ハイパーバイザの作り方〜ちゃんと理解する仮想化技術〜 第１９回 bhyve における仮想 NIC の実装

はじめに

これまでに、ゲスト上で発生した IO アクセスのハンドリング方法、virtio-net の仕組みなど、仮想 NIC の実現方法について解説してきました。今回の記事では、/usr/sbin/bhyve が、仮想 NIC のインタフェースである virt-net に届いたパケットをどのように送受信しているのかを解説していきます。

bhyve における仮想 NIC の実装

bhyve では、ユーザプロセスである/usr/sbin/bhyve にて仮想 IO デバイスを提供しています。また、仮想 IO デバイスの一つである NIC は、TAP を利用して機能を提供しています。bhyve では仮想 NIC である TAP を物理 NIC とブリッジすることにより、物理 NIC が接続されている LAN へ参加させることができます（図１）。

どのような経路を経て物理 NIC へとパケットが送出されていくのか、ゲスト OS がパケットを送信しようとした場合を例として見てみましょう。

1. NIC への I/O 通知

ゲスト OS は virtio-net ドライバを用いて、共有メモリ上のリングバッファにパケットを書き込み、IO ポートアクセスによってハイパーバイザにパケット送出を通知します。IO ポートアクセスによって VMExit が発生し、CPU の制御がホスト OS の vmm.ko のコードに戻ります[*1]。vmm.ko はこの VMExit を受けて ioctl を return し、ユーザランドプロセスである/usr/sbin/bhyve へ制御を移します。

[*1] I/O アクセスの仮想化と VMExit については連載 (“第３回 I/O 仮想化「デバイス I/O 編」”) を参照してください。

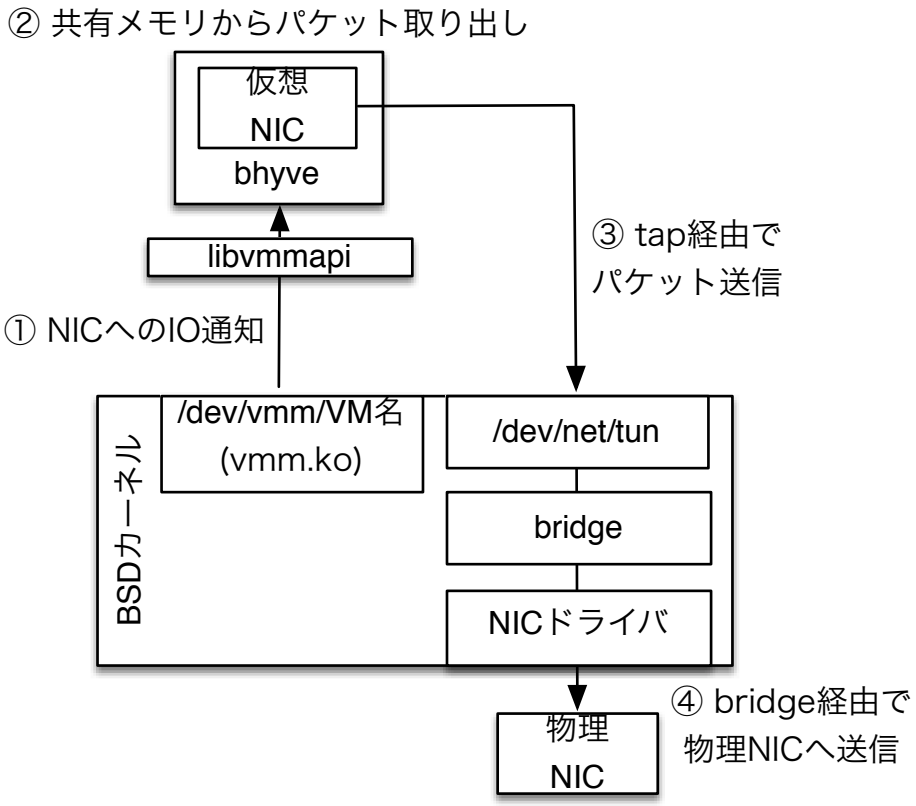

図 1　パケット送信手順

2. 共有メモリからパケット取り出し

ioctl の return を受け取った/usr/sbin/bhyve は仮想 NIC の共有メモリ上のリングバッファからパケットを取り出します[*2]。

3. tap 経由でパケット送信

２で取り出したパケットを write() システムコールで/dev/net/tun へ書き込みます。

4. bridge 経由で物理 NIC へ送信

TAP はブリッジを経由して物理 NIC へパケットを送出します。

[*2] 仮想 NIC のデータ構造とインタフェースの詳細に関しては、連載 (“第１１回 Virtio による準仮想化デバイス その１「Virtio の概要と Virtio PCI」”) ・(“第１２回 Virtio による準仮想化デバイスその２「Virtqueue と virtio-Net の実現」”) を参照して下さい。

受信処理ではこの逆の流れを辿り、物理 NIC から tap を経由して/usr/sbin/bhyve へ届いたパケットが virtio-net のインタフェースを通じてゲスト OS へ渡されます。

TAP とは

bhyve で利用されている TAP についてもう少し詳しくみていきましょう。TAP は FreeBSD カーネルに実装された仮想 Ethernet デバイスで、ハイパーバイザ／エミュレータ以外では VPN の実装によく使われています[*3]。

物理 NIC 用のドライバは物理 NIC との間でパケットの送受信処理を行いますが、TAP は/dev/net/tun を通じてユーザプロセスとの間でパケットの送受信処理を行います。このユーザプロセスが Socket API を通じて、TCPv4 で VPN プロトコルを用いて対向ノードとパケットのやりとりを行えば、TAP は対向ノードにレイヤ 2 で直接接続されたイーサーネットデバイスに見えます。

これが OpenVPN などの VPN ソフトが TAP を用いて実現している機能です（図 2）。

では、ここで TAP がどのようなインタフェースをユーザプロセスに提供しているのか見ていきましょう。TAP に届いたパケットを UDP でトンネリングするサンプルプログラムの例をコードリスト 1 に示します。

■コードリスト 1，TAP サンプルプログラム（Ruby）

```
require "socket"
TUNSETIFF = 0x400454ca
IFF_TAP = 0x0002
PEER = "192.168.0.100"
PORT = 9876
# TUNTAP をオープン
tap = open("/dev/net/tun", "r+")
# TUNTAP のモードを TAP に、インタフェース名を "tapN" に設定
tap.ioctl(TUNSETIFF, ["tap%d", IFF_TAP].pack("a16S"))
# UDP ソケットをオープン
sock = UDPSocket.open
# ポート 9876 を LISTEN
sock.bind("0.0.0.0", 9876)
while true
    # ソケットか TAP にパケットが届くまで待つ
    ret = IO::select([sock, tap])
    ret[0].each do |d|
```

[*3] 正確には TUN/TAP として知られており、TAP がイーサネットレイヤでパケットを送受信するインタフェースを提供するのに対し TUN デバイスは IP レイヤでパケットを送受信するインタフェースを提供します。また、TUN/TAP は FreeBSD の他にも Linux、Windows、OS X など主要な OS で実装されています。

```
        if d == tap # TAP にパケットが届いた場合
            # TAP からパケットを読み込んでソケットに書き込み
            sock.send(tap.read(1500), 0, Socket.pack_sockaddr_in(PORT, PEER))
        else # ソケットにパケットが届いた場合
            # ソケットからパケットを読み込んで TAP に書き込み
            tap.write(sock.recv(65535))
        end
    end
end
```

ユーザプロセスが TAP とやりとりを行うには、/dev/net/tun デバイスファイルを用います。

パケットの送受信は通常のファイル IO と同様に read()、write() を用いる事が出来ますが、送受信処理を始める前に TUNSETIFF ioctl を用いて TAP の初期化を行う必要があります。ここでは、TUNTAP のモード（TUN を使うか TAP を使うか）と ifconfig に表示されるインタフェース名の指定を行います。

ここで TAP に届いたパケットを UDP ソケットへ、UDP ソケットに届いたパケットを TAP へ流すことにより、TAP を出入りするパケットを UDP で他ノードへトンネリングすることが出来ます（図 2 右相当の処理)。

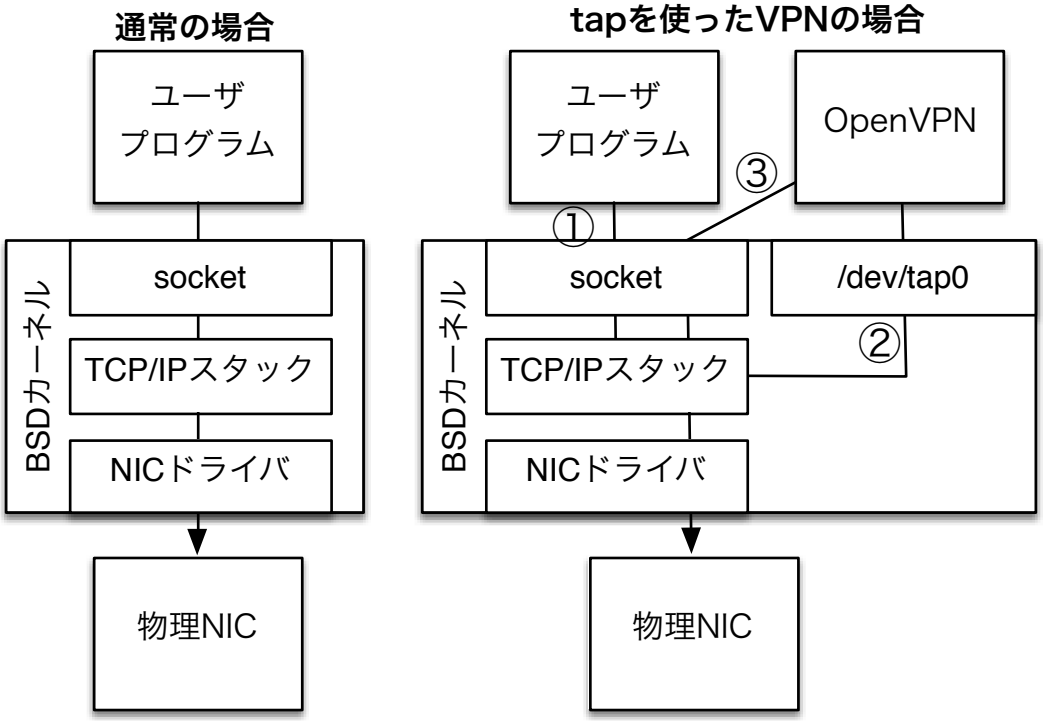

図 2　通常の NIC ドライバを使ったネットワークと TAP を使った VPN の比較

bhyve における仮想 NIC と TAP

VPN ソフトでは TAP を通じて届いたパケットをユーザプロセスから VPN プロトコルでカプセル化して別ノード送っています。

ハイパーバイザで TAP を用いる理由はこれとは異なり、ホスト OS のネットワークスタックに仮想 NIC を認識させ物理ネットワークに接続し、パケットを送受信するのが目的です。このため、VPN ソフトではソケットと TAP の間でパケットをリダイレクトしていたのに対して、ハイパーバイザでは仮想 NIC と TAP の間でパケットをリダイレクトする事になります。

それでは、このリダイレクトの部分について bhyve のコードを実際に確認してみましょう（リスト２）。

■コードリスト２，/usr/sbin/bhyve **の仮想** NIC **パケット受信処理**

```
/* TAP からデータが届いた時に呼ばれる */
static void
pci_vtnet_tap_rx(struct pci_vtnet_softc *sc)
{
    struct vqueue_info *vq;
    struct virtio_net_rxhdr *vrx;
    uint8_t *buf;
    int len;
    struct iovec iov;
    〜 略 〜
    vq = &sc->vsc_queues[VTNET_RXQ];
    vq_startchains(vq);
    〜 略 〜
    do {
    〜 略 〜
        /* 受信キュー上の空きキューを取得 */
        assert(vq_getchain(vq, &iov, 1, NULL) == 1);
    〜 略 〜
        vrx = iov.iov_base;
        buf = (uint8_t *)(vrx + 1); /* 空きキューのアドレス */
        /* TAP から空きキューへパケットをコピー */
        len = read(sc->vsc_tapfd, buf,
        iov.iov_len - sizeof(struct virtio_net_rxhdr));
        /* TAP にデータが無ければ return */
        if (len < 0 && errno == EWOULDBLOCK) {
```

```
    ～ 略 ～
            vq_endchains(vq, 0);
            return;
        }
    ～ 略 ～
        memset(vrx, 0, sizeof(struct virtio_net_rxhdr));
        vrx->vrh_bufs = 1; /* キューに接続されているバッファ数 */
    ～ 略 ～
        vq_relchain(vq, len + sizeof(struct virtio_net_rxhdr));
    } while (vq_has_descs(vq)); /* 空きキューがある間繰り返し */
～ 略 ～
    vq_endchains(vq, 1);
}
```

この関数は sc->vsc_tapfd を kqueue()/kevent() でポーリングしているスレッドによって TAP へのパケット着信時コールバックされます。コードの中では、virtio-net の受信キュー上の空きエリアを探して、TAP からキューが示すバッファにデータをコピーしています。これによって、TAP へパケットが届いた時は仮想 NIC へ送られ、仮想 NIC からパケットが届いた時はゲスト OS に送られます。その結果、bhyve の仮想 NIC はホスト OS にとって LAN ケーブルで tap0 へ接続されているような状態になります。

TAP を用いたネットワークの構成方法

前述の状態になった仮想 NIC では、IP アドレスが適切に設定されていればホスト OS とゲスト OS 間の通信が問題なく行えるようになります。しかしながら、このままではホストとの間でしか通信ができず、インターネットや LAN 上の他ノードに接続する方法がありません。この点においては、2 台の PC を LAN ケーブルで物理的に直接つないでいる環境と同じです。

これを解決するには、ホスト OS 側に標準的に搭載されているネットワーク機能を利用します。1 つの方法は、すでに紹介したブリッジを使う方法で、TAP と物理 NIC をデータリンクレイヤで接続し、物理 NIC の接続されているネットワークに TAP を参加させることます。しかしながら、WiFi では仕様によりブリッジが動作しないという制限があったり、LAN から 1 つの物理 PC に対して複数の IP 付与が許可されていない環境で使う場合など、ブリッジ以外の方法でゲストのネットワークを運用したい場合があります。

この場合は、NAT を使ってホスト OS でアドレス変換を行ったうえで IP レイヤでルーティングを行います[*4]。bhyve ではこれらの設定を自動的に行うしくみをとくに提供しておらず、TAP に bhyve を接続する機能だけを備えているので、自分でコンフィギュレーションを行う必要があります。

リスト 3、4 に/etc/rc.conf の設定例を示します。なお、OpenVPN などを用いた VPN 接続に対してブリッ

[*4] NAT を使わずにルーティングだけを行うこともできますが、その場合は LAN 上のノードからゲストネットワークへの経路が設定されていなければなりません。一般的にはそのような運用は考えにくいので、NAT を使うことがほとんどのケースで適切だと思われます。

ジや NAT を行う場合も、ほぼ同じ設定が必要になります。

■リスト 3，ブリッジの場合

```
cloned_interfaces="bridge0 tap0"
autobridge_interfaces="bridge0"
autobridge_bridge0="em0 tap*"
ifconfig_bridge0="up"
```

■リスト 4，NAT の場合

```
firewall_enable="YES"
firewall_type="OPEN"
natd_enable="YES"
natd_interface="em0"
gateway_enable="YES"
cloned_interfaces="tap0"
ifconfig_tap0="inet 192.168.100.1/24 up"
dnsmasq_enable="YES"
```

まとめ

今回は仮想マシンのネットワークデバイスについて解説しました。次回は、仮想マシンのストレージデバイスについて解説します。

ライセンス

Copyright (c) 2014 Takuya ASADA. 全ての原稿データ はクリエイティブ・コモンズ 表示 - 継承 4.0 国際ライセンスの下に提供されています。

参考文献

“第１１回 Virtio による準仮想化デバイス その１「Virtio の概要と Virtio PCI」.” http://syuu1228.github.io/howto_implemen

“第１２回 Virtio による準仮想化デバイスその２「Virtqueue と virtio-Net の実現」.” http://syuu1228.github.io/howto_imple

“第３回 I/O 仮想化「デバイス I/O 編」.” http://syuu1228.github.io/howto_implement_hypervisor/part3.pdf.