Packet I/O Functionality and Hosting Applications

Setting up Application Hosting Environment

This section illustrates how, with the Packet I/O functionality, you can use Linux applications to manage communication with the IOS XR interfaces. It describes how the OS environment must be set up to establish packet I/O communication with hosted applications.

Verify Reachability of IOS XR and Packet I/O Infrastructure

Feature Name	Release Information	Description
Virtual IP address in the Linux networking stack	Release 7.5.2	Virtual IP addresses allow a single IP address to connect to the current active RP after an RP switchover event. In addition, this functionality enables your network stack to support virtual IP addresses for third-party applications and IOS XR applications that use the Linux networking stack. The following commands are modified: • ipv4 virtual address • ipv6 virtual address • show linux networking interfaces address-only
Automatic Synchronization of Secondary IPv4 addresses from XR to Linux OS	Release 7.5.3	Now the configured interface secondary IPv4 addresses on the Cisco IOS XR software are automatically synchronized to Linux operating system. The third-party applications on Cisco IOS XR can use the secondary IPv4 addresses without any manual intervention. Earlier, you had to configure the secondary IPv4 addresses on the Linux operating system manually.
Automatic Synchronization of Secondary IPv6 addresses from XR to Linux OS	Release 7.11.1	Now the configured interface secondary IPv6 addresses on the Cisco IOS XR software are automatically synchronized to Linux operating system. The third-party applications on Cisco IOS XR can use the secondary IPv6 addresses without any manual intervention. Earlier, you had to configure the secondary IPv6 addresses on the Linux operating system manually.

Interfaces configured on IOS XR are programmed into the Linux kernel. These interfaces allow Linux applications to run as if they were running on a regular Linux system. This packet I/O capability ensures that off-the-shelf Linux applications can be run alongside IOS XR, allowing operators to use their existing tools and automate deployments with IOS XR.

The IP address on the Linux interfaces, MTU settings, MAC address are inherited from the corresponding settings of the IOS XR interface. Accessing the global VRF network namespace ensures that when you issue the **bash** command, the default or the global VRF in IOS XR is reflected in the kernel. This ensures default reachability based on the routing capabilities of IOS XR and the packet I/O infrastructure.

Virtual addresses can be configured to access a router from the management network such as gRPC using a single virtual IP address. On a device with two or more RPs, the virtual address refers to the management interface that is currently active. This functionality can be used across RP failover without the information of which RP is currently active. This is applicable to the Linux packet path.

Automatic Synchronization of Secondary IPv4 and IPv6 addresses from XR to Linux OS

The secondary IPv4 and IPv6 addresses that are configured for an XR interface are now synchronized into the Linux operating system automatically. With this secondary IPv4 and IPv6 address synchronization, the third party applications that are deployed on Cisco IOS XR can now use the secondary addresses. Prior to this release, only primary IPv4 and IPv6 addresses were supported and the secondary IPv4 and IPv6 addresses had to be configured manually in the Linux operating system.

Exposed XR interfaces (EXIs) and address-only interfaces support secondary IPv4 and IPv6 address synchronization:

- EXIs have secondary IP addresses added to their corresponding Linux interface
- Address-only interfaces have secondary IP addresses added to the Linux loopback device. For additional information on address-only interfaces, see show linux networking interfaces address-only.

The restrictions of secondary IPv4 addresses synchronization are:

- Secondary IPv4 addresses are not synchronized from Linux to XR for Linux-managed interfaces.
- The **ifconfig** Linux command only displays the first configured IPv4 address. To view the complete list of IPv4 addresses, use the **ip addr show** Linux command.

For additional information on secondary IPv4 addresses, see ipv4 address (network) and ipv6 address.

You can run **bash** commands at the IOS XR router prompt to view the interfaces and IP addresses stored in global VRF. When you access the Cisco IOS XR Linux shell, you directly enter the global VRF.

Step 1 From your Linux box, access the IOS XR console through SSH, and log in.

Example:

```
cisco@host:~$ ssh root@192.168.122.188
root@192.168.122.188's password:
Router#
```

Step 2 View the ethernet interfaces on IOS XR.

Example:

```
Router#show ip interface brief
Interface IP-Address Status Protocol Vrf-Name
FourHundredGigE0/0/0/0 unassigned Shutdown Down default
FourHundredGigE0/0/0/1 unassigned Shutdown Down default
```

```
FourHundredGigE0/0/0/2 unassigned Shutdown Down default
FourHundredGigE0/0/0/3 unassigned Shutdown Down default
FourHundredGigE0/0/0/4 unassigned Shutdown Down default
FourHundredGigE0/0/0/5 unassigned Shutdown Down default
FourHundredGigE0/0/0/6 unassigned Shutdown Down default
FourHundredGigE0/0/0/7 unassigned Shutdown Down default
FourHundredGigE0/0/0/8 unassigned Shutdown Down default
FourHundredGigE0/0/0/9 unassigned Shutdown Down default
FourHundredGigE0/0/0/10 unassigned Shutdown Down default
FourHundredGigE0/0/0/11 unassigned Shutdown Down default
FourHundredGigE0/0/0/12 unassigned Shutdown Down default
FourHundredGigE0/0/0/13 unassigned Shutdown Down default
FourHundredGigE0/0/0/14 unassigned Shutdown Down default
FourHundredGigE0/0/0/15 unassigned Shutdown Down default
FourHundredGigE0/0/0/16 unassigned Shutdown Down default
FourHundredGigE0/0/0/17 unassigned Shutdown Down default
FourHundredGigE0/0/0/18 unassigned Shutdown Down default
FourHundredGigE0/0/0/19 unassigned Shutdown Down default
FourHundredGigE0/0/0/20 unassigned Shutdown Down default
FourHundredGigE0/0/0/21 unassigned Shutdown Down default
FourHundredGigE0/0/0/22 unassigned Shutdown Down default
FourHundredGigE0/0/0/23 unassigned Shutdown Down default
HundredGigE0/0/0/24 10.1.1.10 Up Up default
HundredGigE0/0/0/25 unassigned Shutdown Down default
HundredGigE0/0/0/26 unassigned Shutdown Down default
HundredGigE0/0/0/27 unassigned Shutdown Down default
HundredGigE0/0/0/28 unassigned Shutdown Down default
HundredGigE0/0/0/29 unassigned Shutdown Down default
HundredGigE0/0/0/30 unassigned Shutdown Down default
HundredGigE0/0/0/31 unassigned Shutdown Down default
HundredGigE0/0/0/32 unassigned Shutdown Down default
HundredGigE0/0/0/33 unassigned Shutdown Down default
HundredGigE0/0/0/34 unassigned Shutdown Down default
HundredGigE0/0/0/35 unassigned Shutdown Down default
MgmtEth0/RP0/CPU0/0 192.168.122.22 Up Up default
```

Note Use the **ip addr show** or **ip link show** commands to view all corresponding interfaces in Linux. The IOS XR interfaces that are `admin-down` state also reflects a `Down` state in the Linux kernel.

Step 3 Check the IP and MAC addresses of the interface that is in `Up` state. Here, interfaces `HundredGigE0/0/0/24` and `MgmtEth0/RP0/CPU0/0` are in the `Up` state.

Example:

```
Router#show interfaces HundredGigE0/0/0/24
...
HundredGigE0/0/0/24 is up, line protocol is up
Interface state transitions: 4
Hardware is HundredGigE0/0/0/24, address is 5246.e8a3.3754 (bia
5246.e8a3.3754)
Internet address is 10.1.1.1/24
MTU 1514 bytes, BW 1000000 Kbit (Max: 1000000 Kbit)
reliability 255/255, txload 0/255, rxload 0/255
Encapsulation ARPA,
Duplex unknown, 1000Mb/s, link type is force-up
output flow control is off, input flow control is off
loopback not set,
Last link flapped 01:03:50
ARP type ARPA, ARP timeout 04:00:00
Last input 00:38:45, output 00:38:45
Last clearing of "show interface" counters never
5 minute input rate 0 bits/sec, 0 packets/sec
5 minute output rate 0 bits/sec, 0 packets/sec
12 packets input, 1260 bytes, 0 total input drops
```

```
0 drops for unrecognized upper-level protocol
Received 2 broadcast packets, 0 multicast packets
0 runts, 0 giants, 0 throttles, 0 parity
0 input errors, 0 CRC, 0 frame, 0 overrun, 0 ignored, 0 abort
12 packets output, 1224 bytes, 0 total output drops
Output 1 broadcast packets, 0 multicast packets
```

Step 4 Verify that the bash command runs in global VRF to view the network interfaces.

Example:

```
Router#bash -c ifconfig
Hu0_0_0_24 Link encap:Ethernet HWaddr 78:e7:e8:d3:20:c0
inet addr:10.1.1.10 Bcast:0.0.0.0 Mask:255.255.255.0
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:4 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:360 (360.0 B) TX bytes:0 (0.0 B)
Mg0_RP0_CPU0_0 Link encap:Ethernet HWaddr 54:00:00:00:bd:49
inet addr:192.168.122.22 Bcast:0.0.0.0 Mask:255.255.255.0
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:3859 errors:0 dropped:0 overruns:0 frame:0
TX packets:1973 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:2377782 (2.2 MiB) TX bytes:593602 (579.6 KiB)
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:242 errors:0 dropped:0 overruns:0 frame:0
TX packets:242 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:12100 (11.8 KiB) TX bytes:12100 (11.8 KiB)
to_xr Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:500
RX bytes:0 (0.0 B) TX bytes:60 (60.0 B)
```

The `to_xr` interface indicates access to the global VRF.

Step 5 Access the Linux shell.

Example:

```
Router#bash
[ios:~]$
```

Step 6 (Optional) View the IP routes used by the `to_xr` interfaces.

Example:

```
[ios:~]$ip route
default dev to_xr scope link metric 2048
6.1.0.0/16dev Mg0_RP0_CPU0_0 proto kernel scope link src 6.1.22.41
20.1.0.0/16dev Hu0_0_0_0 proto kernel scope link src 20.1.1.1
20.2.0.0/16dev Hu0_0_0_20 proto kernel scope link src 20.2.1.1
30.1.0.0/24dev BE500 proto kernel scope link src 30.1.0.1
172.17.0.0/16dev docker0 proto kernel scope link src 172.17.0.1linkdown
```

Note You can also enter the global VRF directly after logging into IOS XR using the **run ip netns exec vrf-default bash** command.

Programme Routes in the Kernel

The basic routes required to allow applications to send or receive traffic can be programmed into the kernel. The Linux network stack that is part of the kernel is used by normal Linux applications to send/receive packets. In an IOS XR stack, IOS XR acts as the network stack for the system. Therefore to allow the Linux network stack to connect into and use the IOS XR network stack, basic routes must be programmed into the Linux Kernel.

Step 1 View the routes from the bash shell.

Example:

```
[ios:~]$ip route
default dev to_xr scope link src 10.1.1.10 metric 2048
10.1.1.0/24 dev Hu0_0_0_24 proto kernel scope link src 10.1.1.10
192.168.122.0/24 dev Mg0_RP0_CPU0_0 proto kernel scope link src 192.168.122.22
```

Step 2 Programme the routes in the kernel.

Two types of routes can be programmed in the kernel:

- **Default Route:** The default route sends traffic destined to unknown subnets out of the kernel using a special `to_xr` interface. This interface sends packets to IOS XR for routing using the routing state in XR Routing Information Base (RIB) or Forwarding Information Base (FIB). The `to_xr` interface does not have an associated IP address. In Linux, most applications expect the outgoing packets to use the IP address of the outgoing interface as the source IP address.

  With the `to_xr` interface, because there is no IP address, a source hint is required. The source hint can be changed to use the IP address another physical interface IP or loopback IP address. In the following example, the source hint is set to 10.1.1.10, which is the IP address of the Hu0_0_0_24 interface. To use the Management port IP address, change the source hint:

  ```
  Router#bash

  [ios:~]$ip route replace default dev to_xr scope link src 192.168.122.22 metric 2048

  [ios:~]$ip route
  default dev to_xr scope link src 192.168.122.22 metric 2048
  10.1.1.0/24 dev Hu0_0_0_24 proto kernel scope link src 10.1.1.10
  192.168.122.0/24 dev Mg0_RP0_CPU0_0 proto kernel scope link src 192.168.122.22
  ```

  With this updated source hint, any default traffic exiting the system uses the Management port IP address as the source IP address.

- **Local or Connected Routes:** The routes are associated with the subnet configured on interfaces. For example, the 10.1.1.0/24 network is associated with the `Hu0_0_0_24` interface, and the 192.168.122.0/24 subnet is associated with the `Mg0_RP0_CPU0` interface .

Configure VRFs in the Kernel

VRFs configured in IOS XR are automatically synchronized to the kernel. In the kernel, the VRFs appear as network namespaces (netns). For every globally-configured VRF, a Linux network namespace is created. With this capability it is possible to isolate Linux applications or processes into specific VRFs like an out-of-band management VRF and open-up sockets or send or receive traffic only on interfaces in that VRF.

Every VRF, when synchronized with the Linux kernel, is programmed as a network namespace with the same name as a VRF but with the string `vrf` prefixed to it. The default VRF in IOS XR has the name `default`. This name gets programmed as `vrf-default` in the Linux kernel.

The following example shows how to configure a custom VRF `blue`:

Step 1 Identify the current network namespace or VRF.

Example:

```
[ios:~]$ip netns identify $$
vrf-default
global-vrf
```

Step 2 Configure a custom VRF `blue`.

Example:

```
Router#conf t

Router(config)#vrf blue
Router(config-vrf)#commit
```

Step 3 Verify that the VRF `blue` is configured in IOS XR.

Example:

```
Router#show run vrf
vrf blue
!
```

Step 4 Verify that the VRF `blue` is created in the kernel.

Example:

```
Router#bash

[ios:~]$ls -l /var/run/netns
total 0
-r--r--r--. 1 root root 0 Jul 30 04:17 default
-r--r--r--. 1 root root 0 Jul 30 04:17 global-vrf
-r--r--r--. 1 root root 0 Jul 30 04:17 tpnns
-r--r--r--. 1 root root 0 Aug 1 17:01 vrf-blue
-r--r--r--. 1 root root 0 Jul 30 04:17 vrf-default
-r--r--r--. 1 root root 0 Jul 30 04:17 xrnns
```

Step 5 Access VRF `blue` to launch and execute processes from the new network namespace.

Example:

```
[ios:~]$ip netns exec vrf-blue bash
[ios:~]$
[ios:~]$ip netns identify $$
vrf-blue
[ios:~]$
```

Running an **ifconfig** command shows only the default `to-xr` interface because there is no IOS XR interface in this VRF.

```
[ios:~]$ifconfig
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
to_xr Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:500
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
[ios:~]$
```

Step 6 Configure an interface in the VRF `blue` in IOS XR. This interface will be configured automatically in the network namespace `vrf-blue` in the kernel.

Example:

The following example shows how to configure HundredGigE 0/0/0/24 interface in `vrf-blue` from IOS XR:

```
Router#conf t
Router(config)#int HundredGigE 0/0/0/24
Router(config-if)#no ipv4 address
Router(config-if)#vrf blue
Router(config-if)#ipv4 address 10.1.1.10/24
Router(config-if)#commit
```

Step 7 Verify that the HundredGigE 0/0/0/24 interface is configured in the VRF `blue` in IOS XR.

Example:

```
Router#show run int HundredGigE 0/0/0/24
interface HundredGigE0/0/0/24
vrf blue
ipv4 address 10.1.1.10 255.255.255.0
!
```

Step 8 Verify that the interface is configured in the VRF `blue` in the kernel.

Example:

```
Router#bash
Thu Aug 1 17:09:39.314 UTC
[ios:~]$
[ios:~]$ip netns exec vrf-blue bash
[ios:~]$
[ios:~]$ifconfig
Hu0_0_0_24 Link encap:Ethernet HWaddr 78:e7:e8:d3:20:c0
inet addr:10.1.1.10 Bcast:0.0.0.0 Mask:255.255.255.0
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
```

```
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
to_xr Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:500
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
[ios:~]$
```

Open Linux Sockets

The socket entries are programmed into the Local Packet Transport Services (LPTS) infrastructure that distributes the information through the line cards. Any packet received on a line card interface triggers an LPTS lookup to send the packet to the application opening the socket. Because the required interfaces and routes already appear in the kernel, the applications can open the sockets — TCP or UDP.

Step 1 Verify that applications open up sockets.

Example:

```
Router#bash
[ios:~]$nc -l 0.0.0.0 -p 5000 &
[1] 1160
[ios:~]$
[ios:~]$netstat -nlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program name
tcp 0 0 0.0.0.0:5000 0.0.0.0:* LISTEN 1160/nc
tcp 0 0 0.0.0.0:57777 0.0.0.0:* LISTEN 14723/emsd
tcp 0 0 0.0.0.0:22 0.0.0.0:* LISTEN 8875/ssh_server
tcp6 0 0 :::22 :::* LISTEN 8875/ssh_server
udp 0 0 0.0.0.0:68 0.0.0.0:* 13235/xr_dhcpcd
Active UNIX domain sockets (only servers)
Proto RefCnt Flags Type State I-Node PID/Program name Path
[ios:~]$exit
Logout
Router#
Router#show lpts pifib brief | i 5000
Thu Aug 1 17:16:00.938 UTC
IPv4 default TCP any 0/RP0/CPU0 any,5000 any
Router#
```

Step 2 Verify that the socket is open.

Example:

```
Router#show lpts pifib brief | i 5000
IPv4 default TCP any 0/RP0/CPU0 any,5000 any
```

Netcat starts listening on port 5000, which appears as an IPv4 TCP socket in the netstat output like a typical Linux kernel. This socket gets programmed to LPTS, creating a corresponding entry in the hardware to the lookup tcp port 5000. The incoming traffic is redirected to the kernel of the active RP where the netcat runs.

Send and Receive Traffic

Connect to the nc socket from an external server. For example, the nc socket was started in the `vrf-default` network namespace. So, connect over an interface that is in the same VRF.

```
[root@localhost ~]#nc -vz 192.168.122.22 5000
Ncat: Version 7.50 ( https://nmap.org/ncat )
Ncat: Connected to 192.168.122.22:5000.
Ncat: 0 bytes sent, 0 bytes received in 0.01 seconds.
```

Manage IOS XR Interfaces through Linux

The Linux system contains a number of individual network namespaces. Each namespace contains a set of interfaces that map to a single interface in the XR control plane. These interfaces represent the exposed XR interfaces (eXI). By default, all interfaces in IOS XR are managed through the IOS XR configuration (CLI or YANG models), and the attributes of the interface (IP address, MTU, and state) are inherited from the corresponding configuration and the state of the interface in XR.

With the new Packet I/O functionality, it is possible to have an IOS XR interface completely managed by Linux. This also means that one or more of the interfaces can be configured to be managed by Linux, and standard automation tools can be used on Linux servers can be used to manage interfaces in IOS XR.

Note Secondary IPv4 addresses cannot be managed by Linux.

Configure an Interface to be Linux-Managed

This section shows how to configure an interface to be Linux-managed.

Step 1 Check the available exposed-interfaces in the system.

Example:

```
Router(config)#linux networking exposed-interfaces interface ?
  BVI              Bridge-Group Virtual Interface
  Bundle-Ether     Aggregated Ethernet interface(s) | short name is BE
  FiftyGigE        FiftyGigabitEthernet/IEEE 802.3 interface(s) | short name is Fi
  FortyGigE        FortyGigabitEthernet/IEEE 802.3 interface(s) | short name is Fo
  FourHundredGigE  FourHundredGigabitEthernet/IEEE 802.3 interface(s) | short name is FH
  GigabitEthernet  GigabitEthernet/IEEE 802.3 interface(s) | short name is Gi
  HundredGigE      HundredGigabitEthernet/IEEE 802.3 interface(s) | short name is Hu
  Loopback         Loopback interface(s) | short name is Lo
  MgmtEth          Ethernet/IEEE 802.3 interface(s) | short name is Mg
  TenGigE          TenGigabitEthernet/IEEE 802.3 interface(s) | short name is Te
  TwentyFiveGigE   TwentyFiveGigabitEthernet/IEEE 802.3 interface(s) | short name is TF
  TwoHundredGigE   TwoHundredGigabitEthernet/IEEE 802.3 interface(s) | short name is TH
```

Step 2 Configure the interface to be managed by Linux.

Example:

The following example shows how to configure a HundredGigE interface to be managed by Linux:

```
Router#configure
Router(config)#linux networking exposed-interfaces interface HundredGigE 0/0/0/24 linux-managed
Router(config-exi-if)#commit
```

Example:

The following example shows how to configure a BVI5 interface to be managed by Linux:

```
Router#configure
Router(config)#linux networking exposed-interfaces interface BVI5 linux-managed
Router(config-exi-if)#commit
```

Step 3 View the interface details and the VRF.

Example:

The following example shows the information for HundredGigE interface:

```
Router#show run interface HundredGigE0/0/0/24
interface HundredGigE0/0/0/24
mtu 4110
vrf blue
ipv4 mtu 4096
ipv4 address 10.1.1.10 255.255.255.0
ipv6 mtu 4096
ipv6 address fe80::7ae7:e8ff:fed3:20c0 link-local
!
```

Example:

The following example shows the information for BVI5 interface:

```
Router#show run interface bvi5
interface bvi5
mtu 1514
ipv4 mtu 1500
ipv4 address 90.9.9.9 255.255.255.0
ipv6 mtu 1500
ipv6 address fe80::4ee1:75ff:fe74:a80c link-local
!
```

Step 4 Verify the configuration in XR.

Example:

The following example shows the configuration for HundredGigE interface:

```
Router#show running-config linux networking

linux networking
 exposed-interfaces
  interface HundredGigE0/0/0/24 linux-managed
  !
 !
!
```

Example:

The following example shows the configuration for BVI5 interface:

```
Router#show running-config linux networking

linux networking
 exposed-interfaces
  interface BVI5 linux-managed
  !
 !
!
```

Step 5 Verify the configuration from Linux.

Example:

The following example shows the configuration for HundredGigE interface:

```
Router#bash
Router:Aug 1 17:40:02.873 UTC: bash_cmd[67805]: %INFRA-INFRA_MSG-5-RUN_LOGIN : User vagrant logged
into shell from vty0

[ios:~]$ip netns exec vrf-blue bash

[ios:~]$ifconfig
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
to_xr Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:500
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

[ios:~]$ifconfig -a
Hu0_0_0_24 Link encap:Ethernet HWaddr 78:e7:e8:d3:20:c0
BROADCAST MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
to_xr Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:500
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

Example:

The following example shows the configuration for BVI5 interface:

```
Router#bash
Router:Aug 1 17:40:02.873 UTC: bash_cmd[67805]: %INFRA-INFRA_MSG-5-RUN_LOGIN : User vagrant logged
into shell from vty0

[ios:~]$ifconfig BVI5
lo Link encap:Local LoopbackBVI5 Link encap:Ethernet HWaddr 4c:e1:75:74:a8:0c
BROADCAST MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
```

```
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

Configure New IP address on the Interface in Linux

This section shows how to configure a new IP address on the Linux-managed interface.

Step 1 Configure the IP address on the interface.

Example:

```
[ios:~]$ip addr add 10.1.1.10/24 dev Hu0_0_0_24
[ios:~]$Router:Aug 1 17:41:11.546 UTC: xlncd[253]: %MGBL-CONFIG-6-DB_COMMIT : Configuration
committed by user 'system'. Use 'show configuration commit changes 1000000021' to view the changes.
```

Step 2 Verify that the new IP address is configured.

Example:

```
[ios:~]$ifconfig Hu0_0_0_24
Hu0_0_0_24 Link encap:Ethernet HWaddr 78:e7:e8:d3:20:c0
inet addr:10.1.1.10 Bcast:0.0.0.0 Mask:255.255.255.0
BROADCAST MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

Configure Custom MTU Setting

This section shows how to bring up the interface and configure a custom MTU in a Linux-managed interface.

Step 1 Configure the MTU setting.

Example:

```
[ios:~]$ifconfig Hu0_0_0_24 up

[ios:~]$Router:Aug 1 17:41:54.824 UTC: ifmgr[266]: %PKT_INFRA-LINK-3-UPDOWN : Interface
HundredGigE0/0/0/24, changed state to Down
Router:Aug 1 17:41:54.824 UTC: ifmgr[266]: %PKT_INFRA-LINEPROTO-5-UPDOWN : Line protocol on
Interface HundredGigE0/0/0/24, changed state to Down
Router:Aug 1 17:41:56.448 UTC: xlncd[253]: %MGBL-CONFIG-6-DB_COMMIT : Configuration committed by
user 'system'. Use 'show configuration commit changes 1000000022' to view the changes.
Router:Aug 1 17:41:56.471 UTC: ifmgr[266]: %PKT_INFRA-LINK-3-UPDOWN : Interface
HundredGigE0/0/0/24, changed state to Up
Router:Aug 1 17:41:56.484 UTC: ifmgr[266]: %PKT_INFRA-LINEPROTO-5-UPDOWN : Line protocol on
Interface HundredGigE0/0/0/24, changed state to Up
Router:Aug 1 17:41:58.493 UTC: xlncd[253]: %MGBL-CONFIG-6-DB_COMMIT : Configuration committed by
user 'system'. Use 'show configuration commit changes 1000000023' to view the changes.

[ios:~]$
[ios:~]$ ip link set dev Hu0_0_0_24 mtu 4096
[ios:~]$
```

```
[ios:~]$Router:Aug 1 17:42:46.830 UTC: xlncd[253]: %MGBL-CONFIG-6-DB_COMMIT : Configuration
committed by user 'system'. Use 'show configuration commit changes 1000000024' to view the changes.
```

Step 2 Verify that the MTU setting has been updated in Linux.

Example:

```
[ios:~]$ifconfig
Hu0_0_0_24 Link encap:Ethernet HWaddr 78:e7:e8:d3:20:c0
inet addr:10.1.1.10 Bcast:0.0.0.0 Mask:255.255.255.0
inet6 addr: fe80::7ae7:e8ff:fed3:20c0/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:4096 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:648 (648.0 B)
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
to_xr Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:500
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

Step 3 Check the effect on the IOS XR configuration with the change in MTU setting on this interface.

Example:

```
Router#show running-config int HundredGigE0/0/0/24
interface HundredGigE0/0/0/24
 mtu 4110
  vrf blue
  ipv4 mtu 4096
  ipv4 address 10.1.1.10 255.255.255.0
  ipv6 mtu 4096
  ipv6 address fe80::7ae7:e8ff:fed3:20c0 link-local
  !
 !
!
Router#
Router#show ip int br | i HundredGigE0/0/0/24
HundredGigE0/0/0/24 10.1.1.10 Up Up blue
```

The output indicates that the interface acts as a regular Linux interface, and IOS XR configuration receives inputs from Linux.

Configure Traffic Protection for Linux Networking

Traffic protection provides a mechanism to configure Linux firewalls using IOS XR configuration. These rules can be used to restrict traffic to Linux applications. You can restrict traffic to Linux applications using native Linux firewalls or configuring IOS XR Linux traffic protection. It is not recommended to use both mechanisms at the same time. Any combination of remote address, local address and ingress interface can be specified as rules to either allow or deny traffic. However, at least one parameter must be specified for the traffic protection rule to be valid.

Note If traffic is received on a protocol or port combination that has no traffic protection rules configured, then all traffic is allowed by default.

This example explains how to configure a traffic protection rule on IOS XR to deny all traffic on port 999 except for traffic arriving on interface HundredGigE0/0/0/25.

Step 1 Configure traffic protection rules.

Example:

```
Router(config)#linux networking vrf default address-family ipv4 protection protocol
tcp local-port 999 default-action deny permit hundredgigE0/0/0/25
Router(config)#commit
```

where —

- **address-family:** Configuration for a particular IPv4 or IPv6 address family.
- **protection:** Configure traffic protection for Linux networking.
- **protocol:** Select the supported protocol - TCP or UDP.
- **local-port:** L4 port number to specify traffic protection rules for Linux networking.
- **port number:** Port number ranges from 1 to 65535 or all ports.
- **default-action:** Default action to take for packets matching this traffic protection service.
- **deny:** Drop packets for this service.
- **permit:** Permit packets to reach Linux application for this service.

Step 2 Verify that the traffic protection rule is applied successfully.

Example:

```
Router(config)#show run linux networking
linux networking
 vrf default
  address-family ipv4
   protection
    protocol tcp local-port 999 default-action deny
     permit interface HundredGigE0/0/0/25
     !
    !
   !
 !
```

Synchronize Statistics Between IOS XR and Linux

This example shows how the bundle-ether interface packet statistics are synchronized between IOS XR and Linux. The packet and byte counters maintained by Linux for IOS XR interfaces display only the traffic sourced in Linux. You can configure to periodically synchronize these counters with the IOS XR statistics for the interfaces.

Step 1 Configure the statistics synchronization including the direction and synchronization interval.

Example:

The following example shows statistics synchronization in global configuration:

```
Router(config)#linux networking statistics-synchronization from-xr
every 30s
```

Example:

The following example shows statistics synchronization in exposed-interface configuration:

```
Router(config)#linux networking exposed-interfaces interface
bundle-ether 1 statistics-synchronization from-xr every 10s
```

where —

- **from-xr:** The direction indicating that the interface packet statistics will be pushed from IOS XR to the Linux kernel.
- **every:** Shows the frequency at which to synchronize statistics. The intervals supported for global configuration are 30s and 60s. The intervals supported for exposed interfaces are 5s, 10s, 30s or 60s. The interval `s` is in seconds.

Step 2 Verify that the statistics synchronization is applied successfully on IOS XR.

Example:

```
Router#show run linux networking
linux networking
 vrf default
  address-family ipv4
   protection
   protocol tcp local-port all default-action deny
    permit interface bundle-ether 1
    !
   !
  !
 !
exposed-interfaces
 interface bundle-ether 1 linux-managed
  statistics-synchronization from-xr every 10s
  !
 !
!
```

For troubleshooting purposes, use the **show tech-support linux networking** command to display debugging information.

Hosting an Application in Docker Containers

This section provides the procedure for hosting an application in docker containers.

The iPerf application is used as an example to demonstrate the hosting at a server and a client router.

Verify Reachability of IOS XR and Packet I/O Infrastructure, on page 2 on the router that hosts the iPerf application. You can enable the following Packet I/O functionalities on the server and client routers prior to hosting the iPerf application in docker containers, for additional features on the routers:

- **Program Routes in the Kernel**—to send or receive traffic to a remote network using a specific interface.
- **Configure VRFs in the Kernel**—to run the iperf application in a non-default VRF.
- **Configure Traffic Protection for Linux Networking**—to secure the router by restricting access to the router on which the iperf application is hosted.

You build the docker image of the application following the standard docker build procedures. The docker image of any application (for example, iPerf) is built only once, after which, that docker image can be copied to other devices where the application can be hosted in docker containers.

Docker Operations

This section describes basic docker operations and the commands required for hosting and maintaining the applications:

Commands for Hosting Applications

- **Pull or Load the image**: This function copies a docker image to a device.
  - Pull—

    The following command pulls the Docker image from a local docker registry. Ensure that the registry is accessible from the router.

    ```
    [ios:~]$docker pull ufi-lnx:5001/alpine
    ! Here ufi-lnx is the docker registry reachable through 10.105.39.169!-->
    Using default tag: latest
    latest: Pulling from library/alpine
    c9b1b535fdd9: Pull complete
    Digest: sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172b11c4367d
    Status: Downloaded newer image for alpine:latest
    [ios:~]$
    ```

    Instead of being pulled from the registry, docker images can be loaded from images saved as tar files.

  - Load—

    You save the docker image as a tar file on the build host using the **docker save -o** *<path for generated tar file> <image name>* command. You copy the tar file into the target router, using the following command:

    You copy the tar file into the target router, using the following command:

    ```
    Router#scp root@10.105.227.122:/var/www/html/alpine.tar
    Tue Mar 10 02:42:38.598 UTC
    Connecting to 10.105.227.122...
    Password:
      Transferred 639972864 Bytes
      639972864 bytes copied in 55 sec (11606958)bytes/sec
    Router#bash
    Tue Mar 10 02:45:25.330 UTC
    [ios:~]$docker load -i /tmp/alpine.tar
    ```

```
Loaded image: alpine:latest
ios:~]$docker images
REPOSITORY           TAG                 IMAGE ID            CREATED
SIZE
alpine               latest              dc721c65d296        11 days ago
622MB
[ios:~]$
```

- **View or List Docker Images**

```
[ios:~]$docker images
REPOSITORY           TAG                 IMAGE ID            CREATED             SIZE
alpine               latest              dc721c65d296        11 days ago         622MB
enipla               latest              fd31184c8c1b        6 weeks ago         581MB
[ios:~]$
```

- **Run Container**

```
[ios:~]$docker run -it alpine bash
root@a1b719df1091:/#
root@a1b719df1091:/#
root@a1b719df1091:/#uname -a
Linux a1b719df1091 4.8.28-WR9.0.0.20_cgl#1 SMP Wed Jan 8 11:16:16 UTC 2020 x86_64 x86_64
 x86_64 GNU/Linux
root@a1b719df1091:/#hostname
a1b719df1091
root@a1b719df1091:/#
[ios:~]$docker ps
CONTAINER ID        IMAGE                 COMMAND              CREATED              STATUS
              PORTS               NAMES
a1b719df1091        alpine                "bash"               About a minute ago   Up
About a minute                       nifty_leavitt
[ios:~]$
```

- **Attach to a Running Container**

The **docker attach** command attaches the terminal to the running container and the **docker exec** command runs commands inside a working container.

```
[ios:~]$docker attach docker1
#bash
root@e1e1924956df:/#
```

The **docker exec -it** *my_container_id* **sh** command executes a shell inside the container:

```
f3b-r1-pod9:/var/lib/docker/volumes]$docker exec -it 57029028609a sh
#
```

- **Stop Containers**

Identify the container using the **docker ps** command and then stop the container using the **docker stop** command.

```
[ios:~]$docker ps
CONTAINER ID        IMAGE                 COMMAND              CREATED              STATUS
              PORTS               NAMES
8782d4902312        alpine                "bash"               7 minutes ago        Up 6
minutes                             hungry_keller
9bd23408d640        alpine                "bash"               17 minutes ago       Up 17
 minutes                            youthful_franklin
[ios:~]$ docker stop 8782d4902312
```

- **Stopping All Containers**

```
[ios:~]$docker stop $(docker ps -a -q)
```

Commands for Maintaining Containers

- **Restart Policies**

  Docker restart policies start the containers automatically after the router reboots.

  ```
  [ios:~]$docker run -d --restart
  ```

- **Clean up Containers and Images**

  You can clean images, containers, volumes, and networks that are dangling (and not associated with a container) by using the **docker system prune** command.

  ```
  [ios:~]$docker container prune
  ```

  and

  ```
  [ios:~]$docker image prune
  ```

- **Monitor Docker (View Docker Statistics)**

  You can view performance metrics, such as utilization of memory and CPU, and container-specific metrics, such as CPU limit and memory limit.

  ```
  [ios:~]$docker stats --no-stream
  CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM
  %                NET I/O             BLOCK I/O           PIDS
  a1b719df1091        nifty_leavitt       0.00%               2.035MiB / 30.78GiB   0.01%
                 648B / 0B           0B / 0B             2
  ```

Note For generic docker commands, see the Docker Release 18.05 documentation. (https://docs.docker.com)

Procedure for Hosting Applications in Docker Containers

1. Build the docker image following the standard docker build procedures. See here.

   The docker image is transferred to the IOS XR router (target router) using one of the following ways:

   - The docker image is pulled from the docker image registry into the target router (or)
   - The docker image is saved as the tar file in the build host and then the tar file is copied into the target router from the build host.

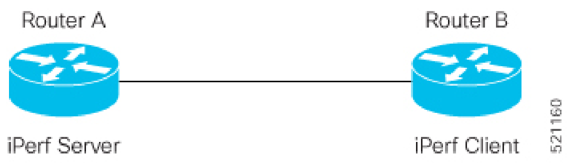
2. Start the docker container and run the application on the router.
3. Verify the hosted application in the docker container.

Run iPerf in Docker Container

As an example of application hosting in docker container, you can install iPerf client on Router A and check its connectivity with an iPerf server installed on Router B.

This figure illustrates the topology used in this example.

Figure 1: iPerf Hosted in a Docker Container



The following steps describe how to run the iPerf server and iPerf client applications on Router A and Router B.

Before you begin

Ensure that you have configured the two routers as shown in the figure-*iPerf application hosted in a Docker Container*.

Step 1 Copy the iPerf application tar file (for example, ubuntu-agnel-image.tar) on Router A.

Example:

```
Router#scp root@10.105.227.122:/var/www/html/ubuntu-agnel-27feb.tar $
Connecting to 10.105.227.122...
Password:
  Transferred 639972864 Bytes
  639972864 bytes copied in 55 sec (11606958)bytes/sec
```

Step 2 Load the docker instance on Router A by using the following command:

Example:

```
Router#bash
[ios:~]$docker load -i /tmp/ubuntu-agnel-27feb.tar
```

Step 3 View all docker images by using the following command:

Example:

```
[ios:~]$docker images ls
REPOSITORY           TAG                 IMAGE ID            CREATED             SIZE
ubuntu-agnel-27feb   latest              dc721c65d296        11 days ago         622MB
ubuntu-agnel-slapi   latest              fd31184c8c1b        6 weeks ago         581MB
```

Step 4 Repeat Steps 1 through 3 on Router B.

Step 5 Configure the application to run as iPerf server on Router A.

Example:

```
[ios:~]$docker run -d -it -p 601:601 ubuntu-agnel-27feb sh
[ios:~]$iperf3 -s -B 172.17.0.2
```

Note “**-p 601:601 ubuntu-agnel-27feb sh**” part of the command maps the Linux port with the docker instance port. 601 on the left hand side is the Docker instance mapping port and 601 on the right hand side is the Linux kernel port.

172.17.0.2 is the IP address of the server.

Step 6 Configure the application to run as iPerf client on Router B and establish connection to iPerf server on Router A.

Example:

```
[ios:~]$docker run -d -it -p 601:601 ubuntu-agnel-27feb sh
[ios:~]$iperf3 -c 172.17.0.2
```

Note "**-p 601:601 ubuntu-agnel-27feb sh**" part of the command maps the Linux port with the docker instance port. 601 on the left hand side is the docker instance mapping port and 601 on the right hand side is the Linux kernel port.

172.17.0.2 is the IP address of the server.

Verify the Application Hosted in the Docker Container

To verify the applications hosted in the docker containers between Router A and Router B, use the **ping** command to check if the connection has been established between iPerf server and iPerf client.

From the iPerf client on Router B, ping the iPerf server on Router A by providing the physical interface IP address to verify the connection between the iPerf server and client applications.

Example:

```
[ios:~]$ping 172.17.0.2
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 172.17.0.2, timeout is 2 seconds:
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 13/21/40 ms
```

172.17.0.2 is the IP address of the client.

Boot Devices Using PXE Server Running in a Docker Container

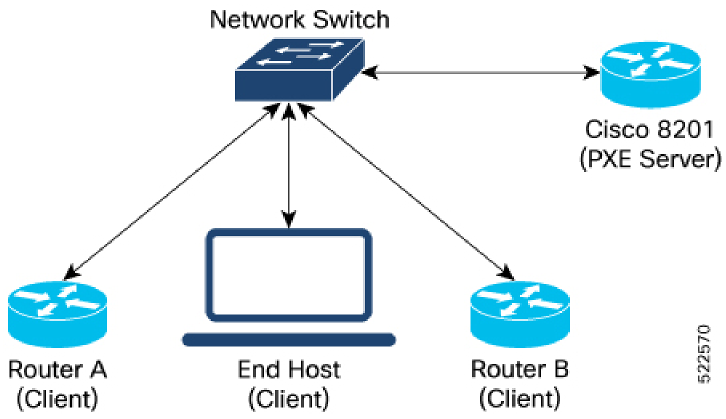
Feature Name	Release Information	Description
Boot Devices Using PXE Server Running in a Docker Container	Release 24.2.1	Starting from Cisco IOS XR Release 24.2.1, the PXE server feature is deprecated and will not be supported in future releases. We recommend not to use this feature starting from Cisco IOS XR Release 24.2.1.

Feature Name	Release Information	Description
Boot Devices Using PXE Server Running in a Docker Container	Release 7.3.4	You can now boot your network devices with a PXE pre-boot execution environment (PXE or iPXE) server running in a Docker container. You use the application manager (appmgr) to manage PXE server docker hosting and functioning through Cisco IOS XR CLIs. This functionality lets you 'freeze' your booting environment in a Docker container instead of having to reinstall the environment for every new machine you want to boot, saving you the trouble of remembering the exact commands and sequences for a PXE boot.

Preboot Execution Environment (PXE) is a client-server interface that enables devices in a network to download the files (like boot image, configurations and so on) from PXE server.

The Client uses DHCP protocol to receive the PXE server details and uses TFTP or HTTP protocol to download the file.

Figure 2: Client-Server Connection



The PXE server docker feature enables the support of PXE/iPXE server functionality on the routers that run Cisco IOS-XR software.

This feature helps compute clusters (clients) to be upgraded from the Cisco 8201 top-of-rack router (server) that hosts the new image for software upgrade, thereby optimizing the operations and management bandwidth. PXE server docker feature is supported on both IPv4 and IPv6 addresses.

In this feature, the PXE server is installed on the Cisco 8201 router (server) in the form of a docker container that is managed by Cisco IOS-XR Application Manager (appmgr). The clients (routers or end-hosts such as Linux devices, VMs and so on) that are connected to this server can request and download the boot image.

The following services are packaged in a single PXE server docker container:

- DHCP
- HTTP (iPXE)
- TFTP (PXE)

These services are used for initial exchange of information and transferring the image between the client and the server.

The PXE server docker feature is available as part of the optional RPM—**xr-pxeserver**. This optional RPM contains:

- Executables for **pxe_svr_mgr** Cisco IOS-XR process.
- PXE server docker image —**pxe-server-docker.rpm**.

  When the optional **xr-pxeserver** RPM is installed, the unsigned docker image (**pxe-server-docker.rpm**) is placed in the appmgr images directory **/pkg/opt/cisco/XR/appmgr/images/pxe-server-docker.rpm**, by the system.
- Helper scripts — **install_pxeserver.py** and **uninstall_pxeserver.py** placed under **"/pkg/bin/"** is used for installing and uninstalling **pxe-server-docker.rpm** from the application manager.

Note The helper scripts are used to manually install or uninstall **pxe-server-docker.rpm** and perform the installation or cleanup instead of using the application manager commands.

Behavioral Specifications

- The Cisco IOS XR DHCP proxy, server, and relay features for both IPv4 and IPv6 are not supported when the PXE server docker container is installed and running on the router.
- PXE server docker is supported only for BVI interfaces on its secondary IPv4 address. The PXE server docker is not supported on the primary IPv4 address of a BVI interface.
- Only one instance of PXE server docker container is supported to run on the router at a given time. Running multiple instances of PXE server docker container on multiple BVI interfaces in parallel is not supported and results in undefined behavior.
- Third-party application RPMs with the application name as "**pxe-server**" must not be installed along with this feature.
- Synchronizing between RPs for large files (iso images) take significant time (approximately 10 mins). If the system performs RPFO immediately after copying a large file to the application folder (**/harddisk:/mirror/server/images**), then these files should be copied again to the current active RP manually.
- The application state is not maintained after an upgrade. If the application is moved to STOP state and then updated using new version of Cisco IOS-XR RPM, then the new version of the application starts again automatically.
- Uninstalling the optional RPM (**xr-pxeserver**) does not stop or remove the PXE server docker container. User has to manually stop the PXE server docker container and uninstall the **pxe-server-docker.rpm** using the application manager commands.

Hosting and Activating the PXE Server Docker on Cisco 8201 Router using Application Manager

To host and activate the PXE server docker container application on Cisco 8201 router using application manager, follow these steps:

Step 1 Install the optional RPM **xr-pxeserver** on the router:

```
Router#install package add xr-pxeserver
Router#install apply restart
Router#install commit
```

When the optional RPM- **xr-pxeserver** is installed and activated, the **pxe_svr_mgr** process is instantiated. The **pxe_svr_mgr** process handles installation and updating the PXE server docker RPM (**pxe-server-docker.rpm**) with application manager. Also, when the optional RPM **xr-pxeserver** is upgraded, the **pxe_svr_mgr** process checks for the new version of the **pxe-server-docker.rpm**. If there is a change in the version number, then it updates the existing version on the router to the new docker RPM version and re-launches the PXE server docker container for the changes to reflect.

Note After activating the **xr-pxeserver** RPM package, the **pxe_svr_mgr** process installs the **pxe-server-docker.rpm** with application manager only when the ongoing install operation is committed and no more install operations are pending.

Step 2 Verify the PXE server docker container application package installed— Use the following command to verify the package installed:

```
Router#show appmgr packages installed
Package
------------------------------------------------------------
pxe-server-2.1.0-ThinXR.x86_64
```

Step 3 Create the following folder structure in the **/misc/disk1/** path and copy the **dhcp.conf** and **image.iso** files to their respective folders, as shown below:

```
/server
|---- config
|     |---- dhcpd.conf
|     |---- dhcpd6.conf
|---- images
|     |---- Image-boot.iso
----logs
```

Note The PXE server docker container uses the "**/server/**" folder for its operations. This path is mounted to PXE server docker using the application manager configuration.

In case of a dual RP system, it is recommended to create this directory structure under "**/misc/disk1/mirror/**". This automatically syncs the PXE server related files to the standby RP node. Therefore, all these files will be available on the new active RP node after RPFO. Otherwise, the user must create the directory structure again and copy all the necessary files for the PXE server docker container.

Step 4 Configure and activate the PXE server docker container application— Use the following set of commands to configure and activate the PXE server docker container application on the interface BVI301:

```
Router#config
Router(config)#appmgr
Router(config-appmgr)#application pxeserver
Router(config-application)# activate type docker source pxe-server docker-run-opts "-it --restart
always --cap-add=NET_ADMIN --net=host --log-opt max-size=20m --log-opt max-file=3 -v
/misc/disk1/mirror/server:/server " docker-run-cmd "-i BV301 -4 172.16.0.0/12 -6 2001:DB8::/48 -l
/server/images -t”
Router(config-application)#commit
  !
!
where,
--cap-add=NET_ADMIN --net=host (mandatory)
-i <interface> (mandatory)
-4 <secondary ipv4 address of BVI>
-6 <ipv6 address of BVI>
-l <location of image stored> (default /server/images)
-t <1 - tftp enabled, 0 - tftp disabled>
```

Step 5 Verify the PXE server docker container status—Use the following command to verify the PXE server docker container status:

```
Router(config)#appmgr application exec name pxeserver docker-exec-cmd status                      C
dhcpd                             RUNNING   pid 94, uptime 0:00:05
dhcpd6                            RUNNING   pid 95, uptime 0:00:05
monitor                           RUNNING   pid 96, uptime 0:00:05
nginx                             RUNNING   pid 97, uptime 0:00:05
syslogd                           RUNNING   pid 98, uptime 0:00:05
tftp-hpa4                         RUNNING   pid 101, uptime 0:00:05
tftp-hpa6                         RUNNING   pid 104, uptime 0:00:05
```

What to do next

The PXE server docker container is active and now the clients can download the boot image and the configuration file from the PXE server (Cisco 8201 router).

CPU-Based Packet Generator

Table 1: Feature History Table

Feature Name	Release Information	Feature Description
CPU-Based Packet Generator	Release 24.2.1	You can now use a CPU-based packet generator for IOS-XR routers to simplify the diagnostic process for routers experiencing problems. This tool allows you to generate a wide range of traffic streams directly within the production environment without physically isolating the routers and moving them to a lab setup. This tool is beneficial in environments that use routers from different vendors or different models from the same vendor. The feature introduces the CLI Options command with different options to generate different types of packets.

Need for CPU-Based Packet Generator

Diagnosing network problems in production environments, such as traffic drops and mis-forwarding issues, is crucial for network management. Traditionally, routers are physically isolated for debugging, requiring moving equipment into lab environments with traffic generators.The CPU-Based Packet Generator can be used in the production environment, eliminating the need to isolate the routers to a lab environment for troubleshooting purposes.

Benefits of CPU-Based Packet Generator

- Versatile Traffic Crafting: Create complex nested packets, such as IPinIPinIPinIP, to test and diagnose a variety of scenarios.
- In-Production Diagnosis: Directly diagnose routers in a problem state without disrupting the network setup.

Restrictions of CPU-Based Packet Generator

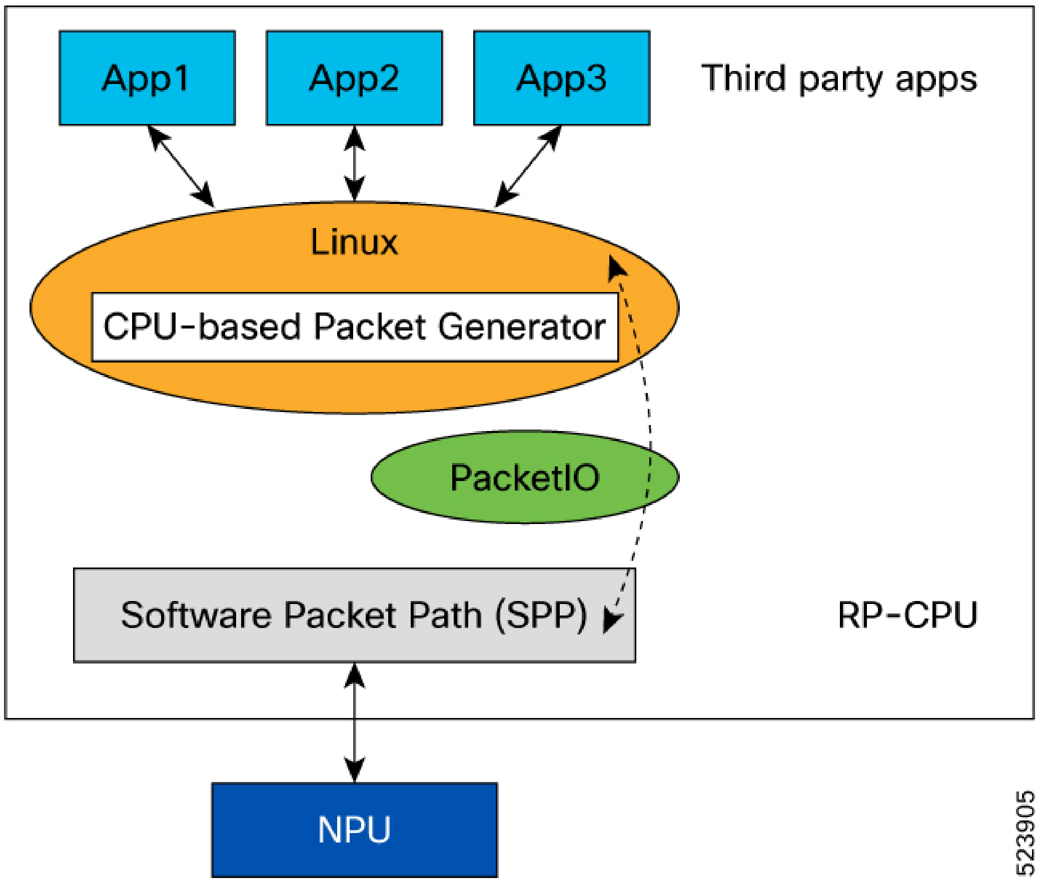
- CPU-based packet generators are not optimized for high-speed packet processing; therefore, they may not match the performance of NPU-based packet generators.
- CPU-based packet generators can potentially introduce higher CPU loads during operation, which may affect the router performance.

- The probe packet rate is 40 kpps for Cisco 8000 Series Routers.

Topology of CPU-Based Packet Generator

The following diagram depicts the software architecture of CPU-based packet generator.

Figure 3: Architecture of CPU-Based Packet Generator



The Cisco IOS-XR PacketIO serves as a host for third-party applications on the XR platform, with PacketIO infrastructure facilitating packet transport and interactions between Linux and XR environments. Leveraging this existing infrastructure, the CPU-based packet generator is implemented as a Linux application and packaged within the supported XR platform base image, ensuring seamless distribution.

The Linux infrastructure maintains a database of all XR interfaces including bundles. The CPU-based packet generator is used to send a specific packet type over a chosen interface.

Capabilities of CPU-based Packet Generator

- **Support different packet types:** The CPU-based packet generator supports various packet types, including:
  - ARP

  - TCP
  - UDP
  - GRE
  - MPLS
  - IPinIP
  - ICMPv4
  - ICMPv6

- **Corrupt or error packet generation:** There are times when routers receive packets that are either corrupted or contain errors for various reasons. To identify and troubleshoot these issues, it becomes necessary to generate similar packets that can be used for debugging purposes. The CPU-based packet generator can create these packets and aid debugging.

  Examples include:

  - IPv4 packet with TTL 0
  - IPv4 packet with wrong checksum
  - IPv4 packet with mismatch between IP option length field and the IP header

How to Use CPU-based Packet Generator?

You can use CPU-based packet generator using:

- **CLI:** Use the **packetgen** command with different options to run the tool from XR bash environment. As the XR interfaces show up as Linux interfaces in bash environment, you can directly use the XR interface names.
- **pcap file:** Use an already captured pcap file in production routers and replay it.

  packetgen -i interface_name -pcap pcap_file

CLI Options

The following table outlines the different options available for the **packetgen** command.

Table 2: Packetgen CLI Options

Option	Description
-accounting	Turn on accounting for packets. Only works if packets come back to the packet generator.
-arp-destination-hw-address string	ARP target hardware address (default: uses interface MAC address)
-arp-destination-ip-address string	ARP target IP address (default: 127.0.0.1 or ::1)

Option	Description
-arp-operation uint	ARP operation (1: request, 2: reply , 3: rarp)
-arp-source-hw-address string	ARP sender hardware address (default : uses interface MAC)
-arp-source-ip-address string	ARP sender IP address (default: uses interface IP)
-burst int	Number of packets to be injected at a time. To be used in conjunction with -sleep.
-count int	Number of packets to be generated.
-data-type string	constant, incrementing, random (default: no payload)
-ethernet-dmac string	Destination MAC address (default: ff:ff:ff:ff:ff:ff)
-ethernet-smac string	Source MAC address (default: use interface MAC address)
-file string	Write packets to file
-gre	Enable GRE
-gre-checksum-present	Enable GRE checksum present bit
-gre-key-present	Enable GRE key present bit
-gre-over-mpls	Enable GRE over MPLS
-gre-protocol uint	Set the protocol type of the GRE payload (default: 0x0800 (IP)
-gre-seq-present	Enable GRE sequence number present bit
-gre-version uint	Set the GRE version number (default 0)
-header string	Custom header for all packets
-hex	Print hex dump of packets
-i string	Interface name for packet injection
-icmp-code uint	ICMP code (default: 0)
-icmp-type uint	ICMP type (default: 0)
-inc-dmac	Increment destination MAC
-inc-smac	Increment source mac
-inner-ethernet-dmac string	Inner Ethernet destination MAC address (default: ff:ff:ff:ff:ff:ff)
-inner-ethernet-smac string	Inner Ethernet source MAC address (default: ff:ff:ff:ff:ff:ff)

Option	Description
-inner-ip-checksum uint	Inner IP checksum (default: compute checksum automatically)
-inner-ip-dont-fragment uint	Set inner IP Don't Fragment flag as 1
-inner-ip-dst string	Inner destination IP address (default: 127.0.0.1 or ::1)
-inner-ip-flow-label uint	Inner IPv6 Flow Label value (default: 0)
-inner-ip-frag-offset uint	Inner IP fragment offset in units of 64-bits (e.g. 1 = 64 bits)
-inner-ip-protocol string	Inner IP protocol . Supports protocol text (TCP, UDP) and code (63 for TCP) (default: TCP)
-inner-ip-src string	Inner source IP address (default: 127.0.0.1 or ::1)
-inner-ip-tos uint	Inner IP Type Of Service (TOS) value (default: 0)
-inner-ip-traffic-class uint	ip-traffic-class (traffic-class) value (default: 0)
-inner-ip-ttl uint	Inner IP time to live (ttl). (Default ttl = 64
-inner-ip-version int	Inner IP version (default: 4)
-inner-vlan-id uint	Inner VLAN id (default: 0)
-inner-vlan-tpid uint	Inner VLAN ethernet type (default: 33024 :Dot1Q)
-inner-vlan-vpri uint	Inner VLANpriority (default: 0
-ip-checksum string	IP checksum (default: compute checksum automatically)
-ip-dont-fragment string	Set IP flag -ip-dont-fragment 0 -> 000 Nothing set -ip-dont-fragment 1 -> 001 More Fragments -ip-dont-fragment 2 -> 010 Dont Fragment -ip-dont-fragment 4 -> 100 set reserved bit
-ip-dst string	Destination IP address (default: 127.0.0.1 or ::1)
-ip-flow-label string	IPv6 Flow Label value (default: 0)
-ip-frag-offset string	Fragment offset in units of 64-bits (1 = 64 bits)
-ip-protocol string	IP protocol. Supports protocol text (TCP, UDP, GRE, VXLAN, ICMP, NDP) and code (63 for TCP) (default: TCP)
-ip-src string	Source IP address (default: use interface ip)
-ip-tos string	IP Type Of Service value (default: 0)
-ip-traffic-class string	IP traffic class (traffic-class) value (default: 0)

Option	Description
-ip-ttl string	IP time to live (ttl). (Default ttl = 64
-ip-version string	IP version should always be set for accurate IP packet creation, ip version (default: 4).
-mpls-exp string	Comma separated MPLS EXP (Experimental) value (default: 0)
-mpls-label string	Comma separated list of Multiprotocol Label Switching (MPLS) labels to be added to the packet. Specified from top to bottom
-mpls-ttl string	Comma separated MPLS TTL (Time To Live) value (default: 64)
-ndp string	Specify the neighbor discovery protocol: nbr-solicit, nbr-advt
-ndp-target-address string	NDP target address (default: for advertisement source IP, for solicitation destination IP
-pcap string	File to replay pcap
-progress	Display a progress bar
-seed int	Seed for pseudo random payload generator
-size int	Size of payload
-sleep string	Time duration to sleep during each burst. To be used together with -burst.
-stdout	Print packets to stdout
-tcp-dport int	TCP destination port (default: 40000)
-tcp-flags string	Set TCP control flags: • U (Urgent): Indicates that the data should be processed urgently. • A (Acknowledgement): Acknowledges the receipt of data. • P (Push): Instructs the sender to push the data to the receiving application immediately. • R (Reset): Resets the connection. • S (Synchronize): Synchronizes sequence numbers to initiate a connection. • F (Finish): Indicates the sender has finished sending data and wants to terminate the connection.
-tcp-sport int	TCP source port (default: 40000)
-udp-dport int	UDP destination port (default: 40000)
-udp-sport int	UDP source port (default: 40000)
-vlan-id uint	VLAN id (default: 0)

Option	Description
-vlan-tpid uint	VLAN ethernet type (default: 33024 :Dot1Q)
-vlan-vpri uint	VLAN priority (default: 0
-vxlan-udp-dport int	UDP destination port for VXLAN (default: 4789)
-vxlan-udp-sport int	UDP source port for VXLAN (default: 0)
-vxlan-vni uint	VXLAN VNI (default: 0)

Sample Commands

This section lists sample commands for some common packet types.

Table 3: Sample Packetgen Commands

Packet Type	Sample Command
ARP	packetgen -i enp0s8 -ip-ttl 32 -arp-operation 1 -progress -count 10000 -inc-smac -arp-destination-ip-address 192.168.56.1
TCP	packetgen -i enp0s8 -ip-ttl 32 -tcp-sport 40000 -progress -count 10000 -inc-smac
UDP	packetgen -i enp0s8 -ip-ttl 32 -udp-sport 40000 -progress -count 10000 -inc-smac
ICMP - PING	packetgen -i enp0s8 -ip-ttl 32 -icmp-type 8 -progress -count 10000 -ip-dst 192.168.56.1
GRE	packetgen -i enp0s8 -ip-ttl 32 -gre -count 100 -inner-ip-ttl 32 -tcp-sport 3222 -progress
IP in IP	packetgen -i enp0s8 -count 100 -tcp-sport 3222 -progress -ip-src="1.1.1.1,2.2.2.2"
ETHER-IP	packetgen -i enp0s8 -ip-ttl 32 -count 100 -inner-ip-version 6 -tcp-sport 3222 -progress -inner-ethernet-smac ff:ff:ff:ff:ff:ff
VLAN	packetgen -i enp0s8 -ip-ttl 32 -tcp-sport 40000 -progress -count 10000 -inc-smac -vlan-id 2
QinQ	packetgen -i enp0s8 -ip-ttl 32 -tcp-sport 40000 -progress -count 10000 -inc-smac -vlan-id 2 -inner-vlan-id 2
VXLAN	packetgen -i enp0s8 -ip-ttl 32 -tcp-sport 40000 -progress -count 10000 -inc-smac -vxlan-vni 3 -vxlan-udp-sport 4444 -inner-ip-version 4 -inner-ethernet-smac ff:ff:ff:ff:ff:ff -data-type constant
NDP	packetgen -i enp0s8 -ip-version 6 -ndp nbr-advt -count 100 -ip-checksum 1 -progress
MPLS	packetgen -i enp0s8 -ip-version 4 -mpls-label 1,2,3,4,5 -tcp-sport 4556 -count 1000 -progress

Command Example

This section shows an example command to send an ICMP ping request from source address 10.0.0.1 to destination address 10.0.0.2 via interface Hu0_0_0_25.

```
Router# bash
[ios:~]$ packetgen -i Hu0_0_0_25 -ip-ttl 32 -progress -count 50 -icmp-type 8 -ip-dst 10.0.0.2
 -ip-src 10.0.0.1 --ethernet-smac 78:c5:51:84:48:c4 --ethernet-dmac 00:00:00:1e:ca:fc
INFO[0000] [ETH IP ICMP]
INFO[0000] Setting SRC IP to 10.0.0.1
INFO[0000] Setting DST IP to 10.0.0.2
INFO[0000] Opening Handle Hu0_0_0_25
INFO[0000] Opened Handle Hu0_0_0_25
INFO[0000] Starting Packet Injection
Sending Packets... 2% | | (1/50, 254 packet/s) [0s:0s] /* Truncated output. */

Address Age Hardware Addr State Type Interface
10.0.0.1 - 78c5.5184.48c4
Interface ARPA HundredGigE0/0/0/25
10.0.0.2 00:50:23 0000.001e.cafc Dynamic ARPA HundredGigE0/0/0/25

Source stats:
Stat Name      Port Name           Control Packet Tx.  Control Packet Rx.  Ping Reply Tx.
20.0.0.2/
Card01/Port01  Ethernet - VM - 001  51                 51                  50

Interface stats:
Input    Punt XIPC  InputQ     XIPC         PuntQ
ClientID Drop/Total Drop/Total Cur/High/Max Cur/High/Max
--------------------------------------------------------------------------------
ipv6_icmp 0/0        0/0       0/0/1000     0/0/1000
icmp      0/50       0/0       0/15/1000    0/0/1000
```