

NetTap: An Efficient and Reliable PC-Based Platform for Network Programming

Stephen Blott, José Brustoloni and Cliff Martin
Information Sciences Research Center
Lucent Technologies — Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, USA
{blott, jcb, cliff}@research.bell-labs.com

Abstract

NetTap is a new platform for prototyping, field-testing, and deploying network services. NetTap is based on a PC running FreeBSD. NetTap can be configured as a bridge, router, or host. NetTap's API allows user-level network applications to send or receive packets, on standard network adapters, without copying or system call overheads. NetTap's watchdog timer and simple bypass switch can be configured to preserve network connectivity in case of NetTap failure. Experiments demonstrate substantial performance advantages of the NetTap API over FreeBSD's network programming APIs (Berkeley packet filters, `ipfw`, and divert sockets). In particular, NetCounter, an application for capturing and aggregating IP network billing records, achieves greater throughput on NetTap at user level than does FreeBSD's IP forwarding at kernel level.

Keywords: Services creation platforms, programming interfaces for networks, network operating systems, real-time billing.

1 Introduction

Prototyping and realistically testing new protocols or network services can be difficult. Existing networks typically lack truly programmable platforms that could serve for such experimentation. Routers and switches are often configurable, but usually don't have an API (Application Programming Interface) that would enable many desirable new services.

Researchers are currently investigating how to make routers programmable [14, 22], but it will probably

take at least several years before such routers become widely available. On the other hand, although several protocol enhancements can be demonstrated by analysis or simulation, certain other new services can only be evaluated using an actual implementation. For example, simulations demonstrated the benefits of WFQ (Weighted Fair Queueing) [10] and RED (Random Early Detection) [12] before manufacturers implemented those features and included them among router configuration options. On the contrary, certain new services, such as billing (i.e., charging users for their network usage), may not be similarly amenable to analysis or simulation. Because the adequacy of a billing strategy depends on how users react to it, and such reaction is difficult to predict or model before being observed, simulations would not be very convincing. For testing such new services, actual deployment and observation in the field are necessary [25].

We experience precisely such challenges in Libretto, a new IP network accounting system designed jointly with colleagues from Kenan Systems [4]. Libretto's main innovation is the aggregation of billing records per user and per service at the capture point. Aggregation can reduce the transmission, storage, and processing rates required for the billing data by orders of magnitude relative to systems that capture billing information per individual flows, e.g. Cisco's NetFlow [8].

Libretto's capture points are called NetCounters. They periodically send billing records to a server called NetMediator. The NetMediator correlates the records of various NetCounters and converts them to a format acceptable to the back-end billing system. The back-end may be a system currently already used for billing in PSTNs (Public Switched Telephone Networks), e.g. Kenan's Arbor/BP [15].

NetCounters ideally would be implementable by modifying router firmware or software. However, few routers currently provide to users facilities for such modifications. Manufacturers might make such modifications and include them as configurable options, but only after successful field trials.

This paper introduces NetTap, a new network programming platform for prototyping, field-testing, and deploying network services. NetTap supports the development, debugging, and maintenance of network applications¹ at user level, avoiding the difficulties associated with kernel-level software. Moreover, to simplify installation in existing networks, NetTap can be configured as a bridge, router, or host. Although inspired by the billing application, we intend NetTap to support many other network applications, e.g. traffic shaping and service level agreement monitoring and policing.

NetTap is based on low-cost PC (Personal Computer) hardware and the freely available FreeBSD [20, 17] operating system. Reliance on PC hardware gives NetTap a free ride on the PC market’s economies of scale and resulting continuous performance improvements. FreeBSD is a stable operating system with mature protocol implementations and a good track record for up-to-date peripheral support.

Making NetTap PC-based does introduce certain reliability concerns. PCs typically crash more often than bridges or routers do. On the other hand, making NetTap FreeBSD-based also introduces some difficulties. FreeBSD’s network programming APIs do not provide certain configuration options or do not support user-level applications as efficiently as is desirable.

This paper contributes a new API that, unlike FreeBSD’s, passes packets to or from user-level network applications with minimal overhead. The new API was incorporated into FreeBSD and imposes essentially no penalty for processing network applications at user level instead of at kernel level. This paper also describes simple hardware additions that significantly enhance the reliability of the PC-based NetTap.

The rest of this paper is organized as follows. Sec-

¹We call network applications those that, unlike host applications, process packets whose source and destination are nodes different from that where the application runs. Network programming is the development of network applications.

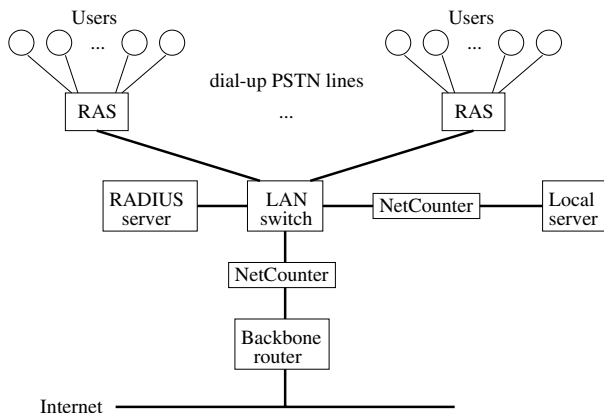


Figure 1: NetCounters are installed in strategic links of an ISP’s point of presence.

tion 2 gives an overview of the Libretto billing system. Section 3 discusses the desirable features in a network programming API for PC-based platforms. Section 4 shows that none of FreeBSD’s three network programming APIs provide all desirable features, while Section 5 describes NetTap’s API, which does support those features. Section 6 explains how we increased NetTap’s reliability. Experiments in Section 7 demonstrate that the NetTap API significantly improves user-level network application performance. Section 8 discusses related work, and Section 9 concludes.

2 The Libretto billing system

This section gives an overview of the Libretto billing system, including its NetCounter and NetMediator components.

NetCounter is the Libretto component that captures billing records. NetCounters are installed in strategic links of a network. For example, each point of presence (PoP) of an Internet Service Provider (ISP) might be wired as illustrated in Figure 1. User hosts connect via dial-up PSTN lines to a remote access server (RAS). Each RAS is connected via a high-speed local area network (LAN) to local servers and to a router that connects the LAN to the Internet backbone. The LAN might be, for example, a full-duplex, switched Fast or Gigabit Ethernet. The local servers provide services such as RADIUS (Remote Dial In User Service) authentication, DNS (Domain Name Service), email, Net news, and Web caching. Note that users do not have permanent IP

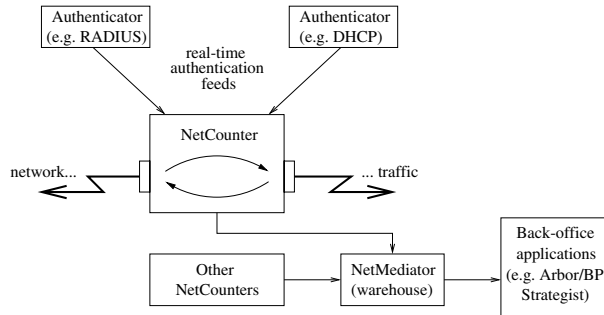


Figure 2: NetCounter receives authentication feeds from the RADIUS server, correlates flows with users, and sends to NetMediator billing records already aggregated per user.

addresses. When a user dials in, the RAS collects the user’s name and password and contacts a RADIUS server to authenticate the user. An IP address is then dynamically assigned to the user. In such architectures, a NetCounter would be installed in the link between the LAN’s switch and the backbone router. If accesses to local servers are to be charged, NetCounters might also be installed in links between the LAN’s switch and local servers. If users connect to the PoP via cable or xDSL, the access architecture changes somewhat, but NetCounter installation remains unchanged.

A NetCounter generates aggregate billing records taking into account every packet that crosses the NetCounter’s link. Each record might contain, for example, the total number of bytes, packets, and flows per user. In order to correlate packets with users, the NetCounter receives authentication information from the RADIUS server (e.g., user x dialed in and was assigned IP address y , or user z hung up), as shown in Figure 2. NetCounter may also separate each user’s records per service (according to TCP or UDP port numbers), or measure characteristics of the user’s traffic (e.g., average bit rate while active or inter-arrival distribution) [4]. Because the measures that are relevant in a given network may not be obvious *a priori*, it is desirable that NetCounter be easily modifiable.

Each NetCounter periodically sends its billing records to NetMediator. Each record corresponds to the immediately previous period only. The NetMediator classifies, correlates, filters, and aggregates records received from NetCounters, and generates billing records in format acceptable to a back-end billing system. The back-end system generates the

actual user bills. The back-end system may also be used in network planning (e.g., determining when and where more capacity is needed) [15].

In principle, NetCounters could be implemented at the switch or router ports that connect to the respective links. However, programmable ports are currently not widely available. Therefore, we implemented NetCounters as bridge applications on Net-Tap platforms. NetMediator and back-end systems are implemented on high-end server hosts.

3 Requirements for a network programming API

This section discusses features that are desirable in an API that supports network applications, e.g. the previous section’s NetCounter, on PC-based platforms.

In general, such network programming API should provide:

1. Support for configuring the platform as a bridge, router, or host. Bridge configuration is often the most convenient because a bridge can be inserted into an existing network’s link without any IP subnet reconfiguration and without any additional hardware. In contrast, a router can be installed only between different IP subnets. Therefore, a router configuration requires creating additional *transit subnets*, an inconvenient chore [18]. PC-based platforms also typically have limited CPU power and memory and I/O bus bandwidth. Therefore, such platforms may not be able to keep up with more than a few high-speed interfaces (e.g., full-duplex Fast Ethernet). This performance consideration may favor bridge and small-aggregate-bandwidth router configurations. Another difference between bridges and routers is that simple hardware devices can, as explained in Section 6, preserve a network’s connectivity by bypassing a network’s bridge when the bridge fails. Similarly bypassing a router is not possible because each router interface must be on a separate subnet.

Many network monitoring applications, including NetCounters, can run on hosts instead of on bridges. In a switched LAN, however, con-

necting a host may take more hardware than does connecting a bridge. Hosts may require an extra switch port configured to mirror the monitored link, or, in the case of Ethernet, a hub. The hub has the disadvantage of making the link operate half-duplex, instead of full-duplex. Another disadvantage of implementing network applications on a host is that, unlike bridge applications, they cannot be upgraded, e.g., to support certain pre-paid billing plans or to enforce service level agreements, which would require selectively delaying or dropping packets.

2. General-purpose functionality. This requirement may sound obvious, but several existing network programming APIs support only a single function, e.g. firewalling.
3. Support for user-level applications. Applications are easier to develop, debug, and maintain at user level than at kernel level.
4. Passing packets between applications and the system without copying. Low-cost PC platforms typically have memory bandwidth that is not much greater than that of today's Fast Ethernets in full-duplex mode (200 Mbps). However, many APIs pass packets to or from applications by copying. Such copying can severely harm end-to-end throughput.
5. Reduced number of system calls and interrupts per packet. The time it takes to field a system call or interrupt on a low-cost PC platform is often not much lower than the minimum time between packets on today's Fast Ethernets (5.2 μ s). For the application to keep up with back-to-back short packets, the API should allow the application to send or receive at least several packets per system call and per interrupt.

4 FreeBSD's network programming APIs and their shortcomings

FreeBSD offers three APIs for network programming: Berkeley packet filters (BPFs), `ipfw`, and divert sockets. Somewhat surprisingly, however, none of them fulfill all the requirements enumerated in the previous section. The following subsections discuss each API in turn.

4.1 BPFs

The BPF [19] API supports general-purpose user-level network applications in host, bridge, or router configurations.

A BPF is a pseudo-device that can be open only by the system administrator. Applications use the `ioctl` system call to set the size of a BPF's *buffer*, specify a BPF's *filter*, bind a BPF to a network interface, put a network interface in promiscuous mode (which stops the interface from discarding packets destined to other nodes), and enable BPF's "immediate mode" (causing `read` calls to return immediately after a packet is received). Whenever a network interface receives a packet, the interface submits the packet to the filter of each BPF bound to the interface. BPF filters are programs written in a safe, interpreted language. Given a packet, a BPF filter computes the length of the packet header (if any, up to the entire packet) that should be copied to the BPF buffer. If the BPF buffer has room, it may hold multiple successive packet headers. Applications use the `read` system call to get and empty the contents of a BPF buffer. The application buffer supplied to the `read` call must have the same size as that of the BPF buffer. A single `read` call may return multiple packet headers. Applications use the `write` system call on a BPF to send a single packet through the respective interface.

Although flexible, BPFs have several inefficiencies. In a bridge configuration, the BPF API will typically copy the packet twice on input and another time on output. Moreover, BPFs do not provide reduction in the number of system calls for sending packets.

4.2 IPFW

FreeBSD's `ipfw` API supports kernel-level firewalling applications in router or bridge configurations². Only the system administrator can use `ipfw`.

The administrator uses `ipfw` to specify a *rule list* in a safe, interpreted language. Each rule in the list contains a pattern and an action. For example, a rule's pattern might be "all Telnet packets coming into my network", and the corresponding action might be "drop". The kernel's IP implementation tries to match every incoming or outgoing packet

²Bridge support started in FreeBSD 2.8.

against each rule in `ipfw`'s rule list. The kernel executes the action of the first matching rule.

Executing applications in the kernel, without copying packets and without system call overheads, would tend to make `ipfw` efficient. However, `ipfw` uses interpreted code, which typically is much slower than native code. `ipfw` also has the drawback of limited applicability (firewalling only).

4.3 Divert sockets

FreeBSD's divert sockets API supports general-purpose user-level network applications in router (but not bridge) configurations. Only the system administrator can use divert sockets.

In FreeBSD, one of `ipfw`'s possible actions is to *divert* the packet to the *divert socket* bound to a specified port. Applications create divert sockets as raw sockets with the pseudo-protocol `IPPROTO_DIVERT`. Applications use the `bind` system call to bind a divert socket to a specified port.

Applications typically receive diverted packets using the `recvfrom` system call. The call returns the address of the interface on which the packet was just received, or `INADDR_ANY` if the packet was outgoing (diverted just before next hop forwarding). The call also returns the divert socket's port number. Applications may arbitrarily modify the packet and reinsert the packet for `ipfw` processing following the rule that caused the packet to be diverted. Applications typically use the `sendto` system call to reinsert packets, supplying the same address and port number previously returned by `recvfrom`.

The main disadvantages of divert sockets are configuration inflexibility (only routers are supported), copy overhead (each packet is typically copied once on input and another time on output), and no reduction in the number of system calls (at least one `read` and one `write` call per packet).

5 The NetTap network programming API

This section describes the NetTap network programming API. Like FreeBSD's APIs, discussed in

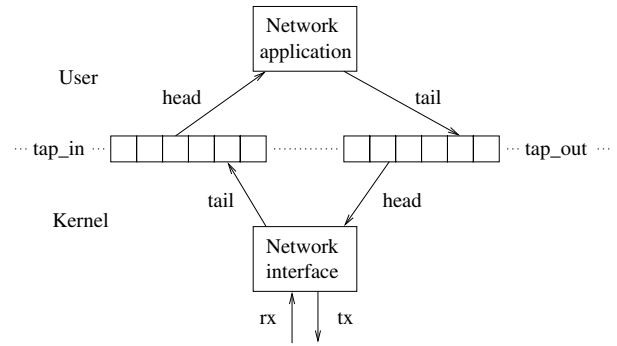


Figure 3: NetTap's `tap_in` and `tap_out` queues allow network applications to send or receive packets without copying or system call overheads.

the previous section, the NetTap API can be used only by the system administrator. However, unlike FreeBSD's APIs, the NetTap API does support all requirements enumerated in Section 3.

The main features of the NetTap API are: (1) All packet buffers are mapped both to the system and to network applications; (2) Instead of passing to each other *copies* of packets, system and network applications exchange *pointers* to packets; and (3) System and network applications communicate asynchronously via a number of circular queues, thus avoiding system call overheads in normal cases, as illustrated in Figure 3. NetTap queues either contain only elements that are enqueued by the system and dequeued by a network application, or vice-versa. Therefore, application accesses do not have to be synchronized with respect to system accesses. The system accesses queues asynchronously when a network interface interrupt occurs or returns, when a NetTap system call returns, or when the system is idle.

The NetTap API provides six system calls:

1. `mbuf_map`
2. `mbuf_unmap`
3. `mbuf_pull`
4. `mbuf_push`
5. `interface_tap`
6. `interface_untap`

In systems derived from BSD Unix [20], including FreeBSD, network interfaces and protocols hold

packets in buffers known as *mbufs*. All mbufs are allocated from a single unpageable region. The `mbuf_map` primitive maps this region to the application's address space. This primitive also creates two circular queues, `mbuf_alloc` and `mbuf_dealloc`, and maps them to the application's address space. Applications allocate mbufs by dequeuing the mbufs' pointers from `mbuf_alloc`, and deallocate mbufs by enqueueing the mbufs' pointers in `mbuf_dealloc`. The `mbuf_map` primitive includes an argument, `mbuf_prealloc`, that specifies the minimum number of mbufs that should be enqueued in the application's `mbuf_alloc` queue. The system asynchronously replenishes that queue, making mbufs available to the application. However, when necessary (e.g., `mbuf_alloc` is empty), applications may use the `mbuf_pull` primitive to force the system to enqueue a specified (strictly positive) number of mbufs synchronously into `mbuf_alloc`. The system asynchronously dequeues pointers from applications' `mbuf_dealloc` queues and deallocates the respective mbufs. However, if necessary (e.g., `mbuf_dealloc` is full), an application may use the `mbuf_push` primitive to force the system to dequeue any pointers from `mbuf_dealloc` synchronously and deallocate the respective buffers. The `mbuf_unmap` primitive unmaps the mbuf region and destroys the application's `mbuf_alloc` and `mbuf_dealloc` queues.

In NetTap, a network interface can be in either *untapped* or *tapped* state. Untapped is the default state, in which applications use FreeBSD's conventional APIs to receive or send packets through the interface. On the contrary, in tapped state, the interface is disconnected from higher-layer protocols (e.g., IP), and FreeBSD's conventional APIs become inoperative on that interface. The `interface_tap` primitive places an interface in tapped state, whereas `interface_untap` reverts the interface to untapped state. `interface_tap` maps the interface's two *tapping* circular queues, `tap_in` and `tap_out`, to the application's address space, as illustrated in Figure 3. When a packet arrives, the interface enqueues in `tap_in` pointers to the mbufs containing the packets. Applications receive packets by dequeuing the respective pointers from `tap_in`. No system calls are necessary to receive packets. However, if desired (e.g., `tap_in` is empty), applications may use the `mbuf_pull` primitive (with a null number of mbufs) to wait for a packet to arrive and the system to enqueue a pointer in some `tap_in` queue mapped to the application. While the application waits for a packet, the system may run other applications or *poll* the network interfaces, thereby possibly reduc-

ing the number of interrupts. Conversely, applications send packets by enqueueing in `tap_out` pointers to the mbufs containing the packets. The system asynchronously dequeues mbuf pointers from `tap_out` queues and sends the respective packets through the respective interfaces. However, if necessary (e.g., `tap_out` is full), applications may use the `mbuf_push` primitive to force the system to dequeue synchronously any pointers from `tap_out` queues mapped to the application, and send the respective packets through the respective interfaces.

NetTap prevents mbuf leakage as follows. Mbuf headers gain a `pid` field containing the identifier of the process that holds the mbuf (if any). The system updates an mbuf's `pid` field whenever the system enqueues the mbuf's pointer in an application's `mbuf_alloc` queue or in an interface's `tap_in` queue. If multiple processes tap the same interface, each process must update an mbuf's `pid` field before dequeuing the mbuf's pointer from `tap_in`. When the system dequeues an mbuf's pointer from an `mbuf_dealloc` or `tap_out` queue, the system clears the mbuf's `pid` field. Processes gain a flag indicating whether the process has mapped mbufs. When a process with such flag set exits, the system scans the mbuf region to find and deallocate mbufs held by that process.

NetTap uses reference counting for maintaining an interface in tapped state while at least one process has that interface tapped. NetTap also keeps a list of interfaces tapped by each process. When a process exits, the system automatically untaps any interfaces still tapped by the process.

NetTap applications can tap any number of interfaces and therefore can implement host, bridge, or router configurations. NetTap applications can be written in any language and implement arbitrary functionality. Applications for a bridge configuration, such as NetCounter, may run directly on top of NetTap's API. Router or host configurations, however, may need additional protocol support. Therefore, NetTap also provides user-level libraries for TCP/IP protocols, IP security [16], and output link scheduling [7]. These libraries run on top of the NetTap API. Applications may easily specialize or modify NetTap's libraries. For example, applications such as NAT (network address translation) [11] or LSNAT (load sharing using IP network address translation) [24] might modify NetTap's user-level IP implementation.

We expect that NetTap platforms will typically be dedicated to an application and have little need for time-sharing scheduling. If multiple threads of control are needed, NetTap applications may use POSIX (user-level) threads, so as to avoid context switch overheads. However, single-thread event-driven implementations can be expected to outperform multiple-thread implementations [21].

On the other hand, if a single processor is unable to keep up with the network traffic tapped by an application, it may be useful to run multiple instances of the application on a shared-memory multiprocessor system, with one application instance per processor. Multiple processes can map mbufs and tap the same interfaces. `mbuf_alloc` and `mbuf_dealloc` queues are private to each process. `tap_in` and `tap_out` queues, however, are shared by all processes that tap the respective interface. Concurrent accesses to tapping queues or other shared data structures must be synchronized. Because NetTap is PC-based, synchronization can be achieved at user level, without system call overheads. The concurrent processes can define a shared integer to be a lock that guards certain shared data structures. Each process uses the i486 (or later) CPU’s `CMPXCHG` (compare and exchange) instruction to acquire or release the lock respectively before or after accessing the shared data structures. The system keeps a list of locks used by each process. When a process exits, the system automatically releases any locks still held by the process.

6 Improving reliability

The NetTap platform is based on PC hardware. This section describes measures taken to prevent NetTap failures and, should NetTap failures happen, to stop them from affecting network reliability.

To prevent failures, NetTap uses:

1. Rack mounting with improved ventilation.
2. Dual hot-swappable power supplies.
3. FLASH-based instead of mechanical disk. The mean time between failures of the FLASH-based disk is much higher than that of a typical mechanical disk.

A watchdog timer allows NetTap to detect and re-

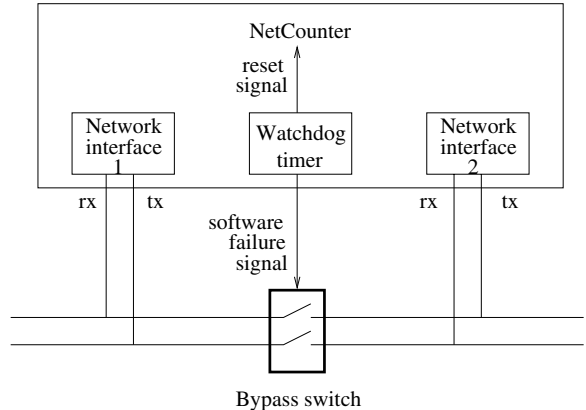


Figure 4: NetTap’s bypass switch may be configured to preserve network link connectivity in case of NetTap failure.

cover from software failures. NetTap applications should periodically reinitialize the watchdog timer. If the timer counts down to zero, presumably NetTap’s application is hung. When the counter reaches zero, the watchdog timer generates a pulsed reset signal, which resets the system, and a latched software failure signal, which may be configured to activate NetTap’s *bypass switch*, as illustrated in Figure 4. The latter signal is reset by software, after the system reboots.

NetTap’s bypass switch may be used when (1) NetTap is configured as a bridge and (2) it is desirable to preserve the network connectivity when the NetTap fails (e.g., when NetTap is used for billing or traffic shaping, but not for firewalling). When the bypass switch is used, it connects the links of two network interfaces while the power is off (hardware failure) or the switch receives the software failure signal from the watchdog timer.

7 Experimental results

This section reports the experimental effects of the API (BPF, `ipfw`, divert sockets, or NetTap) on the throughput and latency of two network applications: (1) the simplest forwarding allowed by the API (i.e., bridging when using BPF or NetTap, or IP forwarding when using `ipfw` or divert sockets), and (2) NetCounter.

In our experiments, the applications ran on either

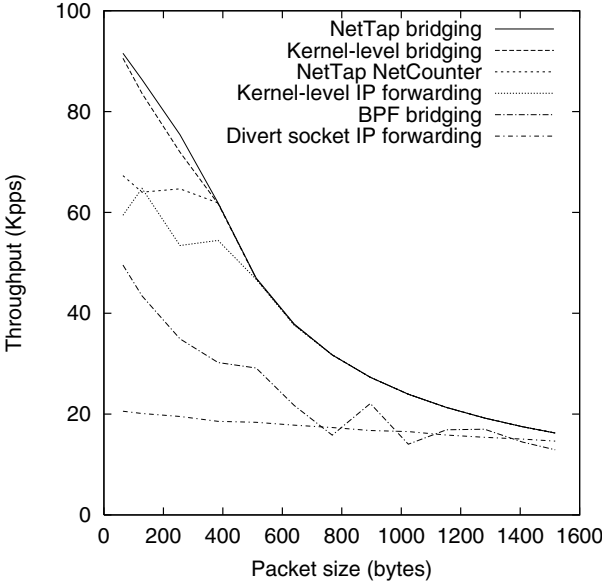


Figure 5: On 333 MHz Celeron PCs, throughput of NetTap bridging is much higher than that of BPF bridging or divert socket IP forwarding. Throughput of NetTap NetCounter is higher than that of kernel-level IP forwarding.

of two PCs. The first PC had a 333 MHz Intel Celeron CPU (SPECint95 rating of 12.3) with 32 KB L1 cache (16 KB instruction + 16 KB data) and 128 KB L2 cache, 64 MB RAM, 80 MB Sandisk Flashdrive, and two Intel Ether Express Pro 10/100 Ethernet adapters with Intel 82559 chipset. The second PC had a 600 MHz Intel Pentium III CPU (SPECint95 rating of 24.6) with 32 KB L1 cache (16 KB instruction + 16 KB data) and 512 KB L2 cache, 128 MB RAM, and the same peripherals as the first PC. Both PCs ran FreeBSD 3.2 augmented with the NetTap API. We measured the throughput and latency using Netcom’s Smartbits SMB-2000 performance analyzer outfitted with two ML-7710 10/100 layer 3 cards configured for full-duplex Fast Ethernet operation. This analyzer follows the benchmarking methodology of IETF’s RFC 1944 [5].

Figure 5 shows the applications’ throughputs (maximum number of packets that each application was able to process per second without any packet being lost) on the 333 MHz Celeron PC. None of the APIs provided the ideal throughput (about 298, 169, or 91 thousands of packets per second (Kpps), respectively) for packets that are 64, 128, or 256 bytes long. NetTap provided a throughput of about 92 Kpps,

roughly 31% of the ideal, for the bridging application and 64-byte packets. For comparison purposes, we also implemented a simple, unbuffered bridging algorithm at kernel level. NetTap’s user-level bridging slightly outperformed kernel-level bridging because both NetTap and kernel-level bridging run essentially without copying or system call overheads, but NetTap has the advantage that its queuing of incoming packets helps avoid the onset of packet loss. For all packet sizes, throughput of NetTap bridging was much higher than that of BPF bridging or divert socket IP forwarding. For the NetCounter application and 64-byte packets, NetTap provided a throughput of about 67 Kpps, more than the throughput of kernel-level IP forwarding (59 Kpps). For either the bridging or the NetCounter application, NetTap provided 100% of the ideal throughput for packets of length 384 bytes or longer.

Figure 5 also shows that, for packets up to 640 bytes long, bridging on BPFs provided much better throughput than did IP forwarding on divert sockets. This is surprising, because BPFs copy each packet three times, whereas divert sockets copy each packet only twice. BPFs’ advantage is due to input batching and substantially fewer system calls. The BPFs were configured in immediate mode with 1 MB buffers. In this experiment, the average number of packets returned by each BPF read call was about 24 for 64-byte packets, 5 for 128-byte packets, and 1 for packets of length 640 bytes or longer. In contrast, each divert socket read call always returns only one packet.

Figure 6 shows the applications’ store and forward latencies on the 333 MHz Celeron PC. Latency was about 28 μ s for kernel-level bridging, 37 μ s for kernel-level IP forwarding, 45 μ s for NetTap bridging, 48 μ s for BPF bridging, 49 μ s for NetCounter on NetTap, and 68 μ s for IP forwarding on divert sockets. NetTap’s throughput advantage over kernel-level IP forwarding comes at the cost of additional latency (about 17 μ s for bridging). This latency is due to NetTap’s tapping queues. The NetCounter application has been heavily optimized and adds only about 4 μ s over bridging. Note that, because BPFs and divert sockets copy packets, they give latencies that increase considerably with packet size, unlike the figure’s other latencies.

We also measured the effect of system calls on latency. If NetTap bridging always calls `mbuf_pull` (specifying null number of mbufs) before dequeuing each packet from `tap_in`, and calls `mbuf_push`

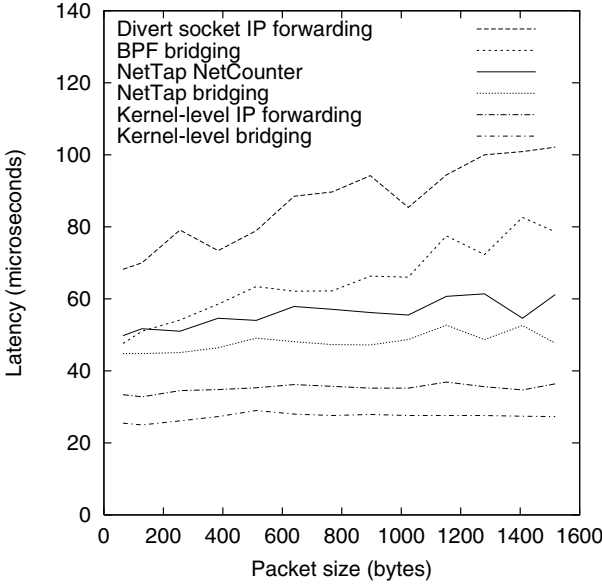


Figure 6: NetTap’s higher throughput than that of kernel-level IP forwarding comes at the cost of somewhat higher latency.

after enqueueing each packet into `tap_out`, the latency rises by about $4.5 \mu s$ — almost the minimum time between 64-byte packets on Fast Ethernet.

Figure 7 shows the applications’ throughputs on the 600 MHz Pentium III PC. Throughput of NetTap bridging for 64-byte packets was about 90 Kpps, actually a bit less than on the 333 MHz Celeron PC. This result suggests that this throughput is not CPU bound, but rather limited by I/O (network adapter and/or PCI bus). Throughput of NetCounter on NetTap for 64-byte packets did improve considerably to 80 Kpps. Because NetCounter involves more computation than does bridging, NetCounter can benefit more from the faster processor. Throughput of kernel-level IP forwarding for 64-byte packets also improved, to 71 Kpps. Throughput improved more slightly for BPF bridging, whose copy penalty may prevent it from fully benefiting from the faster processor. The relative performances were the same as on the 333 MHz Celeron PC.

8 Related work

Router plugins [9] are modules that can be dropped into an operating system’s kernel to modify or cus-

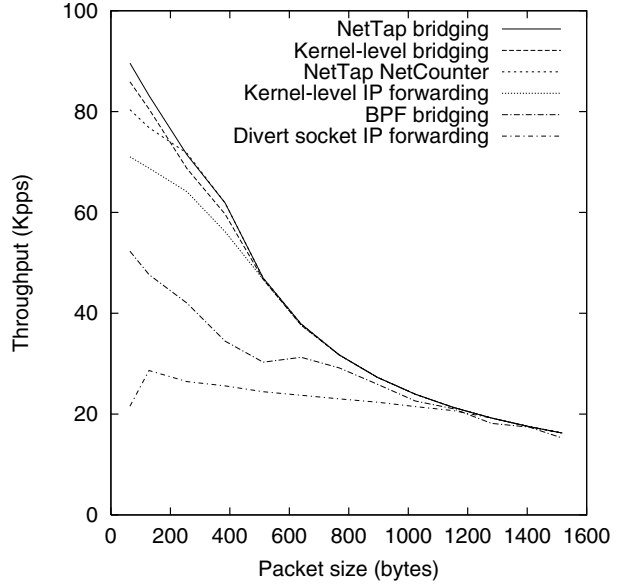


Figure 7: The relative performance on the 600 MHz Pentium III PC is the same as on the 333 MHz Celeron PC. The faster CPU helps NetTap NetCounter more than it does BPF bridging or divert socket IP forwarding.

tomize a router’s protocols. The initial router plugins prototype was PC-based and ran NetBSD. Compared to NetTap, router plugins have the disadvantages of enabling only router configurations and supporting only kernel-level programming. The router plugin’s packet classification scheme may be implemented at user level on top of NetTap’s primitives and can be useful for many applications.

Pronto [14] is an architecture for programmable routers with a cleanly defined interface between a generic forwarding engine and environments for service-specific programs. By keeping programming and forwarding separate, Pronto can enable both flexibility and performance. The Extensible Routers project [22] is also investigating how to make routers programmable, paying particular attention to making router hardware scalable and offering different levels of path optimization. Both Pronto and Extensible Routers ultimately target routers of larger scale than does NetTap.

Several projects are currently investigating active networks, a paradigm whereby users may inject into the network not only packets but also programs for processing those packets [6]. NetTap is a possible platform for active networking. However, ac-

tive networking involves many issues not discussed in this paper, including what language to use and how to distribute and compose active network programs [6, 26, 13], and how to make active networking secure [1]. Applications of active networking include congestion control [2], caching [3], and network management [23].

9 Conclusions

We described Libretto, a new billing system for IP networks. Libretto is an example of a network service that can be evaluated satisfactorily only on a field trial. Unfortunately, today's networks still lack network programming platforms that would easily enable such trials. We argued that bridge configurations often enable convenient deployment, realistic performance requirements, and achievable reliability on PC platforms. FreeBSD, a PC operating system that is well-regarded because of its mature networking code, nevertheless offers network programming APIs that lack many desirable features. We presented the NetTap API, which, unlike FreeBSD's APIs, enables host, bridge, or router configurations and allows arbitrary user-level applications to send or receive packets without any copying and usually without system calls. Our experiments demonstrate the significant performance advantages of the NetTap API for network programming. We also described simple measures that enhance the reliability of the PC-based NetTap platform. NetTap does not require special network interfaces and is straightforward to implement. We believe that an API such as NetTap's would be a valuable addition to most existing PC and workstation operating systems.

References

- [1] D. S. Alexander, W. Arbaugh, A. Keromytis and J. Smith. "A Secure Active Network Architecture: Realization in SwitchWare", in *Network*, IEEE, vol. 12, nr. 3, pp. 37-45.
- [2] S. Bhattacharjee, K. Calvert and E. Zegura. "Congestion Control and Caching in CANES", in *Proc. ICC '98*, IEEE, 1998.
- [3] S. Bhattacharjee, K. Calvert and E. Zegura. "Self-Organizing Wide-Area Network Caches", in *Proc. INFOCOM'98*, IEEE, Mar. 1998.
- [4] S. Blott et al. "User-Level Billing and Accounting in IP Networks", to appear in *Bell Labs Tech. Journal*.
- [5] S. Bradner and J. McQuaid. "Benchmarking Methodology for Network Interconnect Devices". IETF, RFC 1944, May 1996.
- [6] K. Calvert, S. Bhattacharjee, E. Zegura and J. Sterbenz. "Directions in Active Networks", in *Communications Magazine*, IEEE, 1998.
- [7] K. Cho. "A Framework for Alternate Queuing: Towards Traffic Management by PC-UNIX Based Routers", in *Proc. Annual Tech. Conf., USENIX*, June 1998.
- [8] Cisco. "FlowCollector Overview", at http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/nfc/nfc_3_0/nfc_ug/nfccover.htm
- [9] D. Decasper, Z. Dittia, G. Parulkar and B. Platner. "Router Plugins: A Software Architecture for Next Generation Routers", in *Proc. SIGCOMM'98*, ACM, Sept. 1998.
- [10] A. Demers, S. Keshav and S. Shenker. "Design and Analysis of a Fair Queueing Algorithm", in *Proc. SIGCOMM'89*, ACM, Sept. 1989, pp. 1-12.
- [11] K. Egevang and P. Francis. "The IP Network Address Translator (NAT)". IETF, RFC 1631, May 1994.
- [12] S. Floyd and V. Jacobson. "Random Early Detection Gateways for Congestion Avoidance", in *Trans. Networking*, IEEE/ACM, Aug. 1995.
- [13] M. Hicks, P. Kakkar, J. Moore, C. Gunter and S. Nettles. "PLAN: A Packet Language for Active Networks", in *Proc. Intl. Conf. Functional Programming*.
- [14] G. Hjálmtýsson and S. Bhattacharjee. "Control on Demand - An Efficient Approach to Router Programmability", to appear in *JSAC*, IEEE, 1999.
- [15] Kenan Systems. Whitepapers available at <http://www.kenan.com/content/compinfo/whitepapers/index.htm>.
- [16] S. Kent and R. Atkinson. "Security Architecture for the Internet Protocol". IETF, RFC 2401, Nov. 1998.

- [17] G. Lehey. "The Complete FreeBSD", 2nd. ed., Walnut Creek CDROM Books, Walnut Creek, CA, 1997.
- [18] T. Limoncelli. "Tricks You Can Do If Your Firewall Is a Bridge", in *Proc. 1st Conference on Network Administration*, USENIX, Apr. 1999.
- [19] S. McCanne and V. Jacobson. "The BSD Packet Filter: A New Architecture for User-Level Packet Capture", in *Proc. Winter Tech. Conf.*, USENIX, Jan. 1993.
- [20] M. McKusick, K. Bostic, M. Karels and J. Quarterman. "The Design and Implementation of the 4.4 BSD Operating System", Addison-Wesley Pub. Co., Reading, MA, 1996.
- [21] V. Pai, P. Druschel and W. Zwaenepoel. "Flash: An Efficient and Portable Web Server", in *Proc. Annual Tech. Conf.*, USENIX, June 1999, pp. 199-212.
- [22] L. Peterson, S. Karlin and K. Li. "OS Support for General-Purpose Routers", in *Proc. HotOS'99*, IEEE, Mar. 1999.
- [23] D. Raz and Y. Shavitt. "An Active Network Approach for Efficient Network Management", in *Proc. IWAN'99*, LNCS 1653, July 1999, pp. 220-231.
- [24] P. Srisuresh and D. Gan. "Load Sharing using IP Network Address Translation". IETF, RFC 2391, Aug. 1998.
- [25] P. Varaiya, H. Varian, H. Chand, R. Edell and H. Rupp. "INDEX Project Homepage", at <http://www.INDEX.Berkeley.EDU/public/-index.phtml>
- [26] D. Wetherall, J. Gutttag and D. Tennenhouse. "ANTS: A toolkit for building and dynamically deploying network protocols", in *Proc. OPE-NARCH'98*, IEEE, Apr. 1998.