NetDraino: Saving Network Resources via Selective Packet Drops

Jin K. Lee and Kang G. Shin
Department of Electrical Engineering and Computer Science
The University of Michigan, USA
{jinkukl, kgshin}@eecs.umich.edu

Contemporary end-servers and network-routers rely on traffic shaping to deal with server overload and network congestion. Although such traffic shaping provides a means to mitigate the effects of server overload and network congestion, the lack of cooperation between end-servers and network-routers results in waste of network resources. To remedy this problem, we design, implement, and evaluate *NetDraino*, a novel mechanism that extends the existing queue-management schemes at routers to exploit the link congestion information at downstream end-servers. Specifically, *NetDraino* distributes the servers' traffic-shaping rules to the congested routers. The routers can then selectively discard those packets—as early as possible—that overloaded downstream servers will eventually drop, thus saving network resources for forwarding in-transit packets destined for non-overloaded servers. The functionality necessary for servers to distribute these filtering rules to routers is implemented within the Linux *iptables* and *iproute2* architectures. Both of our simulation and experimentation results show that *NetDraino* significantly improves the overall network throughput with minimal overhead.

Categories and Subject Descriptors: C.2.1 [**Network Architecture and Design**]: Network communications ; C.2.3 [**Network Operations**]: Network management

General Terms: Network management, server overload, network congestion

Additional Key Words and Phrases: Network management, Internet servers and routers, congestion and overload control, traffic filtering

1. INTRODUCTION

The rapidly growing number of Internet users and services place increasing demands on web servers and network links, making it possible to overload end-servers and congest network links. Unfortunately, several recent Denial of Service (DoS) attacks and flash crowds on popular servers have shown that the current Internet architecture lacks a configurable mechanism for overload protection. Sudden load-surges can easily overload the servers, which may, in turn, lead to denial of service or loss of data. They can also easily congest network links, and *all* flows that run

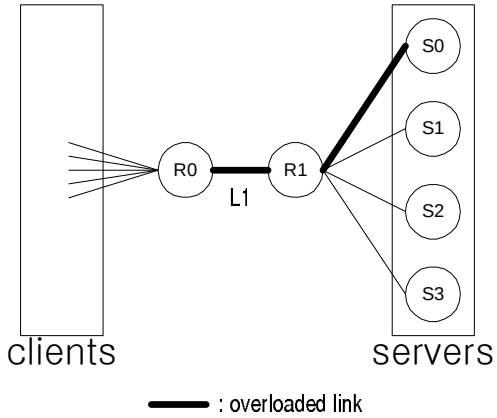


Figure 1. An example network with a congested network link L_1 shared by overloaded (S_0) and non-overloaded (S_1, S_2, S_3) servers.

through the congested links will suffer significant delays or losses. Simple active queue management such as BLUE [Feng et al. 2002], Drop-Tail and RED [Floyd and Jacobson 1993], can handle link congestion by dropping packets in the core of the Internet. However, these mechanisms do not differentiate among different flows/packets, and therefore, there is no way to selectively suppress high-bandwidth and/or low-priority traffic.

A popular configurable approach to mitigating the effect of congestion is to use traffic shaping in either network-routers or end-servers. Traffic shaping enforces prioritization of the transmission/reception of data over a network link. Typically, it associates packets with their traffic classes and regulates the incoming and/or outgoing rate of each traffic class as specified by the traffic-shaping rules. With traffic shaping, routers and servers can distinguish between the individual flows'/aggregates' Quality-of-Service (QoS) requirements and protect themselves from overload. For example, a web server can be configured to discard the requests from non-preferred customers to keep the total incoming request rate within the server's capacity limit.

QGuard [Jamjoom and Reumann 2000] is one such architecture that exploits the power of traffic shaping to provide overload protection and service differentiation at an end-server. It monitors the server's load and dynamically sets up a rate control to accept as much traffic as possible without overloading the server. QoS differentiation is achieved by allowing a higher incoming rate for higher-priority traffic. While simple *iptables* [Andreasson 2001] in the Linux firewall system can also provide server overload protection with QoS differentiation, *QGuard*'s adaptation mechanism maximizes server utilization. Unfortunately, neither *QGuard* nor *iptables* helps reduce network link congestion, as packets are dropped only *after* they are delivered to the end-server at the very edge of the network. Packets that the end-server will eventually drop as a result of its traffic shaping have already consumed network resources on their way to the end-server, possibly causing the dropping of other packets destined for non-overloaded servers. Let us consider

the network shown in Figure 1 where a sudden surge of requests toward the end-server S_0 has congested the network link L_1. S_0 can provide QoS differentiation to incoming requests by its own traffic shaping, but the active queue-management mechanism at the congested link L_1 will indiscriminately discard packets destined for the other end-servers S_1, S_2 and S_3.

To remedy this problem, the router R_0 should shape only the traffic destined for S_0. *Committed Access Rate* (CAR) [Inc. 1998] and *Traffic Control* (TC) [Almesberger 1998] tools are such traffic-shaping features provided by Cisco and Linux routers, respectively. While the network administrator can use these features to set up the routers' rate controls, the routers cannot 'sense' their downstream servers' conditions in real time. Consequently, they might erroneously (i) drop packets destined for lightly-loaded servers and/or (ii) forward more requests to already-overloaded servers.

To combat the above problem, we design, implement, and evaluate *NetDraino*, a novel mechanism that allows a server to inform its upstream routers of its congestion status. Routers can then use this information to decide on packet drops. In the above example, R_0 equipped with *NetDraino* becomes aware of S_0's actual acceptance rate and allows only that rate of traffic toward S_0 to traverse L_1, thus making more bandwidth available for forwarding traffic toward lightly-loaded servers S_1, S_2 and S_3. The main goal of *NetDraino* is to selectively discard those packets—as early as possible in the network (before they reach the servers)—that the overloaded servers will eventually drop. Not only *NetDraino* can relieve S_0 from the burden of dropping excess packets, but, more importantly, it can also minimize possible service degradation that S_1, S_2 and S_3 may suffer. As a prototype to realize this goal, we apply the server-side *NetDraino* to the Linux 2.4 kernel's *iptables* firewalling system and use Linux routers with the *iproute2*'s traffic control (TC) to implement the router-side *NetDraino*.

This paper is organized as follows. Section 2 gives an overview of load-shedding mechanisms, highlighting the key features of *NetDraino*. Section 3 places *NetDraino* in a comparative context with related work. Section 4 details the design of *NetDraino*. Using the *ns* simulator [UCB/LBNL/VINT 2000], Section 5 evaluates the performance of *NetDraino*. Section 6 summarizes the implementation of a *NetDraino* prototype and Section 7 uses the implemented prototype to experimentally evaluate the benefits as well as the overhead of *NetDraino*. Finally, the paper concludes with Section 8.

2. OVERVIEW

Network links can be persistently overloaded for several reasons [Mahajan et al. 2001]. Links can be congested by ill-behaved flows, which do not conform to end-to-end congestion control. By a "flow," we mean a stream of packets identified/specified with its end-point attributes such as source and destination IP addresses, protocol field, and port numbers. To handle the link congestion caused by such ill-behaved flows, traffic-shaping schemes should schedule packets on a per-flow basis. Obviously, a router equipped with flow-based congestion control must maintain state for each flow, making such a scheme unscalable in the core network.

Another traffic type that may congest links is a high-bandwidth traffic class (or

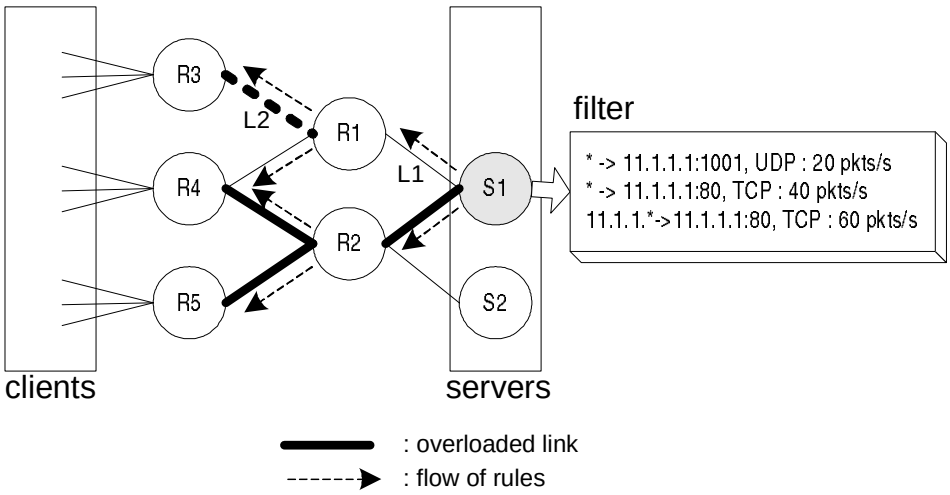


Figure 2. An example network showing intuitive rule propagation.

an aggregate) that can be formed, for example, by flash crowds or distributed DoS (DDoS) attacks, both of which are the main target of *NetDraino*. By "traffic class (aggregate)," we mean a set of flows with common properties, such as the same destination IP address and port. This notion of traffic class is commonly used in policy specifications for firewalls and was initially proposed in [Mogul et al. 1987]. In *NetDraino*, traffic-shaping rules at servers specify how to regulate high-bandwidth or low-priority traffic classes. For example, a rule specifying <destination address: 11.1.1.2, destination port: 80, protocol: TCP> defines a traffic class. If there is a large amount of traffic belonging to this class, it will be identified as a high-bandwidth traffic aggregate and then regulated at routers. Note that whether the traffic is malicious (i.e., DDoS) or not (i.e., flash crowds) does not matter to the network; either case generates high-bandwidth traffic aggregates that can be identified by traffic-shaping rules at target servers. Jamjoom [Jamjoom and Shin 2003] has shown that the high-bandwidth traffic aggregates in the forward path can cause link overload. The scheme in *NetDraino* attempts to mitigate the effect of such link congestion to protect the flows on the overloaded links without compromising the aggregate's target.

2.1 Naive Load-Shedding

We first consider an intuitive (naive) load-shedding mechanism and identify its limitations. Suppose that servers are protected by *QGuard* [Jamjoom and Reumann 2000]. To discard excessive packets before they reach an overloaded server, the server may distribute the *QGuard* rules of the currently-configured filter to its upstream routers. The rules propagate to all routers within N hops of the server, where N is a design parameter. A router receiving the rules classifies packets and regulates their incoming rate according to these rules.

Consider the network in Figure 2. The thick lines in the figure represent heavy packet flows towards S_1. The overloaded server S_1 activates/triggers a filter that

consists of the three rules shown in the figure. Suppose that S_1 decides to distribute a rule that limits the incoming rate of traffic class <* → 11.1.1.1:80, TCP>, to routers R_1 and R_2. Upon receiving the rule, R_1 and R_2 decide whether to install it, based on their links' load conditions. Suppose that R_2 installs the rule, then the traffic rate from R_2 to S_1 identified by the rule <* → 11.1.1.1:80, TCP> will be limited by the rate specified in the rule distributed by S_1. R_1 and R_2 further distribute the rule to R_3, R_4, and R_5. Suppose that the rule was accepted by both R_4 and R_5. When R_4 and R_5 begin to shape the traffic according to this rule, the corresponding traffic from R_4 to R_2 and from R_5 to R_2 is regulated. As a result, traffic towards the lightly-loaded server, S_2, will have a greater chance of reaching S_2 without being dropped in the network. The rule will propagate further to upstream routers of R_3, R_4, and R_5, until it reaches routers located N hops away from S_1.

Note that *QGuard* rules are distributed from a server to *all* of its upstream routers. This is obviously inefficient. In order to achieve wide routers' participation, N might have to be large, which can increase the number of a router's downstream servers flooding their rules to the router. When a router has a large number of upstream routers directly connected to it, the rule distribution would be very expensive. Moreover, the mechanism assumes that all the routers participate in adopting and propagating rules; a single non-conformant router can interfere with the entire set of desirable operations.

In the example network of Figure 2, one may suggest lighter-weight rule propagation since S_1 only needs to distribute the rule to R_2 (because the link L_1 from R_1 to S_1 is not congested). Unfortunately, this choice would limit R_3's ability to selectively drop packets. This is disadvantageous to R_3 when the link L_2 from R_3 to R_1 is congested by the flow <* → 11.1.1.1:80, TCP>. Even though L_1 is not congested, L_2 can be congested by the flow because the capacity of L_2 may be lower than L_1. We, therefore, need a more intelligent mechanism that allows all compliant routers to participate in the load-shedding protocol.

2.2 Sketch of NetDraino

NetDraino allows an end-server to distribute its traffic-shaping rules to its upstream network-routers. For scalability, not only the rules should be distributed efficiently, but also they should not over-burden the routers in installing them. A router cannot accommodate all the rules from all of its downstream servers since it may consume most of its resources just to match each packet against the many filters installed on it. The router should, therefore, install only those rules that may significantly lighten its link congestion condition. Furthermore, since the congestion condition varies with time, the routers' rule sets must also be adapted dynamically.

In *NetDraino*, a congested router informs its downstream end-servers of its IP address. Specifically, the congested router first advertises its congestion condition by marking the packets going through the congested link. If an end-server receiving the router congestion notification happens to be overloaded, it replies back by marking its outgoing packets so that the congested router can be informed of the congested server. Then, the congested router can append its IP address to the packets destined for the overloaded servers. This way, a server learns of the IP addresses of congested routers, and hence, can directly distribute the rules only to

those routers which are congested, instead of distributing rules indiscriminately to all of its upstream routers. Note that the server can customize the rules for the targete router so that they can effectively and correctly soothe congestion at the router. The routers regulate packets according to the rules they received from their downstream servers in order to lighten up their link congestion and cut down the requests toward the congested servers. To minimize the cost to dynamically adapt to varying network condition, *NetDraino* takes a soft-state approach to managing the rules distributed to each router; the rules must be refreshed within a certain timeout period, else they expire and hence will become invalid. *NetDraino* is more desirable than the intuitive scheme because only overloaded servers and congested routers need to classify and regulate packets, and communicate with each other.

While *NetDraino* handles link congestion by separating high-bandwidth traffic aggregates from other normal traffic, it is orthogonal to a traffic differentiation algorithm that identifies high-bandwidth aggregates at target servers: it assumes that traffic-shaping rules at the servers correctly reflect the prioritization policy for each traffic class, and resorts to the definitions in the policy or rules for identifying high-bandwidth aggregates. In this respect, *NetDraino* may complement end-host-based schemes for mitigating DoS attacks [Lakshminarayanan et al. 2003; Yaar et al. 2004]. Note that the rules are not distributed when none of the congested server's upstream routers is overloaded. This avoids unnecessary deployment overhead, because none of the servers may suffer service degradation if their upstream routers are not congested. In this case, the server receiving the heavy traffic can easily protect itself with well-studied overload control schemes [Jamjoom and Reumann 2000; Iyer et al. 2000; Andreasson 2001; Welsh and Culler 2003].

3. RELATED WORK

NetDraino combines traffic shaping at servers with packet scheduling and preferential dropping at routers. Specifically, the routers in *NetDraino* enforce an anticipatory flow control back-pressed by traffic shapers at severs. In this section, we first discuss the related work that prevents congestion and overload of severs and routers. Then, we briefly review the work that also uses a back-pressure control scheme in a distributed system, stressing the distinct features of *NetDraino*.

Traffic regulation at servers was usually studied as a part of QoS management solutions for Internet severs. These solutions allow the network administrators to specify a traffic-shaping policy, which defines rate limits for different traffic classes. For example, Linux *iptables* [Andreasson 2001] and Extreme Network's *ExtremeWare* [Inc. 2002] provide an interface to specify traffic classifiers and associated rate limits. These approaches are based on static rate control; their traffic-shaping policy cannot be integrated with the current overload status of Internet servers, without intervention of administrators. To maximize server utilization, *APF*s [Reumann et al. 2001] in *QGuard* [Jamjoom and Reumann 2000] allow administrators to define dynamic overload responses for their systems. Based on monitoring the input of an overload server, *QGuard* protects the Internet server dynamically with the least restrictive filter possible. While all these schemes protect servers from overload and provide a certain level of QoS differentiation, they have a common limitation of inbound rate controls; the earlier their traffic-shaping policy is enforced in the

network, the better performance they may provide.

There are numerous studies on router scheduling mechanisms. While simple active queue management such as Drop-Tail and RED can handle link congestion by dropping excess packets, RED with ECN [Floyd 1994], RED variants [Feng et al. 1999; Ott et al. 1999] and BLUE [Feng et al. 2002] were proposed to improve the network performance in the presence of congestion. While these schemes alone cannot differentiate packets with various priorities, per-flow scheduling mechanisms and their variants such as Fair Queueing [Demers et al. 1989] and Core-Stateless Fair Queueing [Stoica et al. 1998] enable preferential dropping of packets with flow-based congestion control. Another approach to differentiating and rate-limiting packets is a class-based scheduling mechanism, which regulates the traffic aggregates based on traffic classifiers. Class-based scheduling mechanisms include CBQ [Floyd and Jacobson 1995] and Adaptive Packet Marking [Feng et al. 1997], and *NetDraino* can be viewed as a descendant of these approaches. The contribution of *NetDraino* is to enable a router to have dynamic definitions of aggregates that cause congestion at its downstream severs, instead of fixed definitions specified by administrators.

NetDraino "pushes" traffic-shaping rules at servers—defined either statically or dynamically—into the network so that routers can regulate traffic aggregates before they reach the servers. This concept of back-pressure flow control can be found in literature. First, it dates back to Tymnet and X.25 [CCITT 1998b] packet-switched networks with their hop-by-hop flow control. ATM networks also use the back-pressure scheme for their flow control on VCs. In TCP/IP Internet, Iyer *et al.* [Iyer et al. 2000] argue that incoming traffic into a server had better be regulated early at the server's network interface card (NIC), or at the server's previous node (a proxy server), for effective overload control in servers. In a similar context, Welsh and Culler [Welsh and Culler 2003] use their staged event-driven architecture (SEDA) to apply admission control at as early stages as possible, based on the performance of back stages. These control schemes are to minimize the server's work spent on a request which is eventually not serviced due to overload. On the other hand, *NetDraino* tries to minimize the network resources consumed for such a request, and is orthogonal to the schemes mentioned above; *NetDraino* may seamlessly combine with the existing overload control schemes at servers.

Lakshminarayanan *et al.* [Lakshminarayanan et al. 2003] take the same view of *NetDraino* to propose that end-hosts be responsible for defining traffic filters and pushing them back into the network. But their approach still focuses on protecting the target hosts, thus inserting filters at the last hop router only instead of propagating them far into the network. Pushback [Ioannidis and Bellovin 2002], a mechanism for defending against DDoS attacks, has an approach similar to *NetDraino*, in that routers dynamically adopt and propogate traffic-shaping rules, which identify aggregates and specify rate limits. But their definitions of aggregates do not reflect the condition of servers, and the rules should be propagated router-by-router. Persistent dropping (PD) [Jamjoom and Shin 2003] takes into account the effects of client persistence on the controllability of aggregate traffic. PD complements the existing traffic-shaping solutions by controlling the reaction of the underlying traffic to a rate-limiting policy, and *NetDraino* will also benefit from PD.

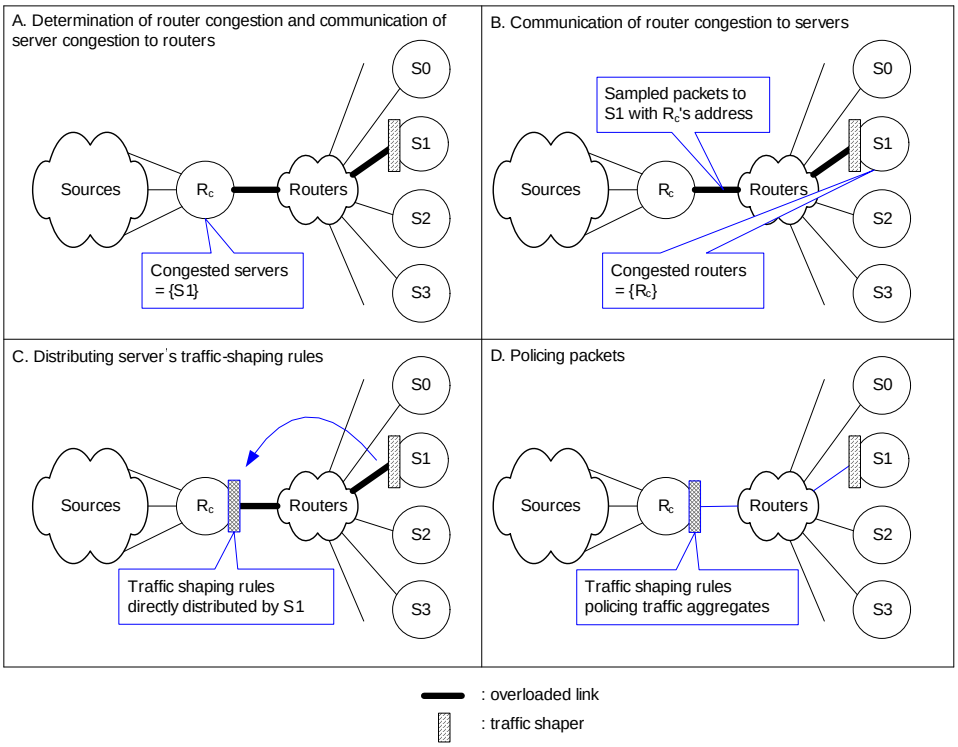


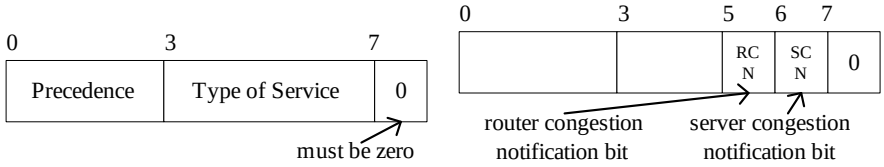
Figure 3. NetDraino example.

4. NETDRAINO

This section details the design of *NetDraino*. Specifically, Sections 4.1 through 4.4 describe the steps taken in *NetDraino* to accomplish the cooperation between routers and servers as outlined in Section 2.2. Figure 3 depicts a simplified pictorial example of how *NetDraino* works in each step. The first two steps are to let congested servers know the IP addresses of the congested routers, and prepare the traffic-shaping rules to be distributed to the congested routers. The last two steps perform actual traffic regulation at the congested routers, and maintain the rules distributed to each congested router.

4.1 Determination of Router Congestion and Communication of Server Congestion to Routers

Before informing servers of its IP address, a congested router should first learn of congested servers. *NetDraino* uses two ToS bits in the IP packet header for this purpose. As shown in Figure 4, *NetDraino* uses one of ToS bits to indicate router congestion and the other to indicate server congestion. Note that if there is no congested server at all, thus no hint on downstream servers' overload conditions, the router cannot take advantage of *NetDraino*. Instead, it can resort to its own AQM policy [Floyd 1994; Floyd and Jacobson 1995; Feng et al. 2002; Jamjoom and Shin 2003] to quench congestion. But fortunately, the very natures of traffic



(a) IPv4 ToS byte (b) Newly-specified ToS byte

Figure 4. Use of ToS bits in *NetDraino*.

aggregates such as flash crowds or DoS attacks suggest that there exist conngested servers corresponding to the traffics causing router congestion.

As in RED, a router equipped with *NetDraino* calculates the average queue length using the exponentially-weighted moving average. This average queue length is compared to a threshold $th_{congest}$, a design parameter, that *NetDraino* uses to determine if the router is congested. When the average queue length is greater than $th_{congest}$, every arriving packet is randomly marked with the Router-Congestion-Notification (RCN) bit set to 1. Thus, packets with RCN bit=1 indicate that they traversed at least one congested router. Random marking of packets in this step is to reduce the chance of marking packets that are not part of heavy traffic.

When a server equipped with a traffic rate limiter receives a packet with RCN bit=1, it checks if the marked packet constitutes excess traffic to the server, i.e., the server is overloaded. If yes, it sets the Server-Congestion-Notification (SCN) bit in the header of its outgoing packets with the destination equal to the source of each marked packet. When a congested router receives the packets with SCN bit=1, it knows which servers are congested and may begin to notify the servers of its IP address as described in Section 4.2. Note that, in this scenario, the congested router should reside both in the forward and reverse paths between the client and the server. This is a reasonable assumption within an AS. But, as AS path asymmetry is known to be common in the Internet, we may have to slightly modify the mechanism for *NetDraino* to be widely deployed by multiple ISPs. For example, in this extended scenario, a client, instead of a router, can be aware of the packets with SCN bit=1, and use an additional ToS bit to notify routers of server congestion. That is, it can mark the packets destined for the congested servers so that the routers along the packets' paths can be aware of the congested servers as well.

The router stores the IP addresses of thus-identified congested servers using a FIFO queue called *CServs* (Congested Servers) whose length is ℓ_Q. Since the number of simultaneously-congested servers is expected to be small, ℓ_Q is usually small. IP addresses in *CServs* age with time. If their age exceeds a pre-set limit, they will be removed to prevent the router from using obsolete servers congestion information. Since *NetDraino* adds to *CServs* the source (server) addresses of packets with SCN bit=1 whenever such packets arrive at a congested router, *CServs* may contain the server addresses for which the router's heavy traffic is not destined. While a mechanism for keeping the history of packets marked with RCN bit could be added to handle these false congested server addresses, such a situation would occur very rarely and hence we omitted it. Figure 5 depicts the algorithm for a

upon arrival of a packet p at dev_{in}
    if (p is marked with a SCN bit) then
        add a source address of p to *CServs* list of dev_{in}
upon departure of a packet p from dev_{out}
    calculate the average queue size Q_{avg}
    if ($Q_{avg} > th_{congest}$) then
        mark p with a RCN bit randomly

Figure 5. The algorithm for a router to identify congested servers.

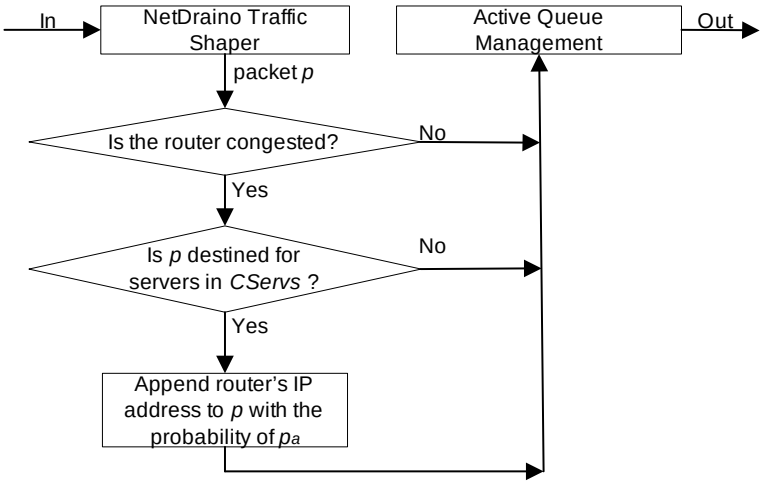


Figure 6. The algorithm for a router to broadcast its IP address.

router to identify congested servers.

4.2 Communication of Router Congestion to Servers

Once a congested router has identified congested servers, it appends its IP address to the IP option fields of packets destined for the servers in *CServs* so that it may receive hints as to which packets the servers will probably discard. We assume that all routers just forward, rather than drop, the packets with unknown IP options, though we do not require that all the routers be aware of *NetDraino* and participate in the mechanism. The servers receiving the packets with the *NetDraino*-filled IP option can identify the congested routers and collect & maintain statistics of how much of their traffic has gone through each congested router. The hints on packet drops are returned to the congested routers after they are tailored to the statistics collected by the servers. Because it is expensive to append additional data to a packet in flight, a router does not alter all in-transit packets. Instead, it modifies each candidate packet with some probability p_a, one of *NetDraino* design parameters. Note that the traffic consuming more bandwidth is more likely to be modified, which is desirable because it is the high-bandwidth traffic that a congested

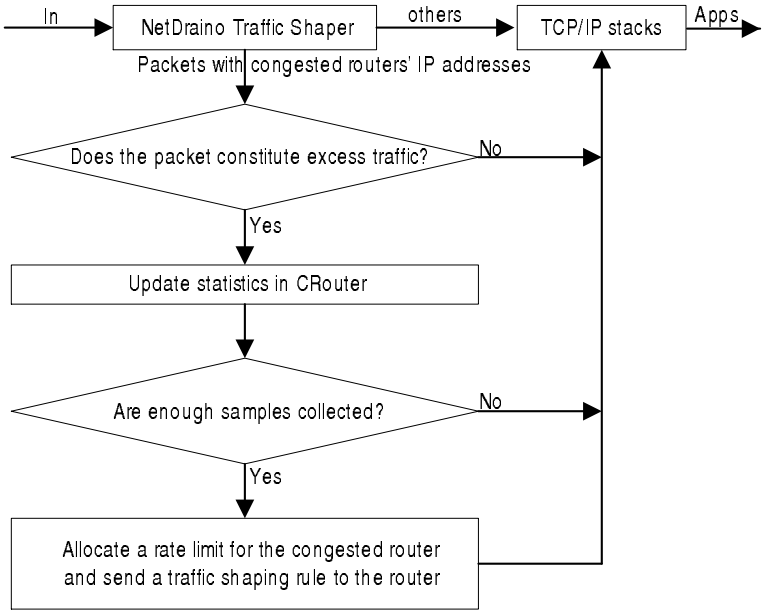


Figure 7. The algorithm for a server to distribute traffic-shaping rules.

router wants to limit.

Figure 6 depicts the proposed algorithm. When packets arrive at the router's output queue, a traffic shaper polices them according to the installed traffic-shaping rules as described in Section 4.4. If the router is not congested or the outgoing packet is not destined for the servers in *CServs*, the router does not append its address. When it decides to modify the packet, some of its upstream routers might have already appended their addresses to the packet. Even then, it just adds its own address to the packet forming a list of router IP addresses. While allowing unlimited addition of information to a packet — though IPv4 allows addition of no more than 40 bytes for IP option fields — is usually a poor design choice, allowing addition of only one router address in a packet and disallowing modification of the already-modified packet will provide servers erroneous statistics. However, more than one IP address will rarely be appended to a packet since p_a is usually very small. Our simulation results in Section 5.3 show that $p_a = 0.1$ suffices to maintain correct statistics at servers, in which case the probability of appending multiple IP addresses to packets is less than 0.01.

4.3 Distributing Server's Traffic-Shaping Rules

For each policed traffic class tc_i, a server maintains a list CR_i of congested routers' IP addresses. CR_i represents the congested routers that class tc_i packets have traversed before reaching the server. Each entry in CR_i associates with it the statistics of the traffic rate from the router represented by the entry. When a server receives a tc_i-packet with a router's IP address, the packet is fed into the corresponding entry in CR_i as a sample. If enough samples are collected in the

entry, the filtering rules will be determined and then distributed.

The server should first allocate a rate limit for the router to which it forwards a shaping rule. Suppose that (i) a rule R_i polices traffic class tc_i by limiting its rate to $rate_i$ in the server; and (ii) the server determines to distribute R_i to one of its upstream routers, $router_U$. The server knows that the total incoming rate of tc_i is $total_{tc_i}$, and uses the statistics maintained in CR_i to infer the fraction, $from_{router_U}$, of this traffic that comes from $router_U$. Then, the ideal rate limit for $router_U$ would be somewhere between $rate_i * from_{router_U}/total_{tc_i}$ and $rate_i$. It is very difficult to get an exact value for this rate limit since it depends on the actual traffic arrival process. We use $k*from_{router_U}/total_{tc_i}$ as the fractional rate limit for $router_U$, where k is an experimentally-determined scaling factor. After determining the rate limit for the congested router, the server distributes the tailored rule to the corresponding router. Figure 7 summarizes this algorithm.

4.4 Policing Packets

Once a router receives the rules from a server, it installs them and polices packets according to them. The received rules constitute the *NetDraino* traffic shaper shown in Figure 6. The server does not maintain the rules once they are distributed. Instead, the router maintains the installed rules by itself. This minimizes the amount of state information each end-server needs to maintain, thus increasing the robustness of *NetDraino*; it can tolerate router or server failures without incurring any further overhead. Failed servers or routers will simply be "forgotten" via expiration of soft states by *NetDraino*.

The rules installed in each router remain valid only for δ seconds after which they time out. Suppose that a rule R_i is to police traffic class tc_i by limiting its rate to $rate_i$ in a router. By installing R_i at time t_0, the router could reduce its congestion. If it removes R_i at $t_0 + \delta$, its outgoing link can become heavily-overloaded by unregulated tc_i traffic because the traffic might not yet have subsided. Instead of completely removing an expired rule, *NetDraino* gradually increases the rate limit specified in the rule. Let $total\_rate_i$ denote the rate limit for tc_i in the server which distributed R_i. *NetDraino* places several gradual rate limit steps between $rate_i$ and $total\_rate_i$, which are incrementally adopted at every δ seconds by each installed rule in the router as it ages. While relaxing the rate limit, both the router and the server might become congested. In such a case, the mechanism restarts. The router will mark packets with its IP address and the congested servers will eventually send updated rules to the router. $rate_i$ is removed from the router either when its rate limit becomes higher than $total\_rate_i$ or when it becomes lightly-loaded.

5. SIMULATION

The performance of *NetDraino* under various configurations is evaluated using the *ns* simulator [UCB/LBNL/VINT 2000] for an example network with 25 clients and 5 servers as shown in Figure 8. Each client requests multiple TCP and UDP connections to each server. Pareto ON/OFF sources with mean ON-time of 500 ms and mean OFF-time of 1000 ms were run to generate TCP traffic, and CBR sources were run to generate UDP traffic. In addition, all sources started transmission of 1-KB packets at a random time within the first 3 seconds of simulation. The routers were configured to store up to 30 packets in each output queue under a

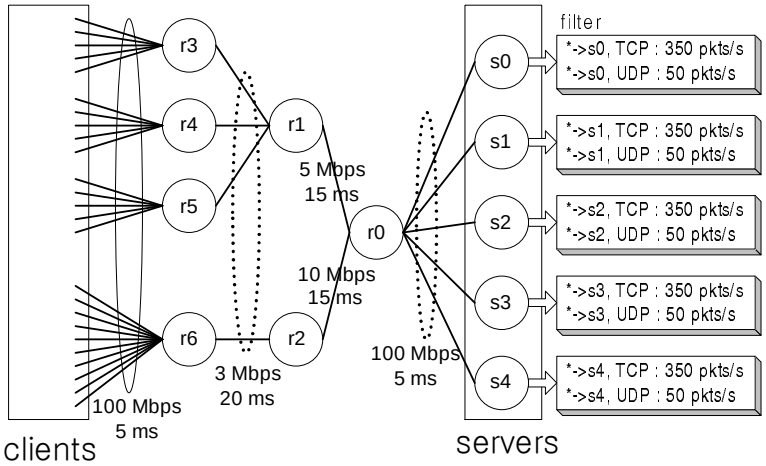


Figure 8. The network topology used for simulation.

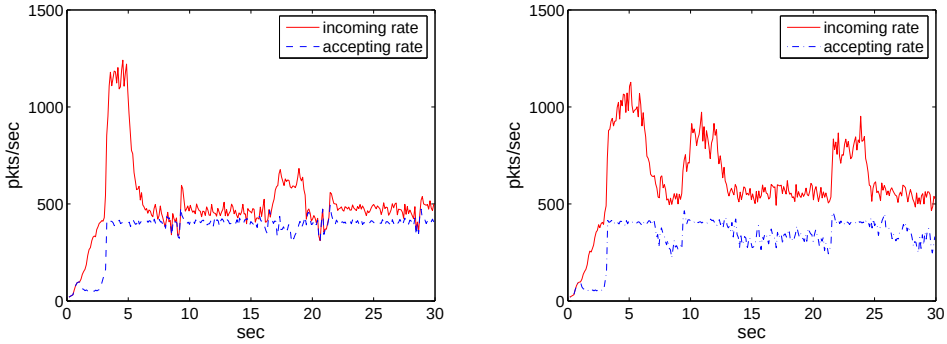
Table I. Default configuration.

# TCP connections to S_0 per client	100
UDP traffic rates to S_0 per client	18 KB/sec
# TCP connections to each other server per client	10
UDP traffic rates to each other server per client	4 KB/sec
k	1.2
$th_{congest}$	15
p_a	0.1

RED queueing discipline modified to support *NetDraino*. Among servers, we made S_0 overloaded by making most of connection requests to S_0. Each server adopted a simple packet filter that polices the rates of TCP and UDP traffic as shown in Figure 8. It simulates *iptables* rather than *QGuard* since it does not adapt itself to the server's load. The default configurations in Table I were used for our simulation unless specified otherwise.

5.1 Incoming traffic rate at a congested server

To understand the behavior of *NetDraino*, Figures 9(a) and 9(b) compare the incoming traffic rates of the congested server S_0 in the example network with and without *NetDraino*. Both the rate of traffic arriving at the server and that of traffic accepted by the server are shown in each graph. The filters were distributed for the first time around $t = 5.0$ in the simulation. In Figure 9(a), the overlap of two traffic rates after $t = 5.0$ shows that traffic to S_0 was regulated by upstream routers and few packets were rejected by S_0, while the difference between two traffic rates in Figure 9(b) shows that unregulated traffic to S_0 was filtered out at S_0, wasting the resources to deliver packets which were dropped at S_0. During the simulation the acceptance rate in Figure 9(a) remained around 400 pkts/s, which was a total traffic rate limit specified by the filter of S_0. This means that traffic was regulated properly; few packets were filtered out in the upstream routers of S_0 when S_0 could



(a) Traffic rate with *NetDraino* (b) Traffic rate without *NetDraino*

Figure 9. Traffic rates of the congested server.

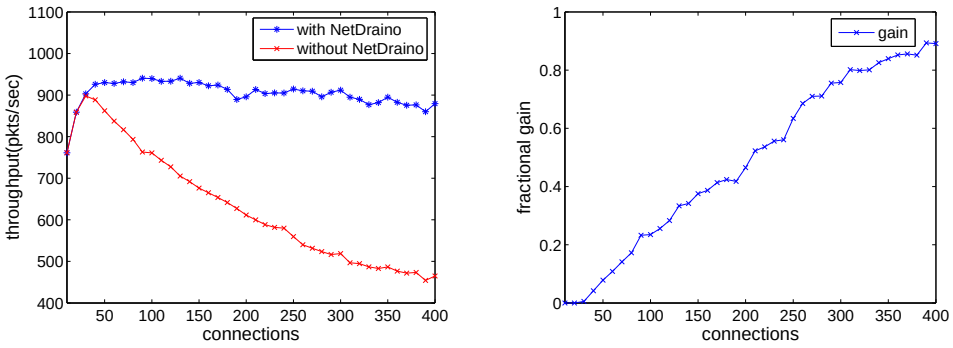
accept all of the packets.

5.2 Throughput

We now show the increase of network throughput by adopting the *NetDraino*'s selective packet drop mechanism. By throughput, we mean the average number of packets per second that are accepted by each server. To consider the effect of traffic towards the congested server, the throughput was measured by increasing the number of TCP connections and the UDP sending rate of each client. The number of TCP connections was varied from 10 to 400 in increments of 10, while the UDP sending rate was varied from 0 KB/s to 78 KB/s in increments of 2KB/s. For readability, We provided only the number of TCP connections in the x-axis of the graph. Figure 10(a) shows the throughput measured from simulations with and without *NetDraino*, while Figure 10(b) shows the fractional throughput gain of *NetDraino*. The fractional throughput gain is the percentage increase in the throughput of the network with *NetDraino*, compared to that without *NetDraino*. In Figure 10(a), the throughput of the network with *NetDraino* remains unchanged as the number of connections to S_0 increases over a certain point (around 40 TCP connections), while that without *NetDraino* decreases as more traffic is bound for S_0. Since the congested links remain fully-utilized for both systems, the gap between the two represents the number of packets that waste the network resources without *NetDraino*. The gain of network throughput with *NetDraino* in Figure 10(b) is the result of the decreasing throughput without *NetDraino*. It shows that the throughput of the network with *NetDraino* is almost 90% higher than that without *NetDraino* when traffic bound for S_0 is the heaviest (400 TCP connections), at which point the ratio of traffic toward S_0 to other traffic is 10 to 1.

Note that the throughput gain keeps increasing as traffic bound for S_0 gets heavier. The throughput goes up to 1.27 and 1.38 when traffic to S_0 increases to 800 and 1200 TCP connections, respectively. Theoretically, it reaches its maximum when traffic to S_0 overwhelms other traffic, which is about

$$\frac{900 \text{ pkts/sec (the maximum bandwidth of network)}}{450 \text{ pkts/sec (the accepting rate of } S_0)}.$$



(a) Throughput (b) Throughput gain

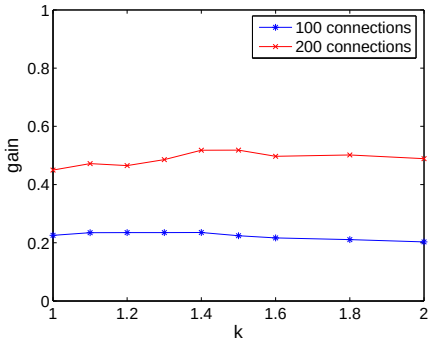
Figure 10. Performance of NetDraino.

5.3 Effects of design parameters

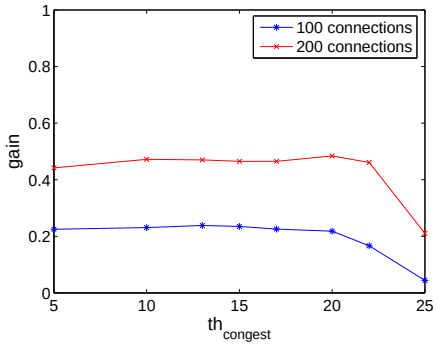
We now illustrate how the choice of *NetDraino* design parameters affects its performance. Each point in Figures 11(a), 11(b) and 11(c) represents a different simulation, with different values for k , $th_{congest}$ and p_a, respectively, where k is the scaling factor used by a server to allocate a rate limit for a congested router, and $th_{congest}$ is the threshold used by a router to decide if it is congested or not in the *NetDraino* semantics. Finally, p_a is the probability for a congested router to append its IP address to the packets destined for the servers in *CServs*. The performance of *NetDraino* was evaluated in terms of the network throughput gain while varying these configuration parameters. To get a more general understanding of the effect, each configuration was tested under two situations: (i) each client generates 100 TCP connections to S_0 (default), and (ii) it doubles the traffic to S_0.

Figure 11(a) shows that *NetDraino*'s performance does not make any significant difference when k is varied from 1.0 to 2.0. This is surprising because we expected a lower gain with a small k around 1.0 as well as with a large k around 2.0. The low gain with a small k may be the result of 'false positives' where the filters distributed by S_0 over-limit the rate of traffic toward S_0, causing the underutilization of S_0. The low gain with a large k may result from the opposite situation; the distributed rules are too generous and too much of excess traffic is dropped at S_0, wasting network resources. However, it turned out that the over-limiting rules distributed under a small k allow more traffic delivered to other servers than S_0, compensating for the underutilization of S_0. When a large k was used, the generous rules at the congested routers did drop some packets bound for servers other than S_0 in order to deliver the packets that S_0 would drop, but its effect was insignificant because the increased rate limit by the generous rules was insignificant compared to the entire traffic.

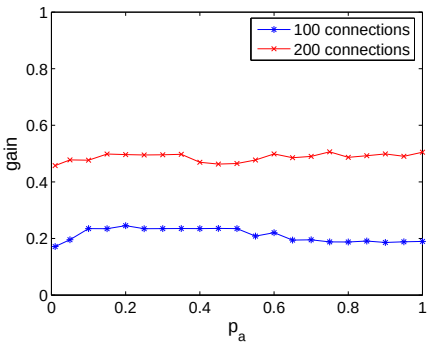
The results in Figure 11(b) are related to the responsiveness of *NetDraino*; the smaller $th_{congest}$, the earlier a router begins to set the RCN bit of its in-transit packets, and therefore, the better responsiveness. Figure 11(b) shows that there was no significant difference in the throughput gain when $th_{congest}$ ranges from 5 to 20, but it decreased if $th_{congest}$ was configured to be over 20. The burstiness



(a) Effect of k, the scaling factor



(b) Effect of $th_{congest}$



(c) Effect of p_a

Figure 11. Effects of design parameters.

of traffic yielded similar responsiveness to the case of $th_{congest}$ values ranging from 5 to 20, while the effects of slow responsiveness appeared in the simulations with $th_{congest}$ configured to be over 20. Figure 11(c) shows the effect of p_a on *NetDraino*'s performance. p_a is closely related to *NetDraino*'s overhead, because it is the most expensive in the mechanism for a router to append its IP address to in-transit packets. A smaller p_a is needed to lower the overhead, but it may result in erroneous statistics to be maintained by servers. Fortunately, the simulation results show that a p_a value as small as 0.1 suffices for *NetDraino* to perform well.

6. IMPLEMENTATION

The router- and server-side *NetDraino* prototypes are implemented as extension modules for the Linux 2.4 kernel's *iproute2* framework and *iptables* firewalling system, respectively. Each router or server participating in the *NetDraino* mechanism loads the kernel module and runs a user-level daemon called *ndagent* which is responsible for communication between a router and a server. Specifically, *ndagent* in a server makes a TCP connection to *ndagent* in a router to distribute traffic-shaping rules. To pass information from the kernel modules to the co-hosted *ndagent* or vice versa, we chose to use the Linux Netlink socket [Dhandapani and Sundaresan

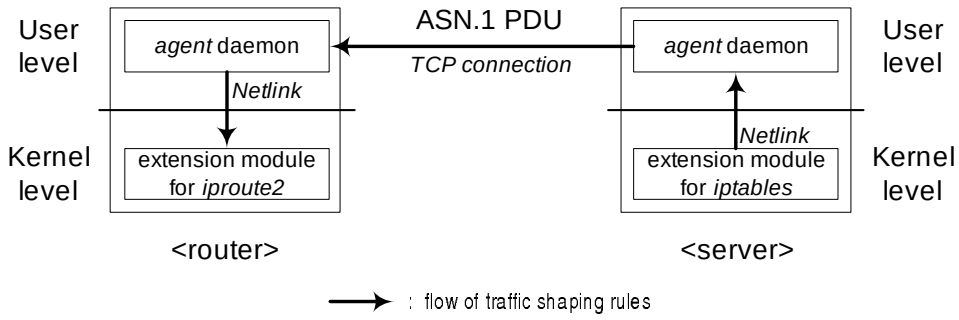


Figure 12. NetDraino architecture.

1999].

Figure 12 provides an abstract view of *NetDraino* architecture. The arrows in the figure represent the flow of traffic-shaping rules. The kernel in a server notifies the co-hosted *ndagent* of the rule tailored to the statistics it collects and maintains. Then, *ndagent* in the server sends the rule to *ndagent* in the target router over a TCP connection. We chose to use ASN.1 [CCITT 1998a] to specify the rule for its flexibility. *ndagent* receiving the rule in the router requests the kernel to regulate the incoming traffic according to the rule.

While we use flexible Linux machines for prototype implementation, we argue that an implementation of the routing extension can scale as well on an alternative architecture such as a Cisco router. For example, as mentioned in Section 1, routers with *Committed Access Rate* (CAR) feature can already rate-limit traffic aggregates defined by user supplied rules. A router's manipulation of a packet's RCN bit is also a trivial task, considering that an Internet router must already decrement the TTL and recalculate the IP header checksum of every packet it forward. To store the list of congested servers and check if a packet is destined for any of them, the routing table structure can be slightly extended to set aside a room for additional fields. But, unfortunately, for the router's exchange of traffic shaping rules with end hosts, our simple ASN.1 protocol unit may have to be replaced with a more formal protocol, which can be an extension of SNMP.

6.1 Router-side NetDraino

For the router-side *NetDraino* implementation, we added a new queueing discipline, called *ndfifo*, to Linux 2.4 kernel's *iproute2* framework. *ndfifo* extends the FIFO (drop-tail) queueing discipline module so that it can perform additional functions described in Section 4, while other queueing disciplines such as RED can also be easily extended to provide such functionalities.

Figure 13 shows the architecture of the router-side *NetDraino* prototype. When *ndfifo* is loaded, it registers to listen to NF_IP_FORWARD hook in the Linux Netfilter architecture, at which it identifies packets with SCN bit set and updates the corresponding *CServs* list. The incoming traffic is regulated by the *NetDraino* rules when it is enqueued according to the *ndfifo* queueing discipline. We implemented *ndfifo*'s dequeue function so that it can randomly append the router's IP address to

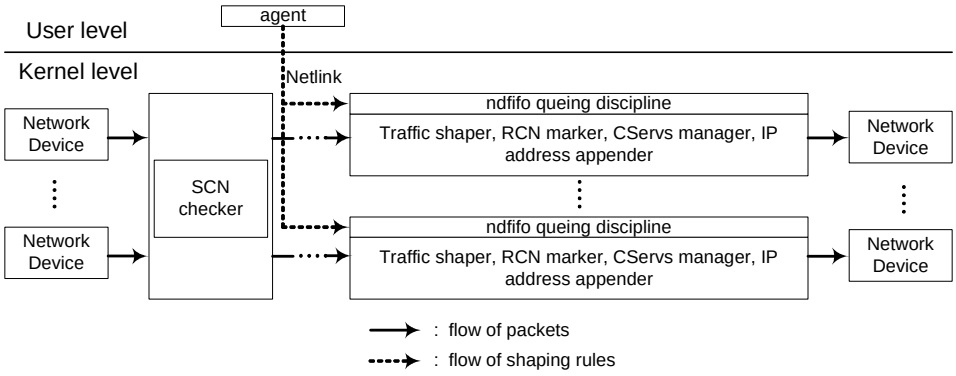


Figure 13. Router-side NetDraino prototype.

CODE(131)	LENGTH	UNUSED
FIRST ROUTER IP ADDRESS		
APPEND PROBABILITY	SEQUENCE NUMBER	
SECOND ROUTER IP ADDRESS		
APPEND PROBABILITY	SEQUENCE NUMBER	
. . .		

Figure 14. The format of new IP option.

a packet's IP option field if the packet is destined for a server in *CServs*. Figure 14 shows the format of the new IP option to deliver router's IP addresses. APPEND PROBABILITY and SEQUENCE NUMBER fields in the option are used by the destination server to maintain statistics as described in Section 6.2. The IP option can contain information of multiple routers, but the total length of IP options in IPv4 must not exceed 40 bytes. If *ndfifo* is unable to append the router's IP address due to the total length limit of IP options, it simply gives up on appending the information; it may cause inaccurate statistics at the server, but its effect would not be significant since such a situation occurs very rarely.

6.2 Server-side NetDraino

We have implemented a new match extension module, called *ndlimit*, for the Linux 2.4 kernel's *iptables* firewalling system. *ndlimit* extends the existing *limit* match extension that implements the well-known token bucket rate-control scheme, while slightly changing the original semantics. *ndlimit* assumes that the target of the match is limited to NF_ACCEPT and unmatched packets should be dropped. These assumptions are valid, considering the fact that *ndlimit* is only to regulate the incoming traffic while the original match module is intended to provide more general functionalities such as limited logging of specific rules. In addition to traffic-shaping functionality, *ndlimit* maintains the statistics of excess traffic and handles the SCN-

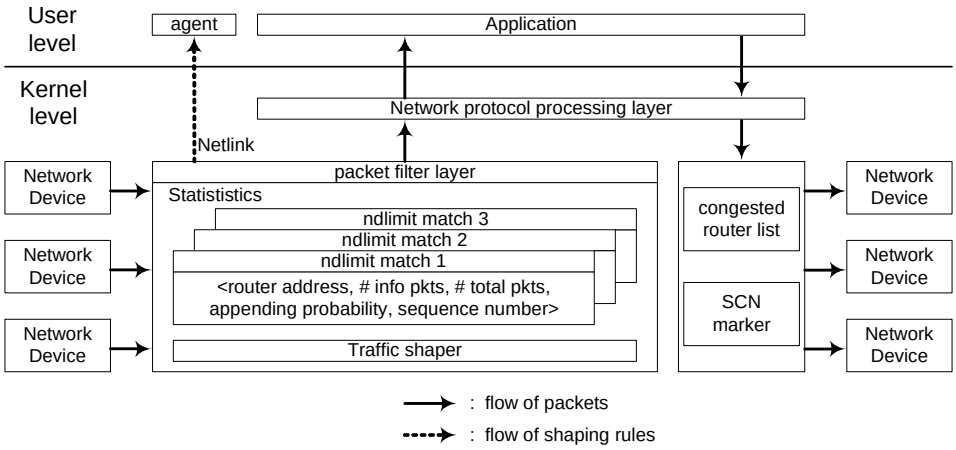


Figure 15. Server-side NetDraino prototype.

bit marking of packets.

Figure 15 shows the architecture of server-side *NetDraino* prototype. It identifies packets with RCN bit set at NF_IP_LOCAL_IN hook, and selectively marks the packets with SCN bit at NF_IP_POST_ROUTING hook in the protocol stack. Each *ndlimit* match rules in the server maintains a list of congested routers' statistics as shown in the figure. Every incoming packet that is fed into a *ndlimit* match increments the count of total packets in the match. If the packet contains a router's information in its IP option field, it also increments the count of *info* packets in the corresponding router's entry. By *info* packets, we simply mean the packets carrying the congested routers' IP addresses. *ndlimit* estimates the number of packets that traversed the congested router as $(\#\ info\ pkts)/(append\ probability)$. If the estimation exceeds a pre-defined threshold, it notifies the agent daemon of the collected statistics to distribute the rule to the routers. Because there is no fixed time window to collect statistics, it may take a long time to distribute the rule if its info packets arrive at a slow rate. But this is reasonable because such traffic would not be very heavy and it is not urgent to deliver the shaping rule. Sequence numbers are used to prevent erroneous statistics. Suppose that a router R_0 starts randomly appending its IP address to packets destined for S_0 at time t_0 but stops it at t_1. If R_0 restarts notifying S_0 of its IP address at t_2, S_0 must notice the absence of this information between t_1 and t_2. Otherwise, it will underestimate the number of packets traversed R_0. For this purpose, the router assigns a new sequence number whenever it starts 'shipping' its IP address, and the server resets statistics and recollects it whenever it detects the change of the sequence number.

7. EVALUATION

7.1 Experimental Setup

To validate *NetDraino* in a more realistic environment, we conducted experiments on a testbed consisting of 4 routers and 2 servers (1 GHz Pentium III generic PCs with 256 MB memory and 100 Mbps Ethernet adapters, hosting Linux 2.4.19)

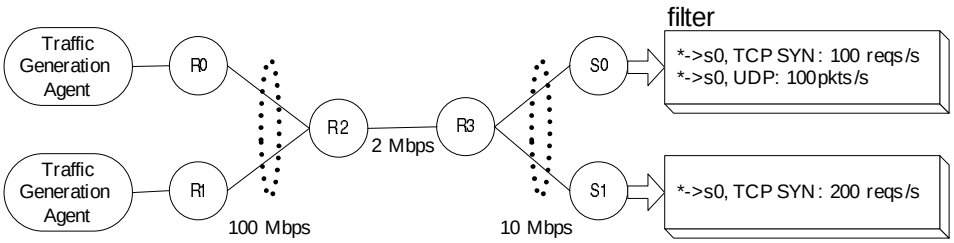


Figure 16. Network topology.

as shown in Figure 16, where R_n are routers and S_n are servers. To make the measurement easier, we lower the bandwidth of the link between the R_2 and R_3 to 2 Mbps, expecting R_2 to be easily overloaded in the presence of heavy transit traffic. We use class-based queue (CBQ) present in Linux 2.4 kernel's iproute2 framework to simulate the link with a low bandwidth, while our *ndfifo* kernel module serves as a queueing discipline of CBQ. Note that the topology consists of 3 layers. The first (leftmost) layer conceptually corresponds to WAN-to-MAN links, the second layer to MAN-to-LAN, and the last one to LAN-to-Host links. Each router is configured to store up to 50 packets in its outgoing queue.

We performed two sets of experiments, where the traffic generation agents simulate either flash crowds or DoS attacks. In both sets of experiments, high-bandwidth traffic is bound for S_0, while the other normal traffic heads for S_1. In the first set of experiments with the flash crowds traffic, two traffic generation agents attached to R_0 and R_1 make requests to S_0 and S_1; 200 requests representing the background traffic towards S_1 are generated every second, while the number of requests to S_0 is varied to account for the aggressiveness of the flash crowds traffic. In the second set of experiments, the agents generate a massive number of 1KB UDP packets destined for S_0 (as well as normal traffic), mimicking DoS attacks to S_0. Specifically, each agent generates 50 requests/sec to S_0 and S_1, while varying the rate of UDP traffic towards S_0 as specified in each experiment. TCP and UDP requests' interarrival times are simply configured to be uniformly-distributed, because the effects of using other more realistic distribution (e.g., a Poisson-distribution) are trivial when the agent generates a very high rate of requests as in our experiments.

Each server adopts a traffic shaper that polices the rates of TCP SYN and UDP traffic as shown in Figure 16. We use the Linux 2.4 kernel's *iptables* firewalling system with our *ndlimit* extension module to regulate the traffic at servers. While S_1 can accept all the incoming packets with this configuration, S_0 may drop some of them and thus simulate to be overloaded, depending on the request rates specified in each experiment.

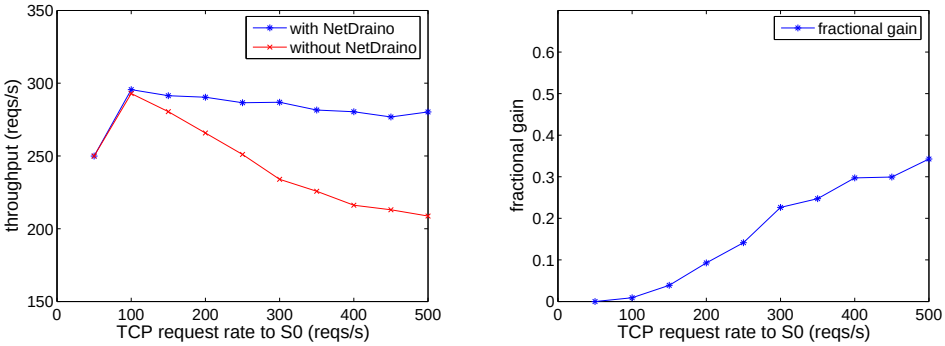
Table II shows the values of several *NetDraino* design parameters chosen in our experiments. These values are shown to be appropriate in Section 5

7.2 Basic Overhead

In *NetDraino*, appending the congested router's IP address to packets in the network and forwarding traffic-shaping rules from the congested servers may increase the network traffic, but the network bandwidth requirement for *NetDraino* is minimal.

Table II. *NetDraino* design parameters chosen in our experiments.

k	1.2
$th_{congest}$	25
p_a	0.125



(a) Total throughput (b) Throughput gain

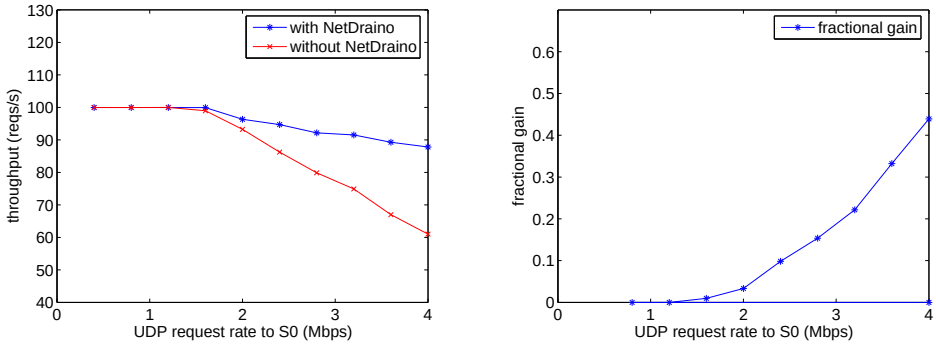
Figure 17. Performance of *NetDraino*—Flash crowds.

Under the testbed setup in Section 7.1, *NetDraino* can add at most 0.4% more network traffic as a result of appending the *NetDraino*-filled IP option to packets, which takes into account the possible packet segmentation. In addition, only one or two *NetDraino* control messages — which contain the traffic-shaping rules — are generated in a minute by a congested server.

To evaluate the overhead *NetDraino* incurs to a router, we collected the CPU time spent in each kernel function at R_2 during the experiments. For this purpose, we used kernprof [SGI 2003], a set of facilities for profiling the Linux kernel. *NetDraino* consumes the router's CPU time to (1) regulate the traffic according to the installed shaping rules, and (2) process each incoming packet to randomly append its IP address. To account for the first case overhead, we varied the number of rules installed at R_2 from 1 to 100 and let R_2 check each incoming packet against all those rules. The second case overhead was measured through the experiments described in the following subsections. The experimental results obtained in both cases show that the overhead of *NetDraino* is negligible: R_2 consumes less than 1% more CPU cycles when it has installed as many as 100 shaping rules than when it has installed none, and it consumes less than 2% more CPU cycles to randomly append its IP address to packets than when it does not use *NetDraino*.

7.3 Performance of *NetDraino*

NetDraino is designed to reduce network congestion by selective packet drops. A congested router will reduce its congestion intelligently to maximize network throughput. To validate the throughput gain by *NetDraino*, we performed two sets of experiments as described in Section 7.1, with and without *NetDraino*. As was done in Section 5.2, the throughput was measured by increasing TCP or UDP



(a) Total throughput (b) Throughput gain

Figure 18. Performance of *NetDraino*—DoS attacks.

request rates towards S_0 by each traffic generation agent. In the first set of experiments with TCP only traffic, the total TCP request rate to S_0 was varied from 50 to 500 reqs/s in increments of 50 reqs/s. In the second set of experiments with UDP traffic overloading the links, the total UDP sending rate was varied from 0.4 Mbps to 4 Mbps in increments of 0.4 Mbps. In each experiment, throughput was measured over ten 120-second intervals and then averaged. Figures 17 and 18 show the measurement results that compare the throughputs with and without *NetDraino* in the experiment sets simulating either flash crowds or DoS attacks. The throughput was represented by the average number of TCP requests accepted by the servers every second. Figures 17(a) and 18(a) plot the total throughputs of S_0 and S_1, while Figures 17(b) and 18(b) summarize the experiments by showing the fractional throughput gain of *NetDraino*.

They show the results similar to those obtained from the simulation in Section 5.2, demonstrating *NetDraino*'s ability of improving the network throughput. In the first scenario, S_0 achieves the same throughput with and without *NetDraino*, limited by the accepting rate of S_0 (100 reqs/s), while S_1 enjoys the most benefit from *NetDraino* as expected. When there are 400 reqs/s towards S_0, the network throughput with *NetDraino* is 30% higher than that without *NetDraino*. In the second scenario, both S_0 and S_1 get the same benefit from *NetDraino*. *NetDraino* throttles UDP traffic at R_2, saving the link bandwidth between R_2 and R_3 for Web traffic. In the presence of 4 Mbps UDP traffic, the servers with *NetDraino* could accept 45% more requests than those without *NetDraino*. Though we have not performed additional experiments with more aggressive traffic towards S_0, we expect that as the traffic to S_0 gets heavier, the network throughput without *NetDraino* continues to drop down to a certain point (the acceptance rate of S_0 in the first experiments set, and 0 in the second case), while that with *NetDraino* remains unchanged. One interesting result is that the total throughput with *NetDraino* also drops when the traffic toward S_0 increases. Because *NetDraino* incurs little overhead to routers, the responsiveness of *NetDraino* accounts for such a drop in the total throughput. With ideal responsiveness, traffic-shaping rules at S_0 should be forwarded to R_2 as soon as R_2 becomes congested, and should be enforced at R_2 as

long as such congestion remains. But due to the distributed nature of *NetDraino*, R_2 is left unprotected from congestion for a short time, resulting in a slight service degradation at servers.

Note that S_0 may also save its resource used to handle the incoming packets, because *NetDraino* drops packets before they reach the servers. To evaluate the resource savings at S_0, we collected the kernel profiles of S_0 during the experiments. The results show that *NetDraino* saves S_0's CPU cycles used in the network kernel, but the savings is less than 1%. This small savings may be due to the fact that we limit the link bandwidth between R_2 and R_3 to 2 Mbps, and S_0 has only a single network interface accepting the traffic. If S_0 is configured to receive a massive amount of traffic, *NetDraino* may save much more CPU cycles of S_0.

8. CONCLUSION

We have proposed, implemented and evaluated *NetDraino*, a new mechanism that links servers' traffic-regulation schemes to routers' queue management mechanisms, in order to maximize the network throughput. Specifically, *NetDraino* distributes servers' traffic-shaping rules to, and enforces them on, congested routers. It largely consists of four steps. First, a congested router identifies congested servers, and conversely, a congested server maintains a list of congested routers. Second, the congested router broadcasts its IP address using the *NetDraino*-filled IP option, while the congested server maintains the statistics showing the traffic rate from each congested router. Third, the congested server tailors its traffic-shaping rules to the target router based on the statistics, and directly distributes them to the router. Finally, each congested router can enforce the rules distributed to itself.

Our simulation and experimentation results have shown that *NetDraino* significantly improves the overall network throughput with minimal overhead. This benefit can be provided to all flows sharing the congested link, without burdening any specific hosts. More importantly, *NetDraino* can be easily deployed in, and is scalable to, a large network, because only those congested servers and routers need to classify packets, police traffic flows, and communicate with each other.

REFERENCES

Almesberger, W. 1998. Linux traffic control — implementation overview. Tech. Rep. SSC/1998/037, EPFL. November.

Andreasson, O. 2001. Iptables tutorial. http://iptables-tutorial.frozentux.net/iptables-tutorial.html.

CCITT. 1998. Recommendation x.208: Specification of abstract syntax notation one (asn.1).

CCITT. 1998. Recommendation x.25: Interface between dte and dce for terminal operating in the packet mode on public data networks.

Demers, A., Keshav, S., and Shenker, S. 1989. Analysis and simulation of a fair queueing algorithm. In *Proceedings of ACM SIGCOMM*.

Dhandapani, G. and Sundaresan, A. 1999. Netlink sockets - overview. http://qos.ittc.ukans.edu/ netlink/html/.

Feng, W., Kandlur, D. D., Saha, D., and Shin, K. G. 1997. Adaptive packet marking for providing differentiated services in the internet. Technical Report CSE-TR-347-97, University of Michigan.

Feng, W., Kandlur, D. D., Saha, D., and Shin, K. G. 1999. A self-configuring red gateway. In *Proceedings of INFOCOM*. Vol. 3. 1320–1328.

Feng, W., Shin, K. G., Kandlur, D., and Saha, D. 2002. The BLUE active queue management algorithms. *IEEE/ACM Transactions on Networking 10,* 4 (August), 513–528.

Floyd, S. 1994. Tcp and explicit congestion notification. *ACM Computer Communication Review 24,* 5, 10–23.

Floyd, S. and Jacobson, V. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking 1,* 4, 397–413.

Floyd, S. and Jacobson, V. 1995. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking 3,* 4, 365–386.

Inc., C. 1998. Committed access rate white paper. http://www.cisco.com/univercd/cc/td/doc/product/software/ios111/cc111/car.htm.

Inc., E. N. 2002. Policy-based qos. http://www.extremenetworks.com/ libraries/whitepapers/technology/ Policy-basedV5.asp.

Ioannidis, J. and Bellovin, S. M. 2002. Implementing pushback: Router-based defense against ddos attacks. In *Proceedings of Network and Distributed System Security Symposium.*

Iyer, R., Tewari, V., and Kant, K. 2000. Overload control mechanisms for web servers. In *Workshop on Performance and QoS of Next Generation Network.*

Jamjoom, H. and Reumann, J. 2000. QGuard: Protecting internet servers from overload. Technical Report CSE-TR-427-00, University of Michigan.

Jamjoom, H. and Shin, K. G. 2003. Persistent dropping: an efficient control of traffic aggregates. In *Proceedings of ACM SIGCOMM.* 287–298.

Lakshminarayanan, K., Adkins, D., and Stoica, A. P. I. 2003. Taming ip packet flooding attacks. In *Proceedings of Workshop on Hot Topics in Networks (HotNets-II).*

Mahajan, R., Bellovin, S., Floyd, S., Vern, J., and Scott, P. 2001. Controlling high bandwidth aggregates in the network. Technical report, AT&T Center for Internet Research at ICSI (ACIRI). February.

Mogul, J. C., Rashid, R. F., and Accetta, M. J. 1987. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles.*

Ott, T. J., Lakshman, T. V., and Wong, L. H. 1999. Sred: Stabilized red. In *Proceedings of INFOCOM.* Vol. 3. 1346–1355.

Reumann, J., Jamjoom, H., and Shin, K. G. 2001. Adaptive packet filters. In *Proceedings of IEEE GLOBECOM.* Vol. 4. 2331–2335.

SGI. 2003. Kernprof. http://oss.sgi.com/projects/kernprof/.

Stoica, I., Shenker, S., and Zhang, H. 1998. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *Proceedings of ACM SIGCOMM.* 118–130.

UCB/LBNL/VINT. 2000. Network simulator, ns. http://www.isi.edu/ nsnam/ns.

Welsh, M. and Culler, D. 2003. Adaptive overload control for busy internet servers. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems.*

Yaar, A., Perrig, A., and Song, D. 2004. Siff: A stateless internet flow filter to mitigate ddos flooding attacks. In *IEEE Symposium on Security and Privacy.*

Jin Kuk Lee received a B.S. degree in Computer Enginering from Seoul National University in 2000 and a M.S. degree in Computer Science from the University of Michigan at Ann Arbor in 2003. Hs is currently a DDS candidate in the UCLA School of Dentistry. His research interests include network traffic control/measurement and internet server design/evaluation.

Kang G. Shin is the Kevin and Nancy O'Connor Professor of Computer Science and Founding Director of the Real-Time Computing Laboratory in the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan. His current research focuses on QoS-sensitive networking and computing as well as on embedded real-time OS, middleware and applications, all with emphasis on timeliness and dependability. He has supervised the completion of 57 PhD theses, and authored/coauthored more than 660 technical papers (more than 240 of which are in archival journals) and numerous book chapters in the areas of distributed real-time computing and control, computer networking, fault-tolerant computing, and intelligent manufacturing. He has co-authored (jointly with C. M. Krishna) a textbook "Real-Time Systems," McGraw Hill, 1997. He has received a number of best paper awards, including the IEEE Communications Society William R. Bennett Prize Paper Award in 2003, the Best Paper Award from the IWQoS'03 in 2003, and an Outstanding IEEE Transactions of Automatic Control Paper Award in 1987. He has also coauthored papers with his students which received the Best Student Paper Awards from the 1996 IEEE Real-Time Technology and Application Symposium, and the 2000 UNSENIX Technical Conference. He has also received several institutional awards, including the Research Excellence Award in 1989, Outstanding Achievement Award in 1999, Service Excellence Award in 2000, Distinguished Faculty Achievement Award in 2001, and Stephen Attwood Award in 2004 from The University of Michigan; a Distinguished Alumni Award of the College of Engineering, Seoul National University in 2002; 2003 IEEE RTC Technical Achievement Award; and 2006 Ho-Am Prize in Engineering. He received the B.S. degree in Electronics Engineering from Seoul National University, Seoul, Korea in 1970, and both the M.S. and Ph.D degrees in Electrical Engineering from Cornell University, Ithaca, New York in 1976 and 1978, respectively. From 1978 to 1982 he was on the faculty of Rensselaer Polytechnic Institute, Troy, New York. He has held visiting positions at the U.S. Airforce Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division within the Department of Electrical Engineering and Computer Science at UC Berkeley, and International Computer Science Institute, Berkeley, CA, IBM T. J. Watson Research Center, Software Engineering Institute at Carnegie Mellon University, and HP Research Laboratories. He also chaired the Computer Science and Engineering Division, EECS Department, The University of Michigan for three years beginning January 1991. He is Fellow of IEEE and ACM, and member of the Korean Academy of Engineering, is serving as the General Chair for the 3rd ACM/USENIX International Conference on Mobile Systems, Applications, and Services (MobiSys'05), was the General Chair of the 2000 IEEE Real-Time Technology and Applications Symposium, the Program Chair of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chair of the 1987 RTSS, the Guest Editor of the 1987 August special issue of IEEE Transactions on Computers on Real-Time Systems, a Program Co-Chair for the 1992 International Conference on Parallel Processing, and served numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991-93, was a Distinguished Visitor of the Computer Society of the IEEE, an Editor of IEEE Trans. on Parallel and Distributed Computing, and an Area Editor of International Journal of Time-Critical Computing Systems, Computer Networks, and ACM Transactions on Embedded Systems.