Network Address Translation (NAT)

Network Address Translation (NAT) modifies addresses, ports, or ICMP identifiers in forwarded packets so that traffic from one network can use the address of another interface.

NuttX supports build-time selectable NAT modes for NAT44 and NAT66:

`full-cone NAT`
: Maps a local address and port, or ICMP identifier, to an external address and port or identifier. The peer endpoint is not part of the NAT entry key.

`symmetric NAT`
: Maps a local endpoint to an external endpoint for a specific peer endpoint. The peer endpoint is part of the NAT entry key, so traffic from the same local endpoint to different peers may use different mappings.

The Nuttx NAT implementation supports:

- TCP
- UDP
- ICMP
  - ECHO (REQUEST / REPLY)
  - Error Messages (DEST_UNREACHABLE / TIME_EXCEEDED / PARAMETER_PROBLEM)
- ICMPv6
  - ECHO (REQUEST / REPLY)
  - Error Messages (DEST_UNREACHABLE / PACKET_TOO_BIG / TIME_EXCEEDED / PARAMETER_PROBLEM)

Workflow

- Outbound
  - **LAN** -> **Forward** -> **NAT** (only if targeting at WAN) -> **WAN**
  - All packets from **LAN** and targeting at **WAN** will be masqueraded with `local ip:port` changed to `external ip:port` ,
- Inbound
  - **WAN** -> **NAT** (only from WAN, change destination) -> **Forward** -> **LAN**
  - Packets from **WAN** will try to be changed back from `external ip:port` to `local ip:port` and send to **LAN**.

Configuration Options

`CONFIG_NET_NAT`
: Enable Network Address Translation. This option depends on `CONFIG_NET_IPFORWARD` .

`CONFIG_NET_NAT44` / `CONFIG_NET_NAT66`
: Enable IPv4-to-IPv4 / IPv6-to-IPv6 NAT. This option depends on `CONFIG_NET_NAT` .

`CONFIG_NET_NAT44_FULL_CONE` / `CONFIG_NET_NAT66_FULL_CONE`
: Select full-cone NAT mode for NAT44 / NAT66.

`CONFIG_NET_NAT44_SYMMETRIC` / `CONFIG_NET_NAT66_SYMMETRIC`
: Select symmetric NAT mode for NAT44 / NAT66.

`CONFIG_NET_NAT_HASH_BITS`
: Set the number of bits used for the NAT entry hash table. The hash table has `1 << CONFIG_NET_NAT_HASH_BITS` buckets.

`CONFIG_NET_NAT_TCP_EXPIRE_SEC`
: Set the expiration time, in seconds, for idle TCP NAT entries. The default value is 86400 seconds, as suggested by RFC 2663, Section 2.6, Page 5. But we may set it to shorter time like 240s for better performance.

`CONFIG_NET_NAT_UDP_EXPIRE_SEC`
: Set the expiration time, in seconds, for idle UDP NAT entries.

`CONFIG_NET_NAT_ICMP_EXPIRE_SEC`
: Set the expiration time, in seconds, for idle ICMP NAT entries.

`CONFIG_NET_NAT_ICMPv6_EXPIRE_SEC`
: Set the expiration time, in seconds, for idle ICMPv6 NAT entries.

`CONFIG_NET_NAT_ENTRY_RECLAIM_SEC`
: Set the time to auto reclaim all expired NAT entries. A value of zero will disable auto reclaiming. Because expired entries will be automatically reclaimed when matching inbound/outbound entries, so this config does not have significant impact when NAT is normally used, but very useful when the hashtable is big and there are only a few connections using NAT (which will only trigger reclaiming on a few chains in hashtable).

Usage

- NAT can be enabled directly from C code:

  `int` **nat_enable**`(FAR struct` **net_driver_s** `*dev)`

  Enable NAT on a network device. Outbound packets forwarded through this device may be translated.

  Returns: Zero is returned if NAT is successfully enabled on the device. A negated errno value is returned on failure.

  `int` **nat_disable**`(FAR struct` **net_driver_s** `*dev)`

  Disable NAT on a network device.

  Returns: Zero is returned if NAT is successfully disabled on the device. A negated errno value is returned on failure.

- NAT can also be enabled from the NSH with the `iptables` command. The rule is added to the NAT table, and the output interface passed to `-o` is the NuttX external interface:

```
nsh> iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

  To remove the rule, use the same rule specification with `-D` :

```
nsh> iptables -t nat -D POSTROUTING -o eth0 -j MASQUERADE
```

> **Note**
>
> The commands above are NuttX `iptables` commands. They configure NAT inside NuttX. They are separate from Linux host `iptables` commands.

Validation

The following setup validates NAT with the NuttX simulator on an Ubuntu 22.04 x86_64 Linux host. IPv4 is used as the main example.

The test uses two Linux network namespaces:

- `private` represents a host behind NuttX NAT.
- `public` represents a host on the external side of NuttX NAT.

This keeps the test local to the host and does not require Internet access.

The topology is:

In this topology:

- `eth0` is the NuttX private-side interface.
- `eth1` is the NuttX external interface. NAT is enabled on `eth1` .
- `tap0` is moved into the Linux `private` namespace and is the peer of `eth0` .
- `tap1` is moved into the Linux `public` namespace and is the peer of `eth1` .

Step 1: Configure NuttX simulator

Configure NuttX simulator with at least two TAP devices, IP forwarding and NAT:

```
CONFIG_NET_IPFORWARD=y
CONFIG_NET_NAT=y
CONFIG_NET_NAT44=y
CONFIG_simulator_NETDEV_NUMBER=2
```

Step 2: Start NuttX simulator

Start the NuttX simulator and make sure it creates two TAP interfaces on the Linux host. Creating or configuring TAP interfaces may require root privileges or capabilities such as `CAP_NET_ADMIN` . The interface names are assumed to be `tap0` and `tap1` in the commands below.

Step 3: Configure Linux namespaces

Run the following commands on the Linux host:

```
NS_PRIVATE="private"
NS_PUBLIC="public"

IF_PRIVATE="tap0"
IP_PRIVATE="192.168.0.1"
IP_NUTTX_PRIVATE="192.168.0.2"

IF_PUBLIC="tap1"
IP_PUBLIC="10.0.1.1"
IP_NUTTX_PUBLIC="10.0.1.2"

sudo ip netns add "$NS_PRIVATE"
sudo ip netns add "$NS_PUBLIC"

sudo ip link set "$IF_PRIVATE" netns "$NS_PRIVATE"
sudo ip link set "$IF_PUBLIC" netns "$NS_PUBLIC"

# Private namespace
sudo ip netns exec "$NS_PRIVATE" ip link set lo up
sudo ip netns exec "$NS_PRIVATE" ip link set "$IF_PRIVATE" up
sudo ip netns exec "$NS_PRIVATE" ip addr add "$IP_PRIVATE/24" dev "$IF_PRIVATE"
sudo ip netns exec "$NS_PRIVATE" ip route add default via "$IP_NUTTX_PRIVATE" dev "$IF_PRIVATE"

# Public namespace
sudo ip netns exec "$NS_PUBLIC" ip link set lo up
sudo ip netns exec "$NS_PUBLIC" ip link set "$IF_PUBLIC" up
sudo ip netns exec "$NS_PUBLIC" ip addr add "$IP_PUBLIC/24" dev "$IF_PUBLIC"
```

The `public` namespace does not need a route back to `192.168.0.0/24` for normal NAT validation, because return traffic is addressed to the translated external address `10.0.1.2` .

Step 4: Configure NuttX network interfaces

Configure the NuttX interface addresses from NSH:

```
nsh> ifconfig eth0 192.168.0.2
nsh> ifup eth0
nsh> ifconfig eth1 10.0.1.2
nsh> ifup eth1
nsh> addroute default 10.0.1.1 eth1
```

Enable NAT on NuttX `eth1` . Either call `nat_enable()` during network initialization or run this command from NSH:

```
nsh> iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
```

Step 5: Validate basic connectivity

Before testing NAT, verify that each direct link works.

From NuttX, ping the private-side peer:

```
nsh> ping 192.168.0.1
```

From NuttX, ping the public-side peer:

```
nsh> ping 10.0.1.1
```

From the private namespace, ping the NuttX private-side interface:

```
sudo ip netns exec private ping 192.168.0.2
```

From the public namespace, ping the NuttX external interface:

```
sudo ip netns exec public ping 10.0.1.2
```

Step 6: Test TCP NAT

Start an iperf server in the `public` namespace:

```
sudo ip netns exec public iperf -B 10.0.1.1 -s -i 1
```

Run the iperf client from the `private` namespace:

```
sudo ip netns exec private iperf -B 192.168.0.1 -c 10.0.1.1 -i 1
```

The server should see the connection as coming from the NuttX external address `10.0.1.2` , not from the private address `192.168.0.1` .

Step 7: Test ICMP NAT

Run ping from the `private` namespace to the `public` namespace:

```
sudo ip netns exec private ping 10.0.1.1
```

On the public side, the ICMP Echo Request should be translated to use the NuttX external address `10.0.1.2` as the source address.

Step 8: Test HTTP NAT

Start a simple HTTP server in the `public` namespace:

```
sudo ip netns exec public python3 -m http.server 8000 -b 10.0.1.1
```

Run HTTP requests from the `private` namespace:

```
for i in $(seq 1 20000); do
  sudo ip netns exec private curl -sS -o /dev/null 'http://10.0.1.1:8000/'
done
```

Step 9: Capture packets

Capture on the private side:

```
sudo ip netns exec private tcpdump -nn -i tap0
```

Capture on the public side:

```
sudo ip netns exec public tcpdump -nn -i tap1
```

For outbound IPv4 traffic, `tap0` should show packets sourced from `192.168.0.1` and `tap1` should show the same flow after translation, sourced from `10.0.1.2` . The inbound return path should show the reverse translation.