Operating System Support for Network Control

a virtual network interface approach for end-host OSs

Takashi Okumura
WIDE Project
taka@wide.ad.jp

Daniel Mossé
University of Pittsburgh
mosse@cs.pitt.edu

Masaki Minami
Keio University
minami@sfc.wide.ad.jp

Osamu Nakamura
Keio University
osamu@sfc.wide.ad.jp

***Abstract*—Because of user demands for better quality of service, network-aware applications have been of increasing necessity. To enable more control, the end-host operating system (OS) is the entity responsible for providing appropriate service level and API to user applications. However, most of the work in this area remains domain-specific and without a generalizable scheme for providing network control as an OS service.**

In this paper, we propose an OS service, namely the virtualization of network interface, that lies between network interface and userland. The virtual network interface is hierarchically attachable to various OS-supported entity, such as threads, processes, and sockets. We argue that the mechanism provides flexible control, as well as the system protection that is required for operating system services.

For a proof of the concept, we show an implementation on a PC-Unix, using the `procfs` file system abstraction. We also carried out a systematic evaluation. The system exhibited the expected control behavior, while keeping the performance small.

I. Introduction

A computer network has traditionally been a transparent service to applications. However, driven by the demand for efficient and high-performance network applications, the situation is changing toward a more "network-aware" paradigm.

The difficulty we confront, whenever we try to find a way for network control, is the given internetworking architecture. One option to circumvent this problem is to radically re-design the network along with the end-host operating system (OS). This approach, taken by the efforts in Active Networks [2], is quite radical and hard to justify for general-purpose systems, requiring extensive system-wide changes. Instead, it would be preferable that the network architecture be kept intact, and that we provide some mechanism that works on end-hosts.

To this regard, there have been several studies, with conservative working implementations. A shortcoming of the studies is their generalizability, because they are targeting specific problems, not the general design issue. Instead, we aim at general rules that aid in the design of network-aware general-purpose OSs. To the best of our knowledge, this is the first paper proposing a design of an OS service for network control that meets the requirements for a core set of OS services, namely system protection, flexible control granularity, various types of control, and reasonable abstraction, aggregation when necessary, and proper API.

T.Okumura. is a student at the School of Medicine, Asahikawa Medical College, Asahikawa, Hokkaido pref., Japan (email: taka@wide.ad.jp). This work was supported by Telecommunications Advancement Organization of Japan, Grant No. JGN-P122518.

The rest of this paper is organized as follows. First, we clarify the problem of OS support for network control, and propose the concept of *virtual network interface* (Section II). Then, we examine the virtual network interface concept so that it works for end-host oriented network control (Section III). After the conceptual work, we present our prototype implementation on FreeBSD, and show performance profiles to validate the model (Section IV). Lastly, we give a brief review of related work (Section V) and conclude the paper (Section VI).

II. Operating System Support for Network Control

A. Fundamental mismatch in perspectives

The operating system research has been working towards proper abstraction of various system resources. Hence, to address the problem of operating system support for network control, the simplest approach is to abstract the network communication, and provide control of such abstraction. However, we contend that this straightforward approach has fundamental shortcomings. We begin the discussion by presenting two antithetical perspectives: the network perspective and the OS perspective.

a) Network Perspective: One way to control network traffic is to create a network abstraction on a per-flow basis and provide services (signaling, API, etc) that work directly on such abstraction. For example, we may extend the socket abstraction so that users can set bandwidth limitations. Admittedly, the low-level control over each connection is useful for some applications, but, as we illustrate below, there is a great room for improvement.

First, such an interface can *easily contradict the resource management semantics of the operating system.* For example, a network-intensive low-priority process can easily starve higher priority processes that require network I/O. This is a network *priority inversion* situation, and it conflicts with resource management semantics of the OS. Clearly, *system protection* is needed for the system to be safe.

Second, *the low-level abstraction is not scalable*. Suppose that we have 1000 concurrent connections with intermittent traffic. It is quite inefficient to designate fixed network resources to each of them. In this case, proper traffic aggregation can greatly improve the resource utilization, as well as the

quality of each session. Likewise, if we have 1000 successive connections of short length, we may want to assign an aggregate flow specification for all of them. Clearly, some aggregation is needed in this case.

Flow classification by network address and/or protocol type is a possible solution in that it provides reasonable aggregation of the traffic. However, although this provides scalable solution, it again violates the system protection.

Third, *a low-level abstraction scheme presupposes knowledge of independent connections* that is not always true. For instance, if we are to control output of a web browser, we need to know how many connections it makes and when. To address the problem, we need another system service to properly locate the connections they make, and monitor the usage. This would probably increase the design complexity of the system, and cause extra overhead and latency.

Lastly, in the area of network resource management, *the service model is still an open problem*. For example, we might have service models such as INTSERV and DIFFSERV, but there are also models for authentication and accounting. The most probable scenario is coexistence of various models, in which each administrative domain chooses its own models and services to offer, based upon its own needs and resource constraints. It is clear that we need a proper abstraction of the underlying mechanism and policies for portability of the control program, otherwise, we would need to hardcode everything, and the portability would be severely compromised.

It can be seen that, although the low-level abstraction of flows provides fine-grain control, it is not always the most desirable solution. Consequently, the network perspective cannot guide the design of the operating system service.

b) Operating System Perspective: On the other extreme, the operating system might provide a higher abstraction. For example, we might abstract an end-to-end flow as a file under `/dev/network/`, with signaling API for end-to-end network control.

Although this sounds promising, it is not as simple as it may seem at a cursory examination. The fundamental problems of this scheme are threefold.

First is the *lack of accountability and controllability* of the network. The network interface is one of the major I/O devices of a modern computer, and shares some characteristics with other I/O devices such as hard drives and serial ports. The network differs in that most of the network devices represent remote resources. For example, the bandwidth guarantee might require output rate regulation at the host and path bandwidth reservation in the intermediate network. The former is controllable by the host, but the latter is typically not. This fact does not make a single and consistent abstraction by the end-host OS impossible, but clearly makes it harder to achieve.

Second, from the OS perspective, we *need a standardized abstraction and interface* (mostly for programming efficiency). However, the service model of the future network are unpredictable, and communication topology differs from protocol to protocol. For example, abstractions for unicast, multicast, and anycast differ greatly, and it would be hard to incorporate them into a single model and one interface. RTSP [15] utilizes a UDP transport for data streaming, and a TCP connection for health check of the stream. It is hard to define a single standardized abstraction and system service, a priori, to satisfy these various needs.

Third, such control requires close cooperation of kernel services and user-level applications. Let us take the example of path bandwidth reservation, again. We may employ an RSVP module at user level for end-to-end signaling, but, in some cases, we also need a kernel module to regulate the host's network I/O. This poses another question of how to separate the functionalities among kernel and user-space.

As briefly illustrated, it is quite challenging to have a satisfactory high-level abstraction and system interface, from the end-host operating system perspective.

B. Requirements for the alternative perspective

The conclusion we reach from the two perspectives presented in the preceding section is that neither high-level abstraction nor low-level abstraction would work satisfactorily as a design principle of an OS service for network control. Hence, we need another approach that should meet the following requirements, gained from the discussion above.

a) System Protection: Major difficulty of the former studies ([3], [14], [4]) is lack of system protection model. Consequently, the priority inversion problem still exists even for the systems with network control support. To address the problem, we need *a system protection model with inheritance*, for the control model. For example, once we set a limit on the resource usage of a process, all its (and its children's) network I/O should not exceed that limit, as we illustrated in our first `netnice` study [12].

b) Flexible Control Granularity: We stated that fine grain control that works directly on a flow is useful, while aggregation of flows (coarse grain control) is indispensable. This suggests that the operating system service should provide flexible control granularity so that it works on a flow, as well as on a set of flows (this point was a major limitation of [12]).

c) Generalizability: The approach, as an operating system service, should be generalizable. To this end, it should provide a topology-free general infrastructure, by focusing on the end-host control.

C. Virtual Network Interface Concept

To meet the requirements, we propose a novel system abstraction, *virtual network interface*. Figure 1 gives a conceptual overview of the approach. A physical machine is shown with a physical network interface (cylinder at bottom). Circles denote execution entities (such as processes and threads), and arrows are data flows. A rounded box wrapping the circles is a virtual machine (VM), an illusion of possessing their own machine by

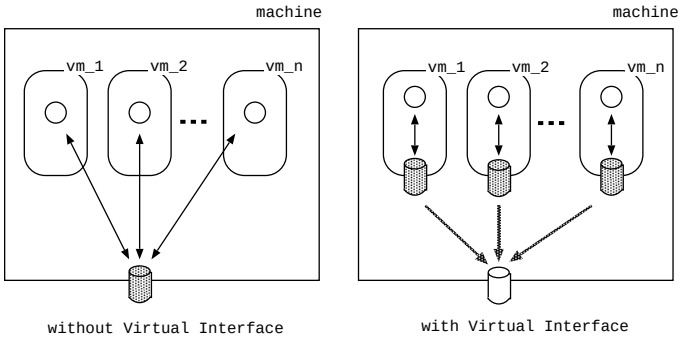


Fig. 1. Conceptual overview of the mechanism.

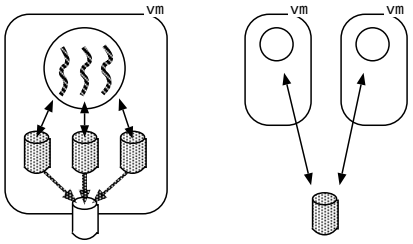


Fig. 2. Hierarchical structuring of the virtual interfaces.

the execution entities.

In a legacy setting (left part of the figure), every process[1] shares a physical network interface (a shadowed cylinder). The network is transparent to the processes, and the best-effort principle governs every I/O activity. Unfortunately, this model has serious drawbacks: there is no control of the network I/O and a network priority inversion problem might occur at any moment.

These shortcomings are due to the lack of a proper abstraction and protection for the real network interface. We believe that this point is a serious flaw that compromises resource protection principle of modern operating systems.

To solve this problem, we introduce virtual network interfaces (the right part of the figure). Each process now possesses its own virtual interface (shadowed cylinders). Processes are capable of configuring flow specification of its virtual interface (within the limits of resource constraints, of course). This way, each process has complete illusion of possessing their own machine, and thus, the system protection between processes is naturally provided.

Next, we extend the concept to meet the requirement of flexible control granularity. A natural solution here is to allow processes to create child interfaces connected to their original one.

On the left of Figure 2, we see three threads (wavy lines in a process circle) having dedicated interfaces, thereby partitioning the resource of the common interface lying on the VM boundary. Note that this solution also holds for sockets, and other OS-supported management units. Since one greedy socket or thread can starve others, we need to allocate virtual interfaces to the smaller units for protection and finer control.

[1] For simplicity of presentation, we will speak of "processes" instead of speaking of "OS-supported entities" in the rest of this paper.

It is easy to extend the idea for larger resource units than processes, such as process groups (right part of Figure 2). Suppose that two processes are cooperating toward a common task. It is natural that we allocate a single virtual interface for both, and give certain amount of resources to it, for the particular job.

The operating system service of virtualizing network interfaces proposed has following properties.

First, the system can provide system protection, while users enjoy flexible control over network I/O, within the limit of allocated resource. Note that cascade of the interfaces provides, simultaneously, hierarchical resource protection and simple solution for flexible control granularity. Second, this local mechanism works independent of communication topology, since the abstraction is limited to local resource. Third, this strategy is backwards compatible, and existing programs can benefit from the proposed mechanism without modification (that is, *retrofitting* is easy).

III. Controlling Network by Virtual Network Interfaces

From the discussion above, we now assume that processes are capable of hierarchically structuring the virtual network interfaces (details about how to consistently structure the interfaces and attach them to processes, threads and sockets is discussed in Section IV), and turn to the configuration issue of the virtual interfaces.

The goal of this section is to clarify the possible control by the mechanism, and extend the limit as much as possible. To this end, first we examine the characteristics of the approach, and then, present several examples illustrating the control.

A. Characteristics of the approach

a) End-host approach in network control: Traffic control by end-nodes has advantage and disadvantage, compared to intermediate node control.

The most notable advantage is stateful nature of end-point control. Consider a server for Electronic Commerce, where customers who have started to choose items have a higher priority to customer than those at the check-out. However, since the server processes know the state of each transaction, they can significantly reduce the processing overhead by utilizing their local information. This is not true of an approach that maintains state in intermediate nodes in the network. With the intermediate node approach, attempting to implement this policy requires extensive inspection of every flow and semantic knowledge of difference types of payload.

A second advantage is traffic aggregation. For example, RTSP [15] utilizes a UDP transport for data streaming, and a TCP connection for health check of the stream. Because the intermediate nodes would not recognize different flows as being aggregated or bundled, only the originator of these flows would have enough information to perform proper control over the session.

Lastly, since congestion avoidance is more efficient than congestion control, in the sense that the former can save wasted

resource consumption during congesting period, traffic sources can provide efficient solution for the problem.

The main disadvantage of the end-host approach is the resource brokerage between the flow originating from the host and other unrelated traffic. Although this is a severe limitation, this does not degrade the advantage of the scheme stated above.

b) Virtual interface as a network control primitive: The management of network-related resources in the end host, achievable through virtual interface, provides only passive control. Consequently, one might claim that the OS should play more active role in the control. However, we doubt that active control by end-host OS is justifiable and technically possible.

Network control by end-host OSs has little utility, as long as the network architecture does not support network-wide QoS. Even with an assumption that the network provides such guarantees and controller interfaces, it is hard to devise a comprehensive mechanism that can accommodate all needs at the kernel level.

System design principles suggest that core functionality meets the most common needs, leaving others to extension mechanisms. Hence, the passive control that can accommodate every network activities and works regardless of policy and network architecture is properly justifiable as a system service. On the other hand, mechanism for active control differs for each control goal and hard to generalize. Hence, it is much more natural to use an extension mechanism, such as user-space libraries or kernel expansion modules, to meet any specific requests for active network control.

c) Resource management by virtual interfaces: The virtual network interface mechanism provides interface for managing network resource of the host. In this regard, the system users has two contradictory goals; better resource partitioning and better resource utilization. Naturally, it is desirable that the virtual interface supports both.

For better partitioning of the network resource, the system needs non work-conserving scheduling, in which the resource may be idle even if there are packets to send (e.g., absolute bandwidth limitations). The non work-conserving scheduler can guarantee specified output rate, independent from activities of other traffic. This provides stable and predictable network performance to processes, and better separation of resources.

Its main shortcoming is resource utilization, in that it may waste or overutilize the resource, depending on the allocation policy. In a setting that requires high resource utilization, a work-conserving scheduler must be employed, since it does not allow an idle network interface. Work-conserving scheduling, such as weighted fair queuing [5] and priority queuing, provides high utilization of the network resources.

Consequently, the virtual interfaces have to support both types of scheduling, in addition to the hierarchical behavior.

B. Examples

Based on the discussion above, we show below how network control is possible with the virtual interface approach.

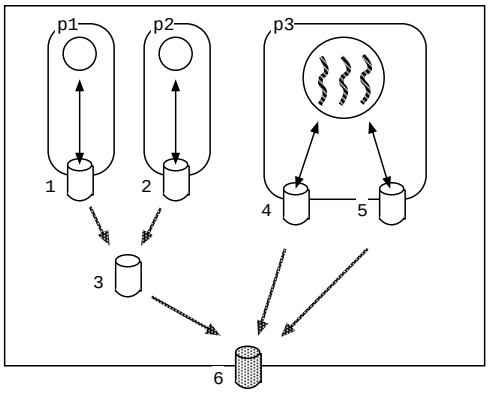


Fig. 3. Configuration sample of a virtual interface structure

a) Client use: Let us take an example of a virtual network interface (VIF) based control for a typical client machine, shown in Figure 3. On the left, we have two daemon processes (p1 and p2) for a network file system access. Suppose that they are mounting a file server on local network, and have bursty traffic pattern. On the right is a client process for a web browser, accommodating several threads, one of which is for a streaming application. Note that bursty traffic by the network file system occasionally saturates the real network interface, and starves the persistent streaming flow. Our goal is to give proper protection to the processes, and provide better quality of service for the system user. To this end, we show below how to configure the VIFs.

First, we want to reserve a certain amount of bandwidth for the streaming application. This is achieved by limiting the file system traffic, and thus, we configure flow specification of VIF 3 so that the bursty traffic is bounded by some predefined bandwidth. Next, even though we want to limit the total resource usage by the file server processes, at the same time, we want to give fair shares to them for better utilization of the limited bandwidth. A reasonable solution here is that we give 50% of VIF 3 to VIF 1 and 50% to VIF 2.

VIF 4 and VIF 5 are for the web browsing. Let us suppose that VIF 4 is for the streaming, and VIF 5 for TCP connections. Again, to assure the streaming quality, it would be better to limit the bandwidth usage of VIF 5, leaving room for VIF 4 to receive time sensitive data. Or, we may give higher priority to VIF 4.

b) Server use: Next, to further illustrate the network control by virtual network interfaces, we give another example for server application, with a QoS manager. We show a scenario in which a constant bit rate service is given for a web server. Figure 4 shows a conceptual overview of the control. A box at the top of the figure represents the QoS manager, named *netnice daemon*, or `netniced`, controlling the VIF and communicating with peer systems on the network segment.

In the configuration, there are three basic VIF types: root, http, and stream. Root VIF on each host is configured as a traffic shaper (that is, using a non work-conserving approach), and configured to have total bandwidth limit allowed to the host.

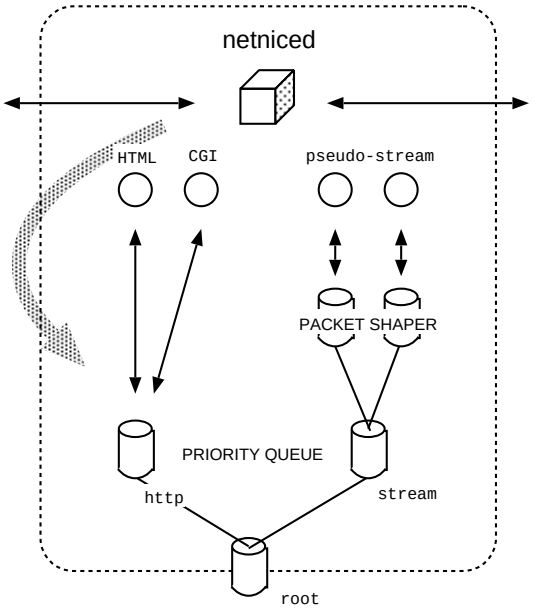


Fig. 4. Constant Bit Rate Stream.

Http is a VIF for HTML files and CGI programs. Stream is a VIF for streaming processes. These VIFs are configured as priority queues, and we gave higher priority to stream VIF. In addition, to limit the upper bound of the bandwidth usage, we configured the system to automatically wrap the streaming processes with a dedicated VIF in a traffic shaper mode. This way, streaming processes generate constant bit rate flow.

The `netniced` manager first generates the basic VIFs at startup, and monitors the execution of each process. When the manager finds a process streaming a continuous media data, but connected to a http VIF, it wraps the process with a dedicated VIF for traffic shaping, and connect it to stream VIF. If the manager found normal process is connected to stream VIF, it returns the process back to http VIF, deleting unnecessary traffic shaper.

For further description of our approach, in this paper we focus on the client application, leaving the details of how to implement and integrate the server applications for the future work.

IV. Implementation and Profiling

We have been discussing the model of the virtual interfaces. Contrary to the simplicity of the abstraction, the implementation of this concept is quite challenging, since the it requires flexible hierarchical packet scheduler. In addition, we need reasonable abstraction with proper API for its control. This section addresses these issues, by showing an actual implementation of the concept we developed on FreeBSD.

The section is organized as follows. First, we describe implementation overview of the kernel module. Second, we describe our system interface, based on `procfs` file system. Third, we give performance profiles of the implemented prototype to validate the model.

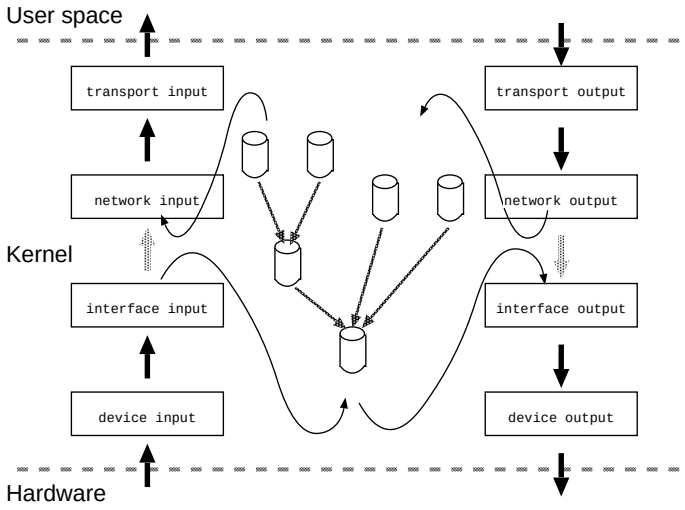


Fig. 5. Implementation Overview.

A. Kernel Module

The implementation overview is given in Figure 5. We briefly describe functional components of interest.

a) Overview: The four thin arrows in the figure denote packet entry and exit points of the virtual network interface (VIF) tree. The packet hook mechanism resides between the network layer and the interface layer.

For each outgoing packet, after the processing of the network protocol (typically, by `ip_output()`), the kernel calls a function for interface output routine, whose address is stored in `struct ifnet`. In our implementation, the addresses are overwritten as `vif_input()`, thus redirecting packets to the VIF subsystem. After proper scheduling in the VIF tree, they are again put back to the outgoing flow using the original function pointer, by `vif_output()` function.

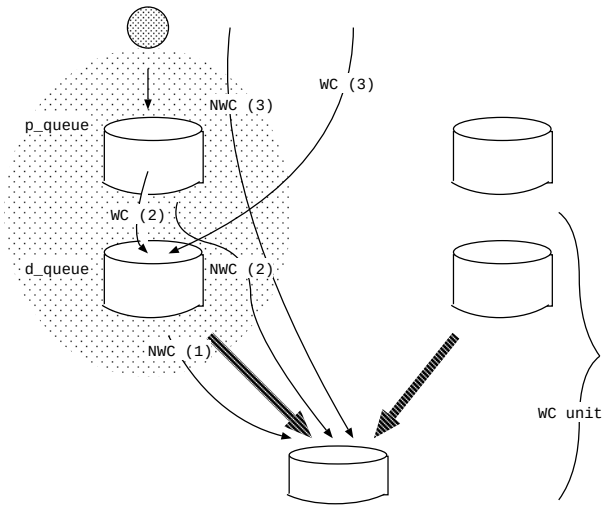
Likewise, input packets are also hooked by the function pointer stealing. In FreeBSD, when a packet arrives at a real interface, the kernel initiates software timer for further processing by `ipintr()`. We overwrite this by `vif_intr()`, to hook the input packets. Processed packets are returned to the upward processing stream, by `vif_output()`.

b) Packet Scheduling: The packet scheduling involves three components; queue, packet, and scheduler. The queue in the system is instantiated as a VIF data structure, `struct vifnet`. It includes actual queue data structures, statistics variables, and scheduling parameters. Note that we have two copies of the queue data structure, one for each direction.

A global scheduler function, `vif_sched()`, determines the rate at which VIFs will be serviced and invokes the `vif_dequeue` function for that VIF at the appropriate time.

Figure 6 shows the hierarchical scheduler algorithm. The goal of the scheduler is to enable work-conserving (WC) scheduling and non-work-conserving (NWC) scheduling in a single framework.

On the top of the figure is a closeup of VIF structure, where




```
vif_enqueue(vif, pkt)
{
   insert_tail(vif->p_queue, pkt);
   vif_sched(vif);
}

vif_dequeue(vif)
{
   switch (vif->type) {
   NWC:
      // (1) break from switch if a packet is found
      if !empty_Q(self->d_queue) break;
   WC:
      if !empty_Q(self->p_queue) break;          // (2)
      if !empty_Q(previous_vif->d_queue) break; // (3)
   default:
      return;
   }

   switch (vif->type) {
   NWC:
      if (vif->root_flg)
            vif_output(pkt);
      else
            vif_enqueue(pkt->next_vif, pkt);
      break;

   WC:
      insert_tail(vif->d_queue, pkt);
      vif_sched(pkt->next_vif);
   }

   schedule(vif);
}
```

Fig. 6. Algorithm of the VIF drainage.

two VIFs have a common parent VIF. Note that a VIF actually comprises two parts; `p_queue` (proximal queue) and `d_queue` (distal queue). The VIFs work as a unit in a scheduling process.

A NWC VIF simply enqueues the packets into `p_queue` of the next VIF at the right time. A WC VIF puts packets to `d_queue` of the VIF when scheduled, and wakes up the next VIF for drainage. The drainage will be done by the next VIF in weighted fair queueing, or in priority queueing. The arrows in the figure show the enqueue and drainage relationship among VIFs, which can be labeled WC and NWC . A number accompanying the labels corresponds to the packet dequeue operation of the scheduler algorithm, written in C-like pseudo-code. We see the two scheduling paradigms being used concurrently, by employing different drainage strategies for WC VIF and NWC VIF, as explained below.

This way we can have a structure of VIFs that works in work-conserving and non work-conserving manner. Note that, although not shown, there are two global schedulers on the system, one for incoming and the other for outgoing traffic.

c) Packet Label: Since packet classification can be a bottleneck of the queuing control, we used a *flow labeling* mechanism. This mechanism is used also for the *routing* of the packets in the VIF tree, as follows.

We changed the process and socket structures to have pointers to attached VIF data structures, `struct vifnet`. We copy the pointer onto each outgoing packet. Then, at the network output into the VIF system, the kernel traverses the VIF tree from the associated VIF down to the root VIF, and creates a list of VIFs on the path. The list is attached to each packet, which is, in turn, enqueued into corresponding VIF. At scheduling time, the packet scheduler simply checks the list for next VIF, without any detailed inspection of the packet for classification.

Incoming packets are handled differently. Since they must be put into the root VIF first, and traverse the VIF tree upward, we need to route the packets properly in the VIF tree to reach the leaf VIF which corresponds to the destination socket. Exploring the entire VIF tree would yield a severe performance penalty. To circumvent this, we utilize *early demultiplexing*. When the packets emerge from the interface input routine, we lookup the protocol control block table, and find the destination socket first. We can easily find the VIF, associated to the particular destination socket, using a pointer for its attached VIF. Then, we get the VIF path from the socket down to the root VIF, and, by reversing the order, we finally get the VIF list on the path from the interface upward to the target socket. Note that the VIF attached to sockets must be a leaf VIF.

Regarding the packet data structure, we slightly modified `mbuf`, so that the scheduler can easily reference the packet data by a pointer, `struct vif_pkt *`, for the processing in the subsystem.

B. System Interface: File System Abstraction

As a system interface, we propose *file system abstraction*. We implemented this interface by extending commonly supported `procfs`. Abstraction strategy and operation model are shown below.

To illustrate the extended `procfs` mechanism, we show in Figure 7 a sample directory representation for the VIF configuration shown in Figure 3. A directory under `/proc/network` represents a VIF. As shown, the VIF structure is perfectly represented in the directory structure. `fxp0` is the name for a real network interface on the machine, marked as interface 6 in Figure 3. Files under a VIF are configuration parameters for the virtual interface (for clarity, only some of them are shown in the figure). It is easy to see that a directory naturally represents a VIF, in Object-Oriented manner.

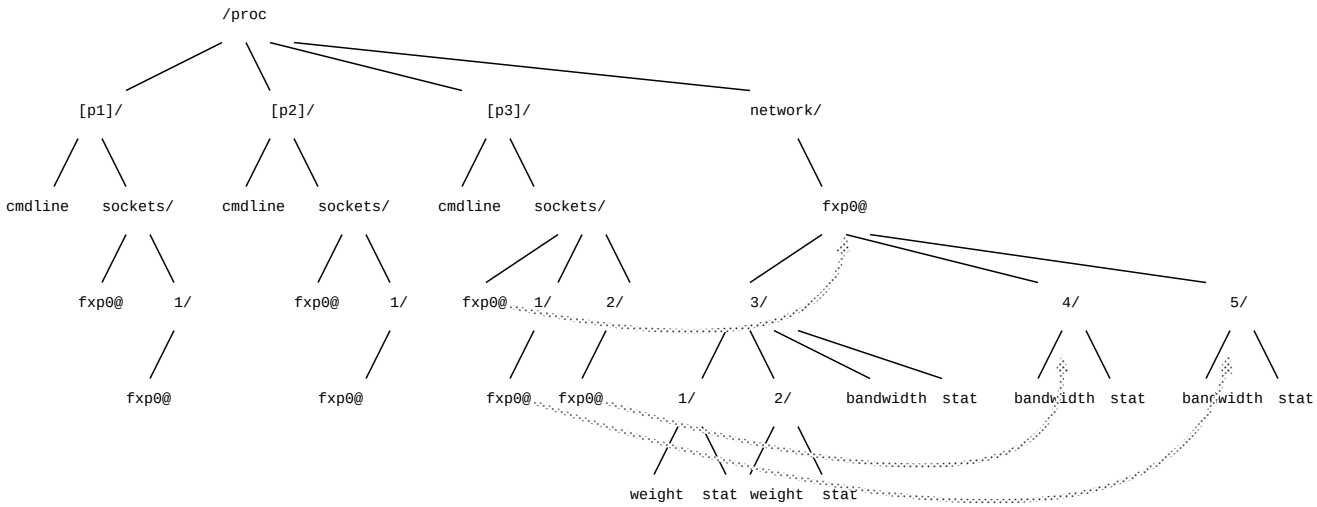


Fig. 7. Directory structure for sample VIF configuration in Figure 3.

Directories under `/proc/[pid]` are process entries. As shown, process 1 (p1) and process 2 (p2) have one socket each, and process 3 (p3) has two sockets. Directory names for the sockets correspond to the file descriptors they are assigned to.

Under the socket directories are links (indicated by @ mark in Unix convention), named `fxp0`. The links denote attachment of the VIFs to the particular resources, such as processes and sockets. For example, we can see that p3 has its network I/O capped by a VIF represented as `/proc/network/fxp0`, and two sockets of p3 by VIFs 4 and 5, respectively.

Association of a VIF and a process (or a socket) is represented by a link, with the same name as the real interface (in this case, `fxp0`). This policy is chosen because we may have several network interfaces. Suppose a machine has two interfaces (`fxp0` and `fxp1`). A process on the machine might want to use `fxp0` heavily, but might need moderate usage of `fxp1`. In this case, we will need separate VIF structures for the two interfaces, and, need two independent VIF associations. It is natural to designate the name of the real interface to discriminate each independent VIF structure.

To sum up, in our system, each process entry possesses a `sockets` subdirectory. Under that directory are subdirectories for each socket, and software links which are pointing their respective target VIFs.

Regarding the operational means, we use normal file management operations, such as directory creation and soft linking. For example, to *create* a VIF, `mkdir` command (or system call, to be exact) is used under `/proc/network` directory. Likewise, `ln` command is used to *attach* a VIF to a process or a socket. For example, an operation, `ln -s /proc/network/fxp0/5` at `/proc/[p3]/sockets/2/` will attach the VIF 5 to the second socket of process 3. Detachment and removal of VIF is possible with `rm` and `rmdir`. To set any value to a certain parameter, just write ASCII value to the representing parameter file. To read, just read the file. As it is a part of `procfs`, all the file entries reflect the operations the user makes.

This abstraction is powerful for the following reasons. First, users (including system administrators and end-users) can easily manage the virtual interfaces with the file management operations that comply with syntax and semantics of the existing system. Second, protection model is simply realized by file permission semantics. Third, the hierarchy of the virtual interfaces is intuitively represented. Fourth, the abstraction is independent of the underlying mechanism for network control. That is, we may have various queuing disciplines without modifying the user interface. Fifth, it allows attachment of VIFs to various resource unit.

Disadvantage of the abstraction is its operational complexity for system users. To compensate this problem, we also provide simple control commands: `netnice`, and `vifctl`.

C. Performance Profiling

In the section, we evaluate the performance of the implementation. First, we show traffic pattern generated by the system. This is followed by a profile of the packet scheduler. Lastly, we show statistics of control overhead.

The hardware specification used in the experiments is: Processor: Intel Celeron processor (733Mhz), Memory: 64MB, Hard-drive: Fujitsu MPG3204AT (20GB), Network interface: 3Com EtherLink XL 905B 10/100, Motherboard: A-Open MX3S. The machines are connected to a fastether switch, Netgear FS516.

a) Traffic pattern: We conducted an experiment to evaluate the output generated by the implemented mechanism. In the experiment, we structured the VIFs under the real network interface, as shown in Figure 3, and configured the VIFs as follows; VIF 1 and 2 each receive 66% and 33% of VIF 3's bandwidth share, respectively. VIF 3 is limited to 256KBps, VIF4 to 512KBps, and VIF 5 to 128KBps. The root vif (VIF 6) has virtually infinite bandwidth for this experiment (configured as 100Mbps). A process on VIF 5 sends packets as a background

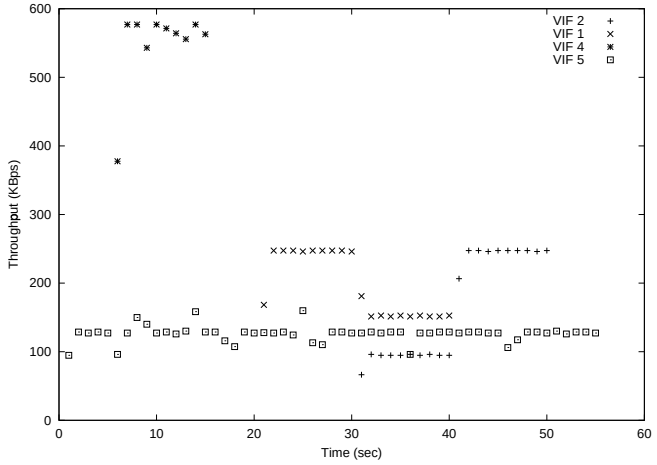


Fig. 8. Output of the VIFs.

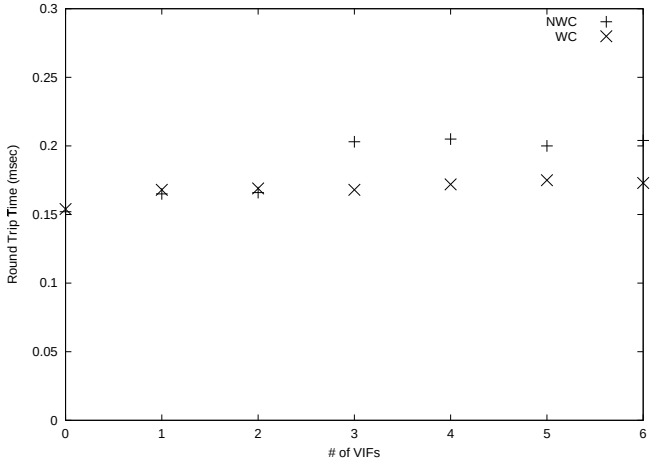


Fig. 9. VIF delay.

traffic at a constant rate, and another process on VIF 4 sends a burst data for 10 seconds at time 5. From time 20 to 40 and 30 to 50, VIF 1 and VIF 2 send packets through VIF 3. Note that all the connections are outbound, and we are showing the output regulation of the mechanism. In order to measure the traffic, we used `tcpdump` command on the destination machine, and plotted the measured throughput of the IP layer.

The result is shown in Figure 8. The results for VIF 3 and 6 are not shown since they are simply the sum of VIF 1+2 and 3+4+5, respectively. We can see that, overall, our implementation regulated the output of processes, as specified. However, there is some slight variability in the measured traffic, which is due to the weighted fair queueing scheduler.

b) Scheduler overhead: Now that we know our implementation performed as specified, next we need to measure the overhead incurred by the system modification, particularly on delay and max throughput that a VIF incurs. To this end, we serialized VIFs, and measured the delay and max throughput, changing the number and the type of VIFs.

Regarding the hardware setting, a machine with `netnice` patch is connected to a 100BaseT switch, and another machine with GENERIC kernel on the other side of the switch is used as a destination. For the measurement of delay, we measured round trip time (RTT) of `ping` packets between two hosts. 100 samples were taken for each configuration. For max throughput, we used the loopback interface (lo0), and configured each VIF to have bandwidth of 512Mbps (much bigger than the hardware capability). We used the `ftp` command to transfer a few-megabytes file, and obtained the throughput statistics from the ftp client, measured at the userland. 10 samples were taken for each configuration.

Figure 9 shows the VIF delay. The X axis is a number of VIF in the line (at VIF 0, we just used the root VIF attached to the real interface), and the Y axis is a minimum RTT in milliseconds. We see a relatively low delay at each VIF, and notice that there is not much difference in delays when we increase the number of VIFs in series.

Figure 10 shows the max throughput of a VIF. The X axis is the number of VIF in the series (at VIF 0, we just used the root VIF attached to the real interface), and the Y axis is the max throughput. As shown, the max throughput through the VIFs is about 200-210Mbps, and the max throughput of a generic kernel is about 225Mbps (not shown here). We can see that the throughput is not severely penalized by the VIF processing, for both work-conserving VIF and non work-conserving VIF.

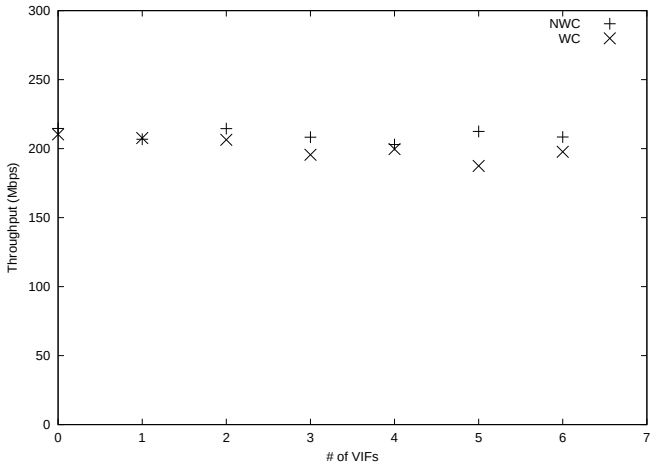


Fig. 10. VIF throughput.

c) Operation overhead: Lastly, the overhead of the management operations was measured, using PERFMON, the CPU performance-monitoring interface of FreeBSD. The operations we tested this time were; creation, deletion, attachment, detachment, read, and write. The number of trials were 100 times each. Note that read and write require `open()` system call before the actual I/O takes place, whereas, for other operations, just a single system call suffices.

The results are shown in Table I. Taking the min values as the representative results (note that the difference between max and min is not significant, except in two cases), we can conclude that the overhead of management of VIFs is about 25 μsec on average, likely to be justifiable for most applications.

Operation	Max.	Av.	Min.	SD
create	65	16	14	5
delete	30	15	14	2
attach	45	34	26	2
detach	32	20	20	1
read	144	35	32	12
write	474	32	25	45

TABLE I
SYSTEM CALL PROFILING (IN μSEC).

D. Discussion

According to the results above, the implementation produced good performance, and the management overhead is proved to be reasonable. In addition, the delay (a few microseconds) for VIF processing is insignificant for most applications.

Next, we note that the early demultiplexing can be used to protect the system from malicious process on the system, as we explored in Section II, although its standard usage is to protect the CPU cycles from malicious network interrupts [10], [6]. Hence, although the early demultiplexing violates the customary layering rules, it provides better system protection. We believe that the virtual interface approach is another illustration that justifies the early demultiplexing on modern operating system.

Lastly, through the implementation phase, we found that it is quite easy to extend the system to support various network protocol, such as IPv6, with this approach. Main reason is that the function hooks (entry points of the VIF structure) are located between the interface layer and the network layer. One exception is the demultiplexing routine for incoming packets, but it turned out that the modification is well localized.

V. RELATED WORK

A file system abstraction of network resources can be seen in [13]. The central idea of the work is "the representation of a resource as a hierarchical file system". For example, the system represents a network connection by a directory, and provides several control files for its management. We owe [13] for the file system abstraction coupled with an ASCII interface. However, we differ in that we pay major attention on hierarchical protection of system resources.

Network control on end-host OS has been explored mainly in two approaches. The first aims at providing a *specific mechanism* to control the network [3], [14], [4], [17]. The other approach tries to provide a *general mechanism* for such control, by increasing programmability of network subsystem of the end-host. To this end, two opposite approaches have been explored: expanded programmability in kernel [9], [8] and user-space protocol processing [7].

The former has flaws in that they lack the perspective of operating system service; particularly, system protection. Qlinux provides hierarchical packet scheduler similar to ours, with the support for flexible control granularity and some system protection [17]. Major difference is that their scheduler provides just non work-conserving scheduling, while ours accommodates mixture of various scheduling disciplines. Further, we support different OS-supported abstractions.

The latter provides general and flexible means to solve problems in network control. However, they do not have proper abstraction for application programs, system administrators, and end-users. We believe that the virtualization of network devices is orthogonal to these extension mechanisms, and worth supporting even in these advanced systems.

The protection of network related resource on OS has been addressed in several former studies. To review, it is beneficial to use two classifications, namely (i) output-oriented or input-oriented and (ii) incremental approach or radical approach.

Resource protection for outgoing traffic deals with the partitioning and fair allocation of CPU cycles consumed to process communications. Since these computations are executed in kernel mode, they tend to cause unfair distribution of CPU resources among processes, particularly when they are network-bound computation and the load is high.

An important representative of the incremental approach is presented in [1]. The architecture of many high-performance server applications has been shifting from traditional "process-per-connection server" model toward "single-process event-driven server" or "single-process multi-threaded server", to eliminate costly overhead of context switching. Nevertheless, operating systems have been assuming that resource protection needs to be done in a process-oriented manner. Consequently, the operating system cannot assure fair allocation of various resources, and this yields an "application vs operating system mismatch". To address the problem, they suggest to separate the protection domain and resource principal, and provide finer resource controllability by a new operating system abstraction called a *resource container* for network-intensive high-performance computing.

A radical approach for the problem is taken by Scout Operating System [11], by D. Mosberger and L.L. Peterson. They use a noteworthy protection paradigm, called *path*, which is applicable to layered systems to optimize the resource used by the processing entity penetrating the layers.

The need for the protection of the system from input traffic has been explored, in the context of *denial of service* attack. This has drawn attention because any host on the network can initiate interrupt-driven processing of the target host, creating a fundamental, but easy, security hole.

Reference [10] addresses the *receive livelock problem*, in which applications can make no progress due to the high arrival rate of interrupts caused by the incoming packets. The work had been strengthened by the proposal of the LRP [6]. They are both categorized in the incremental approach, in the sense that they are realized as an extension of existing network subsystem of a kernel.

Interestingly, radical approach for the problem is found in [16], as a demonstration of Scout Operating System, again.

The paper by O. Spatscheck and L.L. Peterson also addressed defense against the denial of service attacks, utilizing the path concept.

All these approaches share certain similarities, in that the operating system should play central role for protection of network resources, and should realize the partitioning of the CPU utilization, for network-intensive high-performance computing.

In contrast to our approach, first, the approaches mentioned above tend to lack the protection of bandwidth. Secondly, although some of the works claim they are intended for general-purpose operating systems, the target application of these proposals is high-performance server applications, in particular Web servers. In contrast, we are addressing the design principle of general-purpose operating systems used by client machines as well as servers.

VI. Conclusion

We explored a design principle of operating system support for network control, and introduced the *virtual network interface* concept. The virtual interface, attachable to the OS-supported entities, settles the network priority inversion problem and provides consistent resource management semantics to user applications. For configuration of the virtual interfaces, we pointed out that support for work-conserving and non work-conserving scheduling is required and a hierarchical scheduler is indispensable for the system to be functional.

For a proof of concept, we devised a hierarchical packet scheduler algorithm, and implemented the virtual network interface on FreeBSD with file system abstraction. The prototype controlled the network flow, as specified, exhibiting good performance (high throughput and low delays). Further, it provided a reasonable abstraction, conforming to protection semantics and syntax of Unix operating system.

We conclude that virtual network interface approach is a reasonable and promising direction to take, for general-purpose operating systems that require network control. The contributions of our work are: a novel OS abstraction for network control, proposal for consistent abstraction with the `procfs` interface, a hierarchical packet scheduler algorithm, and a prototype implementation.

Source code availability: The patch and related information are available at;
`http://www.asahikawa.wide.ad.jp`.

Acknowledgments: This work was supported in part by the US National Science Foundation, under grant ANI-0087609. We wish to thank Dr. Mark Moir and Dr. Manas Saksena for giving us detailed comments on earlier version of the work. Dr. Akira Kato gave us original idea of the `netnice` queues attachable to sockets, as well as to processes. The first author thanks faculty of Asahikawa Medical College for providing research facilities.

References

[1] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. In *the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans*, October 1999.

[2] T. M. Chen and A. W. Jackson (Editors). *Special Issue on Active and Programmable Networks*. IEEE, July 1998.

[3] K. Cho. A framework for alternate queueing: Towards traffic management by pc-unix based routers. In *USENIX Annual Technical Conference*, pages 247 – 258. USENIX, June 1998.

[4] A. Cox. The Linux traffic shaper. http://lwn.net/1998/1119/shaper.html.

[5] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Computer Communication Review*, 19(4):1 – 12, September 1989.

[6] P. Druschel and G. Banga. Lazy Receiver Processing (lrp): A Network Subsystem Architecture for Server Systems. In *the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA*, February 1996.

[7] T. Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A User-Level Network Interface for Parallel and Distributed Computing. In *SIGOPS 95*, 1995.

[8] J. Elischer and A. Cobbs. The Netgraph networking system. http://www.elischer.org/netgraph/.

[9] M. E. Fiuczynski and B. N. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *1996 Winter USENIX Conference, San Diego, CA*, pages 55 – 64, 1996.

[10] J.C. Mogul. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *the USENIX 1996 Annual Tech. Conf., San Diego, CA*, October 1996.

[11] D. Mosberger and L.L. Peterson. Making Paths Explicit in the Scout Operating System. In *the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA*, February 1996.

[12] T. Okumura, M. Moir, and D. Mosse. Netnice: nice is not only for cpus. In *ICCCN2000 (The 9th International Conference on Computer Communication and Network)*. IEEE Communications Society, October 2000.

[13] D. Presotto and P. Winterbottom. The organization of networks in plan 9. In *the Winter 1993 USENIX Tech. Conf., San Diego, CA*, pages 271 – 280. USENIX, 1993.

[14] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *Computer Communication Review*, 27(1):31 – 41, January 1997.

[15] H. Schulzrinne and R. Lanphier. *RFC2326: Real Time Streaming Protocol (RTSP)*. The Internet Society, 1998.

[16] O. Spatscheck and L.L. Peterson. Defending Against Denial of Service Attacks in Scout. In *the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans*, October 1999.

[17] V. Sundaram, A. Chandra, P. Goyal, and P. Shenoy. Application performance in the qlinux multimedia operating system. In *the Eighth ACM Conference on Multimedia*, pages 127 – 136. USENIX, November 2000.