Developer Manual for NETMAP based network I/O for application layer VNFs

This manual provides design and implementation details of packet distribution mechanism based on NETMAP for application layer VNFs. Please read **user_manual.pdf** for setup and usage of NETMAP based network I/O.

Design

Design of a multi-core VNF application over kernel bypass mechanism such as netmap requires some packet distribution strategy to distribute network traffic among multiple cores for scalability. The distribution mechanism is largely affected by the number of cores processing network traffic, number of queues of vNIC being used for handling network traffic, as well as packet information being used to distribute network traffic among multiple cores. Our implementation of packet distribution mechanism for multi-core VNFs is able to handle network traffic under all such considerations and is not limited by any hardware capability or use of some particular packet information for distribution. In this section, we describe in detail our design for network I/O for VNFs, using both software based and hardware based packet distribution mechanisms.

Software based packet distribution

As shown in figure 1, in case of software based packet distribution, we use single queue or multi-queue ports of VALE switch, or a physical NIC with single queue pair, exposed to VM as a ptnetmap device. When single queue NIC or VALE port is exposed to a multi-core VM, packet distribution from vNIC is done using a pair of netmap threads running on one or two dedicated cores, with all other cores available for use by VNF application running over mTCP stack. When multi-queue VALE ports are exposed to VMs, packet distribution to multiple queues of destination port is done by VALE switch operated by the netmap module in host.

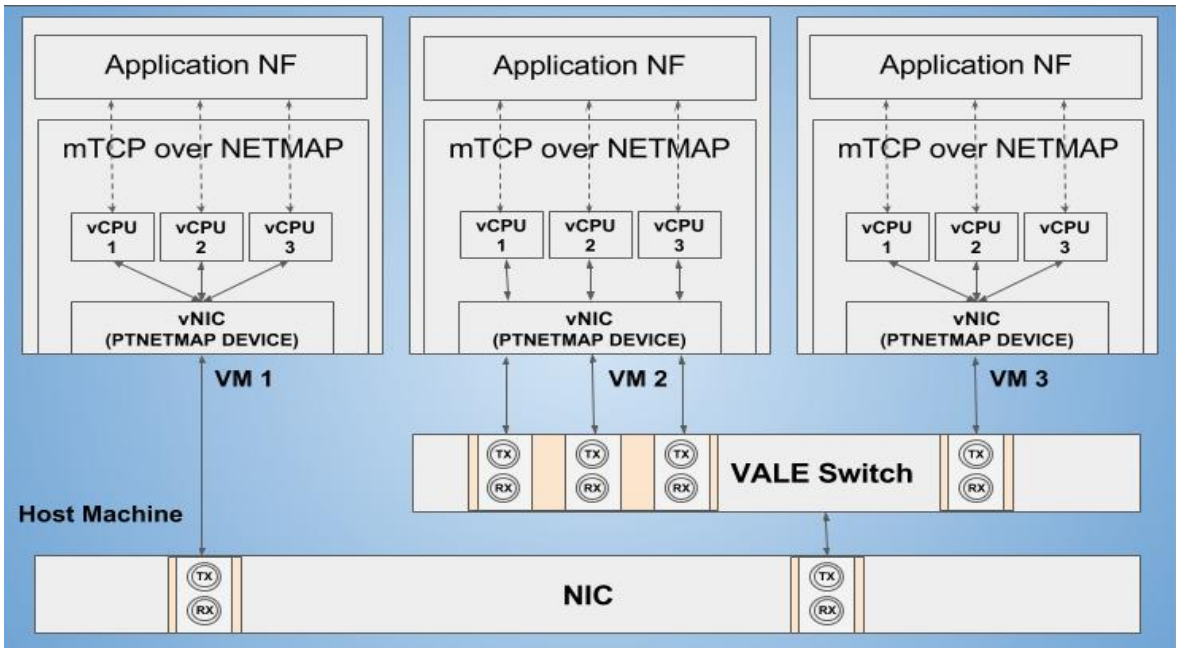

Figure 1: VNF design with software based packet distribution using NETMAP

Software based packet distribution inside VM

Figure 2 shows VNF design with software based packet distribution inside VM, when only single queue vNICs are available to VMs for packet I/O. In the netmap layer of the kernel bypass stack, we create two netmap threads, one for ingress network traffic ethernet layer processing and distribution, and the other for egress network traffic from multiple cores used by application layer VNFs. These threads can run on the same core, or on two different cores. The mTCP stack and application threads run on each of the remaining cores. The netmap TX/RX threads and the mTCP threads communicate using netmap pipes.

The entire mechanism of packet processing and I/O in application level VNF design when packet processing is done in software inside VM is described as follows:

- Threads of multi-core application layer VNFs running on per-core basis use mTCP API for non-blocking send or reception of data.
- mTCP thread running on each core performs packet formation and processing tasks and uses event generation mechanism for communication with application threads. Each mTCP thread uses our netmap API implementation for packet sending and reception at one end of netmap pipes in bursts, while other end of pipes is accessible to RX/TX netmap threads running on one or two dedicated core(s).
- NETMAP RX thread accesses vNIC in NETMAP mode for handling ingress network traffic. It also accesses transmit interface of NETMAP pipes corresponding

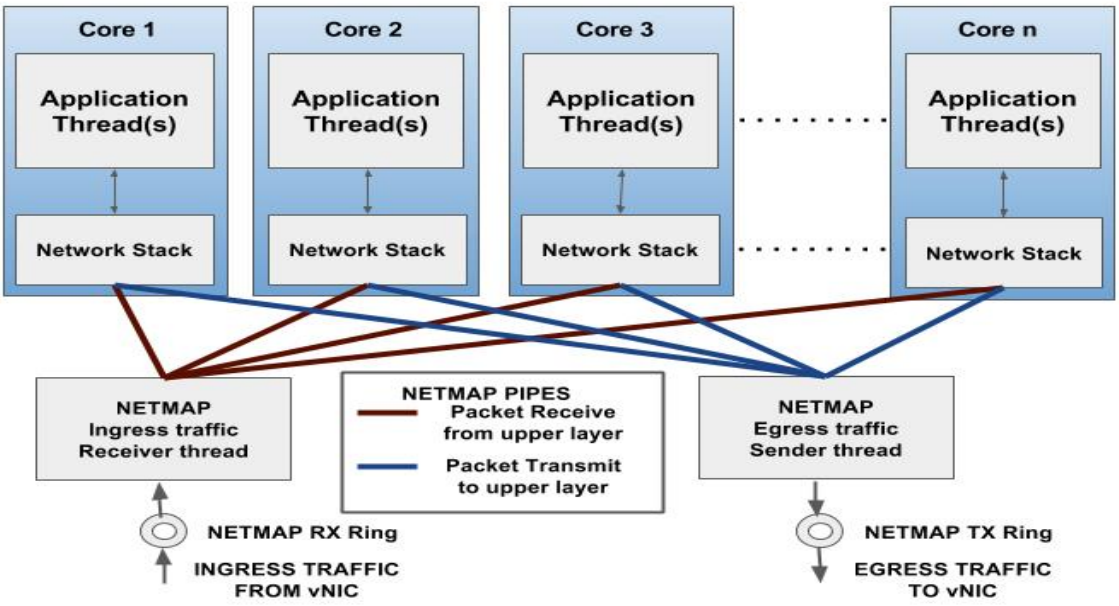

Figure 2: VNF design with software based packet distribution inside VM

to each application core, and forwards packets in bursts to corresponding cores based on software RSS.

- NETMAP TX thread accesses vNIC in NETMAP mode for handling egress network traffic. It also opens receive interface of NETMAP pipes corresponding to each application core, and transmits packets in bursts from all cores to vNIC.

Software based packet distribution in VALE switch:

Figure 3 shows packet distribution mechanism inside VM, when muti-queue VALE ports are exposed to VMs.

The entire mechanism of packet processing and I/O in application level VNF design when packet processing is done in VALE switch is described as follows:

- Threads of multi-core application layer VNFs running on per-core basis use mTCP API for non-blocking send or reception of data.

- mTCP thread running on each core performs packet formation and processing tasks and uses event generation mechanism for communication with application threads. Each mTCP thread uses run to completion model for packet I/O to or from vNIC using our netmap API implementation, which gives access to a queue pair of vNIC for packet I/O, with id same as core id to which an mTCP thread is affinitized.

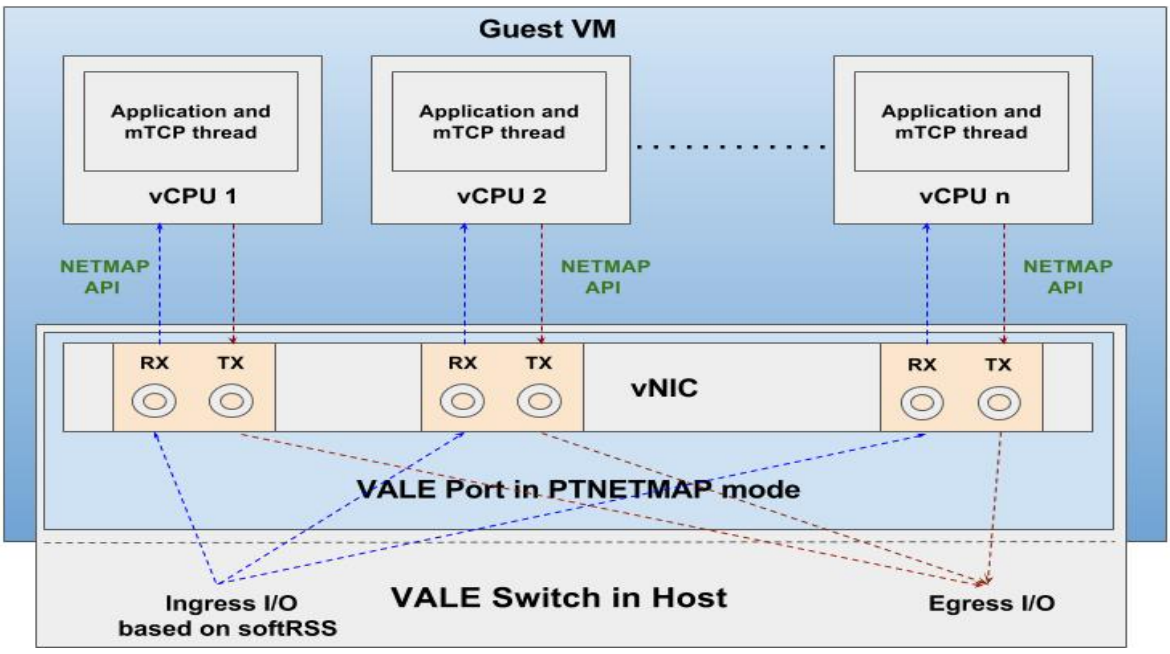

Figure 3: VNF design with software based packet distribution in VALE

- Packet distribution to multiple queues of vNIC of destination VM is done by VALE switch. Original switch logic used packet distribution to multiple queues based on queue-id, wherein a packet was delivered to destination queue with same id as that of source of packet. We modified switching logic to distribute packets based on 4-tuples.

Hardware based packet distribution

Figure 4 shows packet distribution mechanism to multiple cores of a VM, when multi-queue NIC is exposed to the VM. When compared to software based packet distribution in VALE switch, operations inside VM are similar in both cases. The difference lies in operations of queues exposed to a VM. In VALE based multi-queue operations, software queues corresponding to a VALE port are exposed to a VM, whereas in hardware based packet distribution mechanism, hardware queues corresponding to a physical NIC are exposed to a VM. Also, in case of VALE based multi-queue setup, packet distribution to multiple queues is done using software RSS, whereas in case of physical NIC based multi-queue setup, distribution is done using hardware RSS.

Design of user-space Address Resolution Protocol

For completeness of ethernet layer processing for VNFs, there is need to incorporate address resolution protocol(ARP) as part of netmap layer processing. Netmap framework does not provide any pre-existing library for ARP processing. So we designed and

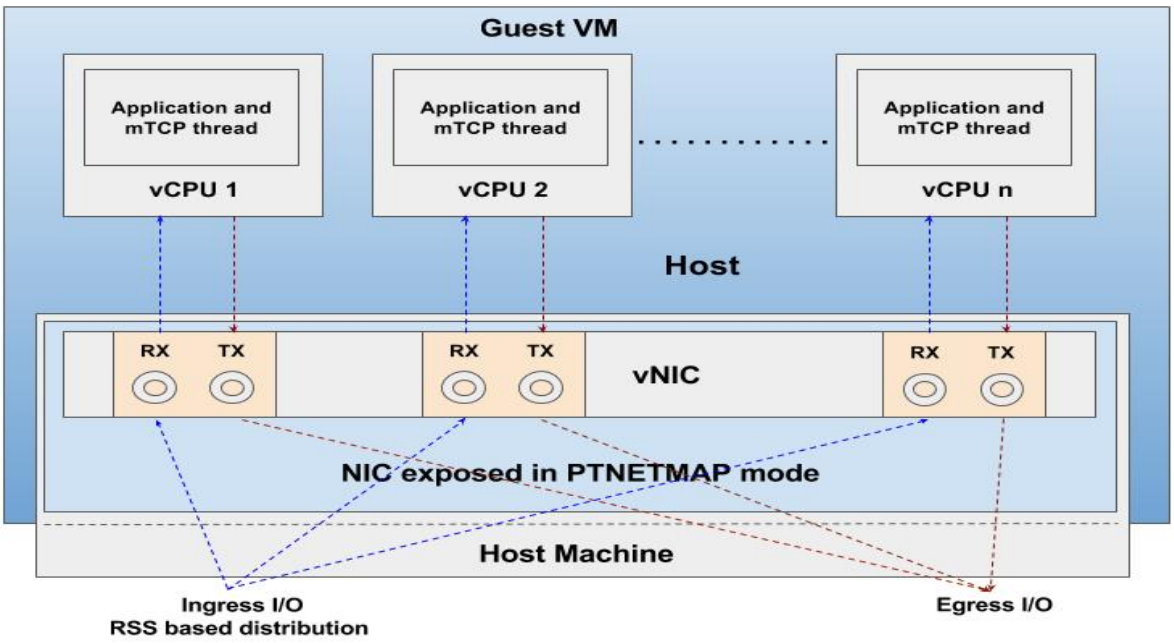

Figure 4: VNF design with packet distribution in physical NIC

implemented ARP functionality and integrated it with netmap layer. ARP functionality works with all hardware or software based packet distribution mechanisms.

Our design of ARP functionality can be bifurcated into two parts:

ARP for single queue based packet distribution in side VM

In this section, we describe our design of ARP functionality when single queue pair corresponding to a vNIC is exposed to a VM, and packet distribution is done using pair of netmap threads running on dedicated core(s) in VM.

Receiver netmap thread performs following operations with respect to ARP functionality:

- On receiving an ARP request or reply packet, it sends IP address - MAC address pair of packet source to sender thread for arptable updation.
- If ARP type packet is an ARP request packet, it conveys this information to sender thread so that sender thread can prepare and send an ARP unicast reply to sender of ARP request.

Sender netmap thread performs following operations with respect to ARP functionality:

- Before sending a packet to vNIC, sender thread checks whether there is arptable entry corresponding to IP address of destination of packet. If there is an entry,

then MAC address corresponding to destination IP is retrieved from arptable and is used to send packet to destination. Otherwise, packet sending is deferred and an ARP request is broadcasted.

- It periodically updates arptable with IP address - MAC address pairs obtained from receiver netmap thread, after receiver thread processes any ARP request or reply packet.
- It periodically checks whether there are some delayed packets for which arptable entry has been created, and sends them.
- It periodically checks whether any ARP request has arrived for which ARP reply needs to be sent, and prepares and sends a unicast ARP reply for the same. Receiver netmap thread conveys this ARP request information to sender thread.

ARP for multi queue based packet distribution

When multi-queue VALE port or a multi-queue physical NIC is exposed to a VM, run to completion model for packet I/O is followed by per-core mTCP thread. Since mTCP uses our netmap API implementation for packet I/O, logic for ARP functionality is implemented in our API, which is described in detail in implementation section. Netmap layer API used by mTCP broadly performs similar tasks with respect to ARP functionality as described in previous section.

Implementation

Implementation is divided into data structures used and functions implemented as part of our API implementation of packet distribution using NETMAP framework. **netmap_api.h** and **netmap_api.cpp** consists of entire implementation of netmap based packet distribution described here.

Data structures used for packet I/O

We maintain two vectors **per_core_send_port** and **per_core_recv_port**, size of which depends upon numbers of cores available for use by application and mTCP inside a VM. These vectors maintain information about send and receive rings available to netmap API at each mTCP core for packet I/O. Ring pair accessed corresponds to one end of netmap pipes in case of VM based distribution, or vNIC in case of VALE or physical NIC based distribution. Vectors **per_core_counts** and **per_core_countr** are packet counters for each mTCP core, and are used to send or receive packets in batches. Receive netmap thread in case of VM based distribution maintains a free queue of netmap buffers, as well as overflow queues for each core. Overflow queue corresponding to a core temporarily stores packets destined to that core while transmit interface of netmap pipe does not have free buffers to transmit packets through netmap pipe.

Packet distribution inside VM

As stated earlier, design of multi-core VNFs, when only single queue pair corresponding to a VALE switch port or physical NIC is exposed to VM as ptnetmap device, requires implementation of packet distribution mechanism inside VM. We achieve this using a pair of netmap threads. Receiver netmap thread handles ingress network traffic from vNIC, whereas sender netmap thread handles egress network traffic to vNIC. Implementation of both netmap threads, handling of packet I/O, and ARP implementation is described in detail below:

Data structures used for ARP implementation

We use google dense map for arptable implementation. We use google dense set and google dense map on per core basis to keep track of pending ARP requests and pending ARP request retries count respectively. A list is maintained on per-core basis for packets delayed for sending due to non-availability of ARP entry.

Ingress network traffic distribution within VM

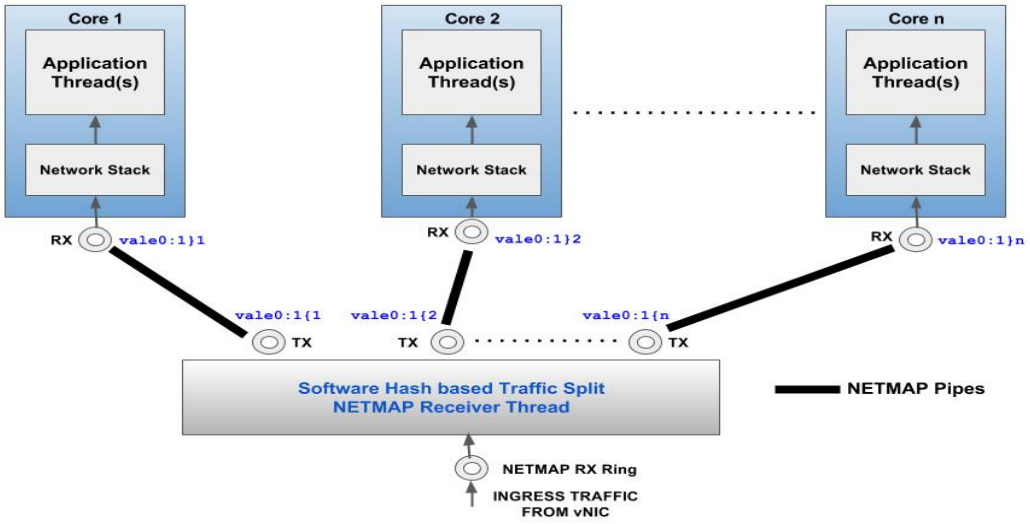

Figure 5: Ingress network traffic distribution within VM

Figure 5 shows implementation of ingress network traffic distribution within VMs.

Ingress I/O traffic is handled by functions **receive_pkts_from_iface**, **get_buffer_rx()** and **syncbufrx()**.

Receiver thread, implemented in function **receive_pkts_from_iface** performs the following operations:

- It opens **vNIC** in **NETMAP** mode and accesses **receive ring** of vNIC for packet reception and distribution among multiple application cores.

- It also creates a VALE switch **vale0** and opens an interface **vale0:1** with option for **extra buffers** and **count of netmap pipes** (for forwarding packets to multiple cores) passed as parameter. This is done because NETMAP region shared by host is fixed in size and cannot be expanded for creating NETMAP pipes and allocating extra buffers to be used during packet receive. So creating a VALE switch inside VM allocates a separate NETMAP region inside guest which is large enough for creation of a number of NETMAP pipes equal to number of cores.

- It then opens virtual interfaces of the form **vale0:1{x**, where **x** is **core id**. This opens one end of NETMAP pipe for each application core and then particular transmit ring can be accessed for packet forwarding to appropriate core.

- In each iteration, thread accepts packets in bursts which are configurable and depend upon number of slots in ring and number of packets available for receiving. It then performs ethernet layer validation, following which it distributes them to appropriate cores based on hash.

- For packets with UDP/TCP and IP headers, it uses a generic hash algorithm based on 4-tuples to identify destination core for packet to be sent. Since hashing is done in user space, packet distribution logic can easily be extended to use any packet information, which makes our design of application level VNF generic and independent of any hardware NIC or software switch support.

- ARP request or reply packets are handled as described in design section.

- If there is space in transmit ring corresponding to NETMAP pipe interface of destination core, packet is sent. Otherwise, it is queued using extra NETMAP buffers and zero copy mechanism and sending to destination core is deferred till slots in transmit ring of particular core become available.

mTCP thread uses functions **get_buffer_rx()** and **syncbufrx()** to get and process packets at receiving end of NETMAP pipes. Their implementation is described below:

- Function **get_buffer_rx()** opens virtual interface of the form **vale0:1}x**, where **x** is **core id**, if not already opened. This gives access to receive ring at receive end of a NETMAP pipe. This function is called by mTCP at beginning of each packet receive operation to get count of packets available for reception. This function then polls receive ring of opened virtual interface corresponding to netmap pipe, and returns count of packets which is less than or equal to receive burst. Subsequent calls to the same function during packet receive operation of mTCP thread returns pointer to first filled receive buffer corresponding to opened interface starting with IP header.

- Function **syncbufrx()** updates **head** and **cur** pointers of ring so that empty slots are available in ring to receive more packets.

Egress network traffic handling within VM

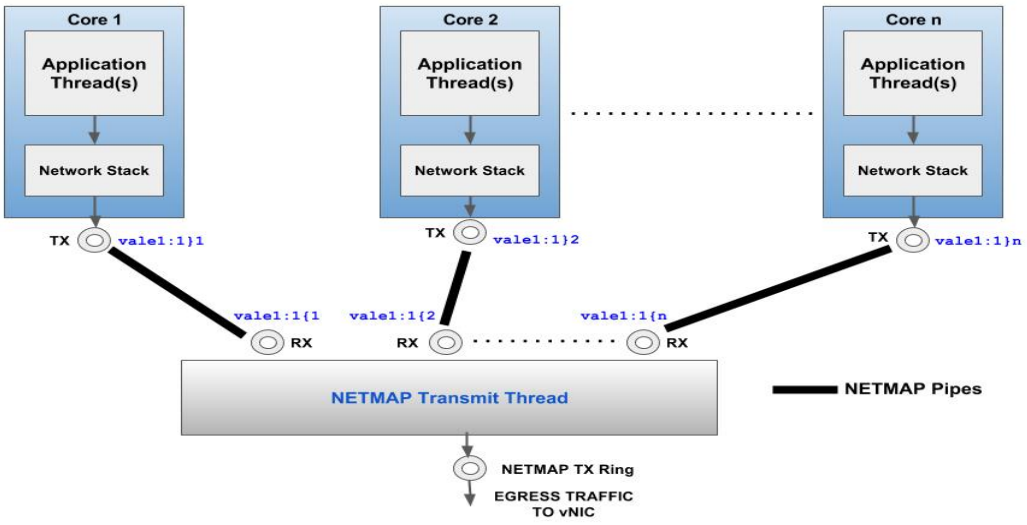

Figure 6: Egress network traffic handling within VM

Figure 6 shows implementation details of egress network traffic distribution within VMs.

Egress I/O traffic is handled by functions **send_pkts_to_iface**, **get_buffer_tx()** and **syncbuftx()**.

Sender thread, implemented in function **send_pkts_to_iface** performs the following operations:

- It opens **vNIC** in **NETMAP** mode and accesses **transmit ring** for packet transmission after receiving them from multiple cores.
- It also creates a VALE switch **vale1** and opens an interface **vale1:1** with option for **count of netmap pipes** (for receiving packets from multiple cores) passed as parameter. This is done for reason same as that in case of receive side implementation.
- It then opens virtual interfaces of the form **vale1:1{x**, where **x** is **core id**. This opens one end of NETMAP pipe for each application core and then receive ring can be accessed to receive packets from each core to be transmitted to vNIC.
- The thread then continues to receive packets from multiple cores through NETMAP pipes.
- After performing ethernet layer processing, it transmits packets in bursts which are configurable and depend on number of slots in ring and number of free slots available for transmitting.

- Generation and sending of new ARP requests or replies, delaying of packets with no ARP entry, updation of arptable, as well as sending of delayed packets is done by this thread, which has already been described in detail in design section.

mTCP thread uses functions functions **get_buffer_tx()** and **syncbuftx()** to access transmit interface of netmap pipes and send packets. Their implementation is described below:

- Function **get_buffer_tx()** opens virtual interface of the form **vale1:1}x**, where **x** is **core id**, if not already opened. This gives access to transmit ring of transmit end of NETMAP pipe. Subsequent calls to this function places the packet in transmit ring corresponding to opened interface, if space is available, or blocks till free slots are available in ring. Packets are pushed through the pipes in bursts, which are configurable.

- Function **syncbuftx()** updates **head** and **cur** pointers of ring so that packets in filled buffers can be pushed through the interface to other end of the pipe.

ARP implemetation for software based packet distribution inside a VM

ARP procedure handling is done by sender and receiver netmap threads, which has already been described in detail in a previous section. We use google dense hash map for arptable implementation. Lockless producer-consumer queues are used for new ARP requests and ARP entries. This is done because receiver thread needs to communicate to sender thread about new ARP entries after receiving ARP packets, and about new ARP requests for which ARP relies are to be generated, in lockless manner as sender thread may simultaneously use these queues for arptable updation and generating and sending ARP replies. Also, sender thread makes 16 retires to obtain MAC address corresponding to an IP address not found in arptable by generating ARP requests, after which it backs off and drops packets corresponding to the particular IP address.

Packet distribution using VALE switch or NIC

Figure 7 shows design of multi-core VNFs, when a multi-queue NIC or VALE switch port is exposed to VM as ptnetmap device. Packet distribution to multiple queues is done in hardware using RSS in case of NIC, whereas in case of VALE switch port, switch handles packet distribution using software RSS. Setup and mplementation of packet distribution strategy for multi-core VNFs in this scenario is described below:

Data structures used for ARP implementation

We use lockless map for arptable implementation, to avoid locking among multiple cores for accessing the map. We use google dense set and google dense map to keep track of pending ARP requests and pending ARP request retries count respectively. Lockless

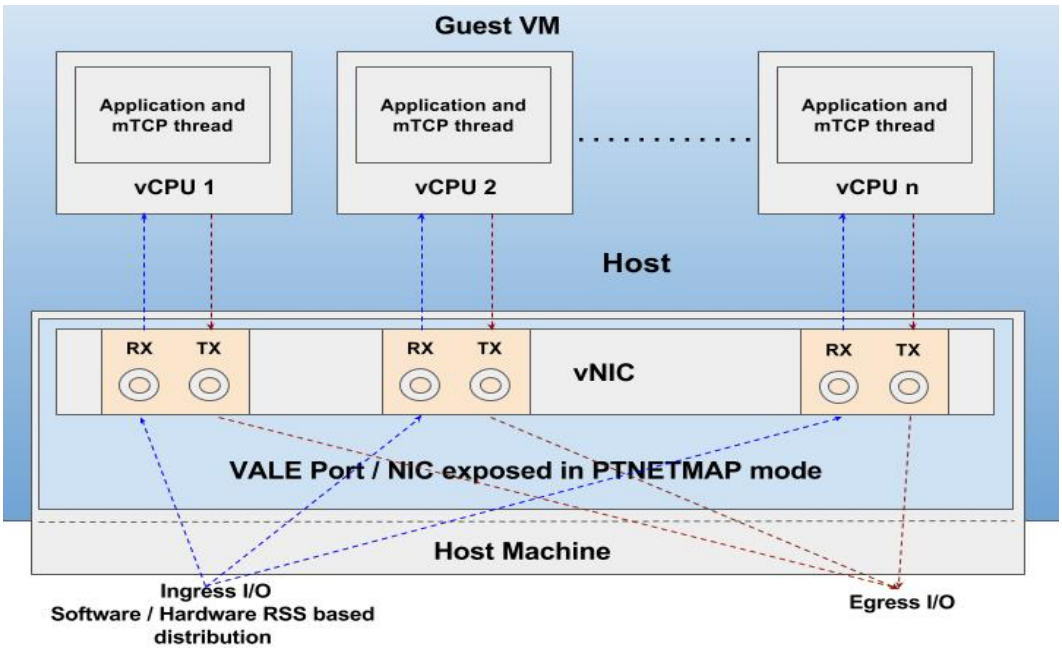

Figure 7: Packet distribution using VALE switch or NIC

queues are used for new ARP entries to be updated and ARP requests pending a reply. A list is maintained for packets delayed for sending due to non-availability of ARP entry.

Setup of multi-queue enabled VALE ports and 4-tuple based packet distribution in VALE switch

- Original switch logic uses packet distribution to multiple queues based on queue-id, wherein a packet is delivered to destination queue with same id as that of source of packet. We have modified switching logic to distribute packets to multiple queues of destination port based on 4-tuples.
- A persistent virtual netmap interface is created using a tool provided by netmap framework. Number of queues and number of buffers per queue are specified while creating the interface. Interface is then attached to a VALE switch as a port to be used by a VM.
- A kernel thread operates on each port of a VALE switch for communication between host and guest VM. For poll based operation of a VALE port, number of cores and core ids to be used by kernel thread in host need to be specified. No such configuration is needed for interrupt based operation of a VALE port.

Setting up number of queues and RSS based on number of cores of VM: With regards to setting up multi-queue NIC based distribution, following configurations

need to be done:

- Appropriate netmap patched network driver need to be inserted with number of queues equal to number of cores in VM to which it is exposed, with RSS enabled on all queues provided to VM.
- Number of buffers in send and receive netmap queues need to be set using **ethtool**.
- Receive flow hashing for both UDP and TCP packets need to be set on 4-tuples. (Default mechanism used is to hash on IP address pair only)
- TCP segmentation as well as receive and transmit checksumming need to be turned off for better performance.

Ingress network traffic distribution within VM

mTCP thread uses functions **get_buffer_rx()** and **syncbufrx()** to get and process packets from vNIC.

- Function **get_buffer_rx()** initially opens netmap memory descriptor corresponding to a receive ring of vNIC. Receive ring identifier is same as core-id to which mTCP thread calling this function is affinitized. This function is called by mTCP for two purposes. It is called once at the beginning of each packet receive operation to get count of packets available for reception. This function then polls particular receive ring of vNIC to get count of packets available for reception. Before polling receive ring, function flushes transmit packets that were previously delayed due to non-existence of ARP entry to transmit ring of netmap pipe, if such an entry is now available. It then returns count of packets to mTCP available for processing after polling, which is less than or equal to receive burst. Each subsequent call to this function by mTCP during packet receive operation performs ethernet layer validation on the packet under consideration. If packet received is an ARP packet, then ARP table is updated with destination information, and **cur** pointer is updated to point to next buffer. If ARP packet is a request packet, an ARP reply is also generated and sent. Otherwise, if packet in buffer is an IP packet with destination IP address same as that of interface from which packet is received, pointer to receive buffer starting from IP header is passed to mTCP. Otherwise, packet is dropped and next buffer is accessed and ethernet validation operations are repeated on it.
- Function **syncbufrx()** updates **head** and **cur** pointers of ring so that empty slots are available in ring to receive more packets.

Egress network traffic handling within VM

mTCP thread uses functions functions **get_buffer_tx()** and **syncbuftx()** to access vNIC for packet transmission.

- Function **get_buffer_tx()** opens netmap memory descriptor corresponding to a transmit ring of VNIC, if not already opened. Transmit ring identifier is same as core-id to which mTCP thread calling this function is affinitized. Subsequent calls to this function places the packet in transmit ring corresponding to opened interface, if space is available, or blocks till free slots are available in ring. Packets are pushed through the transmit ring in bursts, which are configurable.
- Function **syncbuftx()** performs ethernet layer processing of packet before sending it to vNIC. If there is an entry, then MAC address corresponding to destination IP is retrieved from arptable and is used to send packet to destination. While doing so, it also sends packets destined for same IP address that were previously delayed due to non-existence of entry in arptable, if such packets exist. If no entry is found in arptable for outgoing packet, packet sending is deferred and a ARP request is broadcasted. It is to be noted that only one ARP request is broadcasted for packets within a batch with same IP address. Function also updates **head** and **cur** pointers of ring so that packets in filled buffers can be pushed through the interface.