Networking in Osprey

Jan Sacha
Sape Mullender

Alcatel-Lucent Bell Labs
Antwerp, Belgium

Abstract

Network links have more and more bandwidth while processor frequencies do not increase significantly and thus the best way to improve networking performance is to process packets in parallel on multicore machines. This paper describes a networking architecture where incoming network traffic is demultiplexed to user-level network protocol stacks running on different cores using a software packet filter and multiple hardware receive rings. Such architecture allows efficient use of the network interface controller, processor caches, and memory, enabling very efficient and scalable networking. This architecture has been implemented in the Osprey operating system.

1 Introduction

Networking stacks in many operating systems were designed when networks were slow and machines typically had a single CPU with one core. As a consequence, the networking code often has a monolithic, centralized structure where concurrent access is guarded by locks. However, hardware realities are changing. Network links have increasingly high bandwidth and require more and more computing power to process packets. Since processor frequencies do not increase significantly, but the core count per machine grows quickly, the most straight-forward way to improve networking performance is thus to process packets in parallel.

Parallelizing the networking stack is challenging, however, because the networking code can be invoked concurrently from multiple different contexts: User applications pass their requests to the stack using system calls; Packets asynchronously arrive from the network and are typically signaled by interrupts; Finally, protocols such as TCP require timers, which again are delivered asynchronously. Accessing shared networking state from multiple CPU cores is expensive due to the cache coherency semantics. Further, locks prevent parallel execution and might waste a significant fraction of CPU cycles in case of contention.

Hence, in order to achieve high throughput, the networking stack should be organized so that cores share and synchronize as little as possible. A few novel architectures have been proposed recently to reach this goal. In the IsoStack [9] and netmap [6] frameworks, the networking functionality is delegated to a single process which serializes requests to avoid locks and cross-core sharing. In NewtOS [3], cores specialize at performing particular functions, such as filtering packets, running IP, or TCP. In the Affinity Accept design [5], network traffic is divided between symmetrically running cores using hardware support so that each packet is processed on one core only. This latter approach has been shown to allow very high throughput and scalability.

In this paper, we describe an architecture where incoming packets are demultiplexed in software using a packet filter and delivered to a protocol stack running in the application's address space. Such a design does not require hardware support but allows processing packets in parallel with very little inter-core sharing. We describe optimizations in which applications can own private Ethernet buffer rings and use hardware scatter-gather capabilities to avoid memory copying, increasing overall throughput. Our architecture is implemented in the Osprey operating system [7] but we believe it can be ported to other operating systems as well.

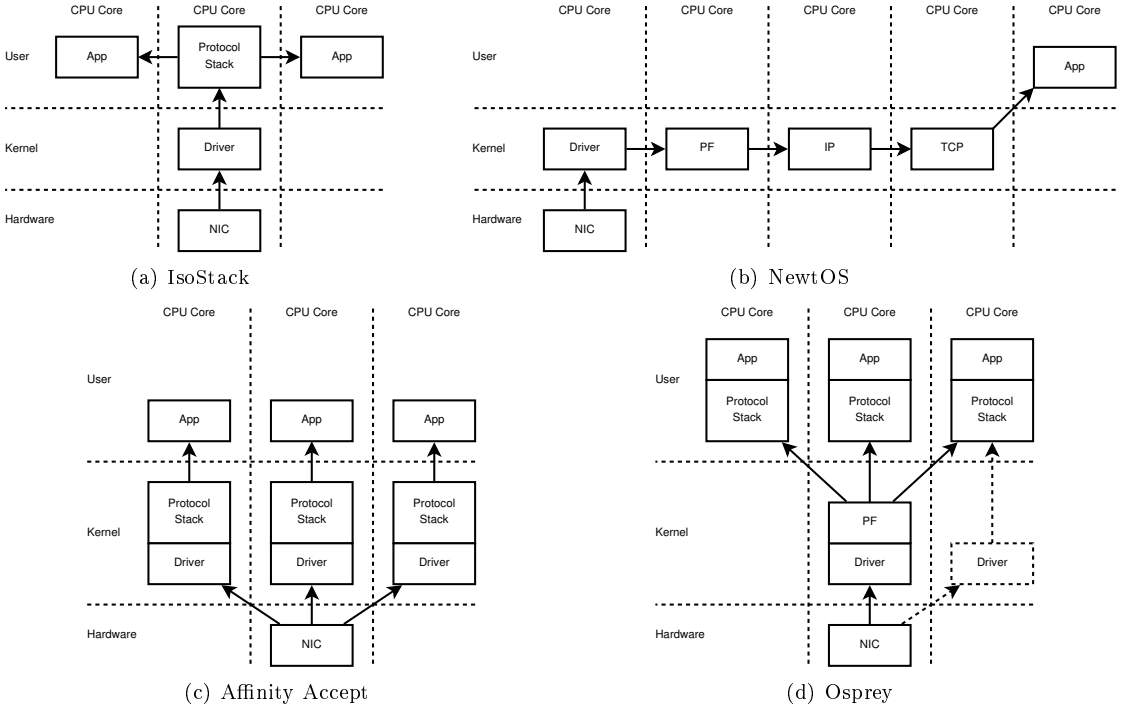


Figure 1: Networking stack organization in sample operating systems.

2 Related Work

The earliest approaches to high-speed networking came from the area of high-performance computing (HPC) where high-bandwidth and low-latency networks, such as MyriNet or InfiniBand, were used to connect powerful compute nodes. Since general-purpose networking stacks in commodity operating systems turned out to be too slow to handle traffic generated in these systems, new networking architectures were developed. HPC applications typically were allowed to interact directly with the network interface controller to bypass kernel abstractions and reduce packet processing overhead. These approaches often provided very little application isolation and security because all hardware was usually owned and controlled by a single HPC application [11, 4, 1].

Today, high-speed networks such as 10G Ethernet become commodity and support for them is added to general-purpose operating systems such as Linux. Figure 1 summarizes a few networking architectures that allow fast packet processing.

In IsoStack [9], the networking stack runs in a single process which serializes events coming from user applications (via asynchronous message queues), the network interface, and the operating system (timers). Such an arrangement allows IsoStack not to use locks and minimizes intercore sharing, allowing efficient use of CPU caches. However, the IsoStack approach does not scale because it can utilize only one core. Similarly to the IsoStack, Osprey runs networking stacks in user processes, but it divides incoming traffic using a packet filter and runs multiple stack instances in parallel to take advantage of multiple cores available in hardware.

NewtOS [3] partitions the networking stack based on functions. Each functional component, such as a device driver, packet filter, IP implementation, and TCP engine, runs on a different core. Network packets are processed by multiple cores in a pipeline. Since cores share very little state and communicate using asynchronous message queues, they can run in parallel. However, this design increases packet processing latency, as every packet traverses multiple cores, and is not able to utilize more cores than the number of networking stack components.

In the Affinity Accept framework [5] all cores perform the same function but network traffic is multiplexed between them using multiple hardware receive queues. Each core maintains a fraction of the networking state using lock-free data structures. Heuristics, such as packet stealing, are developed to match incoming packets with receiving applications and to balance the load between

CPU cores. This approach scales very well and hence enables high throughput. Similar to Affinity Accept, Osprey uses hardware receive queues to multiplex traffic, but it also allows multiplexing in software using a packet filter. Further, Osprey runs protocol stacks in the user space for benefits such as simplified memory allocation, application checkpointing, restarting, and migration.

Netmap [6] is a framework which allows an application to send and receive packets from a network interface very efficiently. It allocates all packet buffers statically at initialization time and maps them into the application's address space to avoid copying. The application receives all network packets from an Ethernet hardware ring and returns empty buffers. The kernel does not multiplex traffic. In Osprey, we use a similar technique to optimize key applications, such as file servers and clients, which can own private Ethernet rings.

Operating systems can improve networking performance by offloading some functions to the network interface controller (NIC). Such offloading can usually be combined with the techniques described above. Modern NICs usually provide checksum calculation and verification, interrupt throttling, header separation, and TCP packet splitting and coalescing. These latter capabilities are known as Large Send Offload (LSO) and Large Receive Offload (LRO) or TCP Segmentation Offload (TSO).

Finally, modern NICs support multiple transmit and receive rings (also called queues) where incoming packets are classified and assigned to the rings based on their source and destination addresses. For example, Intel's Virtual Machine Device queue (VMDq) technology, intended for hypervisors running multiple operating systems, assigns packets to rings based on their destination Ethernet address. Another technology, known as Receive-Side Scaling (RSS), uses a hash from the incoming packet's 5-tuple to select the receive ring. Each ring uses its own buffers and requests interrupts independently, allowing packets to be processed on multiple cores in parallel. Further, packets belonging to the same connection are mapped to the same ring and are hence processed on the same core, improving CPU cache performance.

3 Architecture

Osprey's networking architecture assumes that every network interface has at least one hardware receive (RX) ring and one transmit (TX) ring. Receive ring zero can be shared by multiple applications and is multiplexed by a packet filter. Other receive rings, if available in hardware, can be owned by applications in a way similar to netmap: the application receive all incoming packets with no multiplexing in the kernel, no memory copying, and very little overhead.

Osprey uses only one transmit ring to send all outgoing packets. It does not use multiple transmit rings so that it can decide itself on the order in which packets are enqueued for transmission. If it used multiple hardware transmit rings it would have to let the NIC decide which packets are transferred from the rings to the wire.

In Osprey, every application has its own protocol stack which runs in the user space. Similar to the IsoStack, an Osprey protocol stack can serialize events from the application and kernel, such as packet transmissions, arrivals, and timeouts, in order to work efficiently with no synchronization overhead. The protocol stack runs together with the application—in the same address space and on the same core—which enables zero-copy communication and efficient use of CPU caches. Keeping the networking state in the user space also facilitates application checkpointing, restarting, and migration. From the kernel's view point, the network protocol stack is just part of the application's logic. However, in contrast to IsoStack, every Osprey process can have its own protocol stack, which allows the system to scale better. The only centralized components in the networking architecture are the device drivers and the packet filter.

Osprey inter-process, inter-task (inside the kernel), as well as user-kernel communication is provided by asynchronous message queues. Queue implementation details fall beyond the scope of this paper, however, we mention here that the queues are based on fixed-size shared-memory buffers and fixed-sized cache-aligned messages (64 byte long currently) which are copied on both ends during send and receive operations. Messages typically contain pointers to larger structures such as packet buffers. Asynchronous communication reduces the number of context switches and hence improves performance. In many ways, Osprey's system call API is similar to FlexSC [10].

The asynchronous networking API consists of just a few message types. In order to manage a

```
interrupt_handler()
{
  disable_nic_interrupts();
  send_message(driver, interrupt);
}

transmit(packet)
{
  send_message(driver, packet);
}

driver_main()
{
  loop {
    message = receive_message();
    switch(message.type) {
    case interrupt:
      while(tx_complete())
        send_message(protocol_stack, send_ack);
      ...
      enable_nic_interrupts();
      break;
    case packet:
      tx_ring_append(message);
      break;
      ...
    }
  }
}
```

Figure 2: Driver task pseudocode.

hardware receive ring, an application sends a **ringattach** message containing an interface number and a ring number. Similarly, to use the shared ring zero, the application sends a **netattach** message containing an interface number and a bitmask describing packets that the application needs to receive.

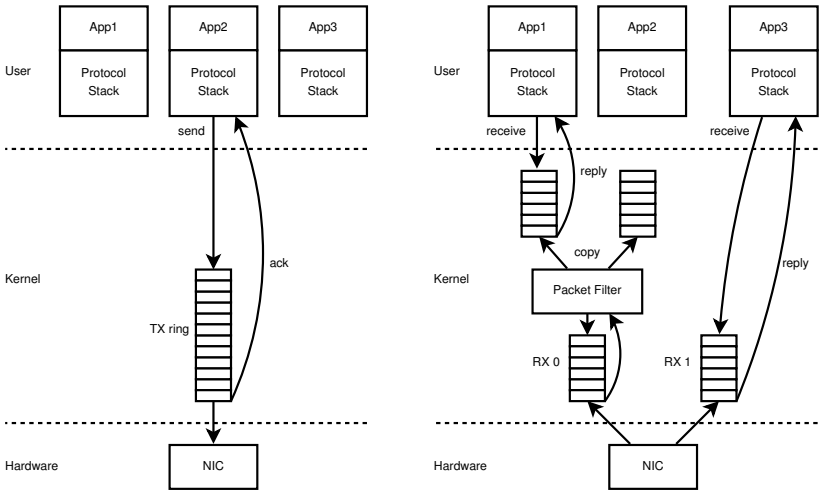
An application sends a packet by issuing a **send** message containing a pointer to the packet buffer and a length. When packet transmission is complete, the kernel sends an acknowledgment message to the application. Similarly, to receive a packet, an application sends a **receive** message to the kernel containing a pointer to an empty buffer and a length. When a packet arrives from the network, the kernel—either the packet filter or the driver managing a private hardware ring—returns a buffer containing packet data. Importantly, the message format is the same for all receive rings, so that applications can easily switch between using shared ring zero and private hardware rings.

Finally, although the networking API is asynchronous, we note that it is very easy to build a synchronous API on top of it. For a synchronous send, an application generates a **send** message and waits for a matching reply, and similarly, for a blocking receive, the application generates a **receive** message and waits for a corresponding reply.

3.1 Device Drivers

Every network interface has its own driver task which communicates with the controller using ports and memory-mapped registers. The driver serializes events coming from hardware and applications to execute efficiently in a lock-free manner. The interrupt handler is trivial. It sends a message to the driver and disables further interrupts from the interface. The driver, upon receiving an interrupt message, inspects the hardware, updates the necessary state, and if a packet transmission or reception is complete, generates a message to the process or kernel task (packet filter) owning the respective ring. When hardware maintenance is finished, the driver enables interrupts again and blocks waiting for a message (see Figure 2).

Similarly, the driver accepts messages coming from user processes and kernel tasks. Upon



(a) Sending (b) Receiving

Figure 3: Osprey's networking architecture overview.

receiving a packet transmission request, the driver appends the packet to a TX ring, and upon receiving a packet receive request the driver appends an empty buffer to an RX ring.

In the current Osprey implementation, every NIC has a single driver task which maintains one TX ring and all available RX rings. However, it is possible to further parallelize the architecture by running a separate driver task per each hardware ring. Since many NICs allow rings to signal interrupts to different cores, drivers tasks could run truly in parallel and service rings independently, improving system scalability on multicore hardware.

3.2 Packet Transmission

Outgoing packets normally are allocated and filled in by the protocol stack in the user space. Kernel tasks are allowed to generate network packets but Osprey rarely uses this possibility. Once a packet is ready to be transmitted, the protocol stack sends a message containing a pointer to the packet body to the driver task managing a chosen network interface (see Figure 3(a)).

The driver does not need to copy the packet into the kernel space but must verify that memory is mapped in and must pin pages to make sure that they are not freed during packet transmission. Pinning pages in Osprey is simple because memory is never swapped out to disk. Further, the kernel verifies that the packet is legal. Since the networking API is asynchronous, the application is allowed to run during packet transmission, and hence could potentially modify the packet after it has been verified by the kernel. To prevent such a possibility, the kernel copies and transmits the header to its own memory.

When packet transmission is complete, the device driver sends a reply message to the process owning the packet. The protocol stack, upon receiving this message, frees the packet buffers and cleans up its internal state. If the user application uses a synchronous networking API, the thread blocked on transmission is woken up.

Older NICs might impose constraints on the packet layout in memory. For example, some NICs require a specific memory alignment, or are able to access only a subset of the address space, or require packets to be contiguous (i.e., do not support scatter-gather). If user-space libraries do not allocate packets in the way expected by these NICs, Osprey falls back to a compatibility mode where the entire packet is copied to the kernel space before transmission.

We are planning to extend the networking architecture to support resource usage management. In the future, the kernel will limit the number of packets and bytes a process can send per time unit. When these constraints are not met, the kernel (e.g., device driver) will delay packet transmission. Further, the kernel will append outgoing packets to the TX ring according to a policy, for example based on process priority or deadline, to allow low-latency network access to privileged processes.

3.3 Packet Reception

In order to receive network packets, a protocol stack needs to provide empty receive buffers to the networking infrastructure (see Figure 3(b)). These buffers are allocated in the user space, and similarly to outgoing packets, must be verified by the kernel. In particular, memory must be mapped in and pages need to be pinned so that buffers are not freed before or during packet reception. Additionally, buffers must be large enough to fit maximum-size packets.

Typically, when a protocol stack is initialized, it allocates a number of receive buffers and sends them to the network interface. Further, each time the protocol stack receives an incoming network packet, it provides a new empty buffer to the interface. By maintaining a fixed number of buffers on the receive ring—either hardware ring managed by the NIC driver or virtual ring managed by the packet filter—the protocol stack reduces the risk of running out of buffers and losing incoming packets.

3.3.1 Shared ring zero

Ring zero is special because it is owned by a kernel task—the packet filter—which multiplexes it between user applications. Such multiplexing is necessary because NICs usually provide a limited number or rings, often only one, and thus may not have enough receive rings for all applications. Further, enabling hardware rings introduces a cost which might be undesired for lightweight or short-lived applications.

Osprey's packet filter classifies packets based on their headers. For each incoming packet, the packet filter extracts key fields from the header, such as protocol types, source and destination addresses, and port numbers, and matches them against bitmasks provided by user applications. The details of the packet classification algorithm and its implementation are described in a separate paper [8].

When the packet filter determines the receiving application, it checks if an user-space buffer is available. If the virtual ring is empty, the packet is dropped. If a user buffer is available, the packet is copied into the user space and a message is sent to the application's protocol stack. The original kernel buffer is always returned to hardware ring zero and reused. This way, the packet filter allocates receive buffers only once—at initialization time.

It is worth noting that currently existing NICs do not have sufficient functionality to implement a packet filter entirely in hardware. In particular, the Intel VMDq technology allows only very simple filtering based on the packet's destination Ethernet (MAC) address, and the RSS technology uses 5-tuple hashes and hence may map packets belonging to different applications to the same ring, requiring further demultiplexing in software. Besides, as already mentioned, most NICs support only a limited number of RX rings.

3.3.2 Private rings

To take advantage of hardware support, Osprey allows applications to use private receive rings. We associate every RX ring with its own MAC address and use the NIC to demultiplex incoming packets. Private rings allow much more efficient networking than the shared ring zero. They do not require software multiplexing, since the application receives all incoming packets, and allow the application to put user-level receive buffers directly on the hardware ring, enabling zero-copy receive.

However, using private receive rings has an impact on the application. First of all, every application owning a private ring must obtain its own IP address—or use a non-IP protocol, such as ATA over Ethernet (AoE). Secondly, Ethernet broadcast packets have a destination address of FF:FF:FF:FF:FF:FF, and hence are all delivered to the same RX ring. In Osprey, we configure the NICs so that broadcast packets are received by ring zero. As a consequence, it is not possible to run protocols such as DHCP or ARP on a private ring because these protocols rely on Ethernet broadcast. To work around this limitation, Osprey kernel handles ARP lookups and runs DHCP on ring zero on behalf of applications that own private RX rings. Given that both ARP and DHCP are critical to network security, we do not view this as a drawback. Most applications should not be allowed to generate ARP and DHCP replies nevertheless.

```
struct Msgpacket {
  void    *addr[4];
  ushort  len[4];
  ushort  flags;
};
```

Figure 4: Message format for incoming and outgoing packets.

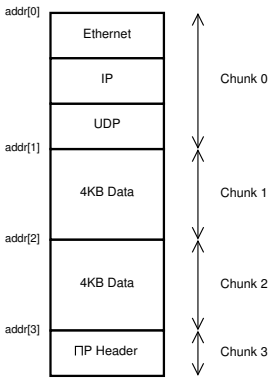


Figure 5: ΠP packet layout.

3.4 Scatter-Gather

So far we have assumed that packet buffers are contiguous in memory and hence can be referenced using a single pointer variable. However, to give more flexibility in packet allocation to the protocol stacks, Osprey supports packets which consist of multiple disjoint chunks of memory. This feature allows for example TCP implementations to combine multiple pieces of data from the user space into a single packet without copying memory.

Messages representing packets have a format shown in Figure 4. A single message can store up to 4 pointers to packet chunks and 4 corresponding lengths. The limit of 4 chunks is imposed by the maximum message size in Osprey, which is currently 64 bytes. The additional `flags` field is used by the device driver to indicate to the protocol stack which checksums have been calculated and verified (e.g., Ethernet CRC, IP header, TCP, or UDP).

Sending and receiving packets consisting of multiple memory blocks requires scatter-gather capabilities in hardware, which most modern NICs already provide. On older hardware, Osprey can always copy fragmented packets into consecutive buffers to overcome NIC limitations.

Modern NICs are also able to parse incoming packets and split them into separate buffers containing the headers and payload. In Osprey, we are planning to use this feature to implement a zero-copy network file protocol called ΠP. In this protocol, a packet consists of network protocol headers, such as Ethernet, IP, and UDP, a ΠP header (known as the operations), and data. Our plan is to align the data buffer on a page boundary so that it can be transferred to an application's address space by updating memory maps. While NICs usually recognize common network protocols, such as IPv4, UDP, and NFS, they regrettably do not support ΠP. However, to make sure that ΠP data is properly aligned, we use a packet wire representation shown in Figure 5. The ΠP header is written at the end of the packet. When the network headers are split off in hardware, data is stored at the beginning of the payload buffer which can be aligned on a page boundary.

4 Status

The networking architecture described in this paper has been mostly implemented in Osprey. We have implemented drivers for a number of Intel NICs including the 82598 10GbE, 82576 1GbE, and older models, as well as the portable kernel components such as the packet filter.

We have also developed a user-space library which provides memory buffer management and a synchronous (blocking or non-blocking) API on top of asynchronous message transactions. We used this low-level library to implement a number of basic network protocols, such as DHCP, ARP, Ethernet, IPv4, and UDP. Finally, we have ported the Lightweight IP library [2] to provide a fully functional user-level TCP/IP stack for Osprey applications.

5 Conclusion

This paper describes a networking architecture where incoming network traffic is demultiplexed using a software packet filter and hardware receive rings to multiple network protocol stacks running in the user space concurrently on different cores. Such architecture allows efficient use of network interface controllers, processor caches, and memory, enabling very efficient and scalable networking. The architecture has been implemented in Osprey. We are planning to run a number of experiments to measure, understand, and validate this architecture's performance.

References

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

[2] A. Dunkels. Design and implementation of the lwip TCP/IP stack, Feb 2001.

[3] T. Hruby, D. Vogt, H. Bos, and A. S. Tanenbaum. Keep net working - on a dependable and fast networking stack. In *Dependable Systems and Networks*, pages 1–12. IEEE, 2012.

[4] 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.

[5] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *EuroSys*, pages 337–350. ACM, 2012.

[6] L. Rizzo. Revisiting network I/O APIs: The Netmap Framework. *Communications of the ACM*, 55(3):45–51, Mar. 2012.

[7] J. Sacha, J. Napper, S. Mullender, and J. McKie. Osprey: Operating system for predictable clouds. In *Proceedings of the Dependable Systems and Networks Workshops*, pages 1–6. IEEE, 2012.

[8] J. Sacha, J. Napper, H. Schild, and S. Mullender. Revisiting user-level networking. In *Proceedings of the 6th International Workshop on Plan 9*, pages 17–24, 2011.

[9] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack: highly efficient network processing on dedicated cores. In *USENIX Annual Technical Conference*, pages 5–5, 2010.

[10] L. Soares and M. Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *OSDI*, pages 1–8. USENIX Association, 2010.

[11] 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.