ELSEVIER

Computer Networks 39 (2002) 769–787

COMPUTER
NETWORKS

www.elsevier.com/locate/comnet

Dynamic buffer management scheme based on rate estimation in packet-switched networks

Jeong-woo Cho ^{*}, Dong-ho Cho

*Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology (KAIST),
373-1 Kusong-dong, Yusong-gu, Taejeon 305-701, South Korea*

Received 24 March 2001; received in revised form 22 October 2001; accepted 28 January 2002

Responsible Editor: G. Morabito

Abstract

While traffic volume of real-time applications is rapidly increasing, current routers do not guarantee minimum QoS values of fairness and drop packets in random fashion. If routers provide a minimum QoS, resulting less delays, more fairness, and smoother sending rates, TCP-friendly rate control (TFRC) can be adopted for real-time applications. We propose a dynamic buffer management scheme that meets the requirements described above, and can be applied to TCP flow and to data flow for transfer of real-time applications. The proposed scheme consists of a virtual threshold function, an accurate and stable per-flow rate estimation, a per-flow exponential drop probability, and a dropping strategy that guarantees fairness when there are many flows. Moreover, we introduce a practical definition of active flows to reduce the overhead coming from maintaining per-flow states. We discuss how proposed scheme motivates real-time applications to adopt TFRC.

© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Congestion and flow control; Packet network; Buffer management; Router mechanism

1. Introduction

TCP is the most widely used transport protocol on the Internet and is appropriate for FTP and Telnet, which both require reliability. However, because it uses an additive increase multiplicative decrease (AIMD) algorithm and induces coarse

timeouts, it can neither ensure smoothly changing sending rate nor can be used for real-time applications [23]. Because most current routers use Drop Tail as a buffer management scheme, which does not guarantee fairness and delay bound, there has been no motivation for real-time applications to use end-to-end congestion control mechanisms. For these reasons, real-time applications use robust congestion control schemes than TCP congestion control [14]. Even though Drop Tail is a simple buffer management scheme, it tends to penalize bursty traffic, such as TCP traffic, does not guarantee fairness, and adds unnecessary delays

^{*} Corresponding author. Tel.: +82-428698067; fax: +82-428670550.

E-mail addresses: ggumdol@comis.kaist.ac.kr (J.-w. Cho), dhcho@ee.kaist.ac.kr (D.-h. Cho).

because it does not drop any packets before the buffer space is fully exhausted.

These problems can be partially solved by using a number of techniques. If a router can maintain a separate queue for each flow, per-flow queueing schemes, such as FQ, SFQ, and DRR can be used [1,13,20]. Although these schemes solve many problems, they require a router to maintain a separate queue, and per-flow queueing and per-flow scheduling are very complex to implement. Furthermore, FQ requires huge buffer size to support many flows. For example, to support one thousand flows with FQ, FQ requires a router to keep several Mbytes buffer size assuming that each IP packet size is about 1 KB. Although SFQ reduces overhead caused by mapping from source–destination address pair to the corresponding queue, it requires even larger router buffer size than FQ to guarantee a comparable fairness compared to FQ. Moreover, in the present situation, most of routers use a single first-in first-out (FIFO) buffer shared by all flows.

Adopting a single FIFO buffer, core-stateless fair queueing (CSFQ) [22] uses per-flow state only in edge routers. Entering the network, packets are marked with an estimate of their sending rate. A core router compares the estimate of each rate with the fair share of that flow and preferentially drops packets if the flow arrives at a higher rate than its fair share. Although CSFQ is much fairer, it requires an extra field in the IP header of every packet and CSFQ must be installed in contiguous fashion on routers.

Random early detection (RED) [6] and flow random early drop (FRED) [10] are the foundation of buffer management schemes because they are practicable and are designed in the consideration of burstiness of TCP flows. RED prevents full exhaustion of buffers and drops packets before congestion becomes severe. However, it does not prevent unresponsive flows from monopolizing buffer space, and TCP-friendly flows attain only a fraction of their fair share [4]. Also, it cannot control queue size effectively and cannot prevent buffer overflow when there are many flows [3]. To address the problem of unresponsive flows, in [4], authors stressed on the need for end-to-end con-

gestion control. Furthermore, they insisted that there should be some mechanism on the network to identify and regulate unresponsive flows. Techniques to identify and punish unresponsive flows have been identified in [5,11]. While these proposals are simple and feasible schemes that solve the problem of unresponsive flows, they can punish unlucky TCP-friendly flows with non-zero probability. Therefore, we do not think that these schemes can be adopted in the present situation. FRED uses per-flow states to solve the problem of unresponsive flows. Although FRED cannot prevent buffer overflow for many flows, it is much fairer than RED and effectively regulates unresponsive flows.

Although RED and its variants can be satisfactory for applications that only require reliability, real-time applications require a router to provide more functions. Moreover, to motivate real-time applications to use TCP-friendly rate control (TFRC) [7,8,19,21] a minimum quality of service (QoS) should be guaranteed. First of all, a router should be able to eliminate unnecessary queueing delays because multimedia applications do not want to experience large queueing delays. Second, a buffer management scheme should be able to regulate unresponsive CBR and UDP flows not to take unfairly large share. Third, a router should support more flows fairly with limited buffer size because IP packet size is relatively large and such a large packet size requires large buffer and results in longer queueing delays. To solve these problems, we propose a new buffer management scheme that ensures better fairness between TCP-friendly flows and unresponsive flows, less delays, and smoother sending rates.

The organization of this paper is as follows: We discuss general requirements of buffer management schemes in packet-switched networks in Section 2. In Section 3, the detailed algorithm we propose is explained with a discussion of mechanics of operation. In Section 4, we show simulation results obtained using our proposed scheme, RED, FRED, and DRR, and analyze the results. Section 5 presents an analysis of various topics relating to our scheme. In the last section, we present a conclusion.

2. Requirements of buffer management scheme

RED is a simple and powerful buffer management scheme that drops packets from each flow in proportion to the amount of bandwidth the flows uses on the output link [10], assuming that all flows exhibit the same behavior as TCP flows do in view of packet drop events. However, RED cannot prevent buffer overflow for many flows, cannot regulate unresponsive flows, and is unfair even among TCP flows because it drops packets randomly [3,4,10,11]. We suggest the following functions that an intelligent buffer management scheme should support:

- (1) low queueing delays;
- (2) control of the queue size to prevent overflow and underflow;
- (3) regulation of unresponsive flows and fairness;
- (4) smooth sending rates for each flow.

In this paper, we define “flow” as a source–destination IP address pair to distinguish each flow transferring multicast traffic data and to guarantee fairness for those flows. For example, address pairs (A,B) and (C,B) are treated as different flows. Although definition of flow can be extended to each TCP port or UDP port, currently, header processing of layer 4 in routers is not common.

2.1. A scalable and fair buffer management scheme

In ideal situations, routers can provide fairness even with a small buffer. But, TCP, which is dominant transport protocol, requires a large buffer because it uses window-based congestion control that causes frequent coarse timeouts when there is insufficient buffer space. This results in short-term unfairness. Although TCP flows require that at least 4 packets per-flow should be buffered in routers to prevent coarse retransmit timeouts [16], most routers provide very small buffers because large buffers without an active buffer management scheme results in unacceptably long delays and long response times.

To guarantee fairness with TCP flows, a buffer management scheme should allow each flow to

buffer at least 4 packets when congestion is not severe. However, just allowing each flow to buffer at least 4 packets can be unfair when bursty TCP flows and unresponsive flows (e.g. CBR flows) coexist. To alleviate this unfairness, a buffer management scheme also should drop packets according to each flow’s estimated throughput.

A router do not provide a large buffer because a large buffer inevitably results in longer queueing delays. Therefore, a buffer management should gracefully adjust per-flow queue sizes according to the number of active flows. When congestion is severe, for example, in case that there are 10,000 flows and the router buffer size is 1000 KB, a flow can buffer only 100 bytes in average. Assuming that an IP packet size is 500 bytes, a flow can buffer only 0.2 packets in average. Guaranteeing fairness in such a condition is not easy due to the following reasons: (1) In such a situation, estimating per-flow rates is not an easy task. Because TCP’s retransmit timeout value is doubled for every consecutive retransmit timeout, estimating per-flow rates and guaranteeing fairness in such a situation are difficult. (2) Maintaining several millions of per-flow states in a router is also not an easy task. If per-flow states are implemented in conventional memory such as RAM, mapping from source–destination address pair to the corresponding state requires $O(\log N)$ (where N is the number of flows) time complexity. If a new and practical definition of flows could reduce this complexity, it would be feasible for routers to maintain such a reduced number of per-flow states.

To minimize unnecessary queueing delays, to guarantee fairness, and to allow a flow to buffer at least 4 packets, we propose a virtual threshold function, shown in Fig. 1. In this figure, we divide router operation into three modes. Each flow can buffer up to $vmax_q/N_{flow}$ bytes. Because each TCP flow does not occupy $vmax_q/N_{flow}$ bytes all the time, exploiting the burstiness of TCP, we can maintain an average queue size to a target value, which is shown as $target_q$. *In no congestion mode*, there is sufficient buffer space to allow each flow to buffer at least 8000 bytes. In this mode, a router can provide highly satisfactory QoS. *In moderate congestion mode*, there is insufficient buffer space

Fig. 1. Virtual threshold function vs. number of active flows.

and the queueing delay increases. In this mode, we allow each flow to buffer a smaller number of packets as the number of flows increases. In *severe congestion mode*, each flow can buffer only a minimum number of packets, that is to say, 4000 KB, and we cannot provide low delays. We also introduce a dropping strategy named “Fair Drop” that guarantees fairness when there are many flows. Fair Drop operates when queue size is larger than fd_{th} . A buffer management scheme should limit router’s queue size to a certain value to prevent buffer overflows. Fair Drop is designed to prevent buffer overflows and to limit maximum queueing delays still maintaining satisfactory fairness values.

As demands on delay and per-flow buffer size can vary, the virtual threshold function can also vary according to these demands.

2.2. Why should a router drop packets periodically?

Achieving smooth sending rates requires periodic dropping of packets. However, RED drops packets randomly as shown in this section.

TCP packet losses are detected based on the following two ways: (1) The TCP sender can detect

them either when it receives *triple-duplicate* acknowledgements (four ACK’s with the same sequence number), or (2) when *retransmit timeouts* occur [17]. We define the *congestion cycle* CC_i as the i th period between two loss indications and define α_i as the number of packets including the first packet loss in CC_i . If RED is in steady state, which means no recent change in the number of flows, packets for flow i are dropped with nearly constant drop probability p . Therefore, α_i is distributed geometrically as follows:

$$P[\alpha_i = k] = (1 - p)^{k-1} p, \quad k = 1, 2, \dots \quad (1)$$

As can be seen from this equation, each flow experiences geometrically distributed inter-drop times. The mean and standard deviation of α_i are as follows:

$$E[\alpha_i] = \sum_{k=1}^{\infty} (1 - p)^{k-1} p k = \frac{1}{p}, \quad (2)$$

$$S[\alpha_i] = \frac{1}{p} \sqrt{1 - p}. \quad (3)$$

We can determine that $E[\alpha_i] = 10$ and $S[\alpha_i] = 9.5$ with $p = 0.1$, indicating that some flows buffer more than a sufficient number of packets and

others buffer fewer than the necessary number of packets. This feature of RED causes unfairness, inefficient buffer usage, and rough sending rates. To avoid these problems, routers should drop packets periodically.

3. BARE algorithm

We propose buffer management based on rate estimation (BARE) scheme which solves the problems discussed in Section 2. BARE consists of virtual threshold function that eliminates unnecessary queueing delay and prevents buffer overflows and underflows, an accurate per-flow rate estimation that estimates each flow's rate, a per-flow exponential drop probability that keeps TCP flows from unresponsive flows, and Fair Drop that guarantees fairness even when the number of flows is very large.

3.1. Description of BARE's operation

BARE's basic algorithm is depicted in Fig. 2. To understand the detailed operation of BARE algorithm, it is necessary to refer to the detailed pseudocode of BARE algorithm in Appendix A.

(1) For each packet's arrival, global queue size q is compared with $vmax_q$ to prevent unnecessary fluctuation of global queue size q and is compared with max_q to prevent buffer overflows. (2) If there is no flow state for this packet, BARE assigns a flow state to this flow. Because a buffer management scheme can support only a finite number of flow states, when all flow states are occupied, BARE assigns a used flow state whose per-flow queue size is 0 to this new flow. Because global queue size q is controlled not to exceed half of buffer size BS by BARE, if the maximum number of per-flow states max_{flow} is set to a proper value, at least one per-flow state would have 0 per-flow queue size. Therefore, a packet whose per-flow state is not registered in per-flow states is never dropped. (3) Because a buffer management scheme should regulate global queue size q to prevent buffer overflows, when global queue size q is larger than Fair Drop threshold fd_{th} , an arriving packet

is dropped. (4) BARE drops packets according to the estimated per-flow rate. (5) For each packet's departure, BARE finds the flow number of this departing packet and updates $\beta[i]$ and global queue sizes. (6) If flow i 's per-flow queue size is 0 for more than timeout value *Timeout_Value*, flow i expires and the flow state for the flow is deleted.

BARE determines a maximum per-flow buffer size depending on the number of currently active flows and drops packets based on the rate estimation of each flow [22]. As an estimate of per-flow share, either a per-flow average queue size estimation in [6] or a per-flow rate estimation in [22] can be used. In fact, using per-flow average queue size requires replacing “ $\exp(-dt/K)$ ” with a constant “ λ ” and replacing rate estimates with average queue estimates in code line 31 in Appendix A as follows. (In addition to this replacement, a portion of code should be modified.)

$$rate[i] \leftarrow (1 - e^{-dt/K}) \frac{p.size}{dt} + e^{-dt/K} rate[i], \quad (4)$$

$$avg_q[i] \leftarrow (1 - \lambda)q[i] + \lambda avg_q[i]. \quad (5)$$

The per-flow buffer occupancy of flow i is proportional to the per-flow output rate of flow i with the FIFO discipline [10]. Therefore, we can guess that these two approaches achieve the same performance. However, the usage of per-flow average queue size as an estimate of the per-flow share is not as precise and efficient as that of per-flow rate estimation. When per-flow average queue size is used as an estimate of per-flow share, filtering of unnecessary noise and quick responsiveness to rapid rate fluctuations cannot be obtained simultaneously. Let assume that end of congestion cycle CC_i is caused only by triple-duplicate ACKs, there are only periodic packet losses, and the round-trip time is fixed to RTT. W_i is the maximum window size in congestion cycle CC_i . With these assumptions, the inter-packet buffering time of TCP varies from RTT/W_i to $(2RTT)/W_i$, so the per-flow share cannot be calculated accurately without dependency on dt . If there are substantial packet losses caused by timeouts, this discrimination becomes more significant. Therefore, we have chosen to use

Fig. 2. BARE algorithm.

rate estimation as a method for estimating per-flow share. Rate estimation in code line 31 in Appendix A is robust to various packet length distributions and is proven to asymptotically converge to the real rate [22].

3.2. Per-flow exponential drop probability

As shown in code line 24 in Appendix A, BARE drops packets for flow i with following drop probability:

Fig. 3. Exponential drop probability.

$$p[i] = \left(\frac{q[i]}{\max_{th}} \right)^{\beta[i]}. \quad (6)$$

Based on per-flow rate estimation and comparison of current average queue size with target_q, BARE either increases or decreases $\beta[i]$. Flow i experiences a high drop probability with a small $\beta[i]$ and experiences a low drop probability with a large $\beta[i]$. Upon decrease, $\beta[i]$ is divided by the rate_{ratio}. Upon increase, $\beta[i]$ is multiplied by the constant value of α .

By introducing $\beta[i]$ as an exponent of drop probability for each flow, drop probability of each flow can be adjusted efficiently as shown in Fig. 3. First, $\beta[i]$ is large when i is a TCP or TCP-friendly flow. In this case, using $\beta[i]$ as an exponent of drop probability makes each TCP and TCP-friendly flow experience periodic packet drops. Because TCP is very sensitive to small drop probability such as 0.1, using $\beta[i]$ as an exponent can prevent unnecessary packet drops for flows using smaller than fair share. For example, if the sending rate of flow i does not exceed fair share (when $q[i] \leq \max_{th}/2$), packets of flow i are dropped with a negligible probability, i.e., $0.5^{10.0} = 0.001$. Therefore, BARE's dropping strategy does not drop flow i 's packets when the number of buffered packets for flow i is less than the average number of buffered packets for other flows. With this dropping method, packets with low rate flows are rarely dropped. Secondly, $\beta[i]$ is small when flow i is unresponsive and is using

more than its fair share. In this case, using $\beta[i]$ as an exponent of drop probability allows high drop probability such as 0.9.

With per-flow exponential adjustment of drop probability, we can achieve a high degree of fairness and smooth sending rates because packets of TCP-friendly flows are dropped nearly periodically. Furthermore, the queue size of each flow is well regulated and each flow is not allowed to buffer more than the necessary number of packets.

3.3. Choosing K

The choice of decay factor K involves several tradeoffs. First, while a smaller K value increases the system responsiveness to rapid rate fluctuations, a larger K value better filters noise and avoids potential system instability. Second, K should be large enough to smooth the estimated sending rates of TCP flows because these rates are estimated to be high when TCP flows have large window sizes just before packet drop events. To control these effects, K can be decided as follows:

$$K \leftarrow C \times \frac{\text{Average Packet Size} \times N}{BW} \quad (7)$$

where N can be substituted with N_{flow} and Average Packet Size is the average IP packet size, and BW is the link speed or service rate of a router. We found that C should be 10–30 through numerous simulations and overall performance is very insensitive to various K values. But (1) using a higher value of K requires a router to maintain unacceptably large number of per-flow states, and (2) guaranteeing fairness when there is insufficient buffer space such that each TCP flow can buffer only up to 0.1 packets in average is meaningless. Moreover, (3) frequent change of K can induce implementation complexity. Considering these three points, as a rule of thumb, we recommend that K should be one to three times value of the average queueing delay, which can be calculated based on dividing the average queue size by the link speed.

3.4. Practical definition of active flows

Because routers have a limited memory, per-flow states should be deleted properly, but neither

too often nor too seldom. With too frequent deletion of the per-flow states, defects caused by FRED's frequent deletion of per-flow states can appear. FRED's defects due to the frequent deletion allows unresponsive flows to take more than fair share because the number of active flows are underestimated and fair share is overestimated. With too infrequent deletion of the per-flow states, the number of active flows are overestimated and fair rate can be underestimated so that all flows could be dropped simultaneously. From code line 31, the rate of flow i is updated according to Eq. (4). In Eq. (4), if $e^{-dt/K}$ is set to $e^{-3} = 0.0498$, dt should be $3K$ and Eq. (4) becomes as follows:

$$\text{rate}[i] \leftarrow 0.9502 \times \frac{\text{p.size}}{dt} + 0.0498 \times \text{rate}[i] \approx \frac{\text{p.size}}{dt}. \quad (8)$$

Therefore, routers do not have to maintain the per-flow state of flow i if $dt = 3K$ s has elapsed since the last buffering operation of flow i . Therefore, *Timeout_Value* used for deleting per-flow states is set to $3K$. This can greatly reduce the overhead coming from maintaining per-flow states.

Although there have been many approaches to buffer management schemes with usage of per-flow states, maintaining per-flow states has been considered to be impractical and not scalable. But, we found that we can reduce the overhead of maintaining per-flow states. *Our main motivation for definition of active flows is that a very weak flow does not need to be regarded as an active flow.* For example, assume that a router's link speed is 20 Mbps, a router buffer size is 160 KB, an IP packet size is 1 KB, K is 150 ms, and *Timeout_Value* is 450 ms. *Timeout_Value* can be regarded as a time window because a flow that has not been buffered for *Timeout_Value* is deleted. This corresponds to maximum queueing delay of 64 ms and average inter-packet time of 0.4 ms. Because a flow that has not been buffered for more than *Timeout_Value* is too weak to be regarded as a flow, we can decide that up to $(450 + 64)/0.4 = 1285$ flows need to be regarded as active flows. For another example, if a router supports an OC-12c link that has a capacity of 622 Mbps and 4 MB buffer, and an IP packet size is 1 KB, this corresponds to maximum queueing delay of 51.4 ms, K of 60 ms, *Timeout_Value* of 180

ms, and average inter-packet time of 12.9 μ s. Therefore, this router has to maintain only up to 14,000 per-flow states. Although this number of per-flow states may be considered as a large number of flows, it is well known that current ATM switches support at least 64,000 VCs [9]. Furthermore, numbers of flows that should be treated as active flows are still overestimated in the above calculations because the probability that every packet belongs to each distinct flow in a time window is low due to TCP's burstiness.

3.5. Fair Drop

Although our new definition of active flows greatly reduced the overhead of maintaining per-flow states, maintaining such a large number of flows results in longer delays. If we allow 1 KB per an active flow with a 20 Mbps link to support 1500 flows, it corresponds to a buffer size of 1.5 MB and maximum queueing delay of 0.6 s that is so long that flows transferring real-time application data would complain. So, such a large buffer is not likely to be used in practice. To become more realistic, a router would try to support such a large number of flows with a smaller buffer so that each flow can buffer only a fraction of a packet in average such as 0.2 or 0.5 packets. As discussed in Section 2.1, guaranteeing fairness for a large number of flows with a small buffer is not trivial. To guarantee fairness with a small buffer, we introduce Fair Drop scheme. *The main motivation for Fair Drop is that recently buffered flows do not need to be buffered again if the buffer size is not sufficiently large and all flows cannot be buffered simultaneously.* When the global queue size is above a specified threshold value, Fair Drop scheme drops packets of flow i if the flow has been buffered recently and its flow state is still maintained by BARE.

4. Simulation results

BARE, RED, FRED, and DRR are compared based on simulation results. While RED is selected as a fundamental scheme due to its simplicity, FRED and DRR is selected as comparable schemes with BARE:

- **RED:** This scheme is significantly more sophisticated than Drop Tail and is designed for routers with a single FIFO buffer. RED drops packets before congestion becomes severe and controls the average queue size between $\max_{th}$ and $\min_{th}$ values. When the average queue size is less than $\min_{th}$, there is no packet drop. When the average queue size is greater than $\max_{th}$, all packets are dropped. When the average queue size is between two thresholds, the packet drop probability is increased linearly in proportional to the average queue size.
- **FRED:** This is an extended version of RED for partial solution of the problem of unresponsive flows. FRED maintains per-flow states for all flows that have a non-zero queue size in the router buffer. Using this per-flow state, FRED preferentially drops the packets of flows that have queue sizes larger than the average per-flow queue size. It unconditionally drops the packets of flows that (1) have queue sizes two times greater than the average per-flow queue size or (2) experience many packet drops. It randomly drops the packets of flows that have larger queue size than the average per-flow queue size with the probability proportional to the average queue size. FRED underestimates the number of active flows and overestimates the per-flow average queue size that is calculated by dividing the average queue size by the number of active flows because FRED deletes per-flow states of flows that have a zero queue size, and TCP flows in timeouts have no packet buffered. Therefore, FRED encourages smooth-rate unresponsive flows, such as UDP-CBR flows. Furthermore, as the fraction of unresponsive flows increases, the average per-flow queue size itself is increased and unresponsive flows are not well regulated.
- **DRR (deficit round robin):** This scheme is a variant of weighted fair queueing (WFQ) discipline. DRR allows WFQ to handle variable packet sizes in a fair manner. DRR is the only one that uses per-flow queueing algorithm while RED, FRED and BARE use a single FIFO buffer. Therefore, DRR guarantees nearly perfect fairness for flows that have at least one packet in the router buffer. Longest queue drop (LQD) is used as a packet drop strategy.

Fig. 4. Simulation topology.

4.1. Simulation configurations

We simulate the configuration shown in Fig. 4. Unless otherwise specified, the following parameters are used. Each output link has a capacity of 20 Mbps, a latency of 2 ms, and a single FIFO buffer of 320 KB. For RED and FRED, $\min_{th}$ is set to 53 KB and $\max_{th}$ is set to 160 KB that corresponds to maximum queueing delay of 64 ms. The buffer size of DRR is set to 320 KB. To compare BARE, RED, FRED and DRR in a fair manner, $\max_{th}$ of DRR is set to 160 KB. TCP-Newreno is used in all simulations because it is the most widely used TCP variant as shown in [18] for its robustness against consecutive packet drops [2]. The data packet size of TCP flows is set to 1000 bytes and the ACK packet size is set to 40 bytes. All BARE parameters are set to the values indicated in Appendix A. We limited BARE's maximum number of per-flow states to 320 to see limiting effects. To avoid the buffer space being fully exhausted, $\max_p$ is set to 0.2 for RED and FRED. The FRED $\min_q$ value is set to 2000 bytes. All four schemes are implemented in ns-2 [24]. RED and FRED operate in byte mode, meaning that packets are buffered and counted in bytes and dropped with a probability proportional to their size. As an example, a 1000 bytes packet is dropped with a probability of 0.2 and a 40 bytes packet is dropped with a probability of 0.008 for RED if the average queue size is slightly smaller than $\max_{th}$. To reduce instantaneous noise and to avoid phase effects, each

simulation is run for $T = 100$ s and each flow is started at a random time. The term Goodput_i of TCP/CBR/TFRC flow i is defined as the number of bytes received by TCP/CBR/TFRC Sink in unit time. We also define N as the number of all flows that are trying to send data packets in simulations while N_{flow} is an estimated number of flows by BARE. Consequently, N is always larger than or equal to N_{flow} .

4.2. Queueing delay and fairness for TCP flows

We simulate only TCP flows and there is no CBR flow. As shown in Fig. 5, BARE reduces the unnecessary queueing delays and maintains a much smaller average queue size compared with RED and FRED. In fact, if the average queue size of BARE is set to target_q of the virtual threshold function, the queueing delays are controlled to the corresponding value. Although the queueing delays increase as the number of flows increases and BARE shows a slightly longer delay than RED when the number of flows is 50–80, BARE maintains smaller delays compared to RED and FRED in the wide range of numbers of flows. BARE eliminates unnecessary queueing delays by regulating each flow's queue size based on the knowledge of the number of flows. Queueing delays of

RED can be reduced by setting max_p to higher values while queueing delays of FRED cannot be noticeably reduced with higher max_p . But, it should be noted that RED does not guarantee fairness as well as QoS, and BARE outperforms FRED and DRR in all cases.

As shown in Fig. 6, under the same condition as mentioned above, we measured the standard deviation of the goodput for each flow, which is normalized by the fair share of that flow. The standard deviation S of $(\text{Goodput}_i/\text{Fair Share})$ is defined as follows:

$$S = \sqrt{\frac{1}{N-1} \sum_{i=1}^N \left(\frac{\text{Goodput}_i}{\text{BW}/N} - 1 \right)^2}, \quad (9)$$

$$\text{Goodput}_i = \text{Goodput}_i(0, T), \quad (10)$$

$$\begin{aligned} \text{Goodput}_i(t, t + \Delta t) \\ = \frac{\text{Total bytes of flow } i \text{ received in } [t, t + \Delta t]}{\Delta t}. \end{aligned} \quad (11)$$

BARE achieves extreme fairness that cannot be compared with RED. When the number of flows is smaller than 100, FRED achieves comparable performance to BARE. While FRED can support only about 100 flows within standard deviation of

Fig. 5. Average queue size vs. number of flows: (a) when N is 5–90 and (b) when N is 100–800.

(a)

(b)

Fig. 6. Standard deviation of (Goodput/Fair Share) vs. number of flows: (a) when N is 5–90 and (b) when N is 100–800.

0.1, BARE support about 350 flows within the same value. BARE's improved fairness over FRED is largely due to Fair Drop. Reasoning from this result, if FRED wants to guarantee fairness values comparable to BARE, it should maintain three or four times larger buffer compared with BARE. Therefore, its maximum queueing delay should be three or four times as large as that of BARE. Although DRR achieves better performance when the number of flows is less than 200, BARE significantly outperforms DRR when the number of flows is larger than 200 because DRR can ensure fairness only when the

number of flows is small and each flow can buffer at least one packet. In contrast to DRR, BARE drops packets whose per-flow states are still maintained and makes room for flows that have not been buffered recently.

Packet loss events are observed with 20 TCP flows. In Fig. 7, packet loss events of source 1 are shown. BARE drops packet more frequently than other scheme to minimize queueing delay. RED and FRED drop packets in a random fashion, as expressed in Eq. (1), while BARE drops packets nearly periodically. With this periodic packet drop, BARE can effectively control per-flow queue

Fig. 7. Loss events of source 1.

sizes and prevent flow i from buffering more than the necessary number of packets. DRR cannot be compared with other schemes because it maintains a per-flow queue for each flow and guarantees perfect fairness for flows that have at least one packets in the router buffer by serving them in round robin discipline.

4.3. Simulations of TCP and CBR flows

We simulate TCP and CBR (which uses UDP as a transport protocol) flows. In Fig. 8, we change the number of flows from 5 to 800. The fraction of CBR flows is set to 40%. Packet size of CBR flows is set to 1000 bytes and inter-packet times are adjusted so that each CBR flows send packets at three times rate of the fair share rate. RED cannot protect TCP flows from unresponsive CBR flows at all. BARE's performance degradation when the number of flows is about 10 is due to TCP's retransmit timeout's. Because TCP's minimum retransmit timeouts are set to 200 ms and round-trip times are about 40 ms, TCP flows cannot get sufficient share. This problem can be solved by setting $vmax_q$ to a higher value when the number of flows

are smaller than 20. BARE and FRED can protect TCP flows from CBR flows. But, if we decide that TCP flows should receive at least half of their fair share, BARE can support up to 650 flows fairly due to its Fair Drop while FRED and DRR can support only up to 100 and 200 flows. Because BARE knows which flows have been recently buffered and drops them even though those flows do not occupy buffer space currently, BARE can keep TCP flows from unresponsive flows even with small buffer sizes. It should be remarked that the average queue size per a flow is smaller than 0.25 packets when the number of flows is 650, which means majority of TCP flows are in retransmit timeouts state.

In Fig. 9, we measure the instantaneous goodput of each flow to observe the instantaneous behavior of TCP and CBR flows. The number of TCP flows is set to 12 and the number of CBR flows is set to 8. All CBR flows send data at 3 Mbps which is three times as large as the fair share value. The measurement interval T_m is set to 0.5 s. Because RED cannot protect TCP flows from unresponsive flows, we have excluded RED from this simulation. The instantaneous goodput of DRR is ex-

Fig. 8. Average of (Goodput/Fair Share) vs. number of flows: (a) when N is 5–90 and (b) when N is 100–800.

Fig. 9. Instantaneous Goodput, in case of $T_m = 0.5$.

tremely smooth because there are only 20 flows and each flow can buffer a sufficient number of packets. The instantaneous goodput of TCP flows for BARE is much smoother than that for FRED. BARE regulates CBR flows within 2 s. When the number of flows is small, BARE's operation is mainly based on both per-flow exponential drop probability which drops packets nearly periodically, and a virtual threshold function that bounds both per-flow queue sizes and the global queue size.

4.4. Instantaneous rates of TCP and TFRC flows

We define coefficient of variation for TCP flow i as follows:

$$\text{CoV}_i = \sqrt{\frac{1}{T/T_m - 1} \sum_{k=0}^{T/T_m - 1} \left(\frac{\text{Goodput}_i(kT_m, kT_m + T_m)}{\text{Goodput}_i(0, T)} - 1 \right)^2}. \quad (12)$$

First, we simulate only TCP flows and results are shown in Fig. 10 because many real-time applications still use TCP as their transport protocols. IP data packet size is set to 500 bytes because real-time applications would decrease its packet size to reduce transmission delay and to cope with bursty packet loss. BARE maintains much smaller CoV_i s than FRED and RED do. When there are small

number of flows such that N is smaller than 60, BARE outperforms FRED due to its periodic packet drops. Also, Fair Drop greatly improves the overall performance of BARE when there are many flows such that N is larger than 180. DRR greatly outperforms three other schemes due to its per-flow queueing.

Secondly, we simulate 40% TFRC [7,8] flows and 60% TCP flows and measure the mean CoV_i s of TFRC flows. All parameters are set to values from [7]. TFRC operates with an equation-based rate control that characterizes TCP sending rates [12,17] based on the following equation:

$$\text{SR} = \frac{s}{R\sqrt{(2p/3)} + t_{\text{RTO}}(3\sqrt{3p/8})p(1 + 32p^2)}. \quad (13)$$

An upper bound on the sending rate SR is used, which is a function of the steady-state loss event rate p , data packet size s in bytes, round-trip time R , and TCP retransmit timeout value t_{RTO} . TFRC estimates the average loss interval, which is a weighted sum of last n loss intervals considering consecutive packet loss events as a single loss event. TFRC uses the average loss interval to calculate the sending rate. We can easily see that TFRC flows should experience periodic packet loss events to estimate p accurately without noisy

Fig. 10. Average coefficient of variation for TCP flows vs. number of flows in case of $T_m = 2.0$.

fluctuation. Results are shown in Fig. 11. As can be seen from this figure, TFRC flows in RED experience noisy instantaneous goodputs, in contrast to BARE and FRED. This feature of BARE should encourage the adoption of TFRC for real-

time applications as a congestion control mechanism.

It should be mentioned that TFRC algorithm is currently not well tuned because TCP models shown in [12,17] is currently not exact. We found

Fig. 11. Average coefficient of variation for TFRC flows vs. number of flows in case of $T_m = 2.0$.

Fig. 12. Instantaneous Goodput of weighted BARE.

that TFRC receives different shares compared with TCP when the number of flows is large and the packet drop probability is high. Furthermore, Eq. (13) is derived with assumption that packet drop events are Bernoulli trials and that assumption is valid only for RED. This assumption is not valid in general as shown in [15]. Research in this area is needed to acquire more accurate TCP model.

4.5. Weighted BARE

BARE can be extended to support flows with different weights. To support differentiated shares, we add a new per-flow variable w_i (a weight value for flow i), and a portion of code is modified. We use two bits of the type of service (TOS) field in the IP header. To support weighted BARE, N_{flow} indicates the total weights of active flows. The values of $\text{rate}_{\text{fair}}$ and max_{th} should be multiplied by w_i . We simulate 8 TCP flows with weights of 1, 2, 3 and 4. Results are shown in Fig. 12. Although weighted BARE cannot differentiate per-flow queueing delays because it uses single FIFO buffer, weighted BARE can be used to effectively support different per-flow shares of goodputs.

5. Miscellaneous topics

5.1. Considerations for implementation

Cooperation and negotiation among several ISPs would probably not be easy. Therefore, a buffer management algorithm should be able to operate individually. BARE can be used in a non-incremental manner in which we can exploit the performance of BARE without installation on several contiguous routers because BARE routers operate individually without any exchange of additional information.

5.2. Queueing delay and IP packet size

Although we preset router buffer sizes in Section 3.4, how many flows a router would support should be considered first. If a router would support 30,000 TCP flows with guaranteeing fairness for each TCP flow, router buffer size should be at least 30,000 IP packets. Assuming that each IP packet size is 1 KB and each flow needs at least two packets buffer, a router buffer size should be at least 60 MB. If a router supports an OC-12c link,

maximum queueing delay would be 0.77 s, which is unacceptably high and flows transferring real-time application data would not be satisfied. This unacceptably high delay is mainly due to large IP packet sizes. To mitigate large queueing delays caused by large IP packet sizes, a buffer management scheme should be able to support many flows with a smaller buffer size. Perhaps, a major difficulty in next generation IP routers would be to reduce queueing delays. As shown in Section 4, BARE could support about 350 flows limiting average queue size below 140 KB which means that each flow buffers 0.4 packets in average.

5.3. Comparison of DRR and BARE

The main advantage of BARE is that it achieves comparable performance to DRR while BARE maintains a single FIFO buffer. Although DRR achieves perfect fairness for flows that have currently at least one packet in the router buffer, it cannot ensure fairness when the number of flows increases above a certain threshold. From the simulation results of BARE and DRR, we can see that BARE achieves comparable performance when the number of flows is small and outperforms DRR when the number of flows is large. To allow DRR to outperform BARE, the buffer size should be increased and queueing delay increases accordingly.

Furthermore, DRR uses per-flow queueing and per-flow scheduling that are considerably hard to implement and does not consider the large amount of legacy routers that use a single and simple FIFO buffer for its each output link. It would be hard to implement per-flow queueing and per-flow scheduling practically. Because DRR guarantees fairness only for currently backlogged flows, DRR would not be an optimum scheme for highly bursty flows such as TCP and its cost per performance ratio would be very high.

6. Concluding remarks and future work

We have proposed a dynamically adjusting per-flow buffer management scheme that can be applied to TCP flows and to flows transferring real-time application data. We have simulated various

configurations with TCP, CBR and TFRC flows. BARE exhibits better fairness, less delays, and better smoothness of sending rates than previous schemes. Introduction of a virtual threshold function that divides router operation into three modes allows the average queue size to fluctuate around the target_q value and eliminates unnecessary delays. BARE also produces more efficient buffer usage that helps routers support more flows than RED, FRED and DRR with the same buffer size. The per-flow rate estimation was accurate in view of estimating the per-flow current share, and noisy and rapid fluctuations were filtered. The per-flow exponential adjustment of the drop probability prevents unresponsive flows from achieving an unfairly large share. BARE also controls the per-flow queue size, preventing flows from buffering more than a sufficient number of packets or buffering fewer than the necessary number of packets.

We also introduced a practical definition of “active flows” and developed a new algorithm for routers to support a larger number of flows in a fair manner in spite of insufficient buffer size. With practical definition of “active flows”, BARE reduces overhead coming from maintaining a large number of per-flow states. Fair Drop greatly improves overall performance when the number of flows are large and the average per-flow queue size is less than one packet. Moreover, the use of bits in the TOS field of the IP header allows easy differentiation of bandwidth allocation. We believe that BARE can support real-time applications and can encourage the use of end-to-end congestion control mechanisms such as TFRC.

Although BARE improves the overall performance of buffer management scheme, additional work to be done remains. Analysis on how TFRC and its variants can better interoperate with BARE, RED, and FRED is needed. Research on tuning the parameters and algorithms of TFRC is needed to satisfy the requirements of real-time applications. More functions should be added to produce smoother sending rates for TCP-friendly flows. Although BARE achieves better fairness on multiple congested links than RED or FRED, research on optimization of BARE is needed to achieve much better fairness on multiple congested links.

Appendix A. A detailed pseudocode of BARE algorithm

In this appendix, we present a detailed pseudocode of BARE algorithm that was used for simulation.

- Constants:
 - $\text{max}_q = 320000$; // maximum aggregate queue size (bytes)
 - $\text{fd}_{\text{th}} = 140000$; // Fair Drop operates if q is larger than this
 - $\text{max}_{\text{target}_q} = 120000$; // maximum value of target_q
 - $\text{max}_{n_{\text{flow}}} = 320$; // maximum N_{flow} that can be maintained
 - $\alpha = 1.2$; // increase factor of β
 - $w_q = 0.004$; // weight for average queue size calculation
 - $\beta_{\text{max}} = 10$; // maximum β
 - $\beta_{\text{min}} = 0.5$; // minimum β
 - $\beta_{\text{init}} = 7.5$; // initial β
 - $K = 0.15$ s; // constant used for rate estimation
 - $\text{BW} = 2500000$ Bps; // service rate (bytes/s)
 - $\text{Timeout_Value} = 0.45$ s; // timeout value used for flow expiration
- Global variables:
 - N_{flow} ; // number of active flows (initially, 0)
 - vmax_q ; // virtual maximum buffer size (bytes)
 - max_{th} ; // maximum queue size for each flow (bytes)
 - target_q ; // target queue size (bytes)
 - $\text{rate}_{\text{fair}}$; // fair rate (share)
 - time ; // current real time (s)
- Global queue sizes:
 - q ; // current global queue size (bytes) (initially, 0)
 - avg_q ; // average global queue size (bytes) (initially, 0)
- Per-flow variables:
 - $q[i]$; // queue size (bytes)
 - $\text{rate}[i]$; // estimated rate (bytes/s)
 - $\beta[i]$; // $\beta[i]$
 - $\text{count}[i]$; // number of bytes processed since last $\beta[i]$ update
 - $q\text{time}[i]$; // last time packet is buffered (s)
- Functions:
 - find_i(p)**; // find the flow number to which p belongs
 - update_gv(mode)**{ // update global variables
 - if** (mode == 1) $N_{\text{flow}} = N_{\text{flow}} + 1$;
 - else** $N_{\text{flow}} = N_{\text{flow}} - 1$;
 - $\text{vmax}_q = \text{vth}(N_{\text{flow}})$;
 - $\text{max}_{\text{th}} = \text{vmax}_q / N_{\text{flow}}$;
 - $\text{target}_q = \min(\text{vmax}_q / 2, \text{max}_{\text{target}_q})$;
 - $\text{rate}_{\text{fair}} = \text{BW} / N_{\text{flow}}$;
 - initialize_pfv(i)**{ // initialize per-flow variables
 - $q[i] = \text{count}[i] = \text{p.size}$;
 - $\text{rate}[i] = 0$;
 - $\beta[i] = \beta_{\text{init}}$;
 - $q\text{time}[i] = \text{time}$;
 - update_gq(mode)**{ // update q and avg_q
 - if** (mode == 1) $\text{value} = \text{p.size}$;
 - else** $\text{value} = -\text{p.size}$;
 - $q = q + \text{value}$;
 - $\text{avg}_q = (1 - w_q)\text{avg}_q + w_q q$;
 - update_beta(rate_ratio)**{ // update $\beta[i]$
 - if** ($\text{rate}_{\text{ratio}} > 1 \ \&\& \ \text{avg}_q > \text{target}_q$){
 - $\beta[i] = \beta[i] / \text{rate}_{\text{ratio}}$;
 - if** ($\beta[i] < \beta_{\text{min}}$) $\beta[i] = \beta_{\text{min}}$;
 - else** {
 - $\beta[i] = \beta[i] \alpha$;
 - if** ($\beta[i] > \beta_{\text{max}}$) $\beta[i] = \beta_{\text{max}}$;
 - random()**; // uniform random number in $[0 \dots 1]$
 - pow(a, b)**; // calculate and return a^b
 - exp(c)**; // calculate and return e^c
- For each arriving packet p :
 - 1: **if** ($q \geq \text{vmax}_q \parallel q \geq \text{max}_q$) {
 - 2: **drop**(p);
 - 3: **return**;
 - 4: }
 - 5: **if** (**find_i**(p) == false) {
 - 6: **if** ($N_{\text{flow}} < \text{max}_{n_{\text{flow}}}$) {
 - 7: $i = \text{new flow number}$;
 - 8: **update_gv**(1);
 - 9: } **else** {
 - 10: $i = \text{randomly selected from } N_{\text{flow}} \text{ flows whose } q[i] \text{ are } 0$;

```

11: }
12: initialize_pfv(i);
13: update_gq(1);
14: return;
15: }
16: if ( $q \geq fd_{th}$ ) {
17:   drop(p);
18:   return;
19: }
20:  $q[i] = q[i] + p.size$ ;
21:  $count[i] = count[i] + p.size$ ;
22: if ( $count[i] \geq 4 \times max_{th}$ ) update_beta
   ( $rate[i]/rate_{fair}$ );
23:  $u = random()$ ;
24: if ( $u < pow(q[i]/max_{th}, \beta[i])$ ) {
25:    $q[i] = q[i] - p.size$ ;
26:   drop(p);
27: } else {
28:   update_gq(1);
29:    $dt = time - qtime[i]$ ;
30:    $qtime[i] = time$ ;
31:    $rate[i] = (1 - \exp(-dt/K)) \times p.size/dt +$ 
      $\exp(-dt/K) \times rate[i]$ ;
32: }
• For each departing packet  $p$ :
33: find_i(p);
34:  $q[i] = q[i] - p.size$ ;
35: if ( $count[i] \geq 4 \times max_{th}$ )
   update_beta( $rate[i]/rate_{fair}$ );
36: update_gq(0);
• For each flow expiration:
// each zero – queue flow expires if
  ( $time - qtime[i] \geq Timeout\_Value$ )
37: update_gv(0);

```

References

- [1] A. Demers, S. Keshav, S. Shenker, Analysis and simulation of a fair queueing algorithm, in: Proceedings of the ACM SIGCOMM'89, Austin, TX, September 1989, pp. 1–12.
- [2] K. Fall, S. Floyd, Simulation-based comparisons of Tahoe, Reno and SACK TCP, ACM SIGCOMM, Comput. Commun. Rev. 26 (3) (1996) 5–21.
- [3] W. Feng, D.D. Kandlur, D. Saha, K.G. Shin, A self-configuring RED gateway, in: Proceedings of the IEEE INFOCOM'99, New York, March 1999, pp. 1320–1328.
- [4] S. Floyd, K. Fall, Promoting the use of end-to-end congestion control in the Internet, IEEE/ACM Trans. Networking 7 (4) (1999) 458–472.
- [5] S. Floyd, K. Fall, Router mechanisms to support end-to-end congestion control, February 1997, LBL Technical Report.
- [6] S. Floyd, V. Jacobson, Random early detection gateways for congestion avoidance, IEEE/ACM Trans. Networking 1 (4) (1993) 397–413.
- [7] S. Floyd, J. Padhye, J. Widmer, Equation-based congestion control for unicast applications, in: Proceedings of the ACM SIGCOMM 2000, Stockholm, Sweden, September 2000, pp. 43–56.
- [8] S. Floyd, J. Padhye, J. Widmer, TFRC, Equation-based congestion control for unicast applications: simulation scripts and experimental code, February 2000. <http://www.aciri.org/tfrc/>.
- [9] V.P. Kumar, T.V. Lakshman, D. Stiliadis, Beyond best effort: router architectures for the differentiated services of tomorrow's Internet, IEEE Comm. Mag. 36 (5) (1998) 152–164.
- [10] D. Lin, R. Morris, Dynamics of random early detection, in: Proceedings of the ACM SIGCOMM'97, Cannes, France, October 1997, pp. 127–137.
- [11] R. Mahajan, S. Floyd, D. Wetherall, Controlling high-bandwidth flows at the congested routers, in: Proceedings of the IEEE ICNP 2001, Riverside, CA, November 2001.
- [12] M. Mathis, J. Semke, J. Mahdavi, The macroscopic behavior of TCP congestion avoidance algorithm, ACM Comput. Commun. Rev. 27 (3) (1997) 25–41.
- [13] P.E. McKenney, Stochastic fairness queueing, in: Proceedings of the IEEE INFOCOM'90, San Francisco, CA, June 1990, pp. 733–740.
- [14] A. Mena, J. Heidemann, An empirical study of Internet audio traffic, in: Proceedings of the IEEE INFOCOM 2000, Tel Aviv, Israel, March 2000, pp. 101–110.
- [15] M. Mitzenmacher, R. Rajaraman, Towards more complete models of TCP latency and throughput, J. Supercomput. 20 (2) (2001) 137–160.
- [16] R. Morris, TCP behavior with many flows, in: Proceedings of the IEEE ICNP'97, Atlanta, GA, October 1997.
- [17] J. Padhye, V. Firoiu, D.F. Towsley, Modeling TCP Reno performance: a simple model and its empirical validation, IEEE/ACM Trans. Networking 8 (2) (2000) 133–145.
- [18] J. Padhye, S. Floyd, On inferring TCP behavior, ACM Comput. Commun. Rev. 31 (4) (2001) 287–298.
- [19] R. Rejaie, M. Handley, D. Estrin, RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the Internet, in: Proceedings of the IEEE INFOCOM'99, New York, March 1999, pp. 1337–1345.
- [20] M. Shreedhar, G. Varghese, Efficient fair queueing using deficit round robin, in: Proceedings of the ACM SIGCOMM'95, Boston, MA, September 1995, pp. 231–242.
- [21] D. Sisalem, H. Schulzrinne, The loss-delay based adjustment algorithm: A TCP-friendly adaptation scheme, in: Proceedings of NOSSDAV'98, Cambridge, England, July 1998, pp. 215–226.
- [22] I. Stoica, S. Shenker, H. Zhang, Core-stateless fair queueing: achieving approximately fair bandwidth allocations in

high speed networks, in: Proceedings of the ACM SIGCOMM'98, Vancouver, Canada, September 1998, pp. 118–130.

- [23] D. Tan, A. Zakhor, Realtime Internet video using error resilient scalable compression and TCP friendly transport protocol, *IEEE Trans. Multimedia* 1 (2) (1999) 172–186.
- [24] UCL/LBNL/VINT Network Simulator-ns (version 2).
<http://www.isi.edu/nsnam/ns/>.

Jeong-woo Cho received the B.S. and M.S. degree in electrical engineering and computer science from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea in 2000 and 2002, respectively. He is currently working toward the Ph.D. degree in electrical engineering and computer science at the same place. His research interests are in the areas of router buffer management, optimization flow control, max-min flow control, and performance evaluation of TCP/IP networks.

Dong-ho Cho received the B.S. degree in electrical engineering from the Seoul National University, in 1979, and the M.S. and Ph.D. degrees, both in electrical and electronics engineering, from the Korea Advanced Institute of Science and Technology (KAIST), in 1981 and 1985, respectively. From 1987 to 1997, he was a Professor of Computer Engineering at the Kyunghee University. Since 1998, he has been a Professor of Electrical Engineering at KAIST. His research interests include wired/wireless communication network, protocol, and services.