CPS: Operating System Architecture for Efficient Network Resource Management with Control-Theoretic Packet Scheduler

Hyungsoo Jung, Hyuck Han, Heon Young Yeom, and Sooyong Kang

Abstract: **The efficient network resource management is one of the important topics in a real-time system. In this paper, we present a practical network resource management framework, control-theoretic packet scheduler (CPS) system. Using our framework, an operating system can schedule both input and output streams accurately and efficiently. Our framework adopts very portable feedback control theory for efficiency and accuracy. The CPS system is able to operate independent of the internal network protocol state, and it is designed to schedule packet streams in fine-grained time intervals to meet the resource requirement. This approach simplifies the design of the CPS system, and leads us to obtain the intended output bandwidth. We implemented our prototype system in Linux, and measured the performance of the network resource management system under various network QoS constraints. The distinctive features of our principles are as follows: It is robust and accurate, and its operation is independent of internal network protocols.**

Index Terms: **Bandwidth reservation, control theory, operating system, packet scheduling, quality of service (QoS).**

I. INTRODUCTION

The primary goal of achieving quality of service (QoS) in a network is to meet diverse QoS requirements of each network flow with limited network resources, and to use given network resources efficiently under such constraints. The advances of multimedia service technology have been boosted recently and the widening use of real-time systems in conjunction with the development of high performance network infrastructure have given momentum to the development of an efficient network management system.

To satisfy the requirements of integrated services, the concept of service differentiation as a network architecture model is necessary in network QoS domain. The requirements that should be satisfied by a differentiation model are 1) throughput requirements, 2) delay bound, and 3) delay jitter of different applications in real-time or soft real-time systems. However, in today's heterogeneous Internet environment where a lot of different policies over the flow control system are employed, there is no *de facto* scheme for a QoS-guaranteed control system that is applicable everywhere.

In that point of view, the role of the operating system (OS) is to regulate and multiplex the access of multiple application programs to the network resources. Unfortunately, current general purpose OSs are not equipped with a resource management module, even though lots of genuine techniques are suggested to support high throughput communication. Current OS' inability to control limited network resources can be an obstacle to exploit high speed network bandwidth effectively.

The goal of our work is, 1) to design a low-overhead and accurate resource management system, 2) to validate our prototype architecture, and 3) to analyze the output performance of new architecture by deploying it in a general purpose OS. This work is accomplished without sacrificing any other of the OS' activities.

In this paper, we present a practical design of an operating system architecture for efficient network resource management with a control-theoretic packet scheduler (CPS) system. The main contribution of the CPS system, as the name infers, is a fine-grained scheduling of bandwidth reserved network flows without incurring additional context switches inside the kernel. Current Linux requires a context switch to activate kernel thread if there is any pending software interrupt (formally handled in the bottom half of the OS) such as a network input/output (I/O) event. Even though such a way of software interrupt handling through the kernel thread has the advantage of reducing the number of context switches by means of delaying a context switch at least until any running process consumes its time quantum or it is blocked and scheduled by the kernel, this delayed handling mechanism cannot make possible a fine-grained packet scheduling. However, the CPS exploits the *trigger state* [1] to perform a packet scheduling. The control owner of the trigger state can be any process, so invoking a packet scheduler does not need additional context switch, and this instantaneous invocation of a packet scheduler, after every network I/O, increases the scheduling granularity of the CPS.

Another contribution of our work is that the CPS system can react smoothly to the system or network load condition by adaptively adjusting a service interval of a packet. This is achieved by applying the proportional-integral-derivative (PID) control theory [2] practically. The feedback control part of the CPS can handle any interference out of the CPS system without incurring misbehaviors.

Our previous work [3] dealt with the control-theoretic approach for QoS control in network routers. In this paper, we

Manuscript received July 11, 2008; approved for publication by Dongman Lee, Division III Editor, May 19, 2009.

This study was supported by the Seoul R&D Program (10561), Seoul, Korea. The ICT at Seoul National University provided research facilities for this study.

H. Jung, H. Han, and H. Y. Yeom are with the School of Computer Science and Engineering, Seoul National University, Seoul, Korea, e-mail: {jhs, hhyuck, yeom}@dcslab.snu.ac.kr.

S. Kang (corresponding author) is with the Division of Computer Science and Engineering, Hanyang University, Seoul, Korea, e-mail: sykang@hanyang.ac.kr.

mainly concentrate on the end-host QoS issues. We deployed our prototype architecture in a Linux kernel. If the control system can accurately regulate the bandwidth of each flow, application services such as multimedia streaming services can easily maintain constant sending rates so that the network management system can efficiently manage the available bandwidth. With these significant features of the CPS system, this paper describes following issues that is concerning: 1) What kind of design optimization we must consider to apply the control-theoretic approach to a real system, 2) how useful the implemented system is, and 3) the performance result of the CPS system.

The rest of this paper is organized as follows: Related work concerning control-theoretic flow control mechanism and OS support for network resource management are discussed in Section II. In Section III, the overall architecture of the proposed CPS system is presented and in Section IV, we describe the main control methodology which is based on the PID control, for our system. In Section V, we present the packet scheduling mechanism which employs soft-timer for fine-grained scheduling. The performance of the proposed system is evaluated through various experiments in Section VI and finally, Section VII concludes our work.

II. RELATED WORKS

Keshav [4] did early work on the control-theoretic approach to the reactive flow control in networks. His theoretical model is based on the control theory, and the presented model is built on networks that do not reserve bandwidth. The model supports fair rate controlling for the network flow by a packet pair probing technique. And the model initially sets the frequency of control once per round trip time (RTT).

Since many artistic packet scheduling approaches such as core-stateless fair queueing (CSFQ) [5] and worst-case fair weighted fair queueing (WF2Q) [6] are known to be theoretic, deploying these approaches in endsystems is hard to achieve. Unlike these approaches, the CPS can control system misbehaviors effectively due to its feedback compensation mechanism without knowing flow charateristics in advance.

Active queue management (AQM) techniques such as random early detection (RED) [7]–[9] mainly focus on detecting congestion and notifying end nodes of the congestion for end nodes to handle the congestion event. However, unlike the CPS system, they fail to control end-to-end user datagram protocol (UDP) flows.

In [10], they proposed a novel model of transmission control protocol (TCP) throughput that is based on the RTT and the packet loss rate, and the model is used in the TCP friendly rate control (TFRC). As explained in Section VI, the CPS does not distort the model and the CPS regulates the delay to achieve QoS requirements.

Qlinux [11] and LinuxRK [12]–[15] projects can be represented as a real implementation of the QoS-guaranteed system built in kernel that concerns only end-host QoS issues. Qlinux deployed the hierarchical start-time fair queuing (H-SFQ) [16], [17] packet scheduler to provide the network QoS. It also applied the lazy receiver processing (LRP) [18] mechanism for exact charging of the time being spent in protocol processing. However, the OS sometimes has to do a spin-checking on whether its packet should be serviced or not. This leads to an overhead in the OS and it reduces OS' resource availability.

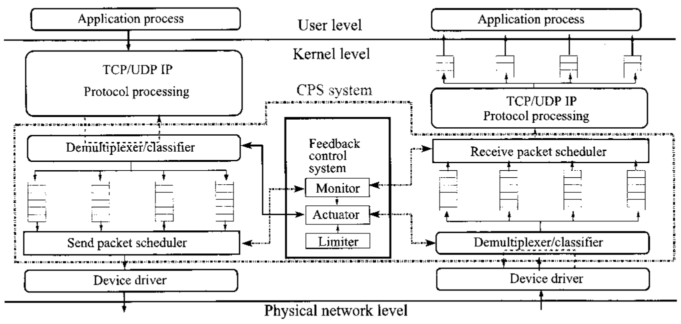


Fig. 1. Overall architecture of the CPS system.

In LinuxRK, the authors created a new kernel thread to make distinctions between the processing of normal network flow and that of reserved network flow, instead of using kernel's default *bottom half handler*. However, the newly created kernel thread cannot accurately control the incoming flow. Moreover, LinuxRK applies a deadline monotonic approach to process each arriving packet for network flows that reserve a prescribed QoS level. So, they used a default kernel timer to check the bytes of processed packets for each flow, and this is another limitation against achieving exact bandwidth controlling.

As for an application level approach, we can consider two works [19], [20]. These works employed control-theoretic approach to make it feasible to achieve service differentiation in a web server application. They implemented well-known control theory and enable us to analyze their experimental results. However, they are constructed not at the kernel level but at the application level.

III. ARCHITECTURAL OVERVIEW

In this section, we describe the overall architecture of our CPS system. First, we explain the limitation of Linux, which has no other QoS-related component inside the original OS, and we enumerate important components of the CPS system in turn.

A. Limitations of Linux

We have implemented the prototype of the CPS system in the Linux kernel. Since Linux has been developed as a general purpose operating system, its ability to meet various demands of time-critical applications leaves a lot to be desired, although fewer resource-demanding application mixes can be easily handled by the current best-effort method. In current versions of Linux, the kernel does not have any inherent network bandwidth control mechanism. To overcome this shortcoming, we used the well-known concept of soft timers [1]. Using the soft timers concept, it is possible to exploit its fine-grained timing ticks for pinpoint scheduling of packet flows and implement an efficient as well as low-overhead packet scheduler for a network QoS without using timer interrupt.

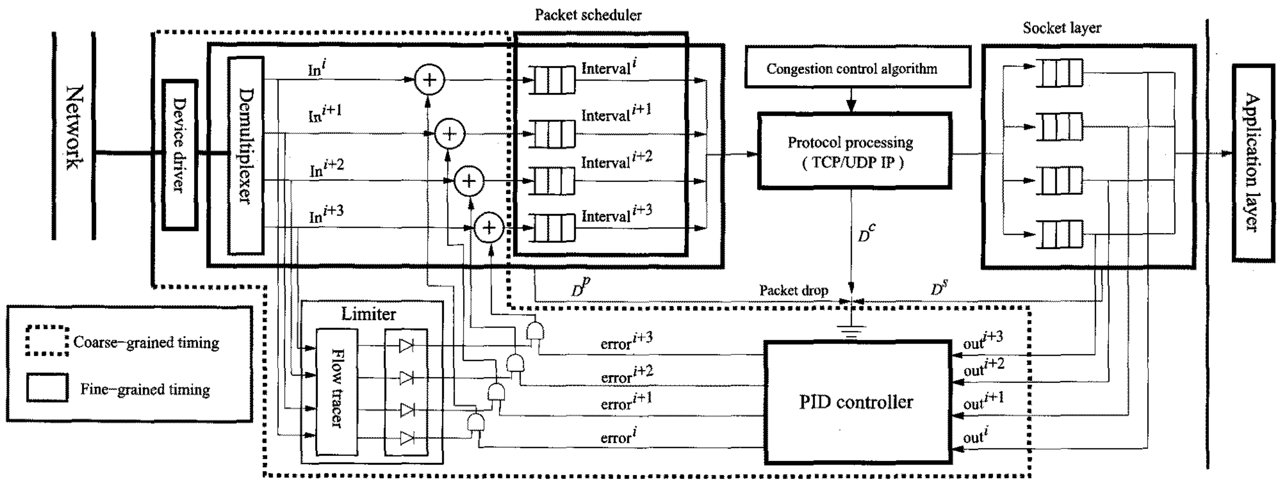


Fig. 2. Receive flow diagram of the CPS system.

B. System Components of the CPS

The design focus of the CPS system is its efficiency and exact timing control for network QoS. As shown in Fig. 1, the CPS system is composed of four main components. The first component is the *soft timer module* that provides fine-grained timing ticks and invokes packet scheduler frequently.

The second one is a packet demultiplexer or a packet classifier. The demultiplexer checks which network stream the incoming or outgoing packet belongs to, based on the information in the packet header. For incoming packets, the demultiplexer verifies checksum to drop abnormal packets, a priori, so that the receive packet scheduler can provide accurate end-to-end flow rate to the application process. After demultiplexing, the classifier determines the service level and records the future service time in which the packet should be serviced, then it enqueues the packet in the designated send/receive queue. In contrast to the packet scheduler which is called by the Actuator, the demultiplexer/classifier is invoked according to the normal kernel mechanism. On the receiver side, it is called by the network interrupt handler, while on the sender side it is called by the end of the kernel control flow of the user program. And the demultiplexer/classifier should invoke the limiter to inform the CPS system of send/receive events.

The third component is a packet scheduler which determines whether or not to service each packet in each queue based on the future service time of each packet.

The last one is the brain of the CPS system: A feedback control system that actually applies control-theoretic scheduling policy to each QoS flow. The feedback control system consists of three key components:

- The monitor continues to sample output value (net bandwidth value observed at the final point of the packet processing) of each flow stream and keeps the history of the average bandwidth for each flow.
- The controller/actuator is a component giving updated value to input point (i.e., demultiplexer/classifier). The updated value reflects the control-theoretic calculation, and the actuator can provide an adaptive control mechanism.
- The limiter acts as an on-off switch to the controller/actuator. It continuously monitors the data rate of each flow to determine whether or not to apply the result of control-theoretic calculation (done by the controller/actuator) to packet scheduling. In an open and unpredictable environment such as the Internet, flows cannot always come into the system constantly. Without the limiter, control system misbehaves and drives the whole system to an aggravated state. The limiter plays an important role in the CPS system.

IV. MAIN CONTROL MECHANISM

In this section, we describe the control theory adopted in the CPS system. We first give an overview of PID control. Then, we describe how we incorporate PID control into CPS in turn.

A. PID Control

With its three-term functionality offering treatment of both transient and steady-state responses, PID control provides a generic and efficient solution to real world control problems; in particular, when the mathematical model of an observed system is not formally defined, and therefore analytical design schemes cannot be used. Because exact modeling of the Internet via mathematical formalism is very difficult due to its complexity, we choose PID control [2] as our primary control strategy.

B. Applying PID Controls to Network QoS

Fig. 2 shows the receive flow diagram of the CPS system. It is an example diagram to show the internal operation of the CPS system; we will give a detailed description of the packet scheduler in a later section. In this section, we describe the mechanism and implementation of key components in our control system. The PID controller is composed of three key components, which was described shortly in Section III-B.

The first one to explain is the monitor. The monitor is a component to observe the net output bandwidth obtained at the final point of the packet processing. On the receiver side, it should be at the socket layer while on the sender side it is at the device driver routine for sending packet streams. The variable out^i is the sampled bandwidth of flow i, and its mathematical form is $\mathrm{out}^i = \mathrm{CopiedByteofFlow}_i/T_s$.

The sampling interval T_s plays a very important role in the control system. If we choose a short sampling interval, the control system is very accurate and responsive in a continuous time domain. However, in a discrete time domain environment like a computer system, too short of a sampling interval could make the control system oscillate and become unstable. Also, in a network subsystem, packets are received at random intervals. If the sampling interval is shorter than the packet's inter-arrival time, the output bandwidth is calculated too often, which results in a wasted CPU cycles. We therefore need to insert a relaxation factor in deciding a sampling interval: In our implementation, we chose one second as a sampling interval.

The second component is the controller/actuator. This is the brain of our control system. The controller/actuator first gathers all output bandwidths of QoS sensitive flows from the monitor. Based on the output values, it calculates updated input values depicted as error^i. We added a converting equation that makes the controller/actuator generate error^i in time interval (seconds) while getting input in bandwidth (bytes/sec). Following the conventional definition in [2], The governing PID equation for flow i is written in the discrete time domain form as follows:

$$\begin{aligned} u^i(t) &= u^i(t-1) + K_P \cdot [\text{out}^i(t) - \text{out}^i(t-1)] \\ &+ \frac{K_P T_S}{T_I} \cdot \text{out}^i(t) \\ &+ \frac{K_P T_D}{T_S} \cdot [\text{out}^i(t) - 2\text{out}^i(t-1) + \text{out}^i(t-2)] \end{aligned} \quad (1)$$

where out(t) is the output bandwidth at time t, K_P is the proportional gain, T_I is the integral time constant, T_D is the derivative time constant, and T_S is a sampling (or observation) interval such that

$$\begin{aligned} \frac{d}{dt}\text{out}(t) &\simeq \frac{\text{out}(t)-\text{out}(t-1)}{T_S}, \\ \int_0^t \text{out}(t)dt &\simeq T_S \sum_0^t \text{out}(i). \end{aligned} \quad (2)$$

The integral gain K_I and the derivative gain K_D can also be expressed as $K_I = K_P \cdot T_S/T_I$ and $K_D = K_P \cdot T_D/T_S$. The three gain constants are parameters that mediate the effect of the control to the system, and are needed to be tuned for each specific system that employs control theory. $u^i(t)$ is the newly-obtained bandwidth value, and our converting equation is error$^i(t)$ = PacketArrivalInterval$/u^i(t)$. The PacketArrivalInterval is a heuristic value of the packet inter-arrival time under the assumption of 1 KB packets at 100 Mbps connection. This parameter is necessary in the converting equation to convert $u^i(t)$ (Mbps) to error$^i(t)$ (millisecond). Another implementation issue is the tuning of gain constants in our PID controller (i.e., K_p, T_i, and T_d). When we deployed the CPS system in the kernel, we tried to set the constant value as 2^k (where k is an integer), so the governing equations can be calculated using only bitwise operations for efficiency. The experimental results of the bandwidth graph under various gain constant settings are presented in the next section.

The last component is the limiter. The purpose of the limiter is to prevent the control system from responding excessively under a temporal variation of the network state. It tracks down the input bandwidth variation for each QoS related flow. If the input bandwidth is under the reserved bandwidth, the limiter switches off the control system. Actually, the meaning of input bandwidth in sender/receiver is different. On the sender side, an input bandwidth is measured at the point where the demultiplexer/classifier is called while on the receiver side it is obtained after the receive interrupt handler is invoked.

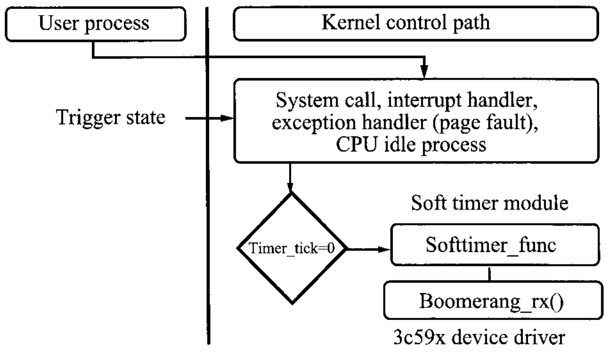


Fig. 3. Various trigger state.

In the CPS system, we applied an exponential averager to trace bandwidth variations. The main reason for this is for the limiter to be able to distinguish temporal bandwidth collapses such as a slow start after timeout with the overall throughput degradation of the network. When the system is in an overall degraded state, the limiter should *switch-off* the effect of the controller/actuator to prevent the system from responding excessively. Let the raw input bandwidth and switching variable for flow i at time t be In$^i(t)$ and $S^i(t)$. Then, the governing equation of the limiter is as follows:

$$\text{In}^i(t) = \frac{\text{In}^i(t-1)}{2} + \frac{\text{In}^i(t)}{2} \& S^i(t) = \begin{cases} 0, & \text{if In}^i(t) < B^i \\ 1, & \text{if In}^i(t) \geq B^i \end{cases} \quad (3)$$

where B^i is the reserved bandwidth for flow i. Finally, the actual value of Interval^i used when the packet scheduler records future service time at the received packet is updated as follows:

$$\text{Interval}^i(t) = \text{Interval}^i(t-1) + \text{error}^i(t) \times S^i(t). \quad (4)$$

In real implementation, we have converted the multiplication operation to a bitwise operation whenever it is possible to reduce overhead inside the kernel.

V. Packet Scheduling

In CPS system, a packet scheduling occurs in the packet scheduler, which is invoked by the actuator. The CPS system does not use a kernel thread level management to do a packet scheduling procedure. Instead of using the bottom half handler (i.e., the software interrupt handler that is called by the dedicated kernel thread in a general OS), we devised a fast checking function using soft-timer. Because the Linux's default timer has relatively a long checking period[1], it is very hard to achieve fine-

[1]The basic timer interrupt interval (10 ms) can be adjusted to be 1 ms by recompiling the entire kernel.

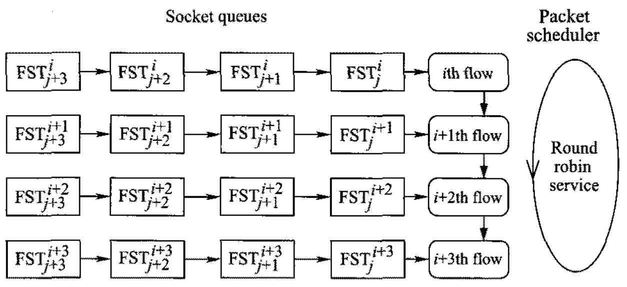


Fig. 4. Round-robin packet scheduling.

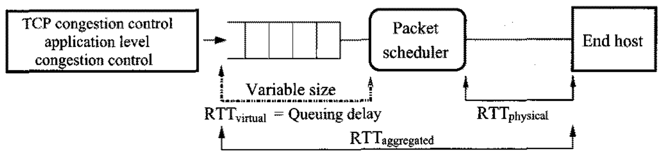

TCP congestion control application level congestion control
Packet scheduler
End host
Variable size
$RTT_{virtual}$ = Queuing delay
$RTT_{physical}$
$RTT_{aggregated}$

Fig. 5. Interaction between TCP and queueing delay of the CPS.

grained throughput control of a packet stream. The fine-grained timing we intent to achieve needs to be shorter than microsecond resolution in modern microprocessor architectures.

In soft-timer, timer is triggered at the pre-designated triggering states, which are shown in Fig. 3. Even within a second, Linux kernel passes these triggering states enormous times, so that we have very fine-grained timing resolution, which is finer than microsecond resolution. This remedies the coarse-grained timer interval problem. Thereby the CPS system can achieve good performance, and it can operate without causing a large overhead.

A. Soft Timer Module

The soft timer plays a critical role in the CPS system. The motivation of implementing the soft timer is to obtain a fine-grained timing service. In the CPS system, two components, the monitor and the packet scheduler, should be invoked periodically by the system. Although the monitor invocation can be coarse-grained, the packet scheduler invocation should be fine-grained to be able to schedule every packet as soon as possible.

The default timer interrupt service provided by Linux is very coarse. Even if the timer interrupt interval can be changed, the overhead caused by context switching cannot be ignored. Therefore, we use soft timer to invoke the monitor and the packet scheduler. For the monitor, we set a long time interval inside the soft timer module while we set the fine-grained time interval for the packet scheduler. A soft timer enables the packet scheduler to be invoked whenever the prescribed system execution states, i.e., trigger state shown in Fig. 3, are reached.

B. Send/Receive Packet Processing

Using the soft timer, we have designed and implemented a fine-grained packet scheduler. The main packet scheduler, which is irrelevant to the OS' default bottom half handler, is invoked whenever the system enters a trigger state, and it should handle both receive and send packet processing.

B.1 Packet Demultiplexing and Classification

Fig. 4 shows the packet scheduling framework in the CPS system. In the packet scheduler, every incoming packet is first checked as to whether it is destined to any registered network stream, and if it is for the registered one, the incoming packet is time-stamped with its future service time, future service time (FST). (Though the incoming packet is not for the registered network stream, its FST is calculated by using the spare bandwidth of the OS.) Then, the packet is enqueued in the corresponding socket queue that stores incoming packets of the flow the packet belongs. Suppose the arrived packet is for the flow i and it is jth packet at time t. Then, the multiplexer/classifier records its FST (FST^i_j) inside the *sk_buff* structure which contains packet contents. When the kernel records the FST^i_j, it needs the current time T_c and the latest timestamp value (LTS^i) for that flow. The LTS^i is the FST of the lastly enqueued packet of the flow i, and it is recorded in a socket structure. If LTS^i is greater than T_c, then FST^i_j becomes $FST^i_j = LTS^i + \text{Interval}^i(t)$, otherwise it becomes $FST^i_j = T_c + \text{Interval}^i(t)$. After FST^i_j is calculated, the LTS^i value should also be updated to $LTS^i = FST^i_j$.

In particular, the CPS system maintains a queue for each network flow for the packet scheduling. Although this new structure makes additonal queueing delay, the higher level congestion control algorithm such as TCP's congestion control algorithm or that of TFRC does not get a harmful effect from this delay. As shown in Fig. 5, the congestion control mechanism of TCP or TFRC sees only an $RTT_{aggregated}$, which is the sum of $RTT_{virtual}$ and $RTT_{physical}$. This makes the congestion window size smoothly configured to the stable point despite the newly introduced delay, $RTT_{virtual}$. Therefore, we can expect that the congestion control algorithm adopted in higher level protocol will not be badly affected from the CPS system, and will interact well with the variation of the $RTT_{virtual}$ (i.e., queueing delay). This expectation has been confirmed by the experiments which will be presented later.

B.2 Packet Processing

General packet processing is performed in the packet scheduler. The term '*received/incoming packet*' refers to any type of packet that comes into the CPS system; in sending process it refers to a packet coming down from the application program while in receiving process it means a packet just arrived from the remote sender. Once every incoming packet is classified and enqueued in each socket queue by the demultiplexer/classifier, the general packet scheduler processes the incoming packet in a different way from the current Linux OS to guarantee the QoS. In the CPS system, every packet is checked against the current time. If the FST^i_j value of a packet is smaller than the current time (i.e., the service time for the packet is already elapsed), this packet deserves to be serviced in this invocation time. Otherwise, all the packets enqueued in this socket queue should wait until the next invocation time. This generic packet scheduling procedure is applied to both incoming and outgoing packet streams in the same way.

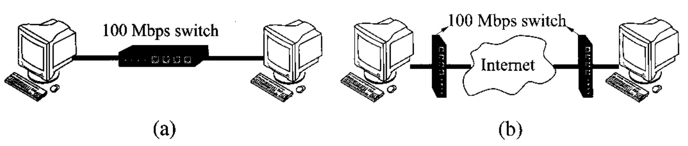


Fig. 6. Experimental environment: (a) LAN environment and (b) WAN environment. In the WAN environment, two computers are located in different cities in Korea (Seoul and Daejon, respectively), and are connected via Internet.

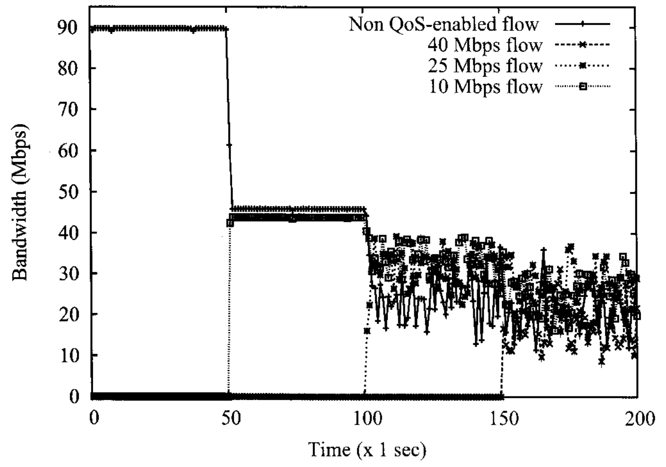


Fig. 7. Acquired network bandwidth of three competing processes which reserved 10, 25, and 40 Mbps, respectively, without feedback control.

VI. EXPERIMENTAL RESULTS

In this section, we present experimental results of the CPS system. We performed a couple of experiments using two machines, which have a 2 GHz CPU and a 3Com 3c59x 100 Mbps Ethernet NIC device, in both LAN and WAN conditions (Fig. 6).

We used a simple TCP send/receive program installed in both machines, which transmits and receives 1 Kbyte data packet, repeatedly. Our experiments focused on the timing accuracy and the ability for exact rate control of the CPS system. We measured effective throughput obtained in the application layer. And we assume that all QoS-enabled flow should start at zero bandwidth, then the flow is controlled by the CPS system.

A. Non-feedback System

In Fig. 7, we measured the throughput of four competing network flows controlled by the non-feedback control system, i.e., the system which has no component that provides feedback control. Among four network flows, one is non QoS-enabled flow and the other three flows reserved 40, 25, and 10 Mbps of bandwidth, respectively. The experiment is performed in the LAN condition to test in a large bandwidth environment. In this experiment, we applied a direct control theory that does not use feedback. The equation, $\mathrm{FST}_j^i = \mathrm{LTS}^i + L_j^i/B^i$ (where B^i and L_j^i mean the reserved bandwidth of the flow i and the length of the jth packet of flow i, respectively), was used inside the packet scheduler, suppressing feedback components in the CPS system. So, L_j^i/B^i is a simplified form of the anticipated service interval. As we can see from the Fig. 7, the resulting behavior is very abnormal. Not only flows could not obtain their subscribed bandwidth but also a large oscillation occurred. This is due to the deficiency of the feedback control mechanism. We carefully conjecture that even if a more accurate mathematical theory is used for direct control, a similar result will occur without a feedback control system.

Table 1. Ziegler-Nichols tuning rule based on step response of the system (first method).

Type of controller	K_P	T_I	T_D
PID	$1.2\frac{T}{L}$	$2 \cdot L$	$0.5 \cdot L$

Table 2. Effects of independent P, I, and D tuning on closed-loop response. (SI, SD, LD, and MC means small increase, small decrease, large decrease and minor change, respectively.)

	Increasing K_P	Increasing K_I	Increasing K_D
Rise time	Decrease	SD	SD
Overshoot	Increase	Increase	Decrease
Settling time	SI	Increase	Decrease
Steady-state error	Decrease	LD	MC
Stability	degrade	Degrade	Improve

B. Tuning of Gain Constant

If the mathematical model of the Internet can be derived, then it is possible to apply various design techniques for determining parameters of the controller. In the literature, there have been many research efforts that modelled the network analytically with proper constraints [7]–[9]. However, the Internet is so complicated that modeling the Internet with all types of traffic can not be easily obtained, then an analytical approach to the design of a PID controller is not easy. As we stated earlier, the merit of our approach is practicality, and the methodology to achieve the goal is to use an appropriate control strategy in real network systems. The reason of selecting PID control is that we can tune gain constants for the PID controller without modeling the Internet completely that requires nontrivial analysis work. This enables us to adopt black-box abstraction to model the Internet to simplify the complexity of analysis. We therefore resort to experimental approaches to the tuning of our PID controller. We use Ziegler-Nichols (Z-N) rules for tuning PID controllers [10]. Z-N rules are based on experimental step responses or based on the value of K_p that results in marginal stability when only proportional control action is used. In Z-N rules, if a controller is the PID controller, the rule we follow is shown in Table 1. In Table 1, L and T denotes delay time and time constant that are determined by observing the response curve. In the CPS system, we make L and T have the same value of 2 seconds because these time values are at least larger than our sampling time of 1 second. According to set to 1.2, 4, and 1. However, the target response bahavior of the CPS system should be stable, which means that the settling time can be longer than what Z-N rules intend to tune.

In addition to Z-N tuing rules, we also adjust the gain constants according to our performance criteria. This is done by observing various response characteristics of the CPS system with a variety of gain constant settings. The effects of gain constants on the closed-loop system are described in Table 2. To assure stability, we incerase the derivative gain K_D compared

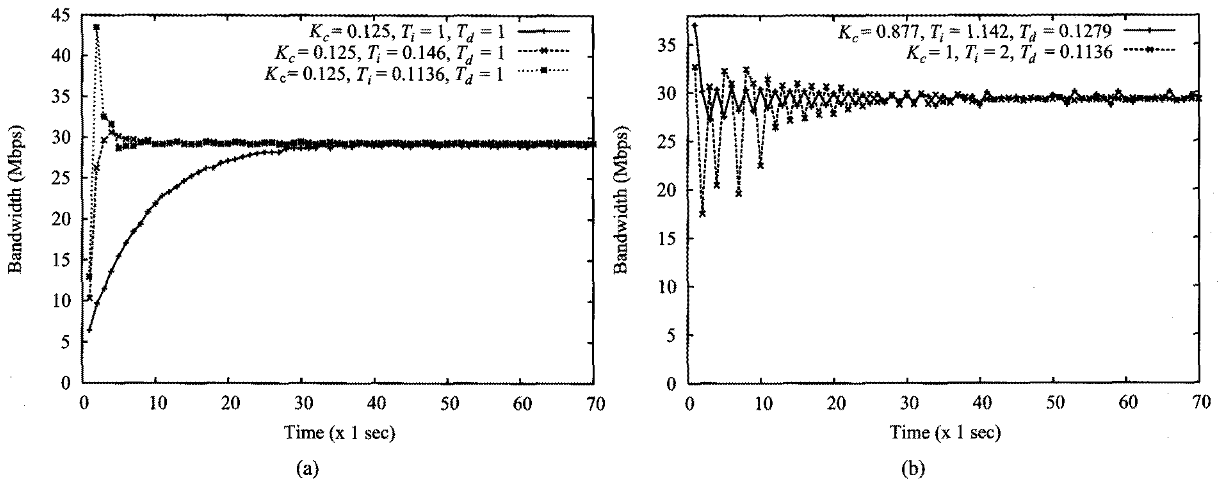


Fig. 8. Behavior of a QoS flow which reserved 30 Mbps bandwidth under PID control with various values of gain constants: (a) Effect of integral gain constant (T_i) and (b) varying gain constants.

to other two gain constants. The tuning method that we adopt in the CPS system is well accepted in the industry. This explains why the argument exists that academically proposed PID tuning rules sometimes do not work well on industrial controllers. Our choice of gain constants also reflects the gap between academia and industry.

Fig. 8 shows the resulting behavior of an application level throughput under various values of gain constants. This experiment was conducted in the LAN environment, and we measured throughput time history of a single flow and plotted all histories in one figure. In Fig. 8(a), we plotted three bandwidth trajectories measured under three conditions of varying T_i 1) $T_i = 1$, 2) $T_i = 0.146$, and 3) $T_i = 0.1136$, respectively, fixing $K_c = 0.125$ and $T_d = 1$. As $K_I = K_P/T_I$ increases, the rising time and the steady-state error decreases while overshoot increases significantly. This follows the effect described in Table 2. The first bandwidth trajectory graph ($K_c = 0.125$, $T_i = 1$, and $T_d = 1$) shows the slowest rising time, and the flow has no overshoot value. Although it converges to the target bandwidth (i.e., 30 Mbps), the settling time is too long. The second one shows the most appropriate behavior and we selected these values as the tuning value of gain constants in the CPS system. It has a fast rising time and a negligible overshoot value. It also converges to the target bandwidth very quickly without fluctuation. The last one shows very large overshoot value, although it shows the fastest rising time. A small value of T_i makes K_cT_s/T_i large, and the system becomes very sensitive to the value of $e(t)$. The last one is not appropriate for the system due to the excessive overshoot.

In Fig. 8(b), we plotted two bandwidth trajectories under different set of gain constants: 1) $K_c = 0.877$, $T_i = 1.142$, $T_d = 0.1279$, and 2) $K_c = 1$, $T_i = 2$, $T_d = 0.1136$, respectively. The first flow graph shows a small amount of oscillation, and it has a long stabilization time compared to the graph shown in Fig. 8(a). It converges to the target bandwidth after 40 seconds. The second one has a larger value of overshoot than the first, and its stabilization time is also longer than that of the first. A long stabilization time is due to the small value of T_d(=0.1136), and large fluctuations result from the large value of K_c(=1). If we increase the value of K_c, the system does not converge to the target bandwidth, oscillating infinitely. This phenomenon is caused by the characteristics of the system. Actually, using large value of K_c, the response behavior of the system largely oscillates and if the sampling period resonates with the oscillation period of the modeled system, the control system cannot converge to a stabilized point. It either diverges or keeps oscillating ceaselessly. We have finally chosen $K_c = 0.125$, $T_i = 0.146$, and $T_d = 1$ as gain constants to make the CPS system stable.

C. Flow Behavior

C.1 Bandwidth Reservation at the Receiver

We measured application level bandwidth variation when the bandwidth reservation is done by the reciever. The sender just keeps sending and the flow rate is controlled by the receiver. The experiment is performed in the LAN condition to test in a large bandwidth environment. Fig. 9(a) shows the bandwidth history of one non QoS-enabled flow and three QoS-enabled flows started at three different times: 50 seconds, 100 seconds, and 150 seconds after the experiment begins. Each QoS-enabled application process reserved 10 Mbps, 25 Mbps, and 40 Mbps of bandwidth, for a total of 75 Mbps. As we can see from the figure, the CPS system accurately controlled QoS-enabled flows competing with non QoS-enabled flow. The throughput of a non QoS-enabled flow is decreased as each of the QoS-enabled flows starts its transmission. Since the normal class flow is expected to be serviced in the boundary of spare bandwidth, our CPS system can be said to provide exact flow rate control to each flow.

Fig. 9(b) shows the result of measuring the application level throughput of network flows under the condition of the gain constants: $K_c = 0.877$, $T_i = 1.142$, and $T_d = 0.1279$. For these gain constants, the system shows some fluctuation, but the system converges to target bandwidth as time elapses. We have two processes that reserve 20 and 40 Mbps of network bandwidth each, and they start communication at 50 seconds and 100 seconds after the initiation of a non QoS-enabled flow. As we can see from the figure, even though the flow behaviors of QoS-enabled streams are oscillating before they converges, only non

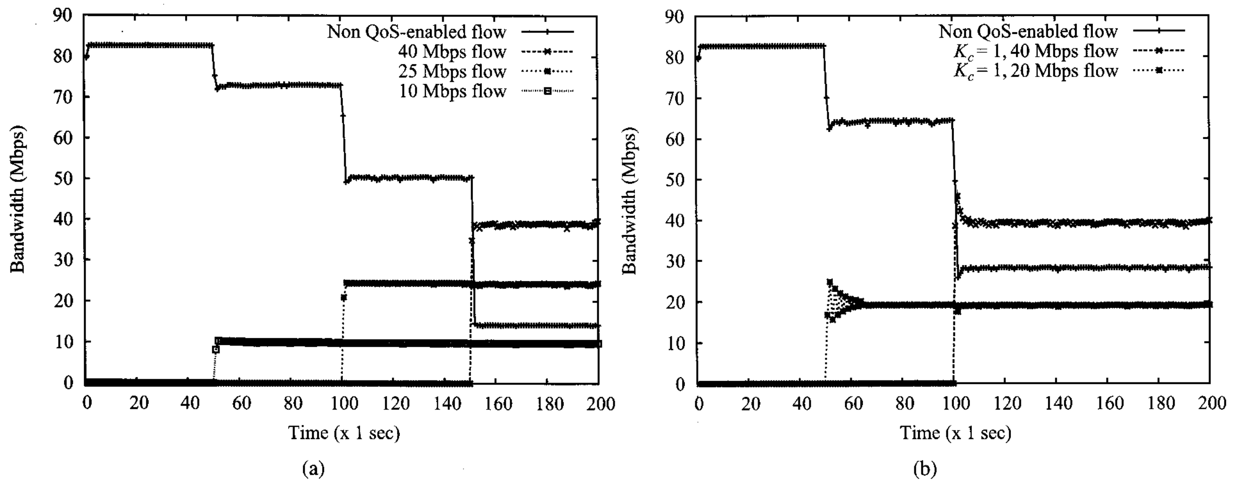


Fig. 9. Flow bahavior of multiple QoS-enabled processes and a non QoS-enabled flow: (a) $K_c = 0.125$, $T_i = 0.146$, and $T_d = 1$ and (b) $K_c = 0.877$, $T_i = 1.142$, and $T_d = 0.1279$.

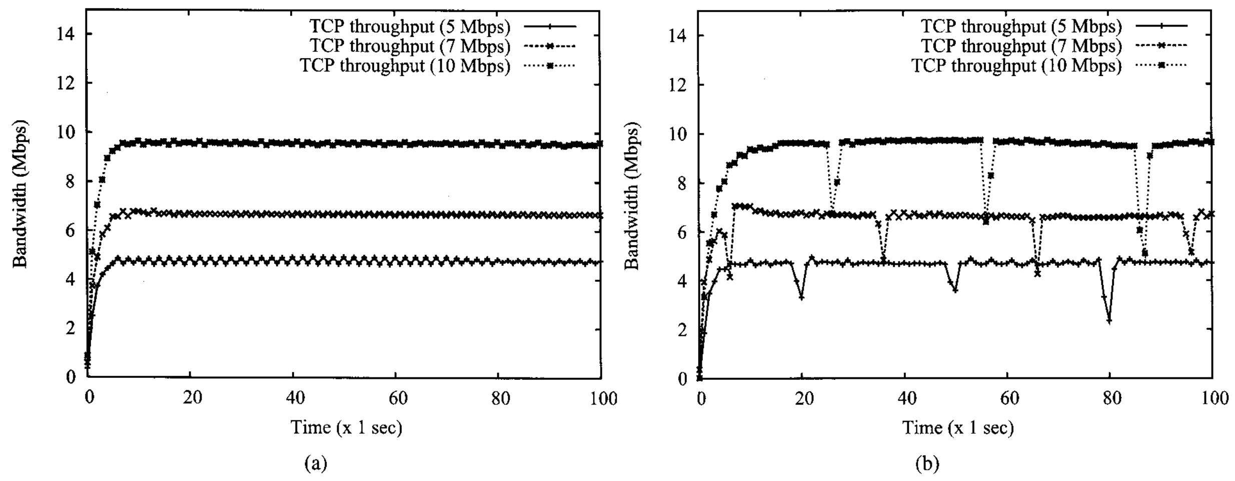


Fig. 10. Acquired throughput at the receiver that receives packets from QoS-enabled TCP server applications: (a) LAN environment and (b) WAN environment.

QoS-enabled flow is affected by oscillation. This means that the CPS system only affects other normal class flows when one of the QoS-enabled flows exhibits an abnormal flow behavior. This is one of the important characteristics that a control system should have: The localization of an anomalous flow effect.

C.2 Bandwidth Reservation at the Sender

Figs. 10(a) and 10(b) show three application level bandwidth histories measured at the receiver located in a remote machine, which is receiving packets from QoS-enabled applications using TCP in LAN and WAN enviroments. In contrast to the earlier experiment, bandwidth is reserved at the sender program and the goodput (effective throughput) is measured at the receiver. Each sender application reserved 5 Mbps, 7 Mbps, and 10 Mbps for a total 22 Mbps of bandwidth. While we measured three bandwidth histories simultaneously in the LAN environment, we performed only one flow each time in the WAN condition due to the lack of available network bandwidth in WAN.

In the LAN environment, each flow obtained 4.75 Mbps, 6.64 Mbps, and 9.46 Mbps of bandwidth, respectively, as an average goodput. The gab between target (reserved) bandwidth and actual bandwidth arised from a small amount of bandwidth that is comsumed by TCP congestion control protocol (fast retransmission and fast recovery). In the WAN environment, our system observes 4.54 Mbps, 6.44 Mbps, and 9.15 Mbps of bandwidth, respectively, on the average. As we can see from Fig. 10(b), some bandwidth collapses are observed at the WAN environment because of either the packet dropping at the autonomous system (AS) border router or a packet loss. After bandwidth collapses, sliding window flow control in TCP recovers from congestion, and the target bandwidth is regained.

D. Packet Scheduling Frequency

Since the packet scheduler is invoked based on the soft timer, we have observed that the frequency of packet scheduling highly depends on the load of the CPU. If the CPU was under full utilization, we could not acquire the satisfactory results we had expected. A soft timer module is inherently designed to avoid the receive livelock problem in core routers where there are no CPU-intensive programs in it. Under an idle CPU condition, the soft timer module is invoked once every 478 nanoseconds, while under a full CPU usage condition, it is invoked once every 8.4 milliseconds. In a pure sense, this indicates that the soft timer cannot be a good candidate to be a fine-grained timing tool under full CPU usage conditions. But in a reasonable sense, we can assume that in a server system, where the bandwidth reservation is performed for the sending process, the probability of having CPU-intensive programs is much lower than having I/O-intensive programs that invoke the packet scheduler at a high frequency.

VII. CONCLUSION

In this paper, we presented the architecture of the CPS system that can satisfy a diverse level of QoS requirements demanded by numerous kinds of applications in Internet environments. Also, we implemented the prototype of the CPS system in a Linux kernel and measured its performance. The CPS system can be ported in the kernel independent of in-kernel network protocols (i.e., congestion or flow control scheme). This design policy, which we call black-box abstraction, makes it practical to apply a feedback control theory to computer systems that have coarse-grained timing granularity. The CPS system needs only a raw input bandwidth rate and a net output bandwidth rate, observed at the end of protocol stack in the kernel. We have learned a couple of significant facts from our work: 1) There are a number of severe constraints to adopt theoretical control theory inside a kernel, 2) the appropriate gain constants differ slightly from what we expected due to the relatively long sampling interval, and 3) the implementation was relatively simple and straightforward: We modified only a couple hundred lines of Linux kernel.

REFERENCES

[1] M. Aron and P. Druschel, "Soft timers: Efficient microsecond software timer support for network processing," *ACM Trans. Computer Systems*, vol. 18, no. 3, Aug. 2000.

[2] K. Ogata, *Modern control engineering*, 4th Ed., Prentice Hall.

[3] H. S. Jung, I. Lee, and H. Y. Yeom, "Control-theoretic approach for a QoS router," in *Proc. IEEE HSNMC*, 2004.

[4] S. Keshav, "A control-theoretic approach to flow control," in *Proc. ACM SIGCOMM*, 1991.

[5] I. Stoica, S. Shenker, and H. Zhang, "Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks," in *Proc. ACM SIGCOMM*, 1998.

[6] J. C. R. Bennett and H. Zhang, "WF2Q: Worst-case fair weighted fair queueing," in *Proc. IEEE INFOCOM*, 1996.

[7] S. Kunniyur and R. Srikant, "Analysis and design of an adaptive virtual queue (AVQ) algorithm for active queue management," in *Proc. ACM SIGCOMM*, 2001.

[8] V. Misra, W. Gong, and D. Towsley, "Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED," in *Proc. ACM SIGCOMM*, 2000.

[9] S. H. Low, F. Paganini, J. Wang, S. Adlakha, and J. C. Doyle, "Dynamics of TCP/RED and a scalable control," in *Proc. IEEE INFOCOM*, 2002.

[10] J. G. Ziegler and N. B. Nichols, "Optimum settings for automatic controllers," *Trans. ASME*, vol. 64, no. 8, pp. 759–768, 1942.

[11] V. Sundaram, A. Chandra, and P. Goyal, "Application performance in the qlinux multimedia operating system," *ACM Multimedia*, Nov. 2000.

[12] S. Ghosh and R. Rajkumar, "Resource management of the OS network subsystem," *IEEE ISORC*, May 2002.

[13] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time systems," in *Proc. SPIE/ACM Conf. Multimedia Comput. and Netw.*, Jan. 1998.

[14] A. Molano, R. Rajkumar, and K. Juvva, "Dynamic disk bandwidth management and metadata pre-fetching in a reserved real-time filesystem," in *Proc. 10th Euromicro Workshop on Real-Time Systems*, June 1998.

[15] A. Molano, K. Juvva, and R. Rajkumar, "Real-time filesystems: Guaranteeing timing constraints for disk accesses in RT-mach," in *Proc. IEEE Real-Time Systems Symp.*, Dec. 1997.

[16] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating system," *ACM OSDI*, Oct. 1996.

[17] P. Goyal, H. M. Vin, and H. Cheng, "Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks," in *Proc. ACM SIGCOMM*, Aug. 1996.

[18] P. Druchel and G. Banga, "Lazy receiver processing (LRP): A network subsystem architecture for server systems," in *Proc. ACM OSDI*, Oct. 1996.

[19] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son, "A feedback control approach for guaranteeing relative delays in web servers," in *Proc. IEEE RTAS*, May 2001.

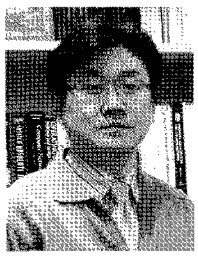
[20] Y. Lu, T. Abdelzaher, C. Lu, L. Sha, and X. Liu, "Feedback control with queueing-theoretic prediction for relative delay guarantees in web servers," in *Proc. IEEE RTAS*, May 2003.

[21] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: A simple model and empirical validation," in *Proc. ACM SIGCOMM*, 1998.

Hyungsoo Jung received his B.S. in Mechanical Engineering from Korea University, Seoul, Korea, in 2002 and his M.S. degree in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 2004. Currently, he is a Ph.D. candidate at Seoul National University. His research interests are computer networks, distributed systems, and large-scale data management systems.

Hyuck Han received his B.S. and M.S. degrees in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 2003 and 2006, respectively. Currently, he is a Ph.D. candidate at Seoul National University. His research interests are distributed computing systems and algorithms.

Heon Young Yeom is a Professor with the Department of Computer Science and Engineering, Seoul National University. He received his B.S. degree in Computer Science from Seoul National University in 1984 and received the M.S. and Ph.D. degree in Computer Science from Texas A&M University in 1986 and 1992, respectively. From 1992 to 1993, he was with Samsung Data Systems as a Research Scientist. He joined the Department of Computer Science, Seoul National University in 1993, where he currently teaches and researches on distributed systems, multimedia systems and transaction processing, etc.



Sooyong Kang received his B.S. degree in Mathematics and the M.S. and Ph.D. degrees in Computer Science, from Seoul National University, Seoul, Korea, in 1996, 1998, and 2002, respectively. He was then a Postdoctoral Researcher in the School of Computer Science and Engineering, Seoul National University. He is now with the Division of Computer Science and Engineering, Hanyang University, Seoul. His research interests include operating systems, multimedia systems, storage systems, flash memories and next generation nonvolatile memories, and distributed computing systems.