PAPER

TCP Congestion Control Mechanisms for Achieving Predictable Throughput Using Inline Network Measurement

Go HASEGAWA[†a)], ***Member***, **Kana YAMANEGI**[††], ***Nonmember***, ***and*** **Masayuki MURATA**[†], ***Member***

SUMMARY Recently, real-time media delivery services such as video streaming and VoIP have rapidly become popular. For these applications requiring high-level QoS guarantee, our research group has proposed a transport-layer approach to provide predictable throughput for upper-layer applications. In the present paper, we propose a congestion control mechanism of TCP for achieving predictable throughput. It does not mean we can guarantee the throughput, while we can provide the throughput required by an upper-layer application at high probability when network congestion level is not so high by using the inline network measurement technique for available bandwidth of the network path. We present the evaluation results for the proposed mechanism obtained in simulation and implementation experiments, and confirm that the proposed mechanism can assure a TCP throughput if the required bandwidth is not so high compared to the physical bandwidth, even when other ordinary TCP (e.g., TCP Reno) connections occupy the link.

key words: *transmission control protocol (TCP), throughput guarantee, congestion control mechanism, Linux*

1. Introduction

The Internet users' demands for network quality has increased due to services becoming progressively diversified and sophisticated because of the remarkable degree to which the Internet has grown, which is due in part to access and backbone network technologies. Applications involving real-time media delivery services, such as VoIP, video streaming and TV meeting systems, all of which have experienced a dramatic level of development, require large and stable amounts of network resources in order to maintain the Quality of Service (QoS). For example, the quality of real-time streaming delivery applications is highly dependent on propagation delay and delay jitter. The available bandwidth on the end-to-end network path is also an important factor in order to smoothly provide rich contents, including voice and video.

There are a number of network-layer technologies, such as IntServ [1] and DiffServ [2], that provide such high-quality network services over the Internet. However, implementation of IntServ or DiffServ architectures would require additional mechanisms to be deployed to all routers through which traffic flows traverse in order to sufficiently benefit from the introduction of IntServ or DiffServ into the network. Therefore, due to factors such as scalability and cost, we believe that these schemes have almost no chance of being deployed on large-scale networks.

On the other hand, a number of video streaming applications use User Datagram Protocol (UDP) [3] as a transport-layer protocol, and UDP controls the data transmission rate according to the network condition [4]–[6]. However, these mechanisms have a large cost when modifying the application program for achieving application-specific QoS requirements, and the parameter settings are very sensitive to various network factors. Furthermore, when such applications co-exist in the network and share the network bottleneck resources, we cannot estimate the performance of the network or that of the applications, because the control mechanisms of such applications are designed and implemented independently, without considering the effect of interactions with other applications.

In our research group, we have previously proposed transport-layer approaches for achieving QoS for such applications. For example, in [7], we proposed a background transfer mechanism using Transmission Control Protocol (TCP) [8], which transfers data using the residual bandwidth of the network without any impact on the co-existing network traffic sharing the bottleneck link bandwidth. Since TCP controls the data transmission rate according to the network condition (congestion level), we believe that the transport-layer approach is ideal for providing high-quality data transmission services in the Internet. Furthermore, by implementing the mechanism into TCP, rather than introducing a new transport-layer protocol or modifying UDP, we can accommodate existing TCP-based applications transparently, and we can minimize the degree of modification to provide high-quality transport services.

In this paper, we focus on achieving predictable throughput by TCP connections. Essentially, TCP cannot obtain guaranteed throughput because its throughput is dependent on, for example, Round Trip Time (RTT), packet loss ratio of a network path, and the number of co-existing flows [9]. Therefore, we intend to increase the probability at which a TCP connection achieves the throughput required by an upper-layer application, while preserving the fundamental mechanisms of congestion control in TCP. We refer to this as *predictable throughput*. By predictable throughput, we mean the throughput required by an upper-layer application, which can be provided with high probability when the network congestion level is not extremely high.

Manuscript received April 7, 2008.
Manuscript revised August 11, 2008.

[†]The authors are with the Graduate School of Infomation Science and Technology, Osaka University, Suita-shi, 565-0871 Japan.
[††]The author is with Matsushita Electric Industrial Co., Ltd., Kadoma-shi, 571-8501 Japan.
a) E-mail: hasegawa@cmc.osaka-u.ac.jp

DOI: 10.1093/ietcom/e91–b.12.3945

In this paper, we propose a congestion control mechanism of TCP for achieving the predictable throughput with high probability, regardless of the network congestion level. We modify the degree of the increase of the congestion window size [10] of a TCP connection in the congestion avoidance phase by using the information on the available bandwidth of the network path obtained by Inline Measurement TCP (ImTCP) [11], [12], which has been previously proposed by our research group. The application examples of the proposed mechanism include TCP-based video/voice delivery services, such as Windows Media Player [13], RealOne Player [14], and Skype [15]. We also show that we can control the sum of the throughput of multiple TCP connections, by extending the mechanism for one TCP connection. This mechanism may be used in the situation in which a stable throughput should be provided for the network traffic between two local area networks interconnected by IP-VPN [16].

We first evaluate the effectiveness of our proposed mechanism by simulation experiments using ns-2 [17] to investigate the fundamental characteristics. We confirm that the proposed mechanism can achieve a TCP throughput of 10–20% of the bottleneck link capacity, even when the link is highly congested and there is little residual bandwidth for the TCP connection. We also show that the proposed mechanism can provide a constant throughput for the traffic mixture of long-lived and short-lived TCP connections. We further implement the proposed mechanism on a Linux 2.6.16.21 kernel system, and evaluate its performance on an experimental network, which is the controlled conditional network. Finally, we confirm the performance of our proposed mechanism in the commercial Internet environment. From these results, we confirm that the proposed mechanism can achieve the required throughput with high probability in an actual network, as in the simulation results.

The remainder of this paper is organized as follows: In Sect. 2, the proposed mechanism to provide predictable throughput is described. The performance of the proposed mechanism through extensive simulation experiments is evaluated in Sect. 3. In Sect. 4, the implementation design in the Linux 2.6.16.21 kernel system is outlined and the performance in an actual network is presented. Finally, conclusions and areas for future study are discussed in Sect. 5.

2. Proposed Mechanisms

Figure 1 shows an overview of the proposed mechanism. We assume that an upper-layer application sends *bw* (packets/sec) and *t* (sec) to the proposed mechanism, which is located at the transport layer. This means that the application requires average throughput *bw* at every interval of *t* sec in the TCP data transmission, and the proposed mechanism tries to achieve this demand. A possible appication of the proposed mechanism is TCP-based video streaming service such as Youtube, where the evaluation slot is set to the length of the playout buffer at the reveiver.

Note that by implementing the proposed mechanism,

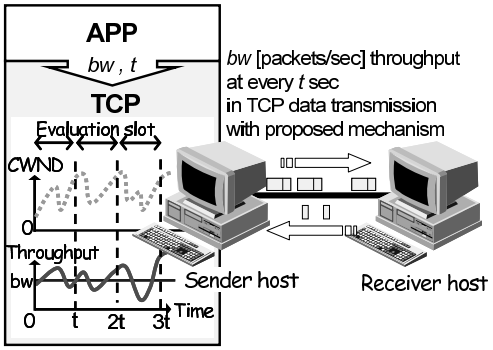


Fig. 1 Overview of the proposed mechanism.

we also need to modify the socket interface to pass the value of required throughput from the upper-layer application to TCP. Here, *bw* is the *required throughput* and the time interval is referred to as the *evaluation slot*, as shown in Fig. 1. We change the degree of increase of the congestion window size of a TCP connection to achieve a throughput of *bw* every *t* sec. Note that in the slow start phase, we use a mechanism that is identical to the original TCP Reno, i.e., the proposed mechanism changes the behavior of TCP only in the congestion avoidance phase. By minimizing the degree of modification of the TCP source code, we expect that the original property of the congestion control mechanism can be preserved. We can also reduce the introduction of implementation bugs by basing our modification on the existing TCP source code.

Since the proposed mechanism changes its behavior in units of the RTT of the connection, we introduce the variable e as $t = e \cdot rtt$, where rtt is the RTT value of the TCP connection.

In Sect. 2.1, the calculation method of target throughput in each evaluation slot is introduced. In Sect. 2.2, the algorithm to achieve the required throughput is proposed.

2.1 Calculating the Target Throughput

We split an evaluation slot into multiple sub-slots, called *control slots*, to control the TCP behavior in a finer-grained time period. The length of the control slot is s (RTT), where s is $2 \leq s \leq e$ (Fig. 2). Figure 2 shows the relationship between the evaluation slot and the control slots. We set the throughput value we intend to achieve in a control slot, which is referred to as the *target throughput* of the control slot. We change the target throughput in every control slot and regulate the packet transmission speed in order to achieve the target throughput. The final goal is to make the average throughput in the evaluation slot larger than or equal to *bw*, the required throughput.

We use the smoothed RTT (sRTT) value of the TCP connection to determine the lengths of the evaluation slot and the control slot. That is, we set the length of the i-th control slot to $s \cdot srtt_i$, where $srtt_i$ is the sRTT value at the beginning of the i-th control slot. At the end of each control slot, we calculate the achieved throughput of the TCP con-

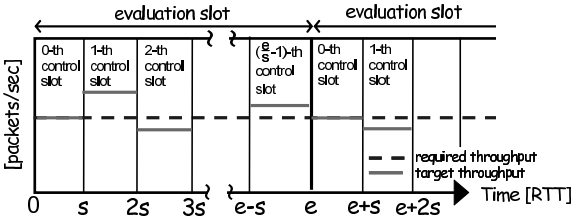


Fig. 2 Evaluation and control slots.

nection by dividing the number of successfully transmitted packets in the control slot by the length of the control slot. We then set the target throughput of the i-th control slot, g_i (packets/sec), as follows:

$$\begin{cases} g_i = bw + (g_{i-1} - tput_{i-1}) \\ g_0 = bw \end{cases}$$

where $tput_i$ (packets/sec) is the average throughput of the i-th control slot. This equation means that the target throughput of the i-th control slot is determined according to the difference between the target throughput and the achieved throughput in the $(i-1)$-th control slot.

2.2 Achieving the Target Throughput by Changing the Congestion Window Size

Although it may seem that one simple method to achieve the target throughput by TCP would be to fix the congestion window size to the product of the target throughput and RTT and to keep the window size even when packet losses occur in the network, such a straightforward method would introduce several irresolvable problems in the network congestion. In addition, such a method would result in severe unfairness with respect to co-existing connections using the original TCP Reno. Therefore, in the proposed mechanism, the degree of modification of the TCP congestion control mechanism is minimal in order to maintain the original properties of TCP. This means that the degree of the congestion window size is increased only in the congestion avoidance phase of a TCP connection; in other words, this mechanism does not modify the TCP behavior in the slow start phase or when a TCP connection experiences packet loss(es).

In the proposed mechanism, the sender TCP updates its congestion window size *cwnd* in the congestion avoidance phase according to the following equation when it receives an ACK packet from the receiver TCP:

$$cwnd \leftarrow cwnd + \frac{k}{cwnd} \quad (1)$$

where k is the control parameter. From the above equation, we expect that the congestion window size increases by k packets in every RTT. The main function of the proposed mechanism is to regulate k dynamically and adaptively, whereas the original TCP Reno uses a fixed value of $k = 1$. In the rest of this subsection, we explain how to change k according to the network condition and the current throughput of the TCP connection.

2.2.1 Increasing the Degree of the Congestion Window Size

Here, we derive k_j^{bw}, nwhich is an ideal value for the degree of increase of the congestion window size when the j-th ACK packet is received from the beginning of the i-th control slot, so that the TCP connection achieves g_i of the average throughput. For achieving the average throughput g_i in the i-th control slot, we need to transmit $(g_i \cdot srtt_i \cdot s)$ packets in $(s \cdot srtt_i)$ sec. However, since it takes one RTT to receive the ACK packet corresponding to the transmitted packet, and since it takes at least one RTT to detect packet loss and retransmit the lost packet, we intend to transmit $(g_i \cdot s \cdot srtt_i)$ packets in $((s-2) \cdot srtt_i)$ sec.

We assume that the sender TCP receives the j-th ACK packet at the n_j-th RTT from the beginning of the control slot, and the congestion window size at that time is $cwnd_{n_j}$. Since the congestion window size increases by k packets every RTT, we can calculate p_{snd}, the number of packets that would be transmitted if we use k_j^{bw} for k in Equation (1) in the rest of the control slot. The length of the rest of the control slot is $(s - 2 - n_j) \cdot srtt_i$ sec. The equation for p_{snd} is as follows:

$$\begin{aligned} p_{snd} &= (s - n_j - 1)cwnd_{n_j} \\ &\quad + \frac{k_j^{bw}}{2}(s - n_j - 1)(s - n_j) \end{aligned}$$

On the other hand, p_{need}, i.e., the number of packets that should be transmitted in order to obtain g_i, is calculated as follows:

$$p_{need} = g_i \cdot srtt_i \cdot s - a_j$$

where a_j is the number of transmitted packets from the beginning of the control slot to when the j-th ACK packet is received. Then, we can calculate k_j^{bw} by solving the equation $p_{snd} = p_{need}$ for k_j^{bw}:

$$\begin{aligned} k_j^{bw} &= 2\{g_i \cdot srtt_i \cdot s - a_j - (s - n_j - 1)cwnd_{n_j}\} \\ &\quad /\{(s - n_j - 1)(s - n_j)\} \end{aligned} \quad (2)$$

In the proposed mechanism, we use the above equation to update k for Eq. (1) when the sender TCP receives a new ACK packet. By this ack-based mechainsm, the proposed mechanism can accomodate the fluctuation of RTTs of the network path.

2.2.2 Limitation of the Degree of Increase Based on the Available Bandwidth

By using Eq. (2) for determining k, the degree of increase of the congestion window size becomes too large when the current throughput of a TCP connection is far below the target throughput. Values of k that are too large would cause bursty packet losses in the network, resulting in a performance degradation due to retransmission timeouts. On the

other hand, when the network has sufficient residual bandwidth, the degree of increase of the congestion window size in Eq. (2) becomes smaller than 1. This results in a lower throughput increase than that for TCP Reno. Therefore, we limit the maximum and minimum values for k, which are denoted by k_{max} and k_{min}, respectively. We simply set $k_{min} = 1$ to preserve the basic characteristics of TCP Reno. However, some applications, such as those transferring sensing observation data and maintaining a monitoring log on a system, generate data at a constant rate and do not require higher throughput than the designated throughput even when the network has enough bandwidth. For such applications we do not set k_{min}, which means that the proposed mechanism does not achieve more than the required throughput even if the residual bandwidth is more than the required throughput.

k_{max}, on the other hand, should be set such that bursty packet losses are not invoked, and the target throughput is obtained. Thus, we decide k_{max} according to the following considerations. First, when the proposed mechanism has obtained the target throughput in all of the control slots in the present evaluation slot, we determine that the available bandwidth of the network path is sufficient to obtain the target throughput of the next control slot. Therefore, we calculate k_{max} so as to avoid packet losses by using the information of the available bandwidth of the network path. The information about the available bandwidth of the network path is estimated by ImTCP [12], which is the mechanism for inline network measurement. ImTCP measures the available bandwidth of the network path between the sender and receiver hosts. In TCP data transfer, the sender host transfers a data packet and the receiver host replies to the data packet with an ACK packet. ImTCP measures the available bandwidth using this mechanism; that is, ImTCP adjusts the sending interval of the data packets according to the measurement algorithm and then calculates the available bandwidth by observing the change of ACK arrival intervals. Because ImTCP estimates the available bandwidth of the network path from the data and ACK packets transmitted by an active TCP connection in an inline fashion, ImTCP does not inject extra traffic into the network. ImTCP is described in detail in [12].

Next, when the proposed mechanism has not obtained the target throughput in the previous control slot, the proposed mechanism will not obtain the target throughput in the following control slots. We then set k_{max} so as to obtain a larger throughput than the available bandwidth of the network path. This means that the proposed mechanism steals bandwidth from competing flows in the network in order to achieve the bandwidth required by the upper-layer application.

In summary, the proposed mechanism updates k_{max} by using the following equation when the sender TCP receives a new ACK packet:

$$k_{max} = \begin{cases} A \cdot srtt_i - cwnd \\ (\text{if } \ tput_{i-1} > g_{i-1}) \\ min(A + (g_i - tput_{i-1}), P) \cdot srtt_i - cwnd \\ (\text{if } \ tput_{i-1} < g_{i-1}) \end{cases} \quad (3)$$

where A and P (packets/sec) are the current values for the available bandwidth and physical capacity, respectively, as measured by ImTCP. In the upper equation in Eq. (3), $A \cdot srtt_i$ indicates the maximum number of packets that the proposed mechanism can occupy within the network capacity without packet losses occurring. In the lower equation in Eq. (3), $(g_i - tput_{i-1}) \cdot srtt_i$ indicates the number of packets required in order to obtain the target throughput when the network has insufficient available bandwidth.

2.2.3 Length of the Control Slot

In general, the length of the control slot (s) controls the trade-off relationship between the granularity of the throughput control and the influence on the competing traffic. For example, if we use a small value for s, it becomes easier to obtain the required throughput because we update the target throughput g_i more frequently. On the other hand, the smaller value of s means that the congestion window size is changed so drastically that we achieve the average throughput in a smaller control slot, which results in a larger effect on other competing traffic. Therefore, we should set s to be as large as possible, while maintaining the required throughput. Since the ideal value of s depends on various factors of the network condition, including the amount of competing traffic, we propose an algorithm to dynamically regulate s.

The algorithm is based on the following considerations. First, when the proposed mechanism has not obtained the target throughput although we set k_{max} by using Eq. (3), a smaller value should be used for s in order to achieve the target throughput. Second, when the proposed mechanism has achieved the target throughput with k_{max} calculated by Eq. (3) and when the congestion window size is satisfied with $cwnd \geq bw \cdot srtt_i$, we can expect that the proposed mechanism could achieve the target throughput, even when we increase the length of the control slot. Therefore, we use a larger s in the next evaluation slot.

2.3 Maintaining Multiple Connections

In this subsection, by extending the mechanism in Sect. 2.1 and 2.2, we depict the mechanism that controls the sum of the throughput of multiple TCP connections. In this mechanism, we assume that multiple TCP connections are maintained at transport-layer proxy nodes such as TCP proxy [18], and the throughput is controlled at the proxy nodes. The proposed mechanism at the proxy node controls the sum of the throughput of multiple TCP connections according to the following assumptions:

- The proposed mechanism is intended to achieve the required throughput, bw, of the sum of multiple TCP connections at every t sec interval.

- The multiple TCP connections use the same values for the length of the evaluation and control slots, which are set based on the minimum sRTT measured by the sender-side proxy node.
- The sender-side proxy node can identify the number of active TCP connections. This assumption is natural when we use explicit proxy mechanisms such as TCP proxy and SOCKS [19].

One of the possible applicaion of this mechanism is Virtual Private Network (VPN) environemnts. When we want to guarantee the throughput of TCP flows in the VPN connections between two networks, we can utilize the proposed mechanism to protect the performance of the VPN from competing VPN connections in the network. Note that one of the advantage of this mechanism is that we do not require the additional mechanisms to the routers and switches in the networks, and we can provide throughput guarantee only with the end-to-end control.

For realizing the above control, we can simply extend Eq. (2) to multiple TCP connections, as follows:

$$k_j^{bw} = \frac{2\{(g_i \cdot srtt_i \cdot s - a_j^{sum})/N_{\rm pm} - (s - n_j - 1)cwnd_i^{n_j}\}}{(s - n_j - 1)(s - n_j)}$$

where a_j^{sum} is the sum of the packets that TCP senders have sent when receiving the j-th ACK, and N_{pm} is the number of active TCP connections. We use this equation for all TCP connections. This equation means that the degree of the increase of the congestion window size is calculated by distributing the number of packets needed for achieving the target slot to the active TCP connections.

3. Simulation Results and Discussions

In this section, we evaluate the proposed mechanism by simulation experiments using ns-2. Figure 3 shows the network model. This model consists of sender and receiver hosts, two routers, and links between the hosts and router. We set the packet size at 1,000 Bytes. The bandwidth of the bottleneck link is set at 100 Mbps, and the propagation delay is 5 msec. A DropTail discipline is deployed at the router buffer, and the buffer size is set at 100 packets. The number of TCP connections using the proposed mechanism is $N_{\rm pm}$, and the number of TCP Reno connections, for creating background traffic, is $N_{\rm reno}$. The bandwidth of the access links is set at 1 Gbps, and the propagation delay is 2.5 msec. For the proposed mechanism, we set $t = 32 \cdot \text{RTT}$ ($e = 32$) for the length of the evaluation slot. In this network model, 32 RTT corresponds to approximately 1 sec. In addition, s, the length of the control slot, is initialized to 16.

3.1 Case of One Connection

We first evaluate the performance of the proposed mechanism for one TCP connection. In this simulation, we set $N_{\rm pm} = 1$, and bw is 20 (Mbps), which is equal to 20% of the bottleneck link capacity. To change the congestion level

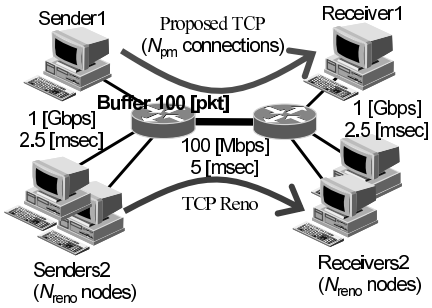
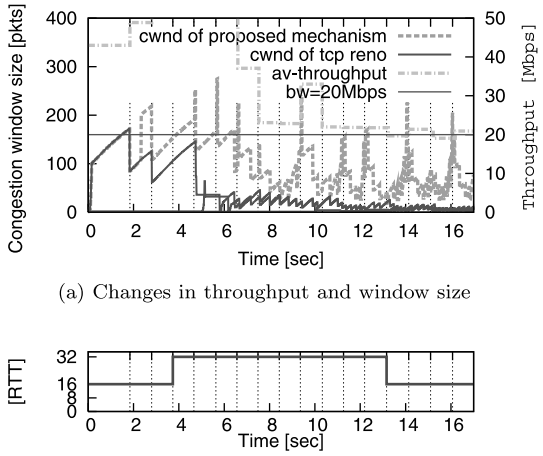


Fig. 3 Network model for the simulation experiments.



(a) Changes in throughput and window size

(b) Changes in control slot length

Fig. 4 Changes in congestion window size, average throughput, and length of the control slot.

of the network, we change $N_{\rm reno}$ to 1, 10, 40 at every 5 seconds. Figure 4 shows the changes in the congestion window size, the average throughput, and the length of the control slot of the TCP connection with the proposed mechanism. In this figure, the vertical grid represents the boundaries of the evaluation slots.

The results for 0-5 seconds, shown in Fig. 4(a), indicate that when one TCP Reno connection co-exists with a TCP connection of the proposed mechanism, the proposed mechanism can obtain the required throughput while performing almost equivalently to TCP Reno. In this period, the available bandwidth is sufficiently large to obtain the required throughput, because there are only two connections in the network, which have capacities of 100 Mbps. Thus, the proposed mechanism sets $k = k_{min}$ (= 1), resulting in fairness while maintaining the TCP Reno connection.

For the results for 5–10 seconds, in which case there are 10 TCP Reno connections, we observe that the proposed mechanism has a faster increase in the congestion window size compared to that of the TCP Reno connections. In this case, this is because it is impossible to obtain the required throughput with behavior identical to TCP Reno due to the increase in the amount of competing traffic. Consequently, the proposed mechanism changes the degree of increase of the congestion window size (k) in order to achieve the re-

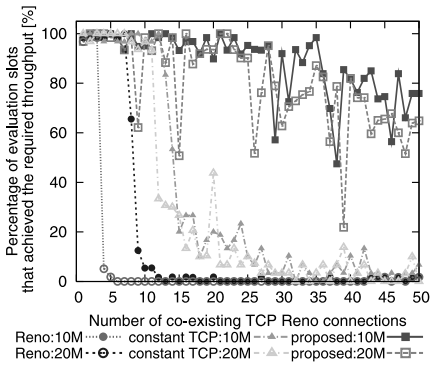


Fig. 5 Percentage of evaluation slots in which the required throughput is achieved.

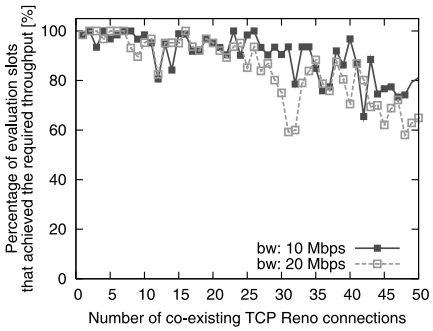


Fig. 6 Case for limiting the throughput of co-existing TCP Reno connections.

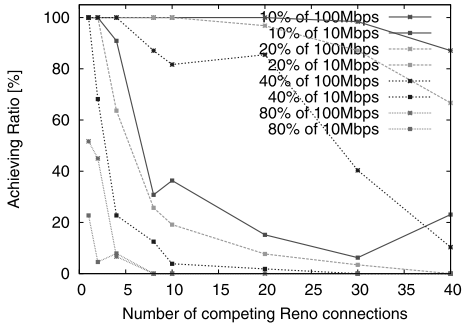


Fig. 7 Case for various target throughput.

quired throughput.

Furthermore, the results after 10 seconds with 40 TCP Reno connections show that the congestion window size of the proposed mechanism increases faster than that of the previous cases, and the length of control slot, s, is changed to a smaller value. This result indicates that the proposed mechanism controls its congestion window size with a smaller length of the control slot to obtain the required throughput because sufficient throughput cannot be achieved by merely changing the degree of increase of the congestion window size. As a result, the proposed mechanism can obtain the required throughput even when there are 40 competing TCP Reno connections. Thus, we have confirmed that the proposed mechanism can effectively obtain the required throughput by changing the degree of increase of the congestion window size and the length of the control slot according to the network congestion level.

We next show the relationship between the performance of the proposed mechanism and the number of co-existing TCP Reno connections in greater detail. We set $N_{\mathrm{pm}} = 1$, and bw is 10% (10 Mbps) and 20% (20 Mbps). Figure 5 shows the ratio of the number of evaluation slots, in which the proposed mechanism obtains the required throughput, to the total number of evaluation slots in the simulation time. In this simulation experiment, the simulation time is 60 seconds. For the sake of comparison with the proposed mechanism, we also show the simulation results obtained using TCP Reno (labeled "Reno") and modified TCP (labeled "constant"), which uses a constant congestion window size of $bw \cdot srtt_{min}$ (packets) even when packet drops occur. Here, $srtt_{min}$ is the minimum sRTT value for the TCP connection.

Figure 5 indicates that the original TCP Reno can obtain the required throughput for 100% of the evaluation slots when a few background connections co-exist, because the original TCP Reno fairly shares the bottleneck link bandwidth with all of the connections. However, when the number of co-existing connections (N_{reno}) increases, TCP Reno cannot obtain the required throughput because it shares the bandwidth with numerous connections. We can also observe that the TCP with a constant window size cannot achieve the required throughput when N_{reno} is larger than 10. In this situation, the network congestion cannot be resolved because the congestion window size is not decreased, even when packet losses occur in the network. In contrast, the proposed mechanism can obtain the required throughput with high probability even when several connections co-exist in the network. This means that the proposed mechanism can control the trade-off relationship between the aggressiveness of the proposed mechanism and the degree of influences on competing traffic.

We also evaluated the proposed mechanism when we limit the maximum value of the congestion window size of co-existing TCP Reno connections. This corresponds to the situation in which the bottleneck link bandwidth is larger than the access link bandwidth. In this simulation, we set the maximum value of the congestion window size of co-existing TCP Reno connections to 100 packets, which limits the throughput of each TCP Reno connection to approximately 4 Mbps. Figure 6 shows the simulation results when bw is set to 10% and 20%, respectively.

Compared with the results shown in Fig. 5, the results in Fig. 6 show that the proposed mechanism has a more stable probability of achieving the required throughput. In this situation, the proposed mechanism can utilize the network bandwidth more effectively because the competing TCP Reno connections are not very strong due to their window size limitation.

As the final result for one connection case, we investigate the performance of the proposed mechanism when we

Table 1 Throughput of competing TCP Reno flows.

Physical link bandwidth	Target throughput	Number of Reno flows	Throughput of proposed flow	Total Reno throughput	Fairness index
100 Mbps	10 Mbps	10	15.2 Mbps	79.7 Mbps	0.999
100 Mbps	10 Mbps	20	13.5 Mbps	82.3 Mbps	0.999
100 Mbps	10 Mbps	40	11.7 Mbps	84.9 Mbps	0.999
100 Mbps	20 Mbps	10	24.8 Mbps	70.0 Mbps	0.999
100 Mbps	20 Mbps	20	23.8 Mbps	72.0 Mbps	0.999
100 Mbps	20 Mbps	40	21.9 Mbps	74.7 Mbps	0.998

change the target throughput. In Fig. 7, we plot the change in the ratio of the number of evaluation slots, in which the proposed mechanism obtains the required throughput, to the total number of evaluation slots, as a function of the number of competing TCP Reno connections. In the figure, we show the results when we set the physical bandwidth of the bottleneck link is 10 Mbps and 100Mbps, and we change the target throughput from 10% to 80% of the physical bandwidth. From this figure, we can see that when the physical link bandwidth is 100 Mbps, we can provide 10%–20% throughput even when the number of competing Reno connections is around 30. However, when the physical bandwidth is 10 Mbps, we can provide around 20% throughput only when the number of competing Reno flows is smaller than 10. This is because the proposed mechanism would increase the congestion window aggressively, which increase the congestion level of the network.

Table 1 shows the average throughput of the flow of proposed mechanism, the total throughput of competing Reno flows, and fairness index of the Reno flows. This table shows that the proposed mechanism steals the bandwidth according to the required throghput, and the competing Reno flows utilize the remaining bandwidth effectively and fairly.

3.2 Case of Multiple Connections

Next, we demonstrate the performance of the proposed mechanism for the multiple TCP connections described in Sect. 2.3. In the simulation, we establish multiple TCP connections between Sender 1 and Receiver 1 in Fig. 3, and the proposed mechanism at Sender 1 controls the throughput of the connections. We set $bw = 20$ (Mbps) and $N_{\rm pm}$= 5 and 10. This setting means that a total throughput of 20 (Mbps) is achieved for the 5 or 10 TCP connections. The maximum value of the congestion window size of co-existing TCP Reno connections is 100 packets. Here, we assume that the TCP sender host knows the current information on the available bandwidth and physical capacity of the network path. This assumption is necessary in order to focus on evaluating the algorithm described in Sect. 2.3.

Figure 8 shows the percentage of the number of evaluation slots, in which the proposed mechanism can obtain the required throughput, to the total number of evaluation slots in the simulation time. This figure shows the results for the following cases: 10 connections without the proposed mechanism (labeled "TCP Reno"); 10 connections, each with the proposed mechanism, where $bw = 2$ (Mbps)

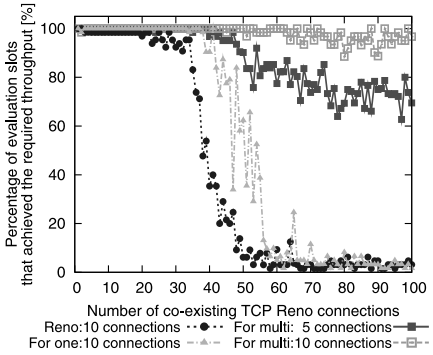


Fig. 8 Performance comparison for multiple connections.

(labeled "For one"); and multiple connections with the proposed mechanism (labeled "For multi"). This figure shows that the original TCP Reno without the proposed mechanism cannot obtain the required throughput when the number of co-existing connections becomes larger than 30. When we use the proposed mechanism for each of the 10 connections, the performance is not as good when the number of competing connections exceeds 40. This is because bursty packet losses occur because the multiple connections simultaneously inject several packets into the network based on the available bandwidth information estimated by each connection. On the other hand, the proposed mechanism for multiple connections can obtain the required throughput with high probability even when the number of the co-existing TCP Reno connections increases. This is because the problem of the proposed mechanism for one connection is solved by sharing k_{max} with the multiple connections, as described in Sect. 2.3. In addition, the performance for $N_{\rm pm} = 10$ is better than that for $N_{\rm pm} = 5$ to achieve the required throughput. This is because the effect of sharing k_{max} becomes larger when a larger number of connections is accommodated.

3.3 Case of a Mixture of Short-Lived and Long-Lived Connections

We finally show the result when short-lived TCP connections such as Web traffic co-exist in the network. In these simulation experiments, the short-lived TCP connections determine their data size and data transmission intervals based on the Scalable URL Reference Generator (SURGE) model [20]. SURGE is a realistic Web workload generation tool that mimics a set of real users accessing a server. Table 2 shows the parameters of the SURGE model.

Table 2 Parameters for the SURGE model.

Component	Function	Parameters
Size-Body	$p(x) = \frac{e^{-(lnx-\mu)^2/2\sigma^2}}{x\sigma\sqrt{2x}}$	$\mu = 9.375$, $\sigma = 1.318$
Size-Tail	$p(x) = \alpha k^\alpha x^{-\alpha+1}$	$k = 133K$, $\alpha = 1.1$
Interval	$p(x) = \alpha k^\alpha x^{-\alpha+1}$	$k = 1$, $\alpha = 1.5$

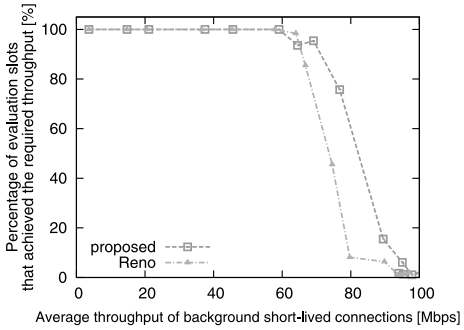


Fig. 9 Performance with co-existing short-lived connections.

3.3.1 Effect of Background Short-Lived TCP Connections

Figure 9 shows the simulation results when the proposed mechanism tries to achieve a throughput of $bw = 20$ (Mbps) for 10 long-lived (persistent) TCP connections and the background traffic generated by short-lived TCP Reno connections. For the purpose of comparison with the proposed mechanism, this figure also shows the results obtained when Sender 1 uses 10 normal TCP Reno connections without the proposed mechanism. This figure shows that the proposed mechanism can provide a higher probability of achieving the required throughput than can TCP Reno. However, compared with Figs. 5–8, the probability drops sharply when the amount of background traffic increases. This is because the traffic from the short-lived TCP connections is more aggressive than that from the long-lived TCP connections due to its bursty nature, and the proposed mechanism is not adept at stealing bandwidth from short-lived connections.

3.3.2 Protecting a Mixture of Multiple Connections

We next consider the case in which the proposed mechanism controls the throughput of the mixture of traffic of long-lived and short-lived TCP connections. We set $bw = 20$ (Mbps) and $N_{\rm pm} = 10$, where five connections are long-lived connections and the remaining five connections are short-lived connections. Figure 10 shows the ratio of the number of evaluation slots, in which the proposed mechanism can obtain the required throughput, and the sum of the average throughput of the short-lived connections. This figure also shows the sum of the average throughput of the five short-lived connections when the long-lived connections of the proposed mechanism do not exist in the network. This

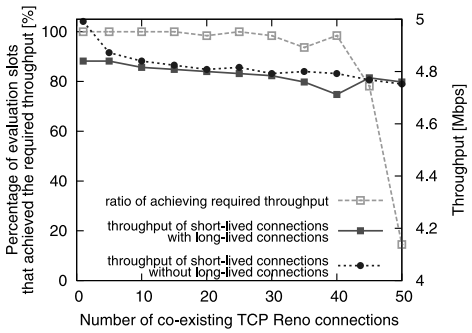


Fig. 10 Case for a mixture of connections.

figure indicates that the proposed mechanism can obtain the required throughput with a high probability for long-lived connections. Furthermore, the average throughput of the short-lived traffic is approximately equivalent to that without the long-lived connections. This means that the proposed mechanism can provide the throughput required by long-lived connections but does not harm the performance of short-lived connections.

4. Implementation and Evaluations on Actual Networks

In this section, we outline the implementation of the proposed mechanism in a Linux 2.6.16.21 kernel system [21], and then evaluate the performance of it in actual networks.

4.1 Implementation Overview

Figure 11 shows the architecture of the proposed mechanism implemented in the Linux 2.6.16.21 kernel system. When new data is generated at the application, the data is passed to the TCP layer through the socket interface [22]. The data is passed to the IP layer after TCP protocol processing by the `tcp_output()` function, and the resulting IP packets are injected into the network. Conversely, an ACK packet that arrives at the IP layer of the sender host is passed to the `tcp_input()` function for TCP protocol processing. The congestion window size of a TCP connection is updated when an ACK packet is passed to the `tcp_input()` function. Therefore, the control program for the congestion window size for the proposed mechanism should be implemented in the `tcp_input()` function. The Linux 2.6.16 kernel system unifies the interfaces for congestion control mechanisms and enables us to implement the congestion control algorithm as a module. In this paper, we implement the proposed mechanism as a module in the Linux 2.6.16.21 kernel system.

The `tcp_input()` function calls the `cong_avoid()` function and updates the congestion window size when an ACK packet arrives. The module for the proposed mechanism determines the congestion window size according to the algorithm described in Sect. 2, and splits the time into the evaluation/control slots in the `cong_avoid()` function. On

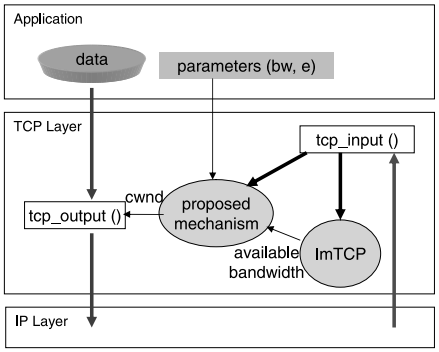


Fig. 11 Outline of implementation architecture.

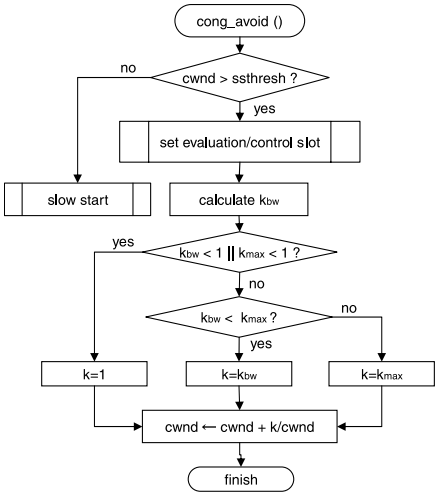


Fig. 12 Flow chart of the `cong_avoid()` function.

the other hand, ImTCP, which we utilize to obtain the available bandwidth of the network path, calculates the available bandwidth in the `tcp_input()` function [7]. The proposed mechanism learns from ImTCP the available bandwidth in the `cong_avoid()` function, and changes the degree of the increase of the congestion window size based on the bandwidth value.

Figure 12 shows the flow chart of the `cong_avoid()` function of the proposed mechanism.

First, the `cong_avoid()` function compares the congestion window size (*cwnd*) and the slow start threshold (*ssthresh*). When *cwnd* is smaller than *ssthresh*, the congestion window size is updated by the slow start algorithm as TCP Reno. On the other hand, when *cwnd* is larger than *ssthresh*, the congestion window size is determined based on the algorithm of the proposed mechanism. In the congestion avoidance phase, the proposed mechanism checks the passed time from the beginning of the present evaluation/control slots and judges the end of the slots. When the passed time is longer than the length of the evaluation/control slots, the proposed mechanism calculates the average throughput in the slot and initializes the variables for the next slots. Next, the increase degree of the congestion window size is determined on consideration of k_{max}, k_{min} and k_j^{bw}, which is calculated according to Eq. (2). Finally, *cwnd* is updated by Eq. (1).

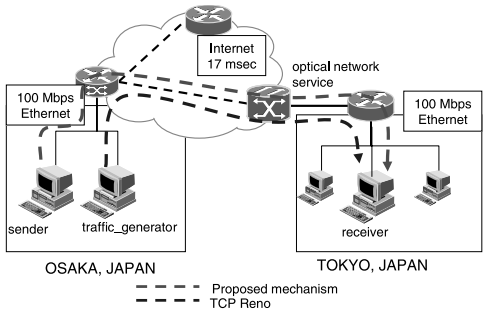


Fig. 13 Experimental system in the Internet environment.

Table 3 PC specifications of the Internet environment.

	sender	traffic_generator	receiver
CPU	P4 3.40 GHz	Xeon 3.60 GHz	Xeon 2.66 GHz
Memory	1024 MB	2048 MB	1024 MB
Kernel	Linux 2.6.16.21	Linux 2.6.17	Linux 2.4.21

4.2 Experiment Results

We confirm the performance of the proposed mechanism in the commercial Internet environment. Figure 13 shows the network environment, which consists of two local area networks in Osaka, Japan and Tokyo, Japan, which are connected to the Internet.

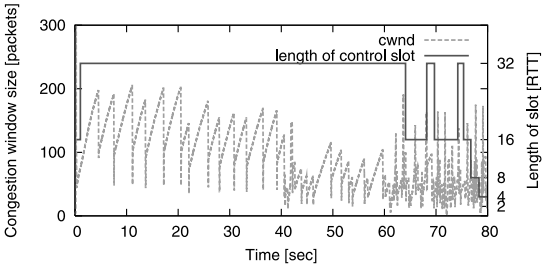
The network environment consists of an endhost that generates cross traffic (traffic_generator), an endhost that uses the proposed mechanism (sender), and an endhost that receives packets from both endhosts (receiver) across the Internet. The path of the commercial Internet network between Osaka and Tokyo passes through 100-Mbps optical fiber services, and the local area networks in Osaka and Tokyo are 100-Mbps Ethernet networks. Table 3 shows the specifications of the endhosts of the experimental system.

Through preliminary investigations, we confirmed the following characteristics regarding the network between Osaka and Tokyo:

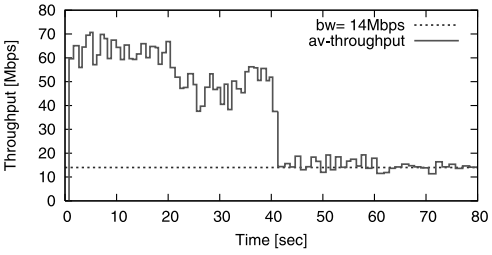
- Seventeen hops exist in the network path from Osaka to Tokyo.
- The minimum value of RTTs is 17 msec.
- The upper limit of the bandwidth between Osaka and Tokyo is 70 Mbps.

In this experiment, we set $e = 32$ for the length of the evaluation slot; s, the length of the control slot, is initialized to 16. The cross traffic is generated by traffic_generator, and the packets are sent to receiver. We set the size of the TCP socket buffer on traffic_generator to limit the maximum throughput of each TCP Reno connection to approximately 3 Mbps, and the amount of the cross traffic is changed by the number of TCP Reno connections. The size of the TCP socket buffer on sender and receiver is large enough.

We first evaluate the behavior of the proposed mecha-



(a) Changes in window size and control slot length



(b) Changes in throughput

Fig. 14 Changes in cwnd, control slot length and throughput in the Internet experiment.

nism against the change in the amount of cross traffic. In this experiment, we use one connection for the proposed mechanism, and set *bw* to 14 (Mbps), which is equal to 20% of the bottleneck link capacity. To change the congestion level of the network, we change the number of TCP Reno connections between traffic_generator and receiver to 0, 5, 25, and 40 at every 20 seconds. Figure 14(a) shows the changes in the congestion window size and the length of the control slot, and Fig. 14(b) shows the changes in the average throughput in each evaluation slot.

From the results for 0–20 seconds in Fig. 14, when there is only one connection for the proposed mechanism, the upper limit of the throughput between Osaka and Tokyo is 70 Mbps because the proposed mechanism can obtain approximately 70 Mbps at most. In addition, the results after 20 seconds in Fig. 14 are almost equivalent to the results on the simulation experiments. That is, the results for 20–40 seconds show that the proposed mechanism can obtain more than the required throughput by keeping the same behavior as the normal TCP connection, and the results for 40–60 seconds show that it can achieve the required throughput by having a faster increase in the congestion window size. From the results after 60 seconds, we observe that the length of the control slot is changed to a smaller value, and the proposed mechanism can achieve the required throughput. Thus, we have confirmed that the proposed mechanism can effectively obtain the required throughput by changing the degree of the increase of the congestion window size and the length of the control slot according to the network congestion level in the commercial Internet environment.

We next evaluate the average throughput in each evaluation slot when we set *bw* to 7 and 14 Mbps and the number of TCP Reno connections to 30 and 50. Figure 15 shows

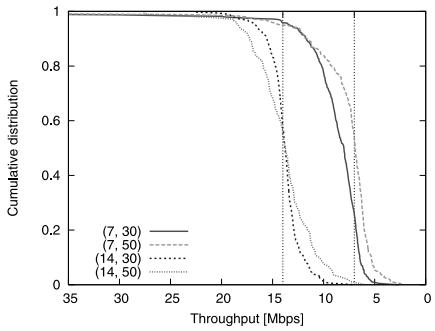


Fig. 15 CDF of the throughput in the Internet experiment.

the cumulative distribution function (CDF) of the average throughput in each evaluation slot.

From Fig. 15, we can observe that the ratio of achieving the required throughput is slightly smaller than that in the simulation results in the previous section. One possible reason is that there are short-lived connections, including web traffic, in the Internet environment. This traffic has a highly bursty nature. Since the proposed mechanism is based on TCP, it cannot adapt to the shorter-term changes of the network condition than its RTT. Another reason is that the measurement accuracy of the available bandwidth of the network path operated by ImTCP becomes slightly degraded in the actual Internet environment. However, most of the evaluation slots which cannot achieve the required throughput can achieve throughput close to the required throughput. Thus, we conclude that the proposed mechanism works well even in the actual Internet environment.

5. Conclusion

In this paper, the author focused on upper-layer applications requiring constant throughput, and proposed an TCP congestion control mechanism for achieving the required throughput with a high probability. The proposed mechanism modifies the degree of increase of the congestion window size of a TCP connection in the congestion avoidance phase by using the information on the available bandwidth of the network path. Through simulation experiments, we demonstrated that the proposed mechanism for one connection can achieve the required throughput with a high probability, even when there is almost no residual bandwidth on the network path. We also reported that the extended mechanism performs effectively to provide the required throughput for multiple TCP connections. In addition, we implemented the proposed mechanism on a Linux 2.6.16.21 kernel system and confirmed from the implementation evaluation that the proposed mechanism works well in actual networks.

We believe that the fairness comparison is desirable when multiple flows in the network utilize the proposed mechanism. Note that the performance of the proposed mechanism is highly dependent on the estimation results of the available bandwidth given by ImTCP. However, the current version of ImTCP does not give accurate estimation of available bandwidth especially when there exists a small

number of ImTCP flows in the simple network such as in Figure 3. From extensive simulation results, we confirm that the proposed mechanism can not provide good performance when multiple flows utilize the proposed mechanism.

The possible solutions for the above problem are as follows. The one is to modify the mechanisms of ImTCP to behave well in multiple-flow situation. The another solution is to utilize other algorithms than ImTCP to estimate the available bandwidth. Note that the proposed mechanism in this paper is independent on the detailed mechanism of bandwidth estimation. In addition, we expect that when the accurate estimation results of the available bandwidth are obtained, the proposed mechanism would provide the same level of the fairness property among co-existing connections as TCP Reno, since the proposed mechanism is quite a simple modification to TCP Reno (just changing the increasing slope of the congestion window). So, we would tackle this problem as one of the important future work.

In addition, we will evaluate the performance of the proposed mechanism in other actual network environments. In addition, we would like to confirm the applicability of the proposed mechanism for actual upper-layer applications, such as a real-time video streaming.

References

[1] J. Wroclawski, "The use of RSVP with IETF integrated services," RFC 2210, Sept. 1997.
[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated service," RFC 2475, Dec. 1998.
[3] J. Postel, "User datagram protocol," RFC 768, Aug. 1980.
[4] M. Handley, S. Floyd, J. Padhye, and J. Widmer, "TCP friendly rate control (TFRC): Protocol specification," RFC 3448, Jan. 2003.
[5] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications," RFC 1889, Jan. 1996.
[6] H.J. Lee, T. Chiang, and Y.Q. Zhang, "Scalable rate control for MPEG-4 video," IEEE Trans. Circuits Syst. Video Technol., vol.10, no.6, pp.878–894, Sept. 2000.
[7] T. Tsugawa, G. Hasegawa, and M. Murata, "Implementation and evaluation of an inline network measurement algorithm and its application to TCP-based service," Proc. NOMS 2006 E2EMON Workshop 2006, April 2006.
[8] J.B. Postel, "Transmission control protocol," RFC 793, Sept. 1981.
[9] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: A simple model and its empirical validation," Proc. ACM SIGCOMM'98, Sept. 1998.
[10] W.R. Stevens, TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley, Reading, Massachusetts, 1994.
[11] C.L.T. Man, G. Hasegawa, and M. Murata, "ImTCP: Tcp with an inline measurement mechanism for available bandwidth," Computer Communications Journal, vol.29, pp.1614–1626, June 2006.
[12] C.L.T. Man, G. Hasegawa, and M. Murata, "A simultaneous inline measurement mechanism for capacity and available bandwidth of end-to-end network path," IEICE Trans. Commun., vol.E89-B, no.9, pp.2469–2479, Sept. 2006.
[13] Microsoft Corporation, "Microsoft Windows Media — Your Digital Entertainment Resource," available from http://www.microsoft.com/windows/windowsmedia/
[14] RealNetworks Corporation, "Rhapsody & RealPlayer," available from http://www.real.com/
[15] Skype Technologies Corporation, "Skype — The whole world can talk for free," available from http://www.skype.com/
[16] B. Gleeson, A. Lin, J. Heinanen, G. Armitage, and A. Malis, "A framework for IP based virtual private networks," RFC 2764, Feb. 2000.
[17] T.V. Project, "UCB/LBNL/VINT network simulator - ns (version 2)," available from http://www.isi.edu/nsnam/ns/
[18] I. Maki, G. Hasegawa, M. Murata, and T. Murase, "Throughput analysis of TCP proxy mechanism," Proc. ATNAC 2004, Dec. 2004.
[19] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, "SOCKS protocol version 5," RFC 1928, April 1996.
[20] P. Barford and M. Crovella, "Generating representative Web workloads for network and server performance evaluation," Measurement and Modeling of Computer Systems, vol.26, pp.151–160, July 1998.
[21] the Linux Kernel Organization, Inc., "The linux kernel archives," available from http://www.kernel.org/
[22] G.R. Wright and W.R. Stevens, TCP/IP Illustrated, Volume 2: The Implementation, Addison-Wesley, 1995.

Go Hasegawa received the M.E. and D.E. degrees in Information and Computer Sciences from Osaka University, Osaka, Japan, in 1997 and 2000, respectively. From July 1997 to June 2000, he was a Research Assistant of Graduate School of Economics, Osaka University. He is now an Associate Professor of Cybermedia Center, Osaka University. His research work is in the area of transport architecture for future high-speed networks and overlay networks. He is a member of the IEEE.

Kana Yamanegi received the ME degree in Information Science and Technology from Osaka University in 2007. She is now at Matsushita Electric Industrial Co., Ltd.

Masayuki Murata received the M.E. and D.E. degrees in Information and Computer Sciences from Osaka University, Japan, in 1984 and 1988, respectively. In April 1984, he joined Tokyo Research Laboratory, IBM Japan, as a Researcher. From September 1987 to January 1989, he was an Assistant Professor with Computation Center, Osaka University. In February 1989, he moved to the Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University. From 1992 to 1999, he was an Associate Professor in the Graduate School of Engineering Science, Osaka University, and from April 1999, he has been a Professor of Osaka University. He moved to Advanced Networked Environment Division, Cybermedia Center, Osaka University in 2000, and moved to Graduate School of Information Science and Technology, Osaka University in April 2004. He has more than two hundred papers of international and domestic journals and conferences. His research interests include computer communication networks, performance modeling and evaluation. He is a member of IEEE, ACM, The Internet Society, and IPSJ.