Revisiting User-Level Networking

Jan Sacha
Jeff Napper
Henning Schild
Sape Mullender

Bell Laboratories
Antwerp, Belgium

Abstract

User-level networking has been applied mostly in the area of high-performance computing since it allows applications to obtain maximum performance from the hardware. We argue that running network protocol stacks in the user space also allows applications to reduce the amount of state that the operating system kernel maintains on their behalf enabling much greater application elasticity and mobility. We describe a lightweight user-level networking framework that does not require any modification to existing networking hardware nor protocols. The main advantages of our framework are its simplicity and ease of implementation.

1 Introduction

User-level implementations of network protocol stacks have existed for many years. Initially, such stacks were used to develop and test novel network protocols which ran outside of the operating system kernel and thus did not harm the kernel's reliability. They were also easier to deploy since they did not require kernel recompilation and reboot and were easier to debug [12, 17]. With the advent of high-performance computing, user-level networking also became a mechanism to increase application performance. By bypassing the kernel and interacting directly with the network interface controller, applications could reduce their overhead and achieve lower communication latency [18, 13, 4].

We claim that there exists yet one more reason to implement network protocol stacks in the user space. As applications gradually move to the cloud, it becomes increasingly important to be able to checkpoint, suspend, resume, marshal, and migrate applications. All these tasks are greatly simplified if the operating system kernel maintains very little or no state on behalf of user processes. This state, however, is very often stored by kernel-level network protocol implementations, for example in the form of connection parameters and acknowledged data buffers. In order to support application elasticity and mobility, the operating system needs to be able to extract application state from its internal structures and consistently redeploy it when needed. By running the entire network stack in the user space, not only do applications limit their kernel-level state but also gain an opportunity to react to mobility events in an application appropriate way.

Further, we argue that running network protocol stacks inside user applications naturally improves modern hardware utilization. According to the current trends in computer architecture evolution, both the core count per host and network bandwidths grow at a high rate. In order to utilize hardware efficiently, network traffic needs to be processed in parallel on multiple cores, which can be achieved by dispatching packets from the network interface to the application cores as early as possible. Such a design also improves memory cache performance because each packet is processed mostly by one core only. Finally, high-performance applications might tune the behavior of their network stacks using application-specific knowledge in order to achieve better performance.

Many existing approaches to user-level networking require either using specialized networking hardware or running customized network protocols. In this paper we describe a light-weight framework for user-level networking that runs on generic hardware and does not require any modifications

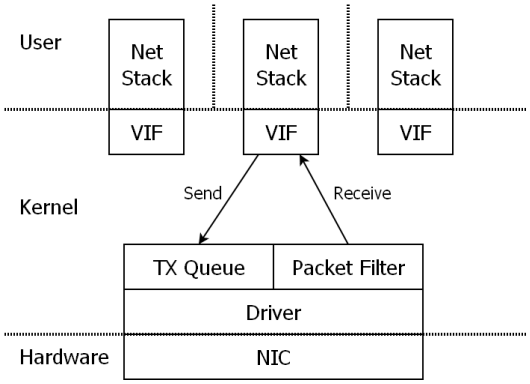


Figure 1: System overview

to the existing network protocols. The main advantages of our framework are its simplicity and low amortized overhead. Our approach is based on the notion of virtual network interfaces that isolate access to physical interfaces from user applications. A virtual interface is defined using two bitmask patters that describe incoming and outgoing packets. Using these patterns, an efficient demultiplexing algorithm dispatches packets received from the physical network interface directly into user applications. The network protocol stacks are implemented entirely in the user space.

The rest of this paper is organized as follows. We discuss related work in section 2, we describe our framework in section 3, and we evaluate it in section 4.

2 Related Work

The earliest approaches to user-level networking focused mostly on the flexibility and ease of implementation for new network protocols rather than performance. They usually provided built-in packet filtering languages [12, 11] or allowed user processes to register custom packet handlers in the kernel to filter the incoming traffic [19]. Some of these built-in languages were later optimized using dynamic code generation [20, 7].

In the high-performance computing area, user-level networking was used as a mechanism to eliminate the kernel from the critical path when sending or receiving network packets. The main challenge in these approaches, apart from maximizing performance, was to provide user isolation and security. Most proposed solutions either relied on custom communication protocols, such as Active Messages [19], Fast Messages [13], and BIP [14], or required packet tagging [18]. Other high-performance solutions relied on specialized hardware [17, 5]. These later approaches eventually lead to a standard, the Virtual Interface Architecture [6, 4], which defined a set of hardware functions needed to support secure and efficient user-level network interface access.

A large research effort was devoted into designing packet classification algorithms for applications such as routing, admission control, intrusion detection, and traffic accounting [8, 10, 15]. It was shown that the complexity in a general packet classification problem grows exponentially with the number of packet classification rules and thus the search space might become extremely large even for moderate rule sets. Most state-of-the-art algorithms address the classification problem by constructing decision trees that allow packet processing with a minimum number of memory accesses. Due to the search space explosion, such decision trees often require very large amounts of memory [15]. Our approach is closer to the PathFinder [2] and the Tuple Space Search [16] algorithms which use bitmask patterns and hash tables and typically require much less memory.

A separate approach to couple the network protocol stack with the application is to use a virtual machine such as Xen [3] or KVM [9]. Typically, the hypervisor provides to each guest system a virtualized network interface which either has its own unique MAC address and is bridged to the physical interface or has the same MAC and IP addresses as the physical interface and is multiplexed using Network Address Translation (NAT). The main disadvantage of these approaches compared to our model is that the hypervisor and the guest operating system add a significant overhead and complexity to the user application.

```
struct Packetdigest{
        struct{ /* Ether */
                uchar   src[6];
                uchar   dst[6];
                uchar   type[2];
        };
        union{
                struct{ /* IPv4 */
                        uchar   src[4];
                        uchar   dst[4];
                        uchar   proto[1];
                        /* TCP or UDP */
                        uchar   sport[2];
                        uchar   dport[2];
                };
                struct{ /* ICMP */
                        uchar   type[1];
                        uchar   code[1];
                };
                struct{ /* AoE */
                        uchar   major[2];
                        uchar   minor[1];
                        uchar   tag[4];
                };
        };
};

struct Packetpattern {
        struct Packetdigest     mask;
        struct Packetdigest     value;
};
```

Figure 2: Sample packet digest definition in the C language

3 System Model

An overview of our networking framework is shown in figure 1. Every physical network interface (NIC) is associated with a collection of *virtual interfaces* (VIF) which allow user processes to send and receive packets. Typically, network protocol implementations generate a virtual interface per each communication endpoint such as a TCP or UDP socket. A virtual interface consists of two bitmask-value patterns, which describe the packets that the interface is allowed to transmit and receive, and a kernel-level queue for incoming packets. The purpose of the bitmask-value patterns is twofold: to verify packets generated by processes and to demultiplex incoming traffic. We also envisage that virtual interfaces will enable the kernel to control the network resource consumption by users processes. For example, the kernel could divide available bandwidth between virtual interfaces according to some policy or provide low-latency network access to real-time processes.

In order to send a packet, a user-level network stack generates all packet headers (including data layer) and passes the packet payload together with the headers to the kernel. The kernel verifies that the packet matches the allowed pattern and attempts to transmit it. If the NIC is busy, the packet is appended to the driver's transmit queue. Normally, patterns for outgoing packets are not allowed to overlap so that processes cannot interfere with each others traffic. If a process attempts to register a virtual interface with a pattern overlapping with another virtual interface, which might for example happen if the process tries to open a connection on a port used by another process, the kernel returns an error.

When an incoming packet is received from hardware, the packet filter compares the packet headers with the bitmask-value patterns defined in the virtual interfaces and selects the virtual

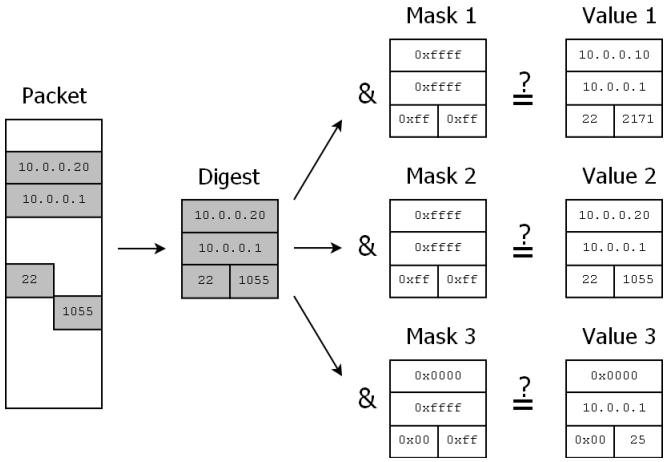


Figure 3: Sample packet digest and bitmask-value patterns

interface that receives the packet. If a user process is already waiting for a packet from the selected virtual interface, the kernel delivers the packet directly to the user-space network protocol stack. If no process is waiting for the packet, the kernel enqueues the packet in the virtual interface.

3.1 Packet filtering

Network access control and traffic demultiplexing is performed using a data structure called *packet digest* which contains all the fields extracted from packet headers that are relevant for packet filtering such as protocol types, source and destination addresses, and port numbers. Figure 2 shows a sample packet digest format in the C language. Since the digest structure is shared by all protocols it contains a union of alternative fields at each protocol layer. Only the data layer is fixed (Ethernet in the discussed example) since it is usually NIC-specific.

Each virtual interface defines two packet patterns: for incoming and outgoing packets. Each such pattern is a pair of packet digest structures, where the first structure is interpreted as a binary bitmask and the second is the corresponding binary value. Note that bitmask-value expressions are sufficiently flexible to represent common pattern types such as exact values for fields and subfields, wildcards (i.e., zeroes in the bitmask) and prefixes (particularly useful for IP subnets). Arbitrary ranges (e.g., port ranges) can be represented by converting them to sets of prefixes.

In order to verify if a packet matches a bitmask-value pattern, a digest is first generated from the packet by extracting relevant fields from the packet header. Since lower-level protocol headers always define the type of the higher-level protocol header such a digest is always unambiguous. A packet digest D matches a bitmask-value pair (B, V) of some virtual interface if $D \,\&\, B = V$ where & is a bitwise AND operator. Again, note that although the packet digest structure contains a union of overlapping fields, packets sent by different protocols (and hence using different fields in the header) always differ in the type field of the lower-layer protocol and thus the bitmask matching algorithm always classifies them correctly.

In order to dispatch a packet received from hardware to a virtual interface, the packet filter generates a packet digest and iteratively tries to match it against receive patterns of all virtual interfaces associated with the NIC. The first virtual interface that matches the digest receives the packet. If no pattern matches the digest, the packet is dropped (see figure 3).

In a simple system configuration virtual interfaces belonging to the same NIC should not have overlapping patterns. If for some reason processes need to create interfaces with overlapping receive patterns, for example to express some complex filtering rules, the packet filter must know the order in which these patterns are applied. If an incoming packet matches multiple virtual interfaces, the first matching interface receives the packet. Optionally, the packet filter may copy the packet and deliver it to multiple matching interfaces, which allows implementing tools such as network sniffers.

Note that an alternative packet filter design is possible where packet digests are not generated but instead mask-value pairs are directly applied to the packet headers. However, packet headers

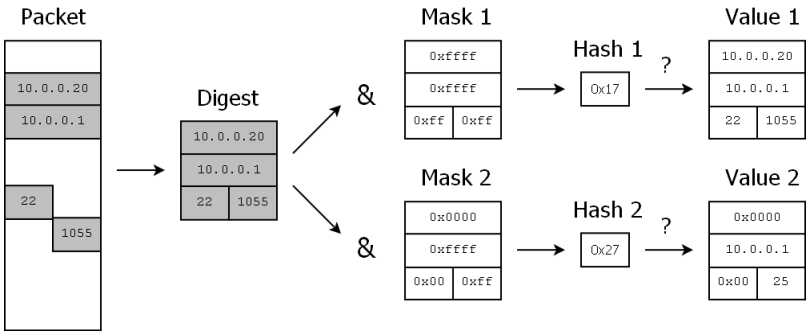


Figure 4: Packet filtering using hash tables

normally contain many fields that are not relevant for packet filtering and thus their comparison would require unnecessary memory accesses. In a typical packet digest definition, such as the one shown in figure 2, a packet digest occupies less than 32 bytes of memory and thus a bitmask-value comparison can be performed by fetching one cache line on most modern architectures. Further, some protocols have variable-length headers (e.g., options) that cannot be classified using simple bitmask-value expressions. On the other hand, generating packet digests requires the kernel to be able to parse protocol headers. We assume though that the kernel needs to have some protocol-specific knowledge for fairness and security reasons.

3.2 Optimizations

The packet demultiplexing algorithm described above requires matching an incoming packet digest with the patterns of potentially all virtual interfaces associated with the NIC. Hence, the algorithm has a linear complexity with the number of virtual interfaces.

One way to improve packet demultiplexing efficiency is to sort the virtual interfaces on the NIC's list based on the number of packets they receive. The packet filter could for example increment a counter in a receiving virtual interface each time it forwards a packet. Occasionally, the packet filter would also re-sort the virtual interface list based on these counters. When processing an incoming packet, the packet filter would then try to match against interfaces that are most likely to receive the packet, increasing the probability of a hit. Given that traffic distributions are often very skewed, such an optimization could yield in a constant amortized filtering time for packets that match interface patterns. However, for packets that do not match any virtual interface (and are thus discarded), the packet filter would still have to iterate over all interface patterns before making a decision and would thus require linear processing time.

Another optimization is based on the observation that many virtual interfaces are likely to have exactly the same bitmasks. For example TCP/IP connections are distinguished by the source and destination IP addresses and source and destination port numbers. Thus, all virtual interfaces corresponding to TCP/IP connections generate the same bitmasks but differ in the associated values (i.e., different port numbers or IP addresses). In order to forward packets efficiently, the packet filter could group all virtual interfaces that share the same bitmask and store their values in a hash table. When processing a packet, the packet filter would calculate the digest, apply a bitmask, calculate a hash from the masked digest, and perform a lookup to check if any value associated with the bitmask matches the digest (see figure 4). Given a sufficient number of bins in the hash table, this filtering method would allow packets to be processed in a constant time per protocol bitmask.

However, pattern matching using hash tables has one limitation. If patterns are allowed to overlap, the order in which patterns are applied to packets may be relevant for the filtering semantics. Grouping patters together using hash tables may thus affect this order. For example, consider a sequence of three patterns that need to be matched in the following order: A, B, and C. Suppose that patterns A and C have the same bitmask and are inserted into a hash table. In such a case, it is impossible to preserve the original pattern matching order because either an A-C hash table lookup is performed before applying pattern B, in which case packets matching both

Trace	Rules	Before hashing		After hashing		Packets
		Bitmasks	Values	Bitmasks	Values	
acl1	751	79	1,246	81	1,250	8,140
acl1_100	98	28	129	28	129	1,000
acl1_10K	9,603	109	12,931	109	13,044	97,000
acl1_1K	916	68	1,222	68	1,222	9,380
acl1_5K	4,415	98	6,109	99	6,139	45,600
fw1	269	221	914	644	3,623	2,830
fw1_100	92	173	302	185	381	920
fw1_10K	9,311	238	32,136	382	998,366	93,250
fw1_1K	791	224	3306	263	3,420	8,050
fw1_5K	4,653	235	15,778	359	22,751	46,700
ipc1	1,550	244	2,180	624	8,329	17,020
ipc1_100	99	54	145	55	146	990
ipc1_10K	9,037	310	12,127	388	16,027	90,640
ipc1_1K	938	186	1,223	190	1,251	9,380
ipc1_5K	4,460	276	5,916	311	8,430	44,790

Table 1: Traces used in the experiments

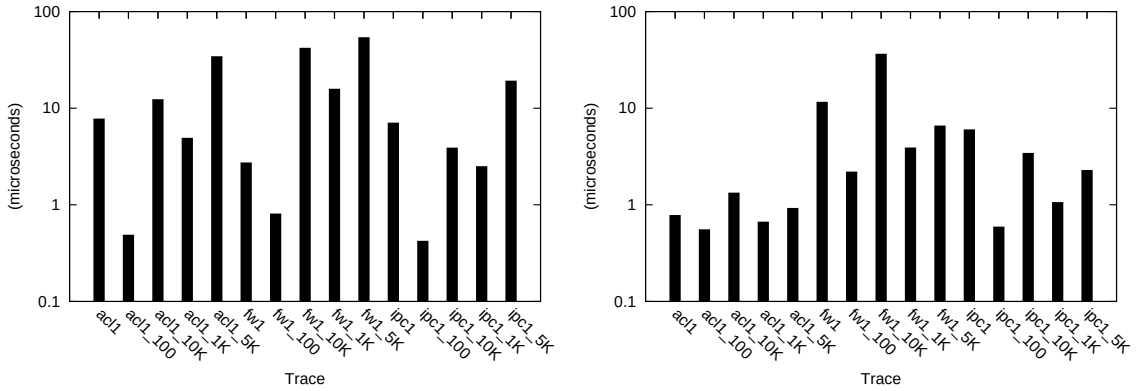
B and C may be classified incorrectly, or pattern B is matched before the hash table lookup, in which case packets matching both A and B may be classified incorrectly.

A solution to this problem is to explicitly add filtering rules for packets that match multiple patterns. In the example above, one could add a new pattern, applied first, for packets that match both A and B. The packet filter would then apply pattern B and at the end it would perform a lookup in the A-C hash table. This solution increases the total number of patterns but it preserves the original filtering semantics.

3.3 User-Kernel Transition

In order to achieve high throughput, the networking code should avoid unnecessary packet copying. While the implementation of the protocol stacks in our framework is entirely application dependent, it is an interesting question whether the kernel can avoid packet copying when sending and receiving packets from the user space. In principle, the kernel can translate the virtual address of the user-space outgoing packet to (possibly multiple) physical addresses and pass them to the NIC for transmission. For security reasons, it might copy the packet header so that the user is not able to modify it. However, some networking cards have limitations on the memory buffers they can use. For example, some NICs can only access a subset of host's memory due to addressing limitations, or have restrictions on the buffer alignment, or do not have a scatter-gather capability and can only transmit packets that are contiguous in the physical memory. If the networking card has one of such limitations, the user either has to allocate packets in a special way (e.g., by asking the kernel) or the kernel has to copy the packet from the user space to a kernel buffer before transmission.

Delivering an incoming packet from the kernel space to the user space without copying is even more problematic. Theoretically it can be done by changing the virtual memory mapping in the receiving process. However, virtual memory can only be modified at a page granularity. If the receiving process requests the packet to be delivered to a specific buffer, both the buffer and the packet payload must be aligned on a page boundary. In order to assure such a packet payload alignment, the kernel would need to know in advance the header length of the packet it is going to receive from the network. We are currently investigating whether it is possible to control incoming packet alignment by assigning MAC addresses or VLAN identifiers to data-intensive network protocols.



(a) Linear pattern matching

(b) Hash table optimization

Figure 5: Average packet demultiplexing time

4 Evaluation

In order to get a preliminary feedback on our framework's feasibility, we implemented the bitmask-based packet filtering algorithm on Plan9 and fed it with traces of packets and filtering rules obtained from a public trace archive [1]. The traces contain three classes of rule sets: Access Control Lists (ACL), Firewall rules (FW) and IP Chains (IPC) summarized in table 1. Columns 2 and 3 show the number of unique bitmasks and values produced from each rule set. Note that due to the range conversion the number of bitmask-value patterns is usually higher than the number of rules in the trace. Furthermore, the number of patterns is increased by the hashing algorithm (columns 4 and 5) which creates new filtering rules for patter intersections in order to preserve the original rule matching order. For some rule sets the number of extra patterns is notably high.

We ran our packet filtering algorithms on an AMD K10 Opteron machine on a single 2.2 GHz core. We first read all the packets from the trace to the main memory and then measured the total time needed to classify all packets. Figure 5 shows the average packet filtering time using a linear pattern matching algorithm and a hash table based algorithm. Clearly, in most cases the hash table optimization improves performance. For most traces, the average packet filtering time is on the order of microseconds. Given a packet size of a few kilobytes, this would allow filtering traffic of approximately one gigabyte per second per core or equivalently 10 gigabits per second per core. Since our prototype is not aggressively optimized (e.g., we do not use SIMD instructions) the throughput may potentially be improved by a more efficient implementation.

5 Conclusions

In this paper we argue that for future applications it will be advantageous to run network protocol stacks in the user space. Many-core machines will need to be able to dispatch incoming packets as quickly as possible to the receiving application cores to maximize performance. We describe a simple user-level networking framework which requires only very limited protocol-specific knowledge in the kernel and allows running entire network protocol stacks in the user space. Measurements on an experimental prototype show that our approach is viable.

References

[1] `http://www.arl.wustl.edu/ hs1/PClassEval.html`.

[2] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. Pathfinder: A pattern-based packet classifier. In *OSDI*, pages 115–123, 1994.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177. ACM, 2003.

[4] P. Buonadonna, A. Geweke, and D. Culler. An implementation and analysis of the virtual interface architecture. In *ACM/IEEE Supercomputing*, pages 1–15, 1998.

[5] C. Dubnicki, L. Iftode, E. W. Felten, and K. Li. Software support for virtual memory-mapped communication. In *International Parallel Processing Symposium*, pages 372–281. IEEE, 1996.

[6] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 18:66–76, 1998.

[7] D. R. Engler and M. F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM*, pages 53–59. ACM, 1996.

[8] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15:24–32, 2001.

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Linux Symposium*, 2007.

[10] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM*, pages 203–214. ACM, 1998.

[11] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *USENIX Winter Conference*, pages 2–12, 1993.

[12] J. Mogul, R. Rashid, and M. Accetta. The packer filter: an efficient mechanism for user-level network code. In *SOSP*, pages 39–51. ACM, 1987.

[13] S. Pakin, V. Karamcheti, and A. A. Chien. Fast messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Parallel & Distributed Technology*, 5:60–73, 1997.

[14] L. Prylli and B. Tourancheau. BIP: A new protocol designed for high performance networking on myrinet. In *IPPSSPDP Workshops*, pages 472–485, 1998.

[15] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet classification algorithms: From theory to practice. In *INFOCOM*, pages 648–656, 2009.

[16] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *SIGCOMM*, pages 135–146. ACM, 1999.

[17] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1:554–565, 1993.

[18] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *SOSP*, pages 40–53. ACM, 1995.

[19] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, pages 256–266. ACM, 1992.

[20] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX Winter Technical Conference*, 1994.