Developing Virtual Network Functions

Introduction

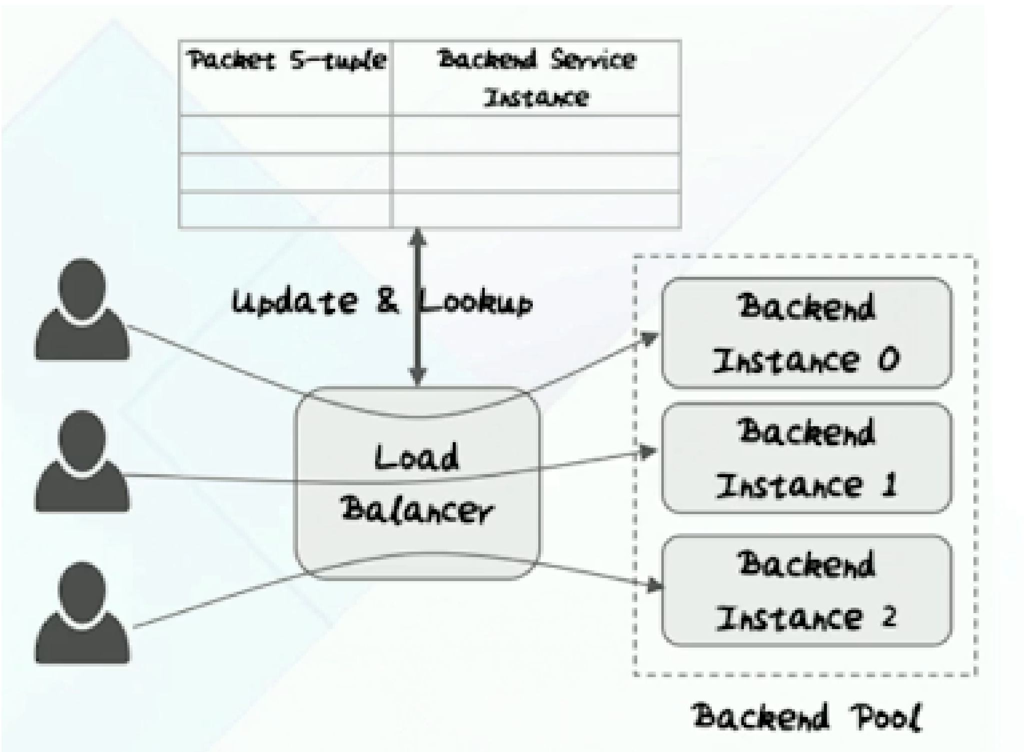
1. In-depth look at virtual network functions
    - How do we implement performance-conscious network functions?

Developing Virtual Network Functions

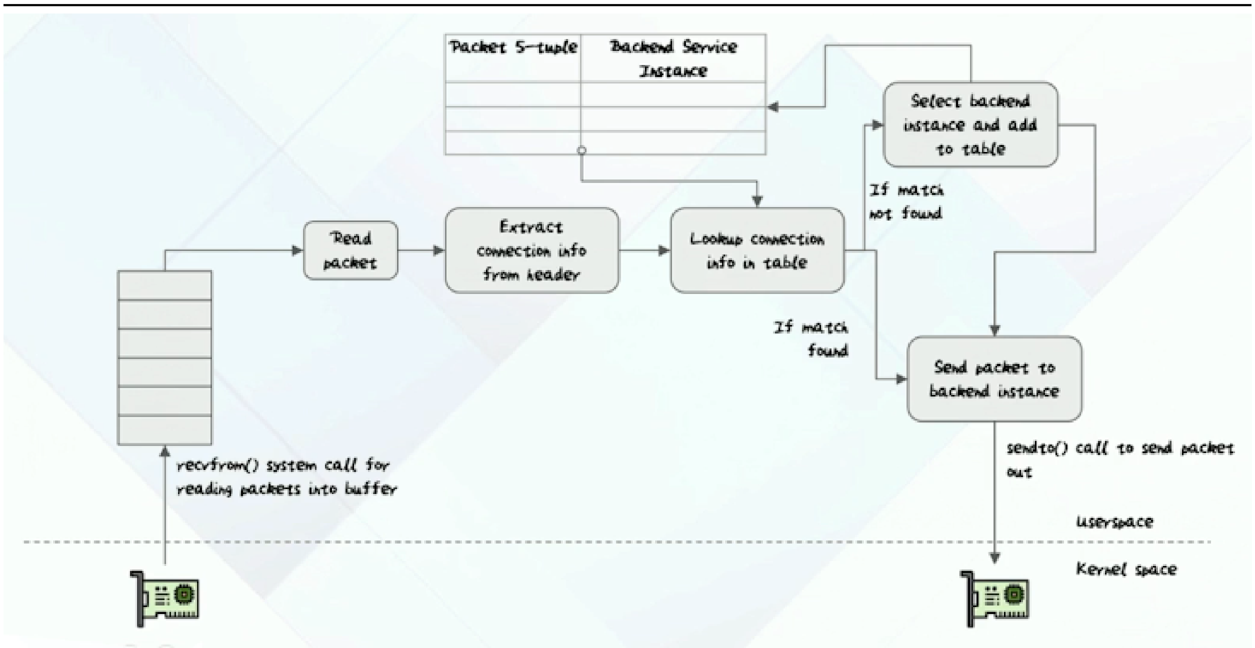
1. VNF: Virtual Network Function
    - Use concrete examples to understand how VNFs are implemented
2. DPDK: Data Plane Development Kit
    - User-space packet processing

Virtual Network Functions

1. Overview
    - Network function implemented in user-space on top of hypervisor
        - For portability
    - Load balancer as a concrete example
        - Keeps a pool of backend service instances (e.g., HTTP server)
        - Distributes incoming packet flows to a specific instance to exploit inherent parallelism in the hardware platform and balance the load across all the service instances
2. Architecture of a load balancer network function
    - Distribute client connections to a pool of backend service instances
        - For example HTTP server
    - Use packet's 5-tuple to choose backend instance
        - Source address, destination address, source port, destination port, protocol
        - Provides connection-level affinity
        - Same connection is sent to same backend instance
    - Most network functions are implemented on top of Linux



Load Balancer Architecture



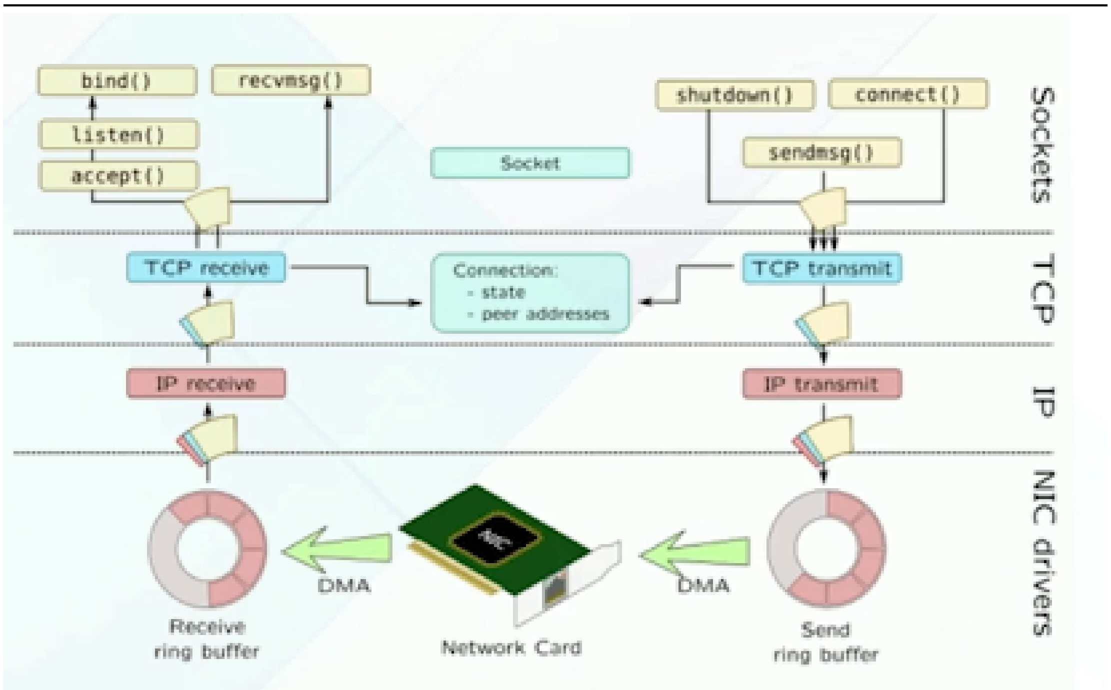
Load Balancer Diagram

Performance Issues in Implementing VNF

1. Eliminating the overhead of virtualization

- Network function is in the critical path of packet processing
- Need to eliminate the overhead of virtualization
  - Intel VT-d allows the NIC to bypass the VMM (hypervisor) by direct- mapping user-space buffers for DMA and passing the device interrupt directly to the VM above the VMM
- Is that enough?
  - Unfortunately no.
  - Let's look at the path of packet processing in an OS like Linux

2. Packet processing in Linux
   - NIC uses DMA to write incoming packet to a receive ring buffer allocated to the NIC
   - NIC generates in interrupt which is delivered to the OS by the CPU
   - OS handles the interrupt, allocates kernel buffer and copies DMA'd packet into the kernel buffer for IP and TCP processing
   - After protocol processing, packet payload is copied to application buffer (user-space) for processing by the application



Linux Packet Processing

3. An example networking app on Linux Kernel
   - A web server on Linux
   - 83% of the CPU time spent in the kernel
   - This is not good news even for a networking app such as a web server
   - This is REALLY BAD news if the app is a network function
4. Performance hits
   - One interrupt for each incoming packet
   - Dynamic memory allocation (packet buffer) on a per packet basis
   - Interrupt service time
   - Context switch to kernel and then to the application implementing the NF
   - Copying packets multiple times
     - From DMA buffer to kernel buffer
     - Kernel buffer to user-space application buffer
     - Note that a NF may or may not need TCP/IP protocol stack traversal in the kernel depending on its functionality

Performance-conscious Implementation of VNF

1. Circling back to Virtualizing Network Functions
    - Intel VT-d provides the means to bypass the hypervisor and go directly to the VM (i.e., the guest kernel which is usually Linux)
        - NF is the application on top of Linux
    - Guest kernel presents a new bottleneck
        - Specifically for NF applications
        - Sole purpose of NF applications is to read/write packets from/to NICs
    - Slowdown due to kernel is prohibitive
        - Demand scaling broke down in 2006 (leakage current started dominating)
        - CPU clock frequencies are not increasing significantly from generation to generation
        - CPU speeds are not keeping up with network speeds
        - NICs can handle more packets per second -> increasing pressure on CPU
    - So it is not sufficient to bypass the VMM for NF virtualization
        - We have to bypass the kernel as well
2. Performance-conscious Packet Processing Alternatives
    - By-passing the Linux kernel
        - Netmap, PF_RING ZC, and Linux Foundation DPDK
    - These alternatives possess common features
        - Rely on polling to read packets instead of interrupts
        - Pre-allocate buffers for packets
        - Zero-copy packet processing: NIC uses DMA to write packets into pre-allocated application buffers
        - Process packets in batches

Data Plane Development Kit

1. DPDK was developed by Intel in 2010
    - Now an open source project under Linux Foundation
    - Libraries to accelerate packet processing
    - Targets wide variety of CPU architectures
    - User-space packet processing to avoid overheads of Linux kernel
2. Features of DPDK
    - Buffers for storing incoming and outgoing packets in user-space memory
        - Directly accessed by the NIC DMA
    - NIC configuration registers are mapped in user-space memory
        - PICe configuration space
        - Can be modified directly by user-space application
    - Effectively bypasses the kernel for interacting with NIC
3. DPDK is a user-space library
    - Very small component in the kernel
        - Used for initialization of user-space packet processing
    - Needed to initialize the NIC to DMA to appropriate memory locations
    - Setup memory mapping for configuration registers on the NIC
        - PCI configuration space
        - Updating those registers is then done in user-space
4. Poll Mode Driver
    - Allows accessing receive (rx) and transmit (tx) queues
    - Interrupts on packet arrival are disabled
    - CPU is always busy polling for packets even if there are no packets to be received
    - Receive and transmit in batches for efficiency

```
while(true) {
    buff <- bulk_receive(in_port)
```

```
    for pkt in buff:
        out_port <- look_up(pkt_header)
        # handle failed lookup somehow
        out_buffs[out_port].append(pkt)
        for out_port in out_ports:
            bulk_transmit(out_buffs[out_port])
}
```

NIC Ring Buffer

1. Overview
    - Each NIC queue is implemented as a ring buffer (not specific to DPDK)
    - Each slot in the ring buffer holds a "descriptor" for a packet
        - Descriptor contains a pointer to the actual packet data and other metadata
        - Actual packet is stored in another buffer data structure
    - Write pointer is advanced when NIC receives packets
    - Read pointer is advanced when CPU reads packets
2. NIC ring buffer
    - Upon packet arrival, NIC populates the next vacant slot with packet's descriptor
    - CPU core running NF polls ring for unread slots
    - When new descriptors are found
        - CPU reads the packet data for those descriptors
        - Returns packets to application
    - No need for locking: producer and consumer are decoupled in ring buffer
    - If no vacant descriptor slots in ring buffer, NIC drops packets
3. Pre-allocated buffers for storing packets
    - Instead of allocating a buffer for each incoming packet, DPDK preallocates multiple buffers on initialization
    - Each rx queue in the NIC can hold no more packets than the capacity of the ring buffer
        - Total size of packet buffers is thereby known = capacity of ring
    - Incoming packet is DMA'd into the buffer along with adding new packet descriptor to ring buffer
    - DPDK uses huge pages to maintain large pools of memory
        - Each page is 2 MB in size (compared to traditional 4 KB pages)
        - Fewer pages -> fewer TLB misses -> improved performance
4. No overhead of copying packet data
    - NIC DMA transfers packets directly to user-space buffers
    - Protocol processing (TCP/IP) is done using those buffered packets in place
        - ...if needed by the network function (Note: not all NFs require TCP/IP processing)
5. Upshot of NF using DPDN and Intelligent NICs
    - All the kernel overheads in packet processing (alluded to earlier) mitigated/eliminated
    - Results in performance-conscious implementation of the VNF
    - Developer of NF can concentrate on just the functionality of the NF
        - DPDK alleviates all the packet processing overheads for any NF

Implementation of VNF

1. DPDK Optimizations
    - Various optimization opportunities are available in DPDK to improve packet processing
    - Each optimization attempts to eliminate a particular source of performance drop in Linux kernel and/or exploitation of hardware features in NICs and modern CPUs
2. Implementing NFs using DPDK on commodity hardware
    - Modern commodity servers contain multi-core CPUs
    - Using multiple cores for packet processing can allow us to match the increasing capacities of NICs
    - NUMA servers

- Multiple sockets each with given nubmer of cores and local RAM
- Accessing remote RAM is much more expensive than local RAM
- Upshot
  - Need to carefully design the packet processing path from NIC to NF taking these hardware trends into account
  - Partnership between system software and hardware

3. DPDK application model
   - Run-to-completion model
     - Polling for incoming packets, processing on packet, and transmission of output packet all done by the same core
     - Each packet is handled by a unique core
   - Pipelined model
     - Dedicated cores for polling and processing packets
     - Inter-core packet transfer using ring buffers
4. Run-to-completion model
   - All cores responsible for both I/O and packet processing
   - Simplest model
   - Each packet sees only one core
     - Works for monolithic packet processing code
     - When all the packet processing logic is contained inside a single thread
     - Simple to implement but less expressive
5. Pipelined execution model
   - Dedicate cores for processing NF logic
   - Some cores are dedicated for reading packets
   - Each packet sees multiple cores
     - Can be used to chain multiple packet processing logics (within an NF)
     - e.g., IN -> firewall -> router -> OUT
   - Inter-core communication done using queue buffers in memory
   - Also useful when packet processing is CPU bound, so having number of polling cores < number of processing cores is a good choice
     - Intrusion Detection System

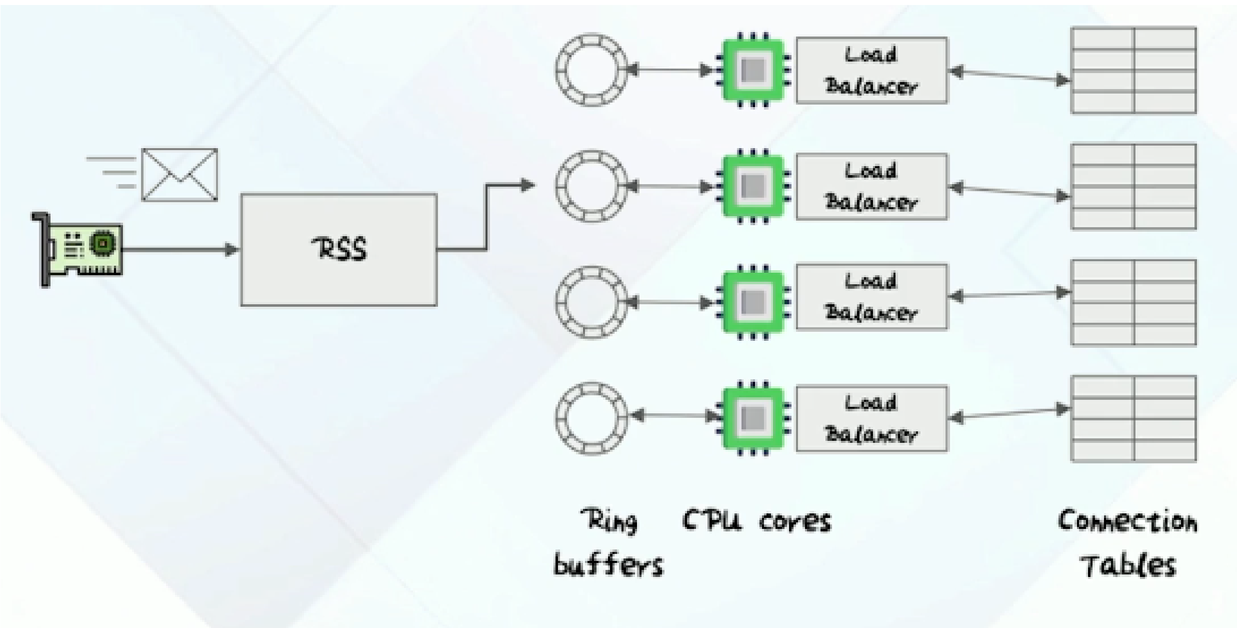
Multi-core Implementation Challenges

1. Overview
   - How to ensure that processing done by two distinct cores don't interfere with each other?
     - If different packets of same connection are sent to different cores, sharing NF state will be a nightmare
   - How to ensure that cores participating in inter-core communication are on the same NUMA node?
   - How to ensure that the cores processing packets are on the same NUMA socket as the NIC?
2. Receive side scaling: Useful hardware technology
   - Enabler of multi-core processing
   - Use hashing to distribute incoming packets to individual cores
     - Hash function takes 5-tuple of packet as input
     - src_ip, dst_ip, src_port, dst_port, proto
   - Each core is assigned a unique ring buffer to poll
     - No contention among threads
   - Different connection -> different queue (ring) -> different core
     - Per-connection state is accessed only by a single core, so state management is easy
3. Multi-core support in DPDK
   - Allows admin to specify the following (hardware/software partnership):
   - Allows mapping of specific RX queue to specific CPU core
     - Port 0 -> Rx queue 1 -> CPU core 6
     - CPU core 6 -> Port 1 -> Tx queue 2

    - Flexible to create as many queues as admin wants
  - Each thread is pinned to a specific core
    - To avoid contention
  - Each thread/core runs the same code
4. NUMA awareness in DPDK
  - DPDK creates memory pools for inter-core communication on the same NUMA socket as the cores involved
  - Ring buffers are allocated on the same socket as the NIC and cores selected for processing
  - Remote memory access is minimized

Putting it Together

1. Load balancer application
  - Multi-threaded, run-to-completion model
    - Each thread performing identical processing
  - Dedicated Rx and Tx queues for each core



Scalable Load Balancer

2. Architecture
  - Incoming packet is directed to a particular read buffer
  - Each thread within a core is doing the following:
    - Receive a packet from the NIC
    - Read packet data for new descriptors
    - Extract connection info from header
    - Lookup connection info in table
    - If match, send packet to backend instance
    - If not match, select backend instance and add to table
  - Ring buffer logic is handled by DPDK
    - Application handles load balancing specifics

Conclusion

1. State of the art: Implement network functions on top if Linux kernel

- General trends for accelerating packet processing to alleviate bottlenecks
  - Mechanisms for bypassing kernel
- Used DPDK as an exemplar
  - Explored how DPDK handles multi-core processors
- Enables very performance-conscious user-space implementations of network functions