` every time a packet is acknowledged. Fig. 4.4 displays congestion avoidance phase of TTFWC. TTFWC slow start and timeout behaviors are the same as RENO. The sender applies TTFWC operation after the slow start phase.

Upon receiving an ACK from the receiver, the TTFWC sender performs the following update. The TTFWC sender calls RENO operation in the slow start phase or when `cwnd` is less than two (Lines 9-16). When the sender is out of RENO mode, it estimates RENO `cwnd` (Lines 17-26). In this scope, the sender estimates based on the window behavior as described Chapter 3. After the number of packets is over the ALI, it increases `cwnd` (Lines 27-). In this time, the TTFWC sender adds extra `cwnd` into base `cwnd` when a tuning parameter is set (Lines 14-17).

The TTFWC sender after increasing `cwnd` period starts to calculate `cwnd` by using the ALI. When the ALI is over the estimation of next loss interval, the sender add extra `cwnd` if the tuning parameter is set more than zero (Line 29-30). I adopt inaccurate ALI estimation (Line 32 and 34). In the substitute calculation, the smooth mode is a little smoother than accurate ALI calculation because the accurate ALI is increased when the sender receives 5.3 ACKs. As well, the responsive mode is more aggressive than accurate ALI calculation because the accurate ALI is increased when the sender receives 2.8 ACKs.

I find that the exact procedure of the ALI calculation wastes processing time. The `cwnd` update with accurate ALI calculation sometimes stops data transmission due to the overhead, TTFWC algorithm in the Virtual Machine (See Section 5.2.3). Through the experience, the limitation of the machine resources affects data transmission. I recommend approximate ALI calculation to minimize the overhead in the congestion avoidance phase. I will also comment about ALI calculation in Section .

I explain the specific functions as below. *tcp_is_cwnd_limited* shown in Lines 7-8 prevents burst data transmission if the current `cwnd` is not much. *ttfwc_invert_imean* executes to be the inverse number of the ALI (Line 38).

4.2 TTFWC Loss Recovery Behavior

The TTFWC sender enters into *ttfwc_recalc_ssthresh* when the SACK or ACK proof contains a loss. Fig. 4.2 shows the procedure of TTFWC loss recovery.

For the request after a loss event, TTFWC performs the following operation: When its `cwnd` is less than two, the TTFWC sender skips the following operation and is in RENO operation (Line 7-10). When the sender detects the first loss event, it calculates a pseudo-loss interval, loss event rate and the number of packets until the increasing `cwnd` period by using the Equation (Line 11-17). In this scope, the sender estimates loss interval by using Equation 2.1 and sets the number of packets until the ALI is over as `li_dif`. After the first loss event, the sender updates loss intervals and calculates the ALI and the next `cwnd`. When the calculated `cwnd` is more than two, the sender reverses TTFWC (Line 19-22). Then, the sender returns the current `cwnd` as `ssthresh` (Line 16).

Fig. 4.6 indicates how to estimate RENO `cwnd` and how to calculate the additional `cwnd` in the function `ttfwc_alpha_cwnd`. The sender is halved RENO `cwnd` and compares to next TTFWC `cwnd` (line 4). A part of lines 13-14 indicates the exception for the estimation. The sender enters this scope when its state is TCP_CA_CWR because the `cwnd` is set 1. When next TTFWC `cwnd` is over the halved RENO `cwnd`, the sender checks Equation 3.1 (Line 10-12). Then, if Equation 3.1 is true shown in line 17 and 19, the sender calculates additional `cwnd`. If not, additional `cwnd` is reset.

```
static void ttfwc_cong_avoid(struct sock *sk, u32 ack, u32 in_flight)
{
   struct tcp_sock *tp = tcp_sk(sk);
   struct ttfwc *ca = inet_csk_ca(sk);
   u32 li_mean, p, cwnd;

   if (!tcp_is_cwnd_limited(sk, in_flight))
       return;
   if (ca->is_reno || ttfwc_lh_length(ca) < 1){
        ca->lh->li_length[0]++;
        if (tp->snd_cwnd <= tp->snd_ssthresh)
            tcp_slow_start(tp);
        else
            tcp_cong_avoid_ai(tp, tp->snd_cwnd);
        return;
    }
    if (ca->lh->li_length[0] >= ca->li_dif/2) {
        if (ca->lh->li_length[0] == ca->li_dif/2)
            ca->reno_cwnd = ca->tcp_cwnd;
        if (tp->snd_cwnd_cnt >= ca->reno_cwnd) {
            if (ca->reno_cwnd < tp->snd_cwnd_clamp)
                ca->reno_cwnd++;
            tp->snd_cwnd_cnt = 0;
        } else
            tp->snd_cwnd_cnt++;
    }
    if (++ca->lh->li_length[0] > ca->li_dif) {
       u32 count = ca->lh->li_length[0] - ca->li_dif;
       if (count == 0 && sysctl_ttfwc_tuning_p > 0)
            tp->snd_cwnd += ca->alpha_cwnd;
         if (sysctl_ttfwc_tuning_p <= 5){
           if (count%6 == 0)  goto common;
         } else {
           if (count%3 == 0)  goto common;
         }
    }
   common:
     p = ttfwc_invert_imean(++ca->li_mean);
     cwnd = ttfwc_calc_x(p);
     if (cwnd + ca->alpha_cwnd > tp->snd_cwnd)
       tp->snd_cwnd++;
}
```

Figure 4.4: *ttfwc_cong_avoid function*

```
static inline u32 ttfwc_recalc_ssthresh(struct sock *sk)
{
  const struct tcp_sock *tp = tcp_sk(sk);
  struct ttfwc *ca = inet_csk_ca(sk);
  u32 next_cwnd, p;

  if (tp->snd_cwnd < 2){
    next_cwnd = tp->snd_cwnd;
    goto out;
  }
  if (++ca->li_counter == 1) {
    ca->tcp_cwnd = tp->snd_cwnd >> 1;
    ca->lh->li_length[1] = ca->li_mean = pseudo_li_calc(ca->tcp_cwnd);
    ca->lh->li_length[0] = 1;
    ca->li_dif = ca->li_mean/100;
    ca->li_counter++;
    goto out;
  } else {
    if (ca->is_reno > 0)
      ca->is_reno = 0;
    ttfwc_loss_hist_update(ca);
    ca->li_mean = ttfwc_avg_loss_interval(ca);
    if(ca->li_mean > 0) {
      p = ttfwc_invert_imean(ca->li_mean);
      next_cwnd = ttfwc_calc_x(p);
    } else
      next_cwnd = 0;
    if (sysctl_ttfwc_tuning_p == 0) goto out;
    else{
      if (ca->tcp_cwnd <= ca->reno_cwnd)
        ttfwc_alpha_cwnd(ca, ca->tcp_cwnd, next_cwnd);
    }
  }
  out:
    ca->tcp_cwnd = next_cwnd;
    if (cwnd <= 2 && ca->is_reno == 0) {
        cwnd = 2;
        ca->li_dif = 0;
        ca->is_reno = 1;
    }
    return max(ca->tcp_cwnd, 2U);
}
```

Figure 4.5: *ttfwc_recalc_ssthresh function*

```
static inline void ttfwc_alpha_cwnd
          (struct ttfwc *ca, u32 cwnd, u32 next_cwnd)
{
 ca->reno_cwnd = ca->reno_cwnd >> 1;
 if (ca->reno_cwnd > next_cwnd)
   ca->alpha_cwnd = 0;
 else{
   u32 dthresh;
   if (ca->alpha_cwnd > 0)
     dthresh = (u32) next_cwnd*100/(cwnd-ca->alpha_cwnd);
   else
     dthresh = (u32) next_cwnd*100/cwnd;
   if (dthresh >= 100)
     ca->alpha_cwnd = 0;
   else {
     u32 thresh = next_cwnd - ca->reno_cwnd;
     if (sysctl_ttfwc_tuning_p >= 5 && dthresh > 72)
       ca->alpha_cwnd = (u32) thresh*sysctl_ttfwc_tuning_p/10;
     else if (sysctl_ttfwc_tuning_p > 0 && dthresh > 81)
       ca->alpha_cwnd = (u32) thresh*pow(sysctl_ttfwc_tuning_p, 2)/100
     else
       ca->alpha_cwnd = 0;
   }
 }
}
```

Figure 4.6: ttfwc_alpha_cwnd

Chapter 5

TTFWC Performance Evaluation

This chapter evaluates the efficiency of TTFWC via simulation and experiment with the implementation. First, I describe simulation testbed and results. Second, I evaluate my implementation in the Linux kernel.

5.1 Simulation Results

I evaluate TTFWC in terms of responsiveness, throughput and TCP-Friendliness, and smoothness through ns-2 simulation [18]. Through all experiments, I execute using three values of the tuning parameter value 0.0, 0.4 and 0.9. All TCP flows leveraged in experiments are SACK TCP [12]. The tuning parameter value 0.4 and 0.9 are representatives of the smoothness-sensitive and the throughput-sensitive node of TTFWC. When the tuning parameter is 0.0, TTFWC is equivalent to the original TFWC shown in Equation 3.1. I denote TTFWC with the tuning parameter a as TTFWC(a).

5.1.1 Responsiveness

I run TTFWC flows on a environment with dynamic changes of available bandwidth to evaluate responsiveness. In the simulation, I run two TTFWC flows over the 15 Mbps bottleneck link. RTTs of the two TFWC flows are 8.4 and 8.8 ms. At time x seconds, 10 Mbps CBR flows are injected, remain for x seconds, and leave. Each trial repeats this

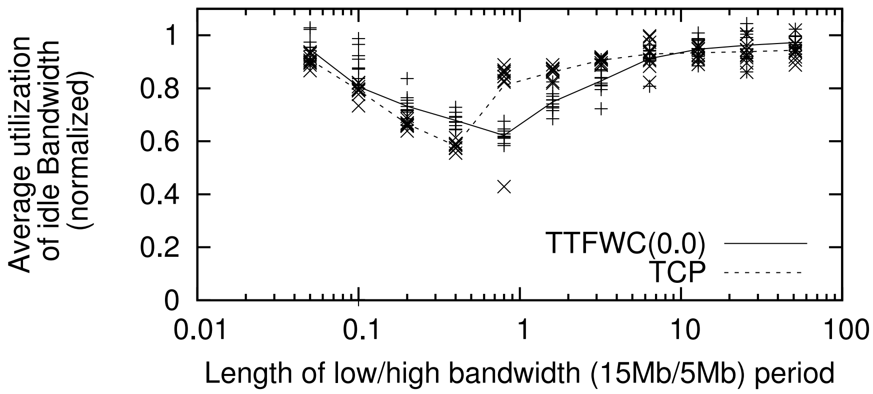


Figure 5.1: TTFWC(0.0) on constant CBR flows

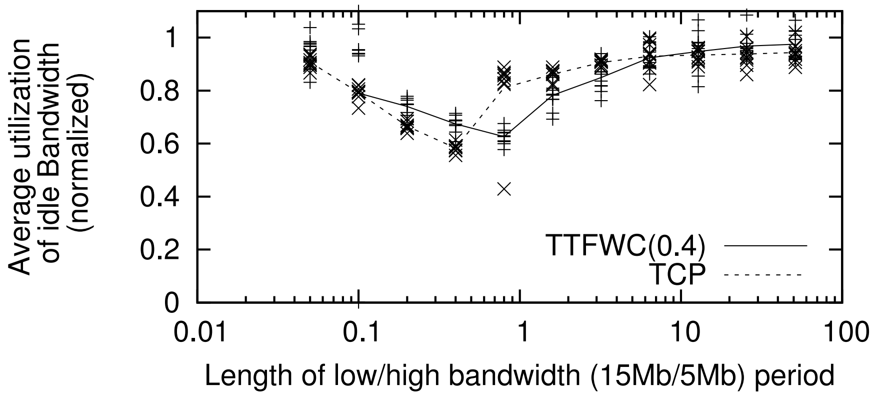


Figure 5.2: TTFWC(0.4) on constant CBR flows

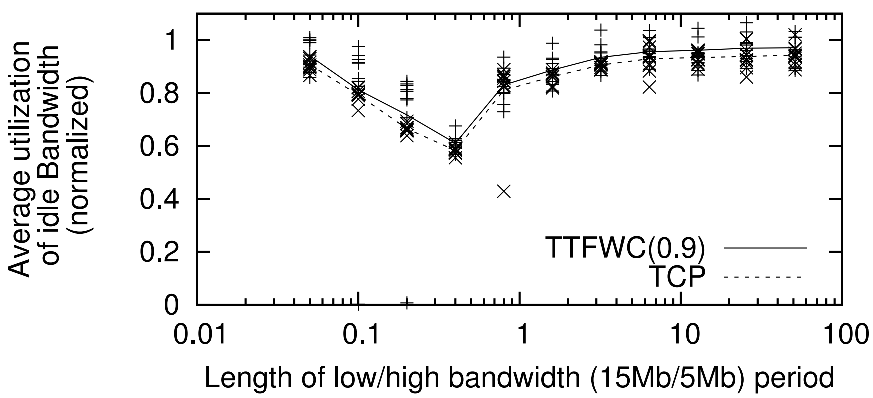


Figure 5.3: TTFWC(0.9) on constant CBR flows

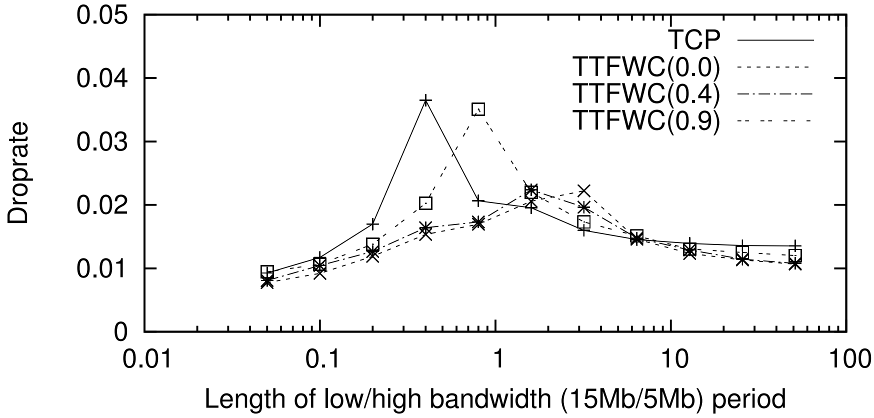


Figure 5.4: The packet loss rate in experiments in Fig. 5.1 - 5.3

interval for 200 seconds, and x varies between 0.05 and 51.2. For comparison, I conduct the equivalent experiments on TCP instead of TTFWC. These scenarios are called SlowCC [4], being widely used to evaluate congestion control behavior on ON/OFF CBR flows.

Fig. 5.1 - 5.3 show utilization of the idle bandwidth of TTFWC and TCP flows. The horizontal axis and the vertical axis represent x and utilization of the idle bandwidth, respectively. Values are normalized, thus 1 means idle bandwidth is fully utilized by TTFWC or TCP flows. TTFWC(0.0) (*i.e.*, original TFWC) and TTFWC(0.4) flows are up to approximately 20% less utilization of the idle bandwidth than TCP flows. On the other hand, TTFWC(0.9) flows, more responsive in the TTFWC design, utilize almost equal idle bandwidth to TCP flows. I conclude from these results that the TTFWC with the greater tuning parameter responds to change of the idle bandwidth more quickly. As the result, they achieved higher throughput than TTFWC flows with the lower tuning parameter. I observed higher packet loss rate with the greater tuning parameter than with the smaller tuning parameter. Fig. 5.4 plots the packet loss rate during the experiments in Fig. 5.1 - 5.3. I believe the reason is that the TTFWC mode toggles between the flat `cwnd` period and `cwnd` increase period frequently. However, I believe this problem is minor, because it happens in a very small range

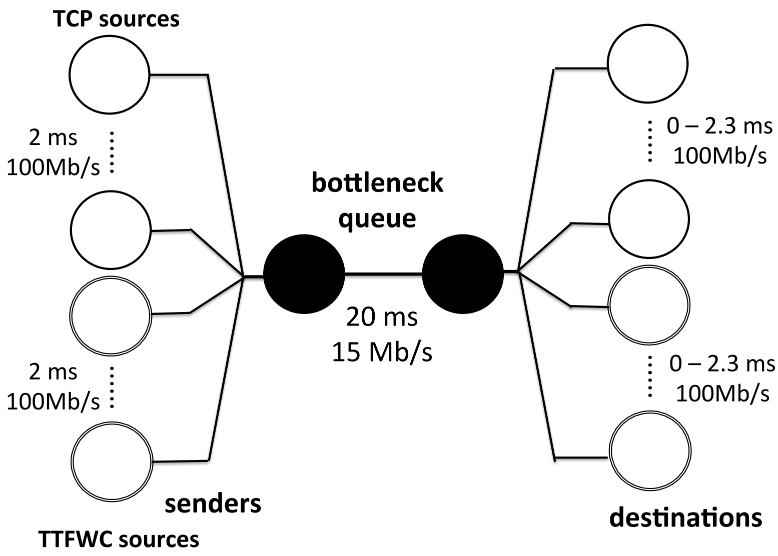


Figure 5.5: Simulation Topology

of congestion patterns.

5.1.2 Throughput and TCP-Friendliness

I evaluate the TCP-Friendliness of TTFWC. I use the dumbbell topology in the simulation shown in Fig. 5.5. I set the same number of TCP and TTFWC flows, which varies from 1 to 64. The bottleneck bandwidth fixes 15 Mbps with a 20 ms delay and droptail queuing. The sender-side access links have 100 Mbps bandwidth with 1 ms delay. The receiver-side access links have 100 Mbps bandwidth with 0 - 2.6 ms delay.

Fig. 5.6 - 5.8 show the throughput of TTFWC and TCP flows. The horizontal axis represents the number of TTFWC and TCP flows. The vertical axis represents the normalized throughput. If the TCP flow's value and TTFWC flow's value overwrap, it means that TCP flows and TTFWC flows achieve equal throughput. The bottom of each figure is the packet loss rate on each trial. From these result, TTFWC is TCP-Friendly regardless of the tuning parameter value, although TTFWC flows achieve a little lower throughput than TCP flows. I observed TTFWC flows with the larger tuning parameter achieve higher throughput than

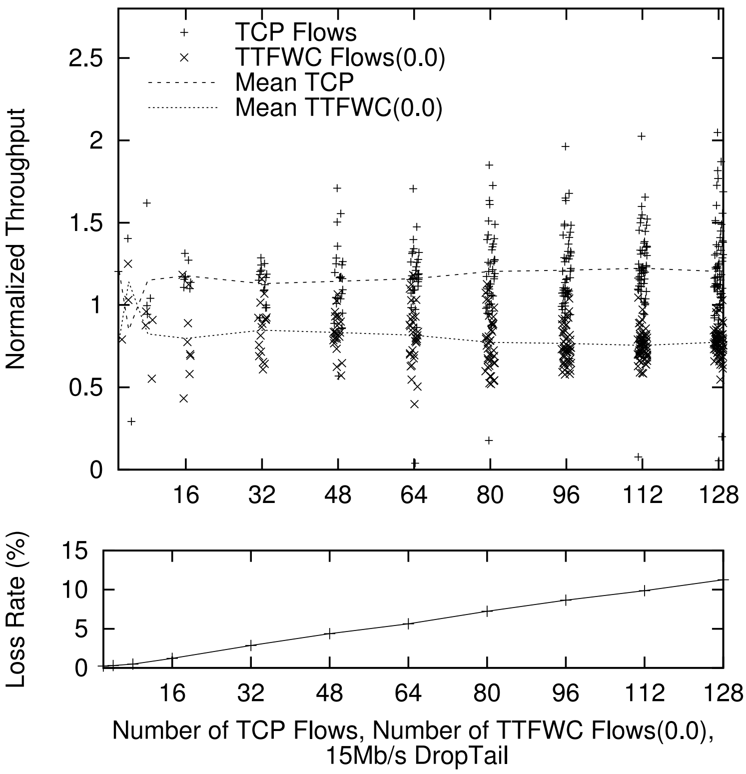


Figure 5.6: TTFWC(0.0) and TCP throughput

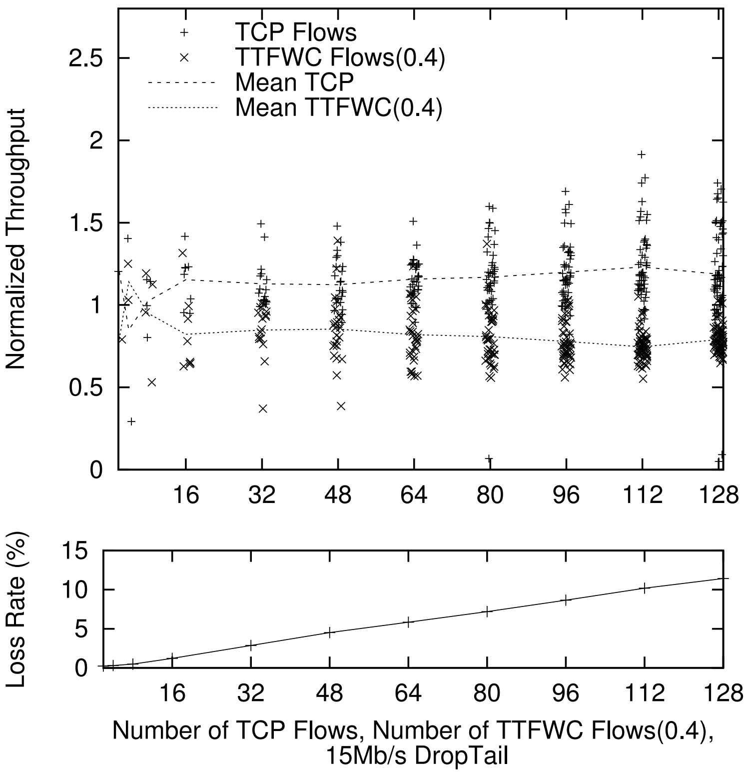


Figure 5.7: TTFWC(0.4) and TCP throughput

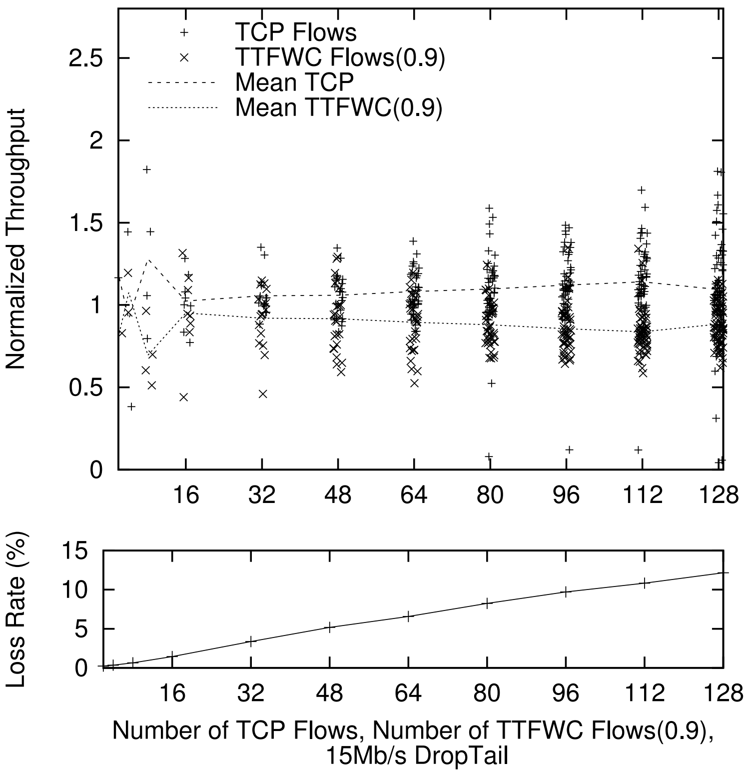


Figure 5.8: TTFWC(0.9) and TCP throughput

those with the smaller tuning parameter (*i.e.*, closer to the original TFWC). TFWC(0.4) flows achieve approximately 2% higher throughput than TFWC(0.0) or the original TFWC. TFWC(0.9) flows achieve approximately 10% higher throughput than TFWC(0.0). Therefore, I conclude that TTFWC certainly achieves the higher throughput than the original TFWC with the larger tuning parameter.

5.1.3 Smoothness

Finally, I evaluate TTFWC smoothness. I adopt the same dumbbell topology as that for evaluation in Section 5.1.2. The number of TTFWC and TCP flows are set to eight. I adopt this value to observe steady-state behavior of both TCP and TTFWC flows, avoiding an unacceptable number of timeouts.

Fig. 13 shows throughput behavior of these eight TTFWC flows. The horizontal axis represents the time, and the vertical axis represents throughput. The bottom graph shows behavior of TCP flows competing with TTFWC(0.9) flows for comparison with TTFWC behavior.

I observe almost the same throughput behavior between TTFWC(0.0) and TTFWC(0.4) as the difference of the tuning parameter value is small. On the other hand, I observe that TTFWC(0.9) flows exhibit larger oscillation as they respond to the packet loss more aggressively. However, their behavior is much smoother than that of TCP flows (See top figure). From these results I conclude TTFWC behaves as intended in its design, which achieves higher throughput with the larger oscillation when the tuning parameter value is greater.

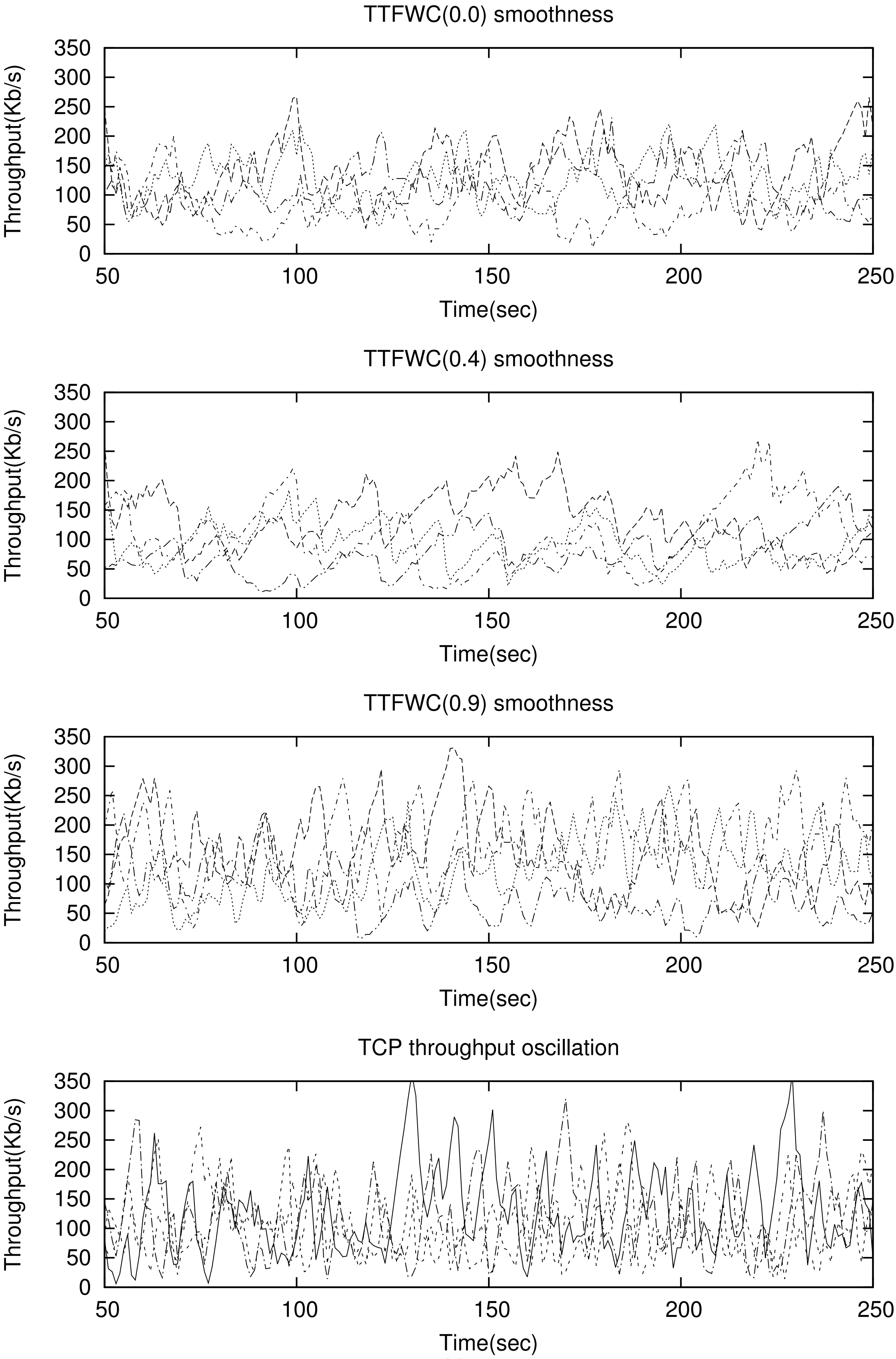


Figure 5.9: Throughput behavior of TCP, TTFWC(0.0), TTFWC(0.4) and TTFWC(0.9)

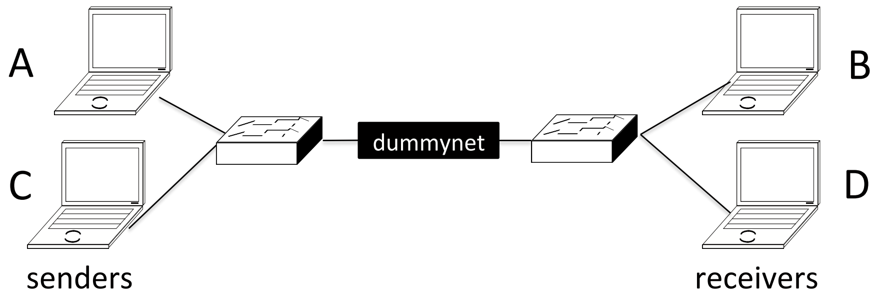


Figure 5.10: Experimental setup of all tests

5.2 Experimental Results

In this section, I evaluate TTFWC with the implementation. Fig. 5.10 shows the experimental setup. I use four machines: two sender nodes marked A and C and two receiver nodes marked B and D. These senders share a bottleneck link that is regulated over `dummynet` [19]. One of the sender nodes is equipped with an Intel Core 2 Duo 1.6 Ghz CPU and 4GB memory. The other node is equipped with an Intel Core i5-2520M CPU and 4GB of memory. Throughout all the experiments in this section, I set the bandwidth of the bottleneck link to 10 Mbps and 250 Mbps to emulate relatively low- and high-speed networks.

I measure TTFWC and TCP throughput by `iperf` [2] in the experiments except for Section 5.2.3. The sender continues to send 16Kbytes data during 200 seconds with the configuration that produces five-second average of its throughput. I observe processing time by using `ftrace` tool [1] in Section 5.2.3.

5.2.1 Fairness

I analyze the protocol fairness between competing TTFWC flows by using two kinds of viewpoints. First, I measure TTFWC throughput with different RTT. When flows with different RTT compete, more bandwidth is unfairly allocated to the flow having smaller RTT. Thus, I can see that a user with longer RTT may not be able to obtain sufficient bandwidth by the congestion control. Second, I measure TTFWC throughput on a network that competes same protocol with same RTT. In this test, we check the fair allocation of the bandwidth for each protocol, when all the flows in the network uses the same congestion control. In these experiments, I inject three parallel flows for each machine and set their delay of 16 ms, 32 ms, 120 ms, 240 ms and 320 ms.

I explain the index of fairness as below. Let $x_i(k)$ be the average throughput of flow i in the five-second period k. If the same link pressure sustains the composition of flows during $k = 1, ..., m$ and changes when k is $m+1$, I define that [1, m] is the maximum-length interval over fixed link state. Suppose there are n active flows in this interval, indexed by $i = 1, ..., n$, let

$$\overline{x_i} = \frac{1}{m}\sum_{k=1}^{m} x_i(k) \tag{5.1}$$

Jain's fairness index [15] for the interval [1, m] is defined,

$$F = \frac{(\sum_{i=1}^{n} \overline{x_i})^2}{n(\sum_{i=1}^{n} \overline{x_i}^2)} \tag{5.2}$$

In this case, F = 1 indicates the ideal condition for the network.

Fairness with Different RTT

I measure the protocol unfairness between competing flows with different RTT. I show a brief scenario as below. The node A transmits three TTFWC flows to the node B. RTT between the node A and B is 120 ms and its value is static. The node C also transmits three TTFWC flows to the node D. To measure RTT unfairness of TTFWC, I change RTTs between the node C and D, and observe throughput of the flows transmitted from the node A and C. I performed equivalent tests with RENO and CUBIC instead of TTFWC, then compare each of fairness by using Equation 5.2.

Fig. 5.11 plots the results of experiments. The horizontal axis of the figure shows RTT in the network and the vertical axis shows fairness. TTFWC achieves inversely proportional throughput to RTT and the results mostly support this observation on a range of the tuning parameter value of TTFWC.

From the figures, we can see that the fairness of TTFWC little differs from tuning parameter values or from other congestion control in the low-speed network. This results indicate that TTFWC remains fairness in the low-speed network.

In the high-speed network, TTFWC marks the same result as well. Interestingly, TTFWC(0.9) over the short RTT is significantly fairer than RENO and CUBIC. I believe that idle bandwidth in the network affects the result. Comparing to low-speed network, the flows generates idle bandwidth between flows with different RTT in the high-speed network. In addition, as TTFWC flows with short delay are more aggressive to available bandwidth than that with long delay. Thus, TTFWC flows with different delay achieve fairer allocation of bandwidth than TTFWC with smaller tuning parameter value.

Fairness with same protocol

In this subsection, I evaluate the fairness between active TTFWC flows. I run three TTFWC flows for each node and inject 10Mbps bottleneck link. The competing flows are set with the configuration of same congestion control and with same RTTs. Then, I change the bandwidth of the bottleneck link to 250Mbps for the equivalent experiment.

Fig. 5.12 plots the protocol fairness in the low- and high-speed network. The horizontal axis and the vertical axis show RTT in the network and the index of fairness, respectively. When I see the fairness of TTFWC in the low-speed network, TTFWC marks less fairness than CUBIC or RENO when the propagation delay is long. I assume that, when two kinds of window behaviors of TTFWC are mixed in the link, the different throughput between competing flows grows up. On the other hand, when the propagation delay is short, the fairness of TTFWC is same with others.

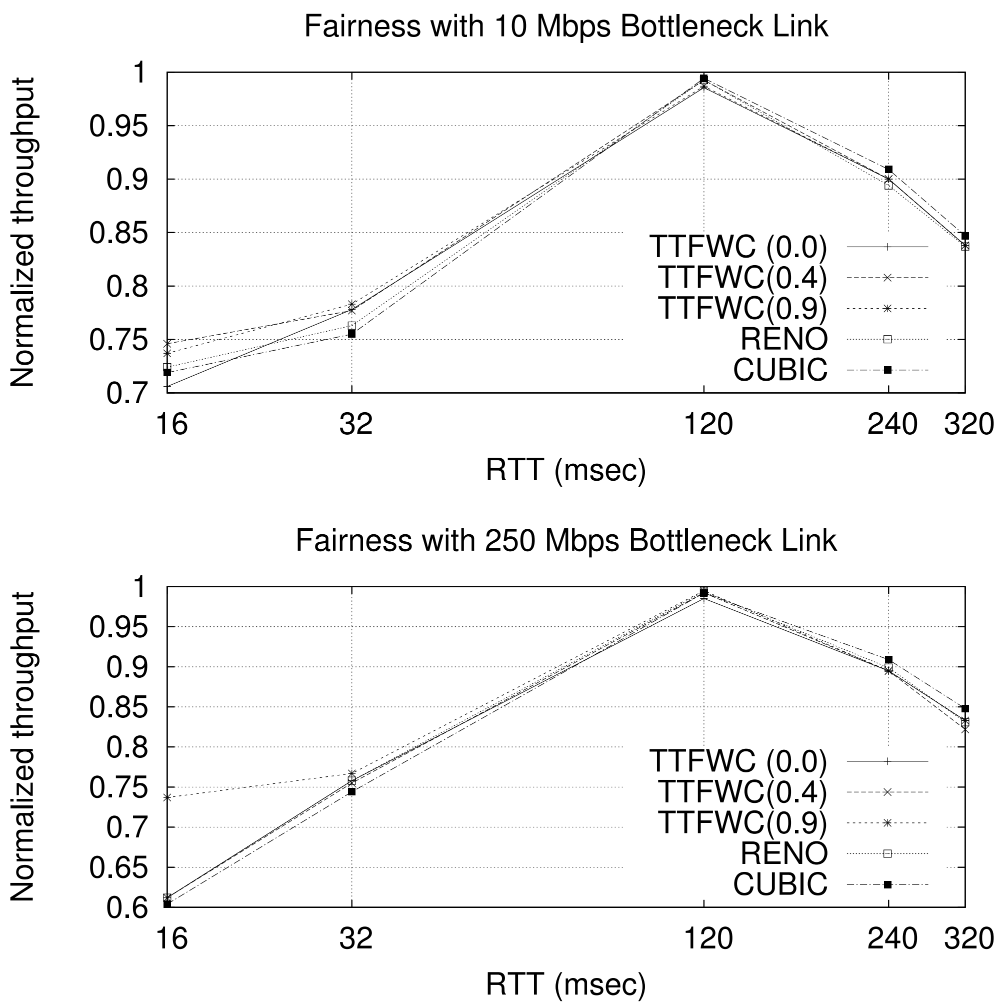
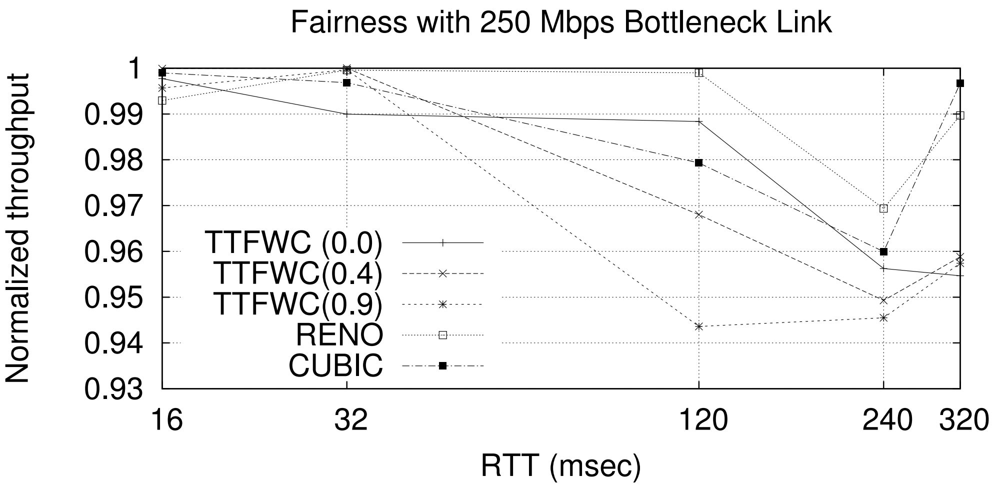


Figure 5.11: Fairness with six active flows three flows with 120 ms delay and the other three flows with varying delays. Results are shown for 10 Mbps (top) or 250 Mbps (bottom) bottleneck bandwidth.



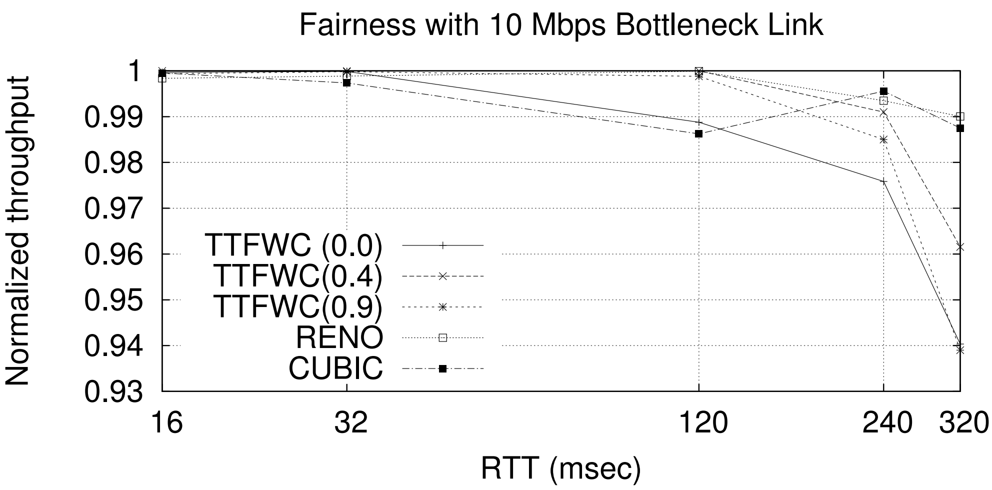


Figure 5.12: Fairness of active six flows under symmetric conditions (same propagation delay, shared bottleneck link, same congestion control algorithm). Results are shown for 10Mbps (top) or 250Mbps (bottom) bottleneck bandwidth.

The same observation with high-speed network can be observed in in the low-speed network. We do not observe the difference between tuning parameter values and TTFWC keeps fairness only when the propagation delay is short.

It is difficult to avoid protocol unfriendliness and insufficient throughput performance on a symmetric network. This is the limitation of TTFWC because TTFWC design assumes to get high throughput in a network that RENO flows competes in the link. I will address this issue in future work.

5.2.2 Throughput and TCP-Friendliness with different RTT

In this subsection, I describe TTFWC throughput and TCP-Friendliness. Unlike the simulation results, I set three TTFWC flows and three RENO flows in the link with different RTTs. Fig. 5.13 - 5.15 show the results when the bandwidth of the bottleneck link is 250Mbps and Fig. 5.16 - 5.18 plot when the bandwidth is 10Mbps. The horizontal axis and the vertical axis represent RTT and normalized throughput, respectively.

TTFWC gets high throughput in proportion to tuning parameters. When I compute average throughput, TTFWC with great tuning parameter value increases 4% of original TFWC throughput in the low-speed network and TTFWC with large tuning parameter increases 10% of TTFWC throughput in the high-speed network.

From the viewpoint of TCP-Friendliness, TTFWC remains totally TCP-Friendly in the low-speed network and acceptable TCP-Friendly in the high-speed network. Although TTFWC throughput is proportion to tuning parameter values, there is little difference between TCP and TTFWC throughput. On the other hand, in the high-speed network, TTFWC with large parameter value gets high throughput comparing to TCP. But the longer propagation delay I set in the network, we do not observe the difference between TCP and TTFWC throughput. I will show you detail observation and consideration as below.

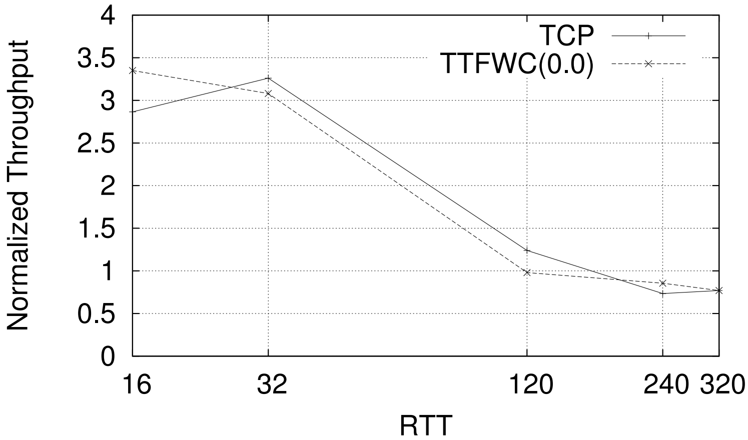


Figure 5.13: TTFWC(0.0) and TCP throughput (Bottleneck Link 250Mbps)

TTFWC reacts to latency of the link in high-speed network when we see Fig. 5.13 - 5.15. In the high-speed network with short delay, the bandwidth utilization of TTFWC(0.9) is allocated more than TCP. I believe the influence of temporal idle bandwidth generated by TCP. In the high-speed network, sawtooth shape throughput behavior makes idle bandwidth between competing flows. As TTFWC window behavior does not make idle bandwidth, it causes the throughput difference between TCP and TTFWC flows.

The same result is observed in low-speed network. When the available bandwidth is limited, TCP leaves very little bandwidth in saw-tooth shaped throughput behavior. Therefore, TTFWC throughput is basically as much as TCP from Fig.5.16 - 5.18.

5.2.3 CPU Processing Time

I measure the processing time in congestion avoidance procedure and in `ssthresh` recalculation with `ftrace`. The `ftrace` tool traces procedures inside the kernel, thus I can use this tool for debugging or analyzing latencies and performance issues from user-space. I measure congestion avoidance procedure over 200,000 calls and `sshthresh` procedure more than 1,500 calls for analysis.

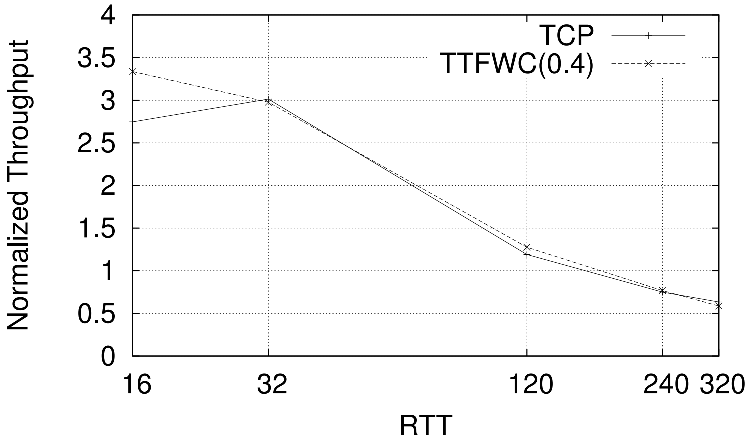


Figure 5.14: TTFWC(0.4) and TCP throughput (Bottleneck Link 250Mbps)

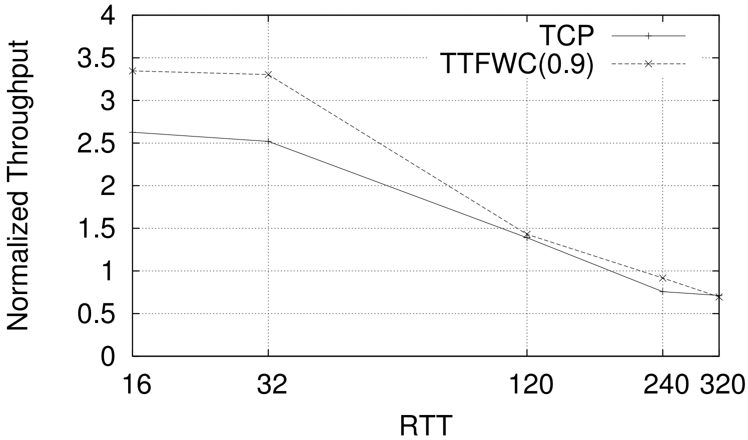


Figure 5.15: TTFWC(0.9) and TCP throughput (Bottleneck Link 250Mbps)

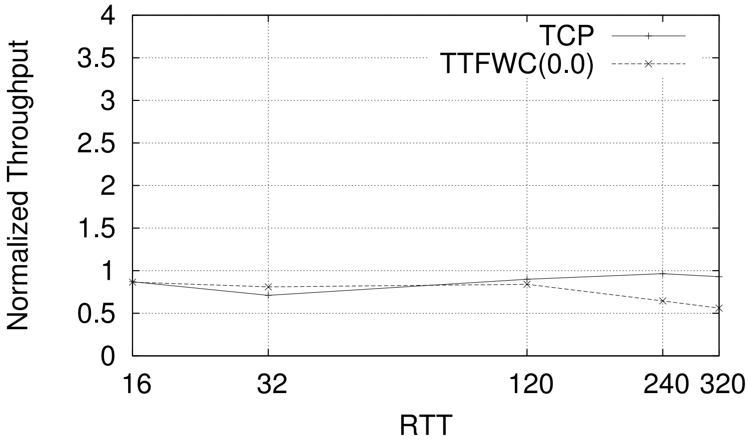


Figure 5.16: TTFWC(0.0) and TCP throughput (Bottleneck Link 10Mbps)

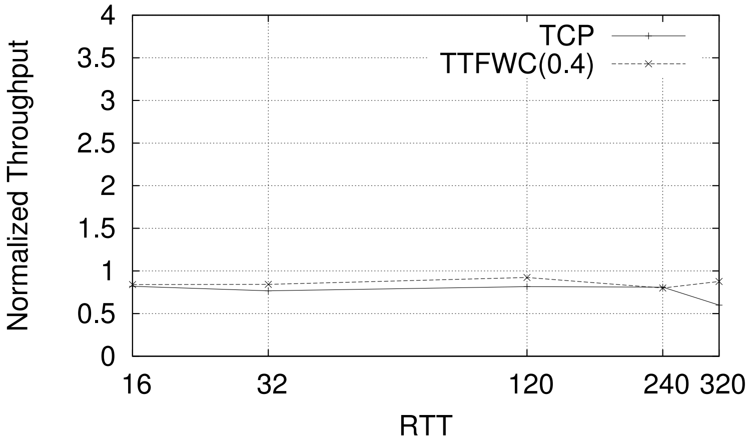


Figure 5.17: TTFWC(0.4) and TCP throughput (Bottleneck Link 10Mbps)

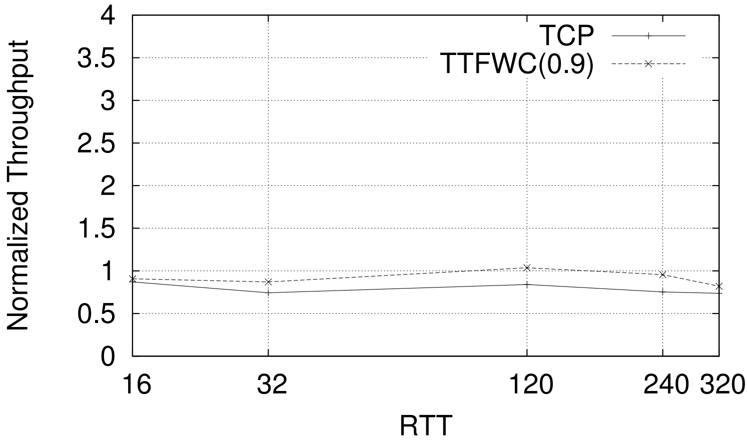


Figure 5.18: TTFWC(0.9) and TCP throughput (Bottleneck Link 10Mbps)

My question is “How long does it take to process TTFWC in comparison with other congestion control algorithms? Is TTFWC impractical for machines?” As TTFWC calculates `cwnd` with loss event history, it is unavoidable to take more processing time than RENO. I compare TTFWC processing time with RENO or CUBIC in this subsection.

Table 5.1 shows that TTFWC takes nine times more processing time than RENO, however, TTFWC takes as long as CUBIC. Recall the discussion some notes about inaccurate ALI calculation in Section 4.1. When I use the accurate ALI calculation in congestion avoidance phase, I find that the ALI calculation adds about 0.78 μs to the current processing time of TTFWC. Although TTFWC does not call the ALI calculation when the number of packets is under the estimation of next loss interval, I find that this procedure makes a machine whose resources are limited. In order to reduce the processing time, I adopt the inaccurate ALI calculation. As the result of this modification, the TTFWC processing time is reasonable, because optional congestion control procedure including CUBIC cannot avoid a little higher overhead than RENO. In addition, since CUBIC is used as the default congestion control of Linux, the processing time of TTFWC to RENO is not so important.

I find that there is only a small difference of processing time between different values of the

Table 5.1: CPU processing time for congestion avoidance procedure

Protocol	Average Time	Standard Deviation
TTFWC(0.0)	1.770 μs	0.271 μs
TTFWC(0.4)	1.951 μs	0.380 μs
TTFWC(0.9)	1.975 μs	0.400 μs
CUBIC	1.771 μs	1.069 μs
RENO	0.240 μs	0.015 μs

tuning parameters. The difference indicates the processing time of RENO `cwnd` estimation and additional `cwnd` procedure. Also, I observe that the standard deviation of TTFWC is proportional to the tuning parameter value. The result tells μs the condition branching that adds base `cwnd`.

Table 5.2 shows the processing time to calculate `ssthresh` for each protocol. In this procedure, TTFWC sender updates the loss histories, then calculates the next window size and additional `cwnd`. Therefore, based on high overhead in TTFWC, I compare how long the processing time of TTFWC differs from others. I observe that TTFWC procedure takes twice as long as CUBIC or RENO and the standard deviation of processing time of TTFWC affects the value of parameters from the tables.

Now, I return the first question. Although the answer to the question is not definitive, I believe that the total processing time of TTFWC is not different from the others. For example, CUBIC have two other procedures: one is parameters' control upon receiving ACKs and the other is when the congestion state of TCP is changed. The average of the former is 1.307 μs and that of the latter is 0.100 μs. In addition, the former procedures are called more frequently than the procedure of `ssthresh` recalculation. When I consider the total processing time of the procedures, I can see that CUBIC is about the sames TTFWC.

Table 5.2: CPU processing time for loss recovery procedure

Protocol	Average Time	Standard Deviation
TTFWC(0.0)	2.200 μs	0.052 μs
TTFWC(0.4)	2.258 μs	0.508 μs
TTFWC(0.9)	2.159 μs	0.439 μs
CUBIC	0.189 μs	0.00 μs
RENO	0.225 μs	0.00 μs

Chapter 6

Conclusion and Future Work

In this thesis, I have proposed and examined TTFWC for multimedia streaming, which allows us to tune the balance between smoothness and the sending rate. The motivation is to improve TFWC throughput for throughput-sensitive multimedia services. As I control TTFWC throughput by using tuning parameter, I can choose whether or not to use smooth congestion control based on the application type.

I find that TTFWC with a large tuning parameter value keeps a high sending rate in the network even in the case of frequent bandwidth changes through the simulation result and real-world experiments. This is a significant contribution, because such services have becomes common with the deployment of Content Delivery Networks (CDNs).

In future work, I address three issues: first is the dynamic adaptation of the tuning parameter, second is the convergence between compatible TTFWC flows, and third issue is to adapt it for use in the high-speed networks.

The first issue is very difficult to improve because the number of frames are not predictable, especially video applications. Although I understand the number of packets stored in the queue, the data rate generated for encoded video varies second by second, making flexible selection of the tuning parameter hard. I will research the inner structure of multimedia application and consider approaches for dynamic control.

The second issue is the limitation of TTFWC design. When we compare throughput-sensitive TTFWC to original TFWC, the TTFWC design towards high throughput affects protocol friendliness and flows' throughput between competing flows.

The third problem is a significant concern for future congestion control algorithms. The limitation of my proposal is its formula-based congestion control algorithm. This means that TTFWC throughput does not exceed RENO throughput. Today, it is important to have not only TCP-Friendliness but also high throughput with fast convergence in the high-speed network. For example, CUBIC is used as the Linux default congestion control algorithm despite not being TCP-Friendly in the long, fat pipe. I think that one of the messages of this research is the requirement to satisfy both stability and high transmission rates. Thus, I will consider what approaches are suitable for multimedia streaming in the high-speed network.

Acknowledgment

I really appreciate many people and great support around me. I will list the people as below.
I express my appreciation to Hideyuki Tokuda for his professional advice, guidance and encouragement. Thanks to his support and kindness, I could challenge to write the thesis and got chances to join the international community. I feel grateful to Rodney D. Van Meter so much. Thanks to his great help and kindness, I can improve my thesis.
I would like to thank professors, associate professors and assistant professors in Keio University, Jun Murai, Osamu Nakamura, Hiroyuki Kusumoto, Kazunori Takashio, Keisuke Uehara, Jin Mitsugi, Keiji Takeda and Jin Nakazawa. Without their encounter and support, I was not what I am. Their classes and their personal character interests me and that is why I study computer science now. I thanks Shoko Miyagawa, too. She took me to Grace Hopper Celebration in Women Computing. Through the conference, I made friends and have have communicate with the great student all over the world. Also, I was able to expand my world through these conferences and these experience gave me fresh inspiration.
I feel grateful so much to my advisor, Michio Honda. Although it takes much time to pick up the topic and to consider the proposal, he always indicates the errors and lead me to have high motivation. Without his supports, I have not be able to accepted my paper and to get chances to join the international conferences. He is also good friend of mine as well as good advisor. I sometimes consulted with him as to my private concerns and my life. Thank you for supporting me beyond the advisor.

I also gratefully acknowledge Hideaki Yoshifuji. When I am at loss for the kernel bug, he taught me what knowledge I need. His guideline and professional advice also help me how to evaluate my implementation.

I appreciate all the member in my laboratory and my friends, too. Without them, I have not been able to built my skill and overcome the hardship.

Of course, I am thankful to my family, too. While it takes more than two hours from home to SFC, my family keeps an eye on my personal life. My mother have to wake up early when I have a class on first period. My father often keeps staying up until I come back home safely. My grand parents always encourage me when I was disappointing the limitation of the skill of mine. My younger sister and my younger brother always make me laugh and revive my heart. They never complain about my life style. Thanks to their supports, I spend happy school life.

February 8, 2012

Midori Kato

Bibliography

[1] ftrace. http://lwn.net/Articles/370423/.

[2] iperf. http://iperf.sourceforge.net/.

[3] V. Tsaoussidis A. Lahanas. Exploiting the efficiency and fairness potential of AIMD-based congestion avoidance and control. *Computer Networks Volume 43, Issue 2*, October 2003.

[4] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic Behavior of Slowly-Responsive Congestion Control Algorithms. In *Proc. ACM SIGCOMM*, 2001.

[5] S. Choi. On studying tfwc protocol smoothness. April 2008.

[6] S. Choi and M. Handley. Designing TCP-Friendly window-based congestion control for real-time multimedia applications. *Proc. PFLDNeT*, May 2009.

[7] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *SIGCOMM*, July 1996.

[8] J. Feng and L. Xu. TCP-Friendly CBR-Like Rate Control. In *Proc. ICNP IEEE*, October 2008.

[9] S. Floyd. Congestion Control Principles. *RFC 2914*, Sep. 2000.

[10] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. *ACM Computer Communication Review*, 30(4):43–56, Oct. 2000.

[11] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. *RFC 2582*, April 1999.

[12] S. Floyd and A. Romanow. TCP Selective Acknowledgment Options. *RFC 2018*, October 1996.

[13] V. Ozdemir I. Rhee and Y. Yi. TEAR: TCP Emulation At Receivers: flow control for multimedia streaming. *Technical Report, Department of Computer Science*, 2000.

[14] D. Towsley J. Padhye, V. Firoiu and J. Kurose. Modeling TCP throughput: a simple model and its empirical validation. August 1998.

[15] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation and modeling.* John Wiley and Sons Inc., 1991.

[16] P. Huang K. Chen, C. Huang and C. Lei. Quantifying Skype user satisfaction. In *ACM SIGCOMM*, September 2006.

[17] A. Medina, M. Allman, and S. Floyd. Measuring the Evolution of Transport Protocols in the Internet. *ACM CCR*, 35(2):37–52, 2005.

[18] The network simulator - ns-2. URL http://www.isi.edu/nsnam/ns/.

[19] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27:31–41, January 1997.

[20] I. Matta S. Jin, L. Guo and A. Bestavros. A spectrum of TCP-Friendly window-based congestion control algorithms. In *Proc. IEEE/ACM Trans. Net*, June 2003.

[21] R. Stewart. Stream Control Transmission Protocol. *RFC 4960*, September 2007.

[22] E. Vieira and M. Bauer. Round-trip Time Variation in SmoothTCP in the Face of Spurious Errors. *Journal of Communications*, 35(2):48–56, 2006.

[23] Y. Yang and S. Lam. General AIMD congestion control. In *Proc. IEEE ICNP*, November 2000.

Published Papers Related to this thesis

- M. Kato, M. Honda and H. Tokuda

  "Extending TFWC towards Higher Throughput",

  IEEE Consumer Communications and Networking Conference (CCNC), Jan 2012, Las Vegas, U.S.A.

Posters and Demos

- M. Kato, M. Honda and H. Tokuda

  "A practical congestion control algorithm for real-time multimedia streaming", Grace Hopper Celebration of Women in Computing (GHC), Nov 2011, Portland, U.S.A.

- M. Kato, M. Honda and H. Tokuda

  "Implementation of congestion control for multimedia streaming" ,

  IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Aug 2011, Toyama, Japan