\1

Proudly uncool and out of fashion

Tun/Tap interface tutorial

Posted by waldner on 26 March 2010, 1:54 pm

Foreword: please note that the code available here is only for demonstration purposes. If you want to be serious, you'll have to make it more robust and integrate it with other code. Also, the description is by no means a definitive reference on the subject, but rather the result of my experimentation. Please report any bug or error you find in the code or otherwise in this article. Thanks.

Link to the source tarball described in the article: **simpletun**.

Update 18/07/2010: Thanks to this post, I've learned that recent versions of **iproute2** can (finally) create tun/tap devices, although the functionality is (still?) blissfully undocumented. Thus, installing tunctl (UML utilities) or OpenVPN just to be able to create tun devices is no longer needed. The following is with **iproute2-2.6.34**:

```
# ip tuntap help
Usage: ip tuntap { add | del } [ dev PHYS_DEV ]
          [ mode { tun | tap } ] [ user USER ] [ group GROUP ]
          [ one_queue ] [ pi ] [ vnet_hdr ]

Where: USER  := { STRING | NUMBER }
       GROUP := { STRING | NUMBER }
```

Tun/tap interfaces are a feature offered by Linux (and probably by other UNIX-like operating systems) that can do *userspace networking*, that is, allow userspace programs to see raw network traffic (at the ethernet or IP level) and do whatever they like with it. This document attempts to explain how tun/tap interfaces work under Linux, with some sample code to demonstrate their usage.

How it works

Tun/tap interfaces are *software-only interfaces*, meaning that they exist only in the kernel and, unlike regular network interfaces, they have no physical hardware component (and so there's no physical "wire" connected to them). You can think of a tun/tap interface as a regular network interface that, when the kernel decides that the moment has come to send data "on the wire", instead sends data to some userspace program that is attached to the interface (using a specific procedure, see below). When the program attaches to the tun/tap interface, it gets a special file descriptor, reading from which gives it the data that the interface is sending out. In a similar fashion, the program can write to this special descriptor, and the data (which must be properly formatted, as we'll see) will appear as input to the tun/tap interface. To the kernel, it would look like the tun/tap interface is receiving data "from the wire".
The difference between a tap interface and a tun interface is that a tap interface outputs (and must be given) full ethernet frames, while a tun interface outputs (and must be given) raw IP packets (and no ethernet headers are added by the kernel). Whether an interface functions like a tun interface or like a tap interface is specified with a flag when the interface is created.

The interface can be **transient**, meaning that it's created, used and destroyed by the same program; when the program terminates, even if it doesn't explicitly destroy the interface, the interfaces ceases to exist. Another option (the one I prefer) is to make the interface **persistent**; in this case, it is created using a dedicated utility (like `tunctl` or `openvpn --mktun`), and then normal programs can attach to it; when they do so, they must connect using the same type (tun or tap) used to originally create the interface, otherwise they will not be able to attach. We'll see how that is done in the code.

Once a tun/tap interface is in place, it can be used just like any other interface, meaning that IP addresses can be assigned, its traffic can be analyzed, firewall rules can be created, routes pointing to it can be established, etc.

With this knowledge, let's try to see how we can use a tun/tap interface and what can be done with it.

Creating the interface

The code to create a brand new interface and to (re)attach to a persistent interface is essentially the same; the difference is that the former must be run by root (well, more precisely, by a user with the **CAP_NET_ADMIN** capability), while the latter can be run by an ordinary user if certain conditions are met. Let's start with the creation of a new interface.

First, whatever you do, the device **/dev/net/tun** must be opened read/write. That device is also called the **clone device**, because it's used as a starting point for the creation of any tun/tap virtual interface. The operation (as with any open() call) returns a file descriptor. But that's not enough to start using it to communicate with the interface.

The next step in creating the interface is issuing a special **ioctl()** system call, whose arguments are the descriptor obtained in the previous step, the **TUNSETIFF** constant, and a pointer to a data structure containing the parameters describing the virtual interface (basically, its name and the desired operating mode - tun or tap). As a variation, the name of the virtual interface can be left unspecified, in which case the kernel will pick a name by trying to allocate the "next" device of that kind (for example, if tap2 already exists, the kernel will try to allocate tap3, and so on). **All of this must be done by root** (or by a user with the CAP_NET_ADMIN capability - I won't repeat that again; assume it applies everywhere I say "must be run by root").
If the ioctl() succeeds, the virtual interface is created and the file descriptor we had is now associated to it, and can be used to communicate.

At this point, two things can happen. The program can start using the interface right away (probably configuring it with at least an IP address before), and, when it's done, terminate and destroy the interface. The other option is to issue a couple of other special **ioctl()** calls to make the interface persistent, and terminate leaving it in place for other programs to attach to it. This is what programs like `tunctl` or `openvpn --mktun` do, for example. These programs usually can also optionally set the ownership of the virtual interface to a non-root user and/or group, so programs running as non-root but with the appropriate privileges can attach to the interface later. We'll come back to this below.

The basic code used to create a virtual interface is shown in the file **Documentation/networking/tuntap.txt** in the kernel source tree. Modifying it a bit, we can write a barebone function that creates a virtual interface:

```
#include <linux /if.h>
#include <linux /if_tun.h>
```

```
int tun_alloc(char *dev, int flags) {

  struct ifreq ifr;
  int fd, err;
  char *clonedev = "/dev/net/tun";

  /* Arguments taken by the function:
   *
   * char *dev: the name of an interface (or '\0'). MUST have enough
   *   space to hold the interface name if '\0' is passed
   * int flags: interface flags (eg, IFF_TUN etc.)
   */

   /* open the clone device */
   if( (fd = open(clonedev, O_RDWR)) < 0 ) {
     return fd;
   }

   /* preparation of the struct ifr, of type "struct ifreq" */
   memset(&ifr, 0, sizeof(ifr));

   ifr.ifr_flags = flags;   /* IFF_TUN or IFF_TAP, plus maybe IFF_NO_PI */

   if (*dev) {
     /* if a device name was specified, put it in the structure; otherwise,
      * the kernel will try to allocate the "next" device of the
      * specified type */
     strncpy(ifr.ifr_name, dev, IFNAMSIZ);
   }

   /* try to create the device */
   if( (err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0 ) {
     close(fd);
     return err;
   }

  /* if the operation was successful, write back the name of the
   * interface to the variable "dev", so the caller can know
   * it. Note that the caller MUST reserve space in *dev (see calling
   * code below) */
  strcpy(dev, ifr.ifr_name);

  /* this is the special file descriptor that the caller will use to talk
   * with the virtual interface */
  return fd;
}
```

The **tun_alloc()** function takes two parameters:

- **`char *dev`** contains the name of an interface (for example, tap0, tun2, etc.). Any name can be used, though it's probably better to choose a name that suggests which kind of interface it is. In practice, names like tunX or tapX are usually used. If `*dev` is **'\0'**, the kernel will try to create the "first" available interface of the requested type (eg, tap0, but if that already exists, tap1, and so on).
- **`int flags`** contains the flags that tell the kernel which kind of interface we want (tun or tap). Basically, it can either take the value **IFF_TUN** to indicate a TUN device (no ethernet headers in the packets), or **IFF_TAP** to indicate a TAP device (with ethernet headers in packets). Additionally, another flag **IFF_NO_PI** can be ORed with the base value. IFF_NO_PI tells the kernel to not provide packet information. The purpose of IFF_NO_PI is to tell the kernel that packets will be "pure" IP packets, with no added bytes. Otherwise (if IFF_NO_PI is unset), 4 extra bytes are added to the beginning of the packet (2 flag bytes and 2 protocol bytes). IFF_NO_PI need not match between interface creation and reconnection time. Also note that when capturing traffic on the interface with Wireshark, those 4 bytes are never shown.

A program can thus use the following code to create a device:

```
  char tun_name[IFNAMSIZ];
  char tap_name[IFNAMSIZ];
  char *a_name;

  ...

  strcpy(tun_name, "tun1");
  tunfd = tun_alloc(tun_name, IFF_TUN);  /* tun interface */

  strcpy(tap_name, "tap44");
  tapfd = tun_alloc(tap_name, IFF_TAP);  /* tap interface */

  a_name = malloc(IFNAMSIZ);
  a_name[0]='\0';
  tapfd = tun_alloc(a_name, IFF_TAP);    /* let the kernel pick a name */
```

At this point, as said before, the program can either use the interface as is for its purposes, or it can set it persistent (and optionally assign ownership to a specific user/group). If it does the former, there's not much more to be said. But if it does the latter, here's what happens. Two additional ioctl()s are available, which are usually used together. The first syscall can set (or remove) the persistent status on the interface. The second allows assigning ownership of the interface to a regular (non-root) user. Both features are implemented in the programs **`tunctl`** (part of UML utilities) and **`openvpn --mktun`** (and probably others). Let's examine the **tunctl** code since it's simpler, keeping in mind that it only creates tap interfaces, as those are what **user mode linux** uses (code slightly edited and simplified for clarity):

```
...
  /* "delete" is set if the user wants to delete (ie, make nonpersistent)
     an existing interface; otherwise, the user is creating a new
     interface */
  if(delete) {
    /* remove persistent status */
    if(ioctl(tap_fd, TUNSETPERSIST, 0) < 0){
      perror("disabling TUNSETPERSIST");
      exit(1);
    }
    printf("Set '%s' nonpersistent\n", ifr.ifr_name);
  }
  else {
    /* emulate behaviour prior to TUNSETGROUP */
    if(owner == -1 && group == -1) {
```

```
      owner = geteuid();
    }

    if(owner != -1) {
      if(ioctl(tap_fd, TUNSETOWNER, owner) < 0){
        perror("TUNSETOWNER");
        exit(1);
      }
    }
    if(group != -1) {
      if(ioctl(tap_fd, TUNSETGROUP, group) < 0){
        perror("TUNSETGROUP");
        exit(1);
      }
    }

    if(ioctl(tap_fd, TUNSETPERSIST, 1) < 0){
      perror("enabling TUNSETPERSIST");
      exit(1);
    }

    if(brief)
      printf("%s\n", ifr.ifr_name);
    else {
      printf("Set '%s' persistent and owned by", ifr.ifr_name);
      if(owner != -1)
          printf(" uid %d", owner);
      if(group != -1)
          printf(" gid %d", group);
      printf("\n");
    }
  }
  ...
```

These additional ioctl()s must still be run by root. But what we have now is a persistent interface owned by a specific user, so processes running as that user can successfully attach to it.
As said, it turns out that the code to (re)attach to an existing tun/tap interface is the same as the code used to create it; in other words, **tun_alloc()** can again be used. When doing so, for it to be successful three things must happen:

- The interface must exist already and be owned by the same user that is attempting to connect (and probably be persistent)
- the user must have read/write permissions on **/dev/net/tun**
- The flags provided must match those used to create the interface (eg if it was created with IFF_TUN then the same flag must be used when reattaching)

This is possible because the kernel allows the **TUNSETIFF** ioctl() to succeed if the user issuing it specifies the name of an already existing interface and he is the owner of the interface. In this case, no new interface has to be created, so a regular user can successfully perform the operation.

So this is an attempt to explain what happens when **ioctl(TUNSETIFF)** is called, and how the kernel differentiates between the request for the allocation of a new interface and the request to connect to an existing interface:

- If a non-existent or no interface name is specified, that means the user is requesting the allocation of a new interface. The kernel thus creates an interface using the given name (or picking the next available name if an empty name was given). This works only if done by root.
- If the name of an existing interface is specified, that means the user wants to connect to a previously allocated interface. This can be done by a normal user, provided that: the user has appropriate rights on the clone device AND is the owner of the interface (set at creation time), AND the specified mode (tun or tap) matches the mode set at creation time.

You can have a look at the code that implements the above steps in the file **drivers/net/tun.c** in the kernel source; the important functions are **tun_attach()**, **tun_net_init()**, **tun_set_iff()**, **tun_chr_ioctl()**; this last function also implements the various ioctl()s available, including TUNSETIFF, TUNSETPERSIST, TUNSETOWNER, TUNSETGROUP and others.

In any case, no non-root user is allowed to configure the interface (ie, assign an IP address and bring it up), but this is true of any regular interface too. The usual methods (suid binary wrapper, sudo, etc.) can be used if a non-root user needs to do some operation that requires root privileges.

This is a possible usage scenario (one I use all the time):

- The virtual interfaces are created, made persistent, assigned to an user, and configured by root (for example, by initscripts at boot time, using **tunctl** or equivalent)
- The regular users can then attach and detach as many times as they wish from virtual interfaces that they own.
- The virtual interfaces are destroyed by root, for example by scripts run at shutdown time, perhaps using **tunctl -d** or equivalent

Let's try it

After this lengthy but necessary introduction, it's time to do some work with it. So, since this is a normal interface, we can use it as we would another regular interface. For our purposes, there is no difference between tun and tap interfaces; it's the program that creates or attaches to it that must know its type and accordingly expect or write data. Let's create a persistent interface and assign it an IP address:

```
# openvpn --mktun --dev tun2
Fri Mar 26 10:29:29 2010 TUN/TAP device tun2 opened
Fri Mar 26 10:29:29 2010 Persist state set to: ON
# ip link set tun2 up
# ip addr add 10.0.0.1/24 dev tun2
```

Let's fire up a network analyzer and look at the traffic:

```
# tshark -i tun2
Running as user "root" and group "root". This could be dangerous.
Capturing on tun2

# On another console
# ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.115 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.105 ms
...
```

Looking at the output of tshark, we see...nothing. There is no traffic going through the interface. This is correct: since we're pinging the interface's IP address, the operating system correctly decides that no packet needs to be sent "on the wire", and the kernel itself is replying to these pings. If you think about it, it's exactly what would happen if you pinged another interface's IP address (for example eth0): no packets would be sent out. This might sound obvious, but could be a source of confusion at first (it was for me).

Knowing that the assignment of a /24 IP address to an interface creates a connected route for the whole range through the interface, let's modify our experiment and force the kernel to actually send something out of the tun interface (***NOTE: the following works only with kernels < 2.6.36; later kernels behave differently, as explained in the comments***):

```
# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
...

# on the tshark console
...
  0.000000     10.0.0.1 -> 10.0.0.2     ICMP Echo (ping) request
  0.999374     10.0.0.1 -> 10.0.0.2     ICMP Echo (ping) request
  1.999055     10.0.0.1 -> 10.0.0.2     ICMP Echo (ping) request
...
```

Now we're finally seeing something. The kernel sees that the address does not belong to a local interface, and a route for 10.0.0.0/24 exists through the tun2 interface. So it duly sends the packets out tun2. Note the different behavior here between tun and tap interfaces: with a tun interface, the kernel sends out the IP packet (raw, no other headers are present - try analyzing it with tshark or wireshark), while with a tap interface, being ethernet, the kernel would try to **ARP** for the target IP address:

```
# pinging 10.0.0.2 now, but through tap2 (tap)
# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

# on the tshark console
...
  0.111858 82:03:d4:07:62:b6 -> Broadcast    ARP Who has 10.0.0.2?  Tell 10.0.0.1
  1.111539 82:03:d4:07:62:b6 -> Broadcast    ARP Who has 10.0.0.2?  Tell 10.0.0.1
...
```

Furthermore, with a tap interface the traffic will be composed by full ethernet frames (again, you can check with the network analyzer). Note that the MAC address for a tap interface is autogenerated by the kernel at interface creation time, but can be changed using the **SIOCSIFHWADDR** ioctl() (look again in **drivers/net/tun.c**, function **tun_chr_ioctl()**). Finally, being an ethernet interface, the MTU is set to 1500:

```
# ip link show dev tap2
7: tap2:  mtu 1500 qdisc pfifo_fast state UNKNOWN qlen 500
    link/ether 82:03:d4:07:62:b6 brd ff:ff:ff:ff:ff:ff
```

Of course, so far no program is attached to the interface, so all these outgoing packets are just lost. So let's do a step ahead and write a simple program that attaches to the interface and reads packets sent out by the kernel.

A simple program

We're going to write a program that attaches to a tun interface and reads packets that the kernel sends out that interface. Remember that you can run the program as a normal user if the interface is persistent, provided that you have the necessary permissions on the clone device **/dev/net/tun**, you are the owner of the interface, and select the right mode (tun or tap) for the interface. The program is actually a skeleton, or rather the start of a skeleton, since we'll only demonstrate how to read from the device, and only explain what the program can do once it gets the data. We assume that the **tun_alloc()** function we defined earlier is available to the program. Here is the code:

```
  ...
  /* tunclient.c */

  char tun_name[IFNAMSIZ];

  /* Connect to the device */
  strcpy(tun_name, "tun77");
  tun_fd = tun_alloc(tun_name, IFF_TUN | IFF_NO_PI);  /* tun interface */

  if(tun_fd < 0){
    perror("Allocating interface");
    exit(1);
  }

  /* Now read data coming from the kernel */
  while(1) {
    /* Note that "buffer" should be at least the MTU size of the interface, eg 1500 bytes */
    nread = read(tun_fd,buffer,sizeof(buffer));
    if(nread < 0) {
      perror("Reading from interface");
      close(tun_fd);
      exit(1);
    }

    /* Do whatever with the data */
    printf("Read %d bytes from device %s\n", nread, tun_name);
  }

  ...
```

If you configure tun77 as having IP address 10.0.0.1/24 and then run the above program while trying to ping 10.0.0.2 (or any address in 10.0.0.0/24 other than 10.0.0.1, for that matter), you'll read data from the device:

```
# openvpn --mktun --dev tun77 --user waldner
Fri Mar 26 10:48:12 2010 TUN/TAP device tun77 opened
Fri Mar 26 10:48:12 2010 Persist state set to: ON
# ip link set tun77 up
# ip addr add 10.0.0.1/24 dev tun77
# ping 10.0.0.2
...

# on another console
```

```
$ ./tunclient
Read 84 bytes from device tun77
Read 84 bytes from device tun77
...
```

If you do the math, you'll see where these 84 byetes come from: 20 are for the IP header, 8 for the ICMP header, and 56 are the payload of the ICMP echo message as you can see when you run the ping command:

```
$ ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
...
```

Try experimenting with the above program sending various traffic types through the interface (also try using tap), and verify that the size of the data you're reading is correct for the interface type. Each read() returns a full packet (or frame if using tap mode); similarly, if we were to write, we would have to write an entire IP packet (or ethernet frame in tap mode) for each write().

Now what can we do with this data? Well, we could for example emulate the behavior of the target of the traffic we're reading; again, to keep things simple, let's stick with the ping example. We could analyze the received packet, extract the information needed to reply from the IP header, ICMP header and payload, build an IP packet containing an appropriate ICMP echo reply message, and send it back (ie, write it into the descriptor associated with the tun/tap device). This way the originator of the ping will actually receive an answer. Of course you're not limited to ping, so you can implement all kinds of network protocols. In general, this implies parsing the received packet, and act accordingly. If using tap, to correctly build reply frames you would probably need to implement ARP in your code. All of this is exactly what **User Mode Linux** does: it attaches a modified Linux kernel running in userspace to a tap interface that exist on the host, and communicates with the host through that. Of course, being a full Linux kernel, it does implement TCP/IP and ethernet. Newer virtualization platforms like **libvirt** use tap interfaces extensively to communicate with guests that support them like **qemu/kvm**; the interfaces have usually names like **vnet0**, **vnet1** etc. and last only as long as the guest they connect to is running, so they're not persistent, but you can see them if you run **ip link show** and/or **brctl show** while guests are running.
In the same way, you can attach with your own code to the interface and practice network programming and/or ethernet and TCP/IP stack implementation. To get started, you can look at (you guessed it) **drivers/net/tun.c**, functions **tun_get_user()** and **tun_put_user()** to see how the tun driver does that on the kernel side (beware that barely scratches the surface of the complete network packet management in the kernel, which is very complex).

Tunnels

But there's another thing we can do with tun/tap interfaces. We can create **tunnels**. We don't need to reimplement TCP/IP; instead, we can write a program to just relay the raw data back and forth to a remote host running the same program, which does the same thing in a specular way. Let's suppose that our program above, in addition to attaching to the tun/tap interface, also establishes a network connection to a remote host, where a similar program (connected to a local tun/tap interface as well) is running in server mode. (Actually the two programs are the same, who is the server and who is the client is decided with a command line switch). Once the two programs are running, traffic can flow in either direction, since the main body of the code will be doing the same thing at both sites. The network connection here is implemented using TCP, but any other mean can be used (ie UDP, or even ICMP!). You can download the full program source code here: **simpletun**.
Here is the main loop of the program, where the actual work of moving data back and forth between the tun/tap interface and the network tunnel is performed. For clearness, debug statements have been removed (you can find the full version in the source tarball).

```
...
  /* net_fd is the network file descriptor (to the peer), tap_fd is the
     descriptor connected to the tun/tap interface */

  /* use select() to handle two descriptors at once */
  maxfd = (tap_fd > net_fd)?tap_fd:net_fd;

  while(1) {
    int ret;
    fd_set rd_set;

    FD_ZERO(&rd_set);
    FD_SET(tap_fd, &rd_set); FD_SET(net_fd, &rd_set);

    ret = select(maxfd + 1, &rd_set, NULL, NULL, NULL);

    if (ret < 0 && errno == EINTR) {
      continue;
    }

    if (ret < 0) {
      perror("select()");
      exit(1);
    }

    if(FD_ISSET(tap_fd, &rd_set)) {
      /* data from tun/tap: just read it and write it to the network */

      nread = cread(tap_fd, buffer, BUFSIZE);

      /* write length + packet */
      plength = htons(nread);
      nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
      nwrite = cwrite(net_fd, buffer, nread);
    }

    if(FD_ISSET(net_fd, &rd_set)) {
      /* data from the network: read it, and write it to the tun/tap interface.
       * We need to read the length first, and then the packet */

      /* Read length */
      nread = read_n(net_fd, (char *)&plength, sizeof(plength));

      /* read packet */
      nread = read_n(net_fd, buffer, ntohs(plength));

      /* now buffer[] contains a full packet or frame, write it into the tun/tap interface */
      nwrite = cwrite(tap_fd, buffer, nread);
    }
  }

...
```

(for the details of the **read_n()** and **cwrite()** functions, refer to the source; what they do should be obvious. Yes, the above code is not 100% correct with regard to select(), and makes some naive assumptions like expecting that read_n() and cwrite() do not block. As I said, the code is for demonstration purposes only)

Here is the main logic of the above code:

- The program uses **select()** to keep both descriptors under control at the same time; if data comes in from either descriptor, it's written out to the other.
- Since the program usese TCP, the receiver will see a single stream of data, which makes recognizing packet boundaries difficult. So when a packet or frame is written to the network, its length is prepended (2 bytes) to the actual packet.
- When data comes in from the tap_fd descriptor, a single read reads a full packet or frame; thus this can directly be written to the network, with its length prepended. Since that length number is a short int, thus longer than one byte, written in "raw" binary format, **ntohs()/htons()** are used to interoperate between machines with different endianness.
- When data comes in from the network, thanks to the aforementioned trick, we can know how long the next packet is going to be by reading the two-bytes length that precedes it in the stream. When we've read the packet, we write it to the tun/tap interface descriptor, where it will be received by the kernel as coming "from the wire".

So what can you do with such a program? Well, you can create a tunnel! First, create and confgure the necessary tun/tap interfaces on the hosts at both ends of the tunnel, including assigning them an IP address. For this example, I'll assume two tun interfaces: tun11, 192.168.0.1/24 on the local computer, and tun3, 192.168.0.2/24 on the remote computer. **simpletun** connects the hosts using TCP port 55555 by default (you can change that using the -p command line switch). The remote host will run simpletun in server mode, and the local host will run in client mode. So here we go (the remote server is at 10.2.3.4):

```
[remote]# openvpn --mktun --dev tun3 --user waldner
Fri Mar 26 11:11:41 2010 TUN/TAP device tun3 opened
Fri Mar 26 11:11:41 2010 Persist state set to: ON
[remote]# ip link set tun3 up
[remote]# ip addr add 192.168.0.2/24 dev tun3

[remote]$ ./simpletun -i tun3 -s
# server blocks waiting for the client to connect

[local]# openvpn --mktun --dev tun11 --user waldner
Fri Mar 26 11:17:37 2010 TUN/TAP device tun11 opened
Fri Mar 26 11:17:37 2010 Persist state set to: ON
[local]# ip link set tun11 up
[local]# ip addr add 192.168.0.1/24 dev tun11

[local]$ ./simpletun -i tun11 -c 10.2.3.4
# nothing happens, but the peers are now connected

[local]$ ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
64 bytes from 192.168.0.2: icmp_seq=1 ttl=241 time=42.5 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=241 time=41.3 ms
64 bytes from 192.168.0.2: icmp_seq=3 ttl=241 time=41.4 ms
64 bytes from 192.168.0.2: icmp_seq=4 ttl=241 time=41.0 ms

--- 192.168.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 41.047/41.599/42.588/0.621 ms

# let's try something more exciting now
[local]$ ssh waldner@192.168.0.2
waldner@192.168.0.2's password:
Linux remote 2.6.22-14-xen #1 SMP Fri Feb 29 16:20:01 GMT 2008 x86_64

Welcome to remote!

[remote]$
```

When a tunnel like the above is set up, all that can be seen from the outside is just a connection (TCP in this case) between the two peer simpletuns. The "real" data (ie, that exchanged by the high level applications - ping or ssh in the above example) is never exposed directly on the wire (although it IS sent in cleartext, see below). If you enable IP forwarding on a host that is running simpletun, and create the necessary routes on the other host, you can reach remote networks through the tunnel.

Also note that if the virtual interfaces involved are of the tap kind, it is possible to transparently bridge two geographically distant ethernet LANs, so that the devices think that they are all on the same layer 2 network. To do this, it's necessary to bridge, on the gateways (ie, the hosts that run simpletun or another tunneling software that uses tap interfaces), the local LAN interface and the virtual tap interface together. This way, frames received from the LAN are also sent to the tap interface (because of the bridge), where the tunneling application reads them and send them to the remote peer; there, another bridge will ensure that frames so received are forwarded to the remote LAN. The same thing will happen in the opposite direction. Since we are passing ethernet frames between the two LANs, the two LANs are effectively bridged together. This means that you can have 10 machines in London (for instance) and 50 in Berlin, and you can create a 60-computer ethernet network using addresses from the 192.168.1.0/24 subnet (or any subnet address you want, as long as it can accommodate at least 60 host addresses). However, do NOT use simpletun if you want to set up something like that!

Extensions and improvements

simpletun is very simple and simplistic, and can be extended in a number of ways. First of all, new ways of connecting to the peer can be added. For example, UDP connectivity could be implemented, or, if you're brave, ICMP (perhaps also over IPv6). Second, data is currently passed in cleartext over the network connection. But when the data is in the program's buffer it could be changed somehow before being transmitted, for example it could be encrypted (and similarly decrypted at the other end).

However, for the purpose of this tutorial, the limited version of the program should already give you an idea of how tunnelling using tun/tap works. While **simpletun** is a simple demonstration, this is the way many popular programs that use tun/tap interfaces work, like **OpenVPN**, **vtun**, or **Openssh**'s VPN feature.

Finally, it's worth noting that if the tunnel connection is over TCP, we can have a situation where we're running the so-called "tcp over tcp"; for more information see "Why tcp over tcp is a bad idea". Note that applications like OpenVPN use UDP by default for this very reason, and using TCP is well-known for reducing performance (although in some cases it's the only option).

Filed under linux, networking, worksforme Tagged networking, tap, tun, tunnel
| Permalink

273 Comments

1. *Brian* says:
   November 14, 2014 at 16:14

   Hi. I think I might be having a similar problem to Pradeep. I created a persistent tunnel with tunctl, and I see my packets via tcpdump coming out on the remote tunnel, but they dont get forwarded anywhere and a netstat -i shows every packet i inject as a TX-DRP.

   I also have ip forwarding on, and i set promiscious mode on the interface as well (though I dont think I need that.)

   I'm also using a raw socket, which I think might be part of the complication.

   However, I read through that thread and see perhaps I should use the IFF_NO_PI flag when the tunnel is allocated. How do I do that if i set it persistently from outside the app?

   Is there an iproute2 command to set that flag? Honestly I dont fully understand what it does... i am reading packets just fine on the tunnel interface on local side that come into my application just fine without that flag set... so why would i need it only for writing? (Both are using raw_sockets)?

   Thanks for any advice you may be able to give.

   FYI: Kernel version is 2.6.32

2. *xiulo* says:
   October 24, 2014 at 14:58

   Hi, I have to do a program B that allows me to change the data from ethernet frames created for another program A before these frames are really send and it has to be totally transparent for the program A. I think I could use the TAP to redirect the frames from the program A to the program B and be able to modificate them before sending them normally.

   my idea is creating a tap interface and puting this tap_interface as a default ethernet interfice for the programA. And put the program B listening in the tap_interface and sending for the physical one after doing my modification on the data.

   A------>B(modifications)--->eth0

   Am I right and I can do this or I am not understanding what really TAP do?

   - *waldner* says:
     October 24, 2014 at 15:48

     There are some points that are not clear in your statement. First, program A most likely uses TCP or UDP (perhaps raw IP), so strictly speaking it's not producing "ethernet frames". This means it has an open socket of some kind, to which it sends data. The operating system, upon receiving this data, does what it deems appropriate with it; this might include creating ethernet frames and sending them out an interface; this interface, in turn, might be a tap interface, which is where program B would get them.
     It is important to note that this frames contain the data originally written from program A, but it's buried into the payload; outside it will have headers (ethernet, IP, TCP/UDP) added later by the operating system (not by program A).
     Once B has the frames, it can change them (assuming it knows where and how to do it - offset into the frame, recalculating checksums and so on), but then it has the problem of how to resend them from userspace so that they appear as they are on the wire. To do that you need the AF_PACKET raw socket (at least under Linux; I suppose equivalent functionality is available under other operating systems). See for example here (but google finds many results): http://austinmarton.wordpress.com/2011/09/14/sending-raw-ethernet-packets-from-a-specific-interface-in-c-on-linux/
     Note that the way B changes the frames can influence the way they come back (notably the source MAC address should be set appropriately depending on the need).

     - *xiulo* says:
       October 27, 2014 at 12:32

       thanks for answering. I think I'm not using transport or net layers only Ethernet. Because the program A is only a master that sends frames to slaves over a ethernet link. Right now I don't have any ip address assigned to the interface I'm using to transmit and the slaves are receiving the messages. Maybe I've just said a very stupid thinng but I haven't read the program A code yet. the program B has to be an error injector if this helps to understand what I want to do.

       I'll look into raw sockets, right now i don't totally understand how those work. I just tested a code I found online to send and receive ethernet raw frames but I've not used it on a virtualinterface yet.
       http://austinmarton.wordpress.com/2012/06/03/receiving-raw-packets-in-linux-without-pcap/
       http://austinmarton.wordpress.com/2011/09/14/sending-raw-ethernet-packets-from-a-specific-interface-in-c-on-linux/

3. *sancelot* says:
   September 18, 2014 at 14:35

   I am a little bit disturbed, I would need following setup to encapsulate frames with specifric protocol, do I need only one

tap interface or 2? or may be I am wrong ?

ETH0 (192.168.1.0/24) ==> ENCODING FRAMES =====> ETH1 (172.168.2.0/24
ETH0 (192.168.1.0/24) <== DECODING FRAMES <===== ETH1 (172.168.2.0/24

- *waldner* says:
  September 21, 2014 at 20:58

  Sorry, I have no idea what you're trying to do here.

4. *Tejas* says:
   August 15, 2014 at 16:46

   Very impressive tutorial. Gave me a better understanding of tun/tap.

   I had a query, hope you would have some insight for the same. I am trying to use tun to create tap interface or to be put precisely, want to make tun behave as tap. Why? I am building an application in android which requires me to send packets via tap. Android unfortunately has support only for tun. How can I make it work as tap. I know the difference between the packet structure of the two (IP vs Ethernet) though I am not sure whether I can make the tun work as tap, by which the server which is configured only to read from a tap device will be able to read successfully.

   Please let me know your insight to the same. Where can i get more information/support for the same topic.

   - *waldner* says:
     August 15, 2014 at 17:08

     It seems that only tun interfaces are available in android. The OpenVPN for android FAQ say:

     > "Actually, one could write a a tap emulator based on tun that would add layer2 information on send and strip layer2 information on receive. But this tap emulator would also have to implement ARP and possibly a DHCP client. I am not aware of anybody doing any work in this direction."

     A quick google search turns up just something like that, https://code.google.com/p/guizmovpn/source/browse/trunk/openvpn/tapemu.c, however I have no experience with android development and thus don't know how good or viable that approach ist, nor can help you more than that.

     I also have no idea whether rooting the device would make tap possible or not.

     - *Tejas* says:
       August 15, 2014 at 17:30

       Yes, rooting the device does provide support for tap, but that is not how I want to proceed since this will be a client app and rooting the device is not an option in that case.

       Thank you for your insights though.

   - *Alex* says:
     September 21, 2014 at 11:12

     Hi Tejas,

     Did you find a solution for your problem ?
     I also would like to solve this and tried to think about a generic solution for Android.
     It would help if you let me know your specific need for the tap, is that a VPN or anything else ?

     Thanks,
     Alex

5. *TJ* says:
   August 14, 2014 at 22:35

   So for the part where you force the kernel to actually send something out of the tun interface, you mentioned "NOTE: the following works only with kernels < 2.6.36; later kernels behave differently, as explained in the comments". I didn't really understand why. I have Linux version 3.8.0 and I am trying to understand how the tun interface works. What happens when we ping to a tun interface.

   Is there a workaround or updated code for higher linux kernel versions?

   Great tutorial BTW. Didn't find any other better page which explains tun/tap to novice developers like me.

   - *waldner* says:
     August 14, 2014 at 22:56

     It's simple: with older kernels, the interface was always considered to be up, so one could just throw anything out of it and tcpdump would get it. Starting from kernel 2.6.36, a change was added that sets the interface "up" only when there is a program connected to the interface, which is effectively the equivalent of a "carrier" or a "wire" for the tun interface. It's no different from what you see with a regular ethernet interface: if the cable is disconnected, you get a "no-carrier" status. With this new feature, it's no longer possible to ping an address belonging to the tun interface subnet and see the packets going out, unless a program has previously been attached to the interface.

- *Marc* says:
  September 30, 2014 at 23:53

  Hi Waldner, thanks for all of this information. I have a situation whereby kernel version 2.6.32 shows ARP broadcasts in tcpdump when I ping an unused ip address within the tap interface subnet. As you describe, following the same procedure on version 3.13.0 of the kernel shows no ARP traffic in tcpdump.

  I've tested with the code you provided in simpletun.c, and it of course successfully activates the tap0 interface (cat /sys/class/net/tap0/operstate => up) during tun_alloc(), yet when I use libpcap, and pcap_open_live() to attach to tap0, the interface does not come up. I thought calling pcap_activate() thereafter might make a difference, but the interface stays down. http://www.tcpdump.org/manpages/pcap_activate.3pcap.txt

  A non-code solution would work too, for example 'ip link set dev tap0 up' but that doesn't bring the interface up for me either. http://unix.stackexchange.com/questions/37659/

  Is there anything you can think of that might help?

  Thanks
  Marc

  - *waldner* says:
    September 30, 2014 at 23:59

    I'm not sure if you refer to the interface being "up" or having a carrier. To set it up, you just use "ip" of "ifconfig" (after the interface is made persistent). However if you don't connect a program to the interface's file descriptor, the interface flags show "NO CARRIER" and no traffic is sent (it's equivalent to a regular ethernet interface without a plugged cable).
    If you run a program that opens /dev/net/tun and actually attaches to the tun interface, you perform the equivalent of plugging the cable into a physical ethernet interface; that provides the carrier to the interface and traffic can flow. I hope I'm understanding your issue correctly.

    - *Marc* says:
      October 1, 2014 at 18:31

      Hi Waldner, thanks for your reply, and your time! I'm talking about the interface having a carrier, as the no-carrier state seems to be the reason why tcpdump (by virtue of libpcap) is unable to see any traffic on tap0. Until I run your program (./simpletun -i tap0 -a -s) which binds to tap0 taking it out of the no-carrier state, at which point tcpdump starts showing the ARP traffic generated due to pinging a non-existing host (10.0.0.2).

      Do you know of a way to force tap0 to act as it did in earlier versions of the kernel? I'd like it to be up (ie, to get the "carrier") without needing to see some process attached to the special fd so traffic can be sniffed regardless of the carrier state? Perhaps there is a way to force this on the tun driver, or perhaps there is a method libpcap which can accomplish this, maybe pcap_activate()?

      When your program (simpletun) is not running I see no traffic in tcpdump, and the tap0 operstate shows as down:
      # cat /sys/class/net/tap0/operstate
      down

      NB. 'ip' doesn't change the no-carrier state, or make a difference to the packet capture in tcpdump.
      # ip link set dev tap0 up
      # cat /sys/class/net/tap0/operstate
      down
      No error is given, the operstate stays down, and as expected tcpdump doesn't see the packets.

      When I start your program however, tcpdump starts showing traffic and operstate switches to up:
      # cat /sys/class/net/tap0/operstate
      up

      Can you think of anything which may help? Many thanks,
      Marc

      - *Marc* says:
        October 2, 2014 at 01:08

        it seems that the TAP (or TUN) device must be sent the TUNSETIFF control code (defined in linux/if_tun.h) using ioctl() in order for the interface to come up (ie, to get the "carrier").

        i've searched the code of both libpcap and tcpdump for references to either TUNSETIFF or if_tun.h. neither exist, which leads me to conclude that neither of these libraries are capable of activating/attaching to the TAP device in order to facilitate packet capture, and require a another program perform the attachment. reasonable of course since the interface operstate is akin to connecting a network cable, as i understand it.

        so I think this only leaves the possibility that the tun.c driver may be coaxed into behaving like it did prior to version 2.6.36 of the linux kernel, before this feature was introduced.

is this likely?

thanks waldner

- *waldner* says:
  October 2, 2014 at 19:09

  I think the change that was introduced in 2.6.36 was to make the behavior of the tun interface consistent with that of a real ethernet interface. As I said, attaching a program to the tun interface (or, more precisely, as you found, running the TUNSETIFF ioctl) is the equivalent of "plugging in" the cable, as it's the task of the connecting program to send/receive the packets to/from the interface. So, just as one doesn't run tcpdump on an ethernet interface with the cable unplugged, one shouldn't attempt to do its equivalent on a tun interface.
  That said, I don't know of any workaround, short of creating a simple stub program (can even be much simpler than simpletun) whose only purpose in life is to keep the tun interface "connected" for the duration of the capture or whatever you are doing with it.

  - *Marc* says:
    October 2, 2014 at 23:19

    hi waldner, thanks again. I'm working cross-platform in c# with mono, so I was keen to avoid a separate stub program in c, python or anything external to the main binary really. thanks to your posts and comments here I was able to understand the problem. now i've got pure c# (once i worked out the appropriate pinvoke system call imports) to attach to the tun interface and let libpcap do its job. works a charm too, so many thanks again for your help, i'd have been really stuck without you!

    cheers,
    marc

6. *Pradeep* says:
   April 21, 2014 at 07:41

   Hi waldner,
   Thanks for a great tutorial. I was trying to create two tap interfaces, attach it to a linux bridge, send frame via one tap interface and hoping to receive from the other tap interface. Your tutorial helped me code it up. Since, i send raw packets, i couldn't use read/write. I did a tcpdump on the sending tap interface and verified the sending side is ok. But, the receive side doesn't seem to receive anything. Here's my code snippet, appreciate if you or anyone can spot anything wrong. I have removed the error checks and constructed the frame manually. Both the tap interfaces are in UP/RUNNING state.

   Sending side:
   ---

   unsigned char buffer[100] = unsigned char buffer[60] = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x9a, 0x89, 0x0f, 0x89, 0x3a, 0x8c, 0x08, 0x06, 0x00, 0x01, 0x08, 0x00, 0x06, 0x04, 0x00, 0x01, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0xc0, 0xa8, 0x01, 0x01, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xc0, 0xa8, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

   int idx;
   int sockfd;

   if((fd = open("/dev/net/tun", 2)) < 0) {
   .. }

   memset(&ifr, 0, sizeof(ifr));
   ifr.ifr_flags = IFF_TAP ;
   strncpy(ifr.ifr_name, "SEND_TAP", IFNAMSIZ);
   if((err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0){
   ... }
   if((err = ioctl(fd, TUNSETPERSIST, (void *) &ifr)) < 0){
   .. }

   sockfd = socket(PF_PACKET, SOCK_RAW , htons(ETH_P_ALL));
   memset(&ll, 0, sizeof(ll));
   ll.sll_family = PF_PACKET;
   ll.sll_ifindex = 22; /* ifindex of SEND_TAP using ip link */
   ll.sll_protocol = htons(ETH_P_ALL);
   if (bind(sockfd, (struct sockaddr *) &ll, sizeof(ll)) < 0) {
   ....
   nread = sendto(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr *) &ll, sizeof(ll));

   Receiving Side:
   ---

   unsigned char buffer[100];

```
int idx;
int sockfd;

if( (fd = open("/dev/net/tun", 2)) < 0 ) {
.. }

memset(&ifr, 0, sizeof(ifr));
ifr.ifr_flags = IFF_TAP ;
strncpy(ifr.ifr_name, "RCV_TAP", IFNAMSIZ);
if( (err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0 ){
... }
if( (err = ioctl(fd, TUNSETPERSIST, (void *) &ifr)) < 0 ){
.. }

sockfd = socket(PF_PACKET, SOCK_RAW , htons(ETH_P_ALL));
memset(&ll, 0, sizeof(ll));
ll.sll_family = PF_PACKET;
ll.sll_ifindex = 26; /* ifindex of RCV_TAP using ip link */
ll.sll_protocol = htons(ETH_P_ALL);
if (bind(sockfd, (struct sockaddr *) &ll, sizeof(ll)) < 0) {
....
nread = recvfrom(sockfd, buffer, sizeof(buffer), 0,
(struct sockaddr *) &ll, &addrlen);
..
```

- *waldner* says:
  April 21, 2014 at 13:22

  I'm not sure I understand why you need raw sockets here. With tap, you can already send/receive full ethernet frames.
  Also note that if you use sockets you're putting yourself at the other side of the processing that a tun/tap interface does, that is, you're not using anything of what tun/tap is for.

  - *Pradeep* says:
    April 21, 2014 at 17:09

    Thanks Waldner. Do you mean using write/read calls? I tried that as well.

    On sending side:

    unsigned char buffer[60] = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xb2, 0xcf, 0x1c, 0xd4, 0xed, 0x1a, 0x08, 0x06, 0x00, 0x01, 0x08, 0x00, 0x06, 0x04, 0x00, 0x01, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0xc0, 0xa8, 0x01, 0x01, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xc0, 0xa8, 0x01, 0x02, 0x00};
    nread = write(fd, buffer,sizeof(buffer)); /* fd is the same as the code i gave above for creating tap */

    On receiving, i simply do:
    nread = read(fd, buffer,sizeof(buffer)); /* Again fd is the same */

    The same problem persists, nothing seen on the receive side. Moreover, if i change the DST MAC to that of the listening tap, like in:

    unsigned char buffer[60] = {0xe2, 0x5a, 0x11, 0x19, 0xca, 0x28, 0xb2, 0xcf, 0x1c, 0xd4, 0xed, 0x1a, 0x08, 0x06, 0x00, 0x01, 0x08, 0x00, 0x06, 0x04, 0x00, 0x01, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0xc0, 0xa8, 0x01, 0x01, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xc0, 0xa8, 0x01, 0x02, 0x00};

    I also see that the initial 4 bytes of the frame is chopped like below (using tcpdump). Well, the Rx side doesn't get anything as well.

    listening on TX_TAP, link-type EN10MB (Ethernet), capture size 65535 bytes
    08:54:20.987776 IP0 bad-len 1
    0x0000: ca28 b2cf 1cd4 ed1a 0806 0001 0800 0604
    0x0010: 0001 0202 0202 0202 c0a8 0101 ffff ffff
    0x0020: ffff c0a8 0102 0000 0000 0000 0000 0000
    0x0030: 0000 0000 0000 0000

    That's what made me think i should use raw sockets since this has problems at the sending side itself.

    - *waldner* says:
      April 21, 2014 at 17:22

      What I mean is that you are on the wrong side of things.
      Socket programming is not a replacement for the processing that tun/tap allows you to do.
      With tun/tap your code plays the part of the wire/intermediary, whereas with sockets you are the original producer/final consumer of the packets.

- *Pradeep* says:
  April 21, 2014 at 19:32

  Thanks Waldner. My intention is to send a regular ethernet frame (Supposed to be multicast, but i wanted a basic unicast to work) over a tap interface and receive it on another tap interface. Both of the tap interfaces are connected to a Linux bridge. How do i achieve it or what am i doing wrong above?
  I actually started with regular read/write like your example. But, in write() i was constructing my own frame. That didn't work for above reasons.

  Then i switched to Raw sockets and i gave the ifindex as that of the tap interface(ll.sll_ifindex = 22;)
  As i said, the reason for RAW socket is i may want to send raw ethernet frame (say lldp).

  I am sure i am not understanding something right. If i just write to the tap and another thread read from the same tap, it all works. But, with a bridge in between, i don't even see the frame going to the bridge. Qemu provides the same connectivity with VM's.

  - *waldner* says:
    April 21, 2014 at 21:07

    So you only need the code to create tun interfaces. My examples that use read()/write() have nothing to do with what you're trying to do. You would need that code, for example, if you wanted to implement a bridge yourself. But in this case, the kernel already does that.
    I'd say there's nothing tun/tap specific in your problem (apart from the device creation code, which can however also be done externally), that is, you would likely experience the same problems if using two regular physical ethernet interfaces connected to a bridge (which, btw, is a good way to check whether your code really works).
    Without knowing the details of the kernel bridge implementation, I would check, in random order:

    iptables rules - even if it's a bridge, there's a setting that causes iptables rules to be evaluated for packets traversing a bridge, and it's enabled by default in many distributions.
    target MAC address - to decide out which bridge port to send out a frame, the kernel checks the target MAC address and sees on which port it has learned it. Note that this is NOT the MAC address of the interface/port participating in the bridge. If you want the kernel to send a frame with a MAC of 00:11:22:aa:bb:cc out of a given bridge port, you need to tell the bridge (via ARP or other means) that 00:11:22:aa:bb:cc is located behind that port. The MAC addresses of the interfaces that are themselves bridge ports are not used for anything (AFAIK), except to derive the MAC address of the bridge. On the other hand, it's true that if the destination MAC is unknown, the bridge should send the frame out all its ports.
    MTU - the payload of the ethernet frame you send should not exceed the bridge MTU otherwise it's silently dropped, this is probably obvious but it's worth mentioning.

    - *Pradeep* says:
      April 22, 2014 at 01:35

      Good suggestion, Waldner. I agree. When i created a tap interface using "tunctl -t xxx -u yyy", it wasn't put in RUNNING state, after i did a ifconfig up and my process opening that tap interface. This is when i landed in your site and your code got me past this issue. I already tried flushing the iptable rules. Thanks for your help. When i find the issue, i can post it.

      - *Pradeep* says:
        April 23, 2014 at 08:23

        I think this has to do with the way i am using the tap interface.

        I used the same program with two machines connected to each other and my program for sending/receiving (raw sockets) was working fine. Then, this is what i did:
        Create a bridge on one m/c with multiple interfaces including a tap interface

        eth3, eth4, tap of BRIDGE1 eth3 (another m/c)

        If I do a ping or write something using my program on the other m/c it appears in the tap interface of Bridge 1 when I read it using my program.
        So, bridge is probably acting ok. If frame is received on any physical interface connected to the bridge, it does the proper forwarding to other ports.
        If my program writes to the tap, quoting from above
        "To the kernel, it would look like the tun/tap interface is receiving data "from the wire".

        Then, not sure why the bridge is not seeing the frame. It works fine for physical interfaces. I added the tap to the bridge in a regular way "sudo brctl addif XXX tap". It's not just Linux bridge, i also used OVS, still the same.

- *waldner* says:
  April 23, 2014 at 16:19

  If by "my program writes to the tap" you mean "I use raw sockets to write a frame to the tap interface", then what's written in my article does NOT apply. As I said, the article covers a packet/frame processing mode that is NOT at all what you do using raw sockets.

- *Pradeep* says:
  April 23, 2014 at 21:27

  Yes, I understand this may be out of scope with regards to what the article covers, my apologies for that. I was just trying to find a way to write to a tap interface when a user program constructs a frame. Now, for a change, I have a solution!! But, don't understand the complete details.

  Let's forget everything about RAW sockets that I said before.

  I created the tap as given in your example but w/o IFF_NO_PI. I just used the read/write as given in your example. Same topology with eth3 of M/C1 connected to eth3 of M/C2. But, M/C2 has a linux bridge whose port is eth3 as well as two tap interfaces one for rx and another for tx.

  Now, I send a frame from M/C1, the code on M/C2 using read on the tap interface (rx tap), receives the frame.
  Sending frame looks like:
  1111 1111 1111 a80c 0dc0 fa41 8100 0051
  The code when it dumps the buffer after the read() on tap interface produces:
  00008100 111111111111a80c0d
  Even though tcpdump on tap shows the frame as is, when read call returns it puts an extra 4 B at the beginning. I am not sure, but my guess is 2B of 0 and 2B of ethertype. And, when I do a "write" of the buffer back to the same tap interface, it all goes out fine and it works beautifully.

  But, when I do a write on the tap interface (tx tap) in M/C2, like:
  unsigned char buffer[] = {0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0xa8, 0x0c, 0x0d, ….};
  The code when it dumps the buffer after the read() on another tap (rx one) interface produces:
  00000800 1111a80c0dc0f
  It has truncated the 4B of DMAC and has put some 4B of data at the beginning. If i construct the frame with 4 leading Bytes like:
  unsigned char buffer[] = {0x00, 0x00, 0x81, 0x00, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0xa8, 0x0c, 0x0d, ….};
  It goes out fine and the packet looks correct.

  Is there anything I am missing? Any insights in the leading 4B?

  - *waldner* says:
    April 25, 2014 at 13:45

    Since you're not using IFF_NO_PI, the kernel adds a 4-byte preamble to the frame. Use IFF_NO_PI and you should see the frame unchanged.

7. *Mike* says:
   April 18, 2014 at 18:03

   Hi Waldner,
   I tried your simpletun example and I couldn't make it work. I must miss something that I'd appreciate your help:

   - what's your routing table look like when you run simpletun (netstat -rn)?

   - I started the server side simpletun on the remote host just fine, but when I started simpletun on the local side, it said "host unreachable".

   More details:

   - remote host is connected to local host via Enet
   - remote host has ip 'a', local host has ip 'b'
   - TUN 'virtual1' was created on remote host, with ip '10.0.0.10'
   - TUN 'virtual1' was created on local host, with ip '10.0.0.20'

   - started simpletun on remote host. ok
   - started simpletun on local host with: simpletun -i virtual1 -c b (where b is the real ip address of the remote host)

   and it successfully connected to TUN 'virtual1', but failed the 'connect' call when it tried to connect to the server via ip 'b'.

   what did I do wrong?

Thank you!
mike-

- *waldner* says:
  April 18, 2014 at 20:40

  This sounds like a simple network connectivity issue, are you sure you can connect from IP "a" to IP "b" on the port used by the simpletun "server" (bt default TCP port 55555)?

  - *Mike* says:
    April 19, 2014 at 00:38

    Thanks for the reply.I couldn't ping 10.0.0.20 from the local (10.0.0.10) either... I could ping 'b' from 'a' just fine.

    - *waldner* says:
      April 19, 2014 at 06:53

      The fact that you can ping "b" from "a" does not imply that TCP (and thus simpletun) works at all. You must make sure you have TCP access from "a" to "b" on the port used by simpletun (55555 by default, can be changed with -p).

      - *Mike* says:
        April 19, 2014 at 07:00

        Thank you again! What's an easy way to check TCP between 2 hosts on port 55555?

        - *waldner* says:
          April 19, 2014 at 07:24

          You need to have a program running on b listening on port 55555 (like simpletun in "server" mode) and try to connect from a. So on "b" run

          simpletun -i tun3 -s

          (replace "tun3" with the correct name for your tun/tap interface)

          and on "a" run

          simpletun -i tun11 -c b

          replace "b" with b's IP address (NOT the address of b's tun/tap interface), and tun11 with the name of the tun/tap interface on "a".
          If the last command fails with "connection refused" or some other error, you have a basic connectivity problem between "a" and "b".
          If it says nothing, chances are it has correctly connected and you can start exchanging traffic between a's tun/tap IP and b's tun/tap IP.
          You can also add the -v option to both client and server command line, that should give you some more info.

8. *Guido* says:
   March 13, 2014 at 15:45

   Hi Waldner, thank you for this great tutorial. The problem I found is that ip tuntap is not available on centos. Is there somewhere I can download it from?
   I've tried openvpn for this scenario but it didn't work.
   APP1 -> OPENVPN -> SERVER1 -> NETWORK -> SERVER2 -> OPENVPN -> INTERNET
   What I want to do basically is have server2 mask the source of app1. Problem is that app1 uses a lot of connections, like 10k and openvpn can't handle that much and is very slow. We tried UDP, TCP and various MTUs but we can't do it. I was wondering if ip tuntap was a good option.

   - *waldner* says:
     March 13, 2014 at 17:37

     I don't know, perhaps there is some repository that has a recent copy of iproute2. In any case, you can create tun interfaces with openvpn --mktun or using umlutilities, buth of which should definitely be available for centos.
     As for the rest of your question, sorry but the lack of useful information is such that I'm not able to make any sense out of it.

     - *Guido* says:
       March 13, 2014 at 18:46

       ok let me see if I can complete the information.
       Right now I'm using GRE or IPIP tunnels.
       I have an application running on server 1, let's say a tool that run DNS queries.
       This application binds to the IP 10.0.0.1. This traffic is PBRd to 10.0.0.2 on server 2.
       On server 2 I forward all the traffic from 10.0.0.0/24 to the internet.
       So if on server X (outside my network) a connection on the port 53 is received from server1, on their logs it will show up with the IP of server 2.

Now what I want to do, is use TUN/TAP instead of IPIP or GRE beacuse I'm working iwth openvz containers and they don't allow GRE or IPIP. With openvpn the performance was very very bad and I want to develop or use something that can suit the number of connections I require.
Does it make sense?

- *waldner* says:
  March 13, 2014 at 22:21

  It's still not clear to me how tunnels fit into the scenario you describe (and why you are not using plain NAT to do source the address rewriting), but in any case:

  - GRE and IPIP should definitely work with openvz
  - Whatever you did with those tunnels, can be done with two tun/tap interfaces, one at each end of the tunnel
  - While with GRE and IPIP it's the kernel that does all the work, with tun/tap things happen in userspace and it's your code that has to implement the tunnel
  - It is possible to run multiple instances of OpenVPN on the same machine
  - If you end up implementing the same I/O model as OpenVPN, you'll likely experience the same problems you had with it (perhaps a bit mitigated by the fact that you don't do encryption), so you have to implement something better.

  Just some hints.

9. *Eric* says:
   March 12, 2014 at 12:29

   Hi Walder,

   Congrats for the great post! It is of great help for new comers like me. Would you mind to explain how the kernel (in particular in regard) to iptables handle packets that arrive on real interfaces addressed to the a tun interface?

   For example, I have eth0 and eth1 both wired to a second computer and declared as auto in /etc/network/interfaces however only eth0 has an ip address defined (10.0.0.1) and a a tun interface with address 192.168.0.1.

   1) If the external computer pings 192.168.0.1 from eth0, assuming iptables has a rule to drop all incoming from eth0, will this packet reach the tun interface?
   2) If the external computer pings 192.168.0.1 from eth1, given the fact that the interface is not up, will this packet reach the tun interface? What happens if there is a rule to drop all from eth1, will it reach the tun (after all the real interface is down)?

   Thanks in advance,

   Eric

   - *waldner* says:
     March 12, 2014 at 12:45

     The tun interface is just another interface. In your testcase, if you had eth5 instead of the tun interface, nothing would change.
     If a packet enters eth0 and iptables drops all packets entering eth0, they will be dropped and thus will not reach eth5, or tun0, or whatever other interface.
     If a packet enters eth1 and it is down, it is dropped even before iptables has a word on it.
     In short, iptables-wise there's no difference between a tun interface and another physical interface.

     - *Eric* says:
       March 12, 2014 at 13:52

       Walden, thanks for the answer, I see now that my question is probably related to iptable instead of tun interfaces. Still there are some points I'm not fully understanding (please forgive my ignorance ;)).

       For example, given the rule "-A INPUT -i eth0 -j DROP", if (from an external host) I ping 192,168.0.1 on eth0 wire, the packet will came trough targeting the eth0 MAC address, right? the destination IP address however does not match the one assigned to eth0. Since ip tables is only aware of the network layer, would iptables discard it or let it pass? What "-i eth0" means in practice to iptables?

       Thanks for the help and patience,

       Eric.

       - *waldner* says:
         March 12, 2014 at 18:13

         Your iptables rule drops anything that comes in from eth0, regardless of source/target MAC or IP address. If the target MAC is not that of eth0, it will probably be dropped even before iptables has a chance to see it.
         I suggest you read up some basic networking concepts. For iptables, a good reference (though a bit outdated) is https://www.frozentux.net/iptables-tutorial/iptables-tutorial.html.

         - *Eric* says:

March 12, 2014 at 19:38

Walder,

Thanks for the link, in fact due to a coincidence I read this exact same tutorial just a few days ago. I do have a regular knowledge about networking in overall (not specific to iptables or tun/tap, but a good one when comes to network stacks like OSI or TCP/IP).

You see, as strange as it may sound, what you described is what I expected to see. However I do have server in production whose behavior is not that. It has a tun interface with similar iptable's rules and it's traffic is not being dropped. The server runs a fork of ChilliSpot, which is a program to create captive portal under a walled garden environment. It starts a tun interface (not tap, as far as I can tell) and than provide dhcp service to wireless clients. Those client's traffic is than filter by this program.

This program also setup a few iptables rules and one of those rules is the mentioned "drop all", which at first I tough it would drop any packet. But the server is running ok, clients connect to the access point and have they're traffic sent to internet as expected. So I'm kind of lost here trying to understand how this is working (it's a fact however that it is working). According to the ruleset (http://pastebin.com/Tqty9B8A) traffic on eth2 should never reach the tun0 nor it's associated executable, but it does.

At first I assumed this strange behavior could be explained by iptables handling tun interfaces in some strange way, but from your explanations that is not case. I guess I'll need to dig a little further.

Anyhow thanks for your time.

Cheers,

Eric

- *waldner* says:
  March 12, 2014 at 23:28

  Remember that only packets destined to the local machine hit the INPUT chain, otherwise they enter the FORWARD chain. Your original question did not contain any reference to the actual problem so I simply thought of a standalone machine receiving packets destined for the local box, while now it is possible to appreciate the real nature of your question. My guess is that these packets that you think should be dropped but aren't are simply not entering the INPUT chain.

  Here's a link to another article on how to debug iptables (ie, see which chains and rules the packets traverse). It also contains a link to a diagram which shows the big picture of iptables packet processing: http://backreference.org/2010/06/11/iptables-debugging/

10. *Dino* says:
    February 5, 2014 at 15:25

    No problem here, just wanted to say thank you for this brilliant tutorial. Very well written, easy to follow, great examples. Cheers mate!

11. *Cyrill* says:
    January 9, 2014 at 13:50

    Hy

    Thanks a lot for the very nice tutorial of a useful but poorly documented Linux module.

    Currently I have to make usage of the Tun/Tap module but run into a major issue.

    As I already wrote down the problem on Stackoverflow I kindly ask you to take a look at my questions there:
    http://stackoverflow.com/questions/21001713/usage-of-tun-tap-and-multicast-messages

    Can you help me a step further in this matter?

    I will be really glad for any hints and comments.

    Best, Cyrill

    - *waldner* says:
      January 11, 2014 at 11:30

      From what I can understand, it looks like your issues are at the application level, not at the newtork level. Does at least app2 receive the messages from app1?

      - *Cyrill* says:
        January 17, 2014 at 13:48

        Hi Waldner

Thanks very much for getting back to me in this matter.

I updated the question on stackoverflow and included snippets of the code that is involved in setting up the TUN/TAP devices.

With my current usage, neither APP1 nor APP2 are able to read back the data sent to them over tap1 and tap2, respectively.

May I ask you to take another look on stackoverflow and check whether I setup anything wrong?

Note that I also included an strace()-extract of the fundamental system-calls issued by APP1 and APP2. The problem is, that select() never sees any information ready to be read by the TAP devices.

I'm really thankful for all the time you invest!

Best, Cyrill

- *waldner* says:
  January 18, 2014 at 14:00

  If packets are seen at the tap interfaces, it means that the mediator is working (I'm assuming that, when the mediator is running, packets sent by APP1 are seen at tap2 and packets sent by APP2 are seen at tap1).
  However without the code of the application it's not possible to reproduce the setup and try to investigate the problem.

- *waldner* says:
  January 18, 2014 at 19:24

  It also occurs to me that you may want to have a look at the logs for messages related to possible rp/martian problems (although it shouldn't be an issue here).

12. *DmitryP* says:
    November 27, 2013 at 06:29

    Hi,

    We have the following configuration: Blade Server running multiple VMs on these VMs may run multiple QEMUs.

    My qestion is how should be configured and connected QEMU interfaces to be viewed from an "External Network".

    eth0(Physical Port) eth0 VM eth0 QEMU .

    tia,
    Dmitry

    - *waldner* says:
      November 30, 2013 at 11:59

      There are a few ways, the easiest is probably to just bridge the physical eth0 and the QEMU guest interface together (into eg br0) and give br0 a public IP or make it reachable from the outside somehow.

13. *Anwar Ali* says:
    November 25, 2013 at 23:20

    Hi Waldner,

    On my laptop, I'm using Ubuntu 12.04 (I've created one VM).

    Thanks for the excellent tutorial. I've created two taps and from my application I'm trying to capture the ethernet frame received on tap2. When I run the program, I assigned IP addresses as follows:

    ip link set dev tap1 up
    ip link set dev tap2 up
    ip addr add 10.0.0.1 dev tap1
    ip addr add 20.0.0.1 dev tap2

    Then, when I tried to ping 10.0.0.2 my application did receive the ARP request on tap1 but my application was not able to print the packet on tap2 (the one that was sent from tap1). The interesting thing is that when I typed "ifconfig -a" tap2 RX packets was incremented when the ping is isssued. Here is my code:

```
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
```

```
#include
#include

static void error(const char* msg)
{
   perror(msg);
   exit(1);
}

int tun_alloc(char *dev, int flags)
{
   struct ifreq ifr;
   int fd, err;
   char* clonedev = "/dev/net/tun";

   fd = open(clonedev, O_RDWR);
   if( fd == -1 )
   {
      error("open");
   }
   memset(&ifr, 0, sizeof(ifr));
   ifr.ifr_flags = flags;
   if(*dev)
   {
      memcpy(ifr.ifr_name, dev, IFNAMSIZ);
   }
   err = ioctl(fd, TUNSETIFF, (void*)&ifr);
   if( err == -1 )
   {
      error("ioctl");
   }
   memcpy(dev, ifr.ifr_name, IFNAMSIZ);
   dev[IFNAMSIZ] = '';


   return fd;
}

int main(int argc, char* argv[])
{
  char dev[IFNAMSIZ+1] = "tap1";
  char dev2[IFNAMSIZ+1] = "tap2";



  struct pollfd fds[2];
  int ret;
  unsigned char buffer[2000];
  int i = 0;

  fds[0].fd = tun_alloc(dev, IFF_TAP);
  if( fds[0].fd == -1 )
  {
     error("tun_alloc");
  }
  fds[1].fd = tun_alloc(dev2, IFF_TAP);
  if( fds[1].fd == -1 )
  {
     error("tun_alloc");
  }

  int nread = 0;
  int nwrite = 0;
  fds[0].events = POLLIN;
  fds[1].events = POLLIN;

  while(1)
  {
    ret = poll(fds, 2, -1);
    fprintf(stderr, "\n%s %04x  events\n",dev, fds[0].revents);
    fprintf(stderr, "\n%s %04x  events\n",dev2, fds[1].revents);
    if( ret == -1 )
    {
       error("poll");
    }
    if( fds[0].revents & POLLIN )
    {
       nread = read(fds[0].fd, buffer, sizeof(buffer));
       if( nread == -1 )
       {
          close(fds[0].fd);
          error("read error");
       }
       fprintf(stderr, "\n%d bytes received from %s\n", nread, dev);
       for(i = 0; i < nread; i++)
       {
          fprintf(stderr, "%02x ", buffer[i]);
       }
       nwrite = write(fds[1].fd, buffer, nread);
       if( nwrite == -1 )
       {
          close(fds[0].fd);
          error("write error");
       }
       fprintf(stderr, "\n%d bytes written to %s\n", nwrite, dev2);
       for(i = 0; i < nwrite; i++)
       {
          fprintf(stderr, "%02x ", buffer[i]);
       }
    }
    if( fds[1].revents & POLLIN )
    {
       nread = read(fds[1].fd, buffer, sizeof(buffer));
```

```
        if( nread == -1 )
        {
           close(fds[1].fd);
           error("read error");
        }
        fprintf(stderr, "\n%d bytes received from %s\n", nread, dev2);
        for(i = 0; i < nread; i++)
        {
           fprintf(stderr, "%02x ", buffer[i]);
        }


     }

   }

   return (EXIT_SUCCESS);
}
```

- *waldner* says:
  November 30, 2013 at 12:14

  When your application writes the packet to tap2, the kernel sees an incoming packet, and performs the usual checks (checksum, MAC address, routing, rp filter etc.). If the result of one of these checks is that the packet should be dropped, the kernel drops it. Since you're working with tap, I'd suggest the first thing to look at is MAC addressing. The frame you receive on tap1 has the destinationa MAC of tap1 (obviously, or your application wouldn't receive it). Now you write it unchanged to tap2 (which has a different MAC from tap1). The kernel sees an incoming frame on tap2 with a destination MAC address that does not match that of tap2, so the kernel drops the frame (note that the same would happen with eth0, or any other ethernet interface).
  Probably, if you set tap2 in promiscuous mode (via iproute2 or from your code) the frame would be accepted and your application would see it. But as usual, it all depends on what you're trying to achieve.

14. *doug* says:
    October 4, 2013 at 19:42

    anyone has this code translated to visual c++ for windows?

    - *waldner* says:
      October 13, 2013 at 00:06

      I doubt this code would work on windows anyway...

15. *Gourmet* says:
    September 8, 2013 at 16:27

    Pretty useful!
    You'll have to add that IFF_NO_PI must be used, if any, ONLY for tun interfaces.
    For tap interfaces, it will add four extra leading bytes that are unuseful but to add garbage to an Ethernet frame (and shift the whole frame with 4 bytes).

    db

16. *Cherif* says:
    September 2, 2013 at 14:48

    Hi Waldner,

    Thanks for these information.

    I am trying to make a http client/server architecture. In order to simulate some network artefacts between the client and the server, i will use "Network Simulator NS3". So packets, sent or received, from client to server, have to pass through NS3 (the latter is installed on a different machine).

    So I will have this architecture (3 machines) : (Browser@clientMachine) (tap-left NS3 tap-right) (Server@Server machine)

    The architecture is similar to the one described in https://github.com/nabam/ns3/blob/master/examples/tap/tap-csma-virtual-machine.cc (line 32).
    The difference is that the latter architecture describes a connection between lxc containers and the host machine. And what i am trying to do is almost the same thing but using real machines instead of lxc containers.
    The tap and bridge configuration is described in the file : https://github.com/nabam/ns3/blob/master/examples/tap/virtual-network-setup.sh (without the lxc configuration)

    What command line should I use in order to force packet, going from client to server for example, to pass through tap-left, and that the output of tap-right, can arrive to the serverMachine. (Which means that the serverMachine should be see-able by the tap-right).

    I don't know if it's clear as a question...

    Thanks.

    - *waldner* says:
      September 2, 2013 at 20:06

I don't know NS3, are tap-left and tap-right on the same machine or on different machines?

- *Cherif* says:
  September 2, 2013 at 20:37

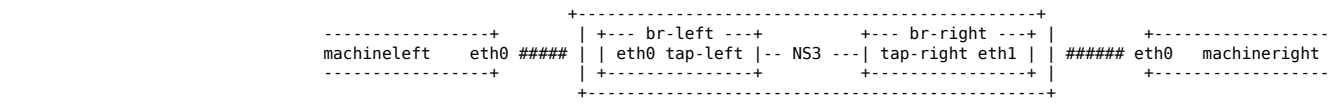
  Both are on the same machine.
  The architecture is better described on this page ("Big picture" section) : http://www.nsnam.org/wiki/index.php/HOWTO_Use_Linux_Containers_to_set_up_virtual_networks

  But, instead of containers as shown in the picture, i would like to do the same thing with real machine: MachineRight, MachineLeft, and HostOS (NS3 with tap-left and tap-right).

  I didn't found a way to link eth0 of machineLeft to tap-left, and same thing for the right side.

  - *waldner* says:
    September 2, 2013 at 20:59

    Ah, maybe I see. So on the NS3 machine, you need two bridges, each bridge will include a tap interface (which in turn goes into NS-3) as well as a physical interface, so roughly you'd have this (interface names are just an example):



    So the NS3 box should have (at least) two physical ethernet interfaces. The connections marked with #### can be direct port-to-port ethernet cables (machineleft's eth0 to NS3's eth0 and machineright's eth0 to NS3's eth1), or there can be a switch in between.

    To create br-left and br-right you do somthing like:

```
# brctl addbr br-left
# brctl addif br-left eth0
# brctl addif br-left tap-left
# brctl addbr br-right
# brctl addif br-right eth1
# brctl addif br-right tap-right
```

    Hope this is what you were looking for.

    - *Cherif* says:
      September 2, 2013 at 21:21

      Yes that is what I need as architecture. Actually, I am working with virtual machines (VMware player), so I can't make a direct connection using cables.
      How can I connect eth0 of MachineLeft to NS3's Eth0 virtually ?

      - *waldner* says:
        September 2, 2013 at 21:27

        Well you have to use the virtualization platform's facilities; normally, with enterprise VMware products you'd have virtual switches, but I know nothing about VMware player. As a shot-in-the-dark suggestion, see if you can use a bridge on the physical host as a "virtual switch" to which to connect the relevant interfaces of the guests. This is just a guess as I don't know anything about VMWare player,

17. *Harry* says:
    September 1, 2013 at 14:25

    Hello waldner,

    Thanks for such a wonderful tutorial !!!

    I am like a novie in networking and wondered here on following stuffs :

    1) Why IP address for tap interface is choosen as 10.0.0.1 ? And How to decide an IP for Private interface and subnet mask ?
    2) Why can't we set IP address for it in the same subnet as in for the real physical interface conected with an internet ? I found that setting it in the same subnet causes to stop internet access.
    3) (Here) When ping is used with 10.0.0.2 (or any address within 10.0.0.2 - 10.0.0.254) why it gets travelled to tap ineterface ?

    Kindly provide me detailed insight over this questions. Thanks in advance.

    - *waldner* says:
      September 2, 2013 at 20:30

      Hello,

      1) Those are completely arbitrary and entirely depend on what you want to do. In this case, 10.0.0.0/24 is a common IP range, and 10.0.0.1 is just an easy address from that range.

2) If you assign IP addresses that are in the same range to different interfaces you can be in trouble unless you properly configure routing, This is an advanced topic and I would not recommend it unless you have very specific needs that require that, and know what you're doing.
3) Because the kernel sees that the address is not local, and the route to reach it points to the tap interface. This is exactly the same that happens with any interface, be it virtual or physical..

18. *Patrick* says:
August 13, 2013 at 12:56

Hi Waldner,

thank you for the tutorial. I've some questions and hope you can help me. I can create TAP devices normaly I use two,e.g. tap0 and tap1. On Tap0 I want send Ethernet II Pakets to Tap1. I've bothe Programms wirtten and bind to the devices. I want build a connection between the Tap devices so I can send direct my Packets. It's should work isolated that I can buil two more Tap devices an connect to each outher and this is isolated to the first one.

Is it possible with Tap devices? when yes do you have a tipp for me?

I use a Ubuntu 12.04.01 LTS.

- *waldner* says:
August 13, 2013 at 19:45

It should definitely be possible, yes. Remember that for the kernel to send out a packet, it has to believe that the IP is located "behind" the respective tap interface, so do not use the IP addresses you assigned to the tap interfaces for your applications.

19. *vivek* says:
July 18, 2013 at 06:32

hi Walder,
does TAP interface works with DHCP. DHCP client is running in the application program waitng for data at port 67. but, TAP does not pass DHCP offer to DHCP client.

- *waldner* says:
July 18, 2013 at 09:01

It should definitely work, as long (as always) as the client connected to the TAP interface is able to reach the DHCP server and viceversa. This usually means that you have to bridge the TAP interface. And of course the application connected to TAP must implement DHCP correctly (send DHCPDISCOVER etcetera).

20. *Erik* says:
July 16, 2013 at 12:54

Hello,

it is possible to read from a TAP-Device with multiple Threads? (from one Process)
I have created two Threads (with pthread) and both call 'read()' but only one Thread (always the same) comes back with Data and the other Thread is blocked for ever.

Is this behavior normal?
Is the second Thread only used in Situations with higher Load?
Exist similar Problems at writing to an TAP-Device?

Regards
Erik

- *waldner* says:
July 16, 2013 at 14:56

It's a file descriptor, so if you have multiple processes or threads competing for it you also have to manage syncronization, locking etc. yourself.

  - *Erik* says:
July 19, 2013 at 13:10

Hello,

i do not understand what you mean.
In my opinion is read a threadsafe Function/Syscall and the Linux-Kernel must do that not multiple Callers read the same Data.
But if the TAP-Device is a "normal" File-Descriptor than should it have the same Rules as all other File-Descriptors to, i try to find a Linux specific Documentation for this Topic.

Thanks for your Replay.
Erik

    - *waldner* says:
July 19, 2013 at 14:36

The problem is with the meaning of "is it possible to read", as you say. Technically is it of course possible. Whether the results are what you expect, is an entirely different matter.
Without checking, I would think that read() is thread-safe, however you can't go wrong if you implement synchronization and locking yourself, rather than relying upon the underlying implementation.

21. *Ronex* says:
July 12, 2013 at 07:35

Hello,

I am facing some issues while using tap interface over windows.
I have used source /tap driver provided by openvpn, I start with a simple program which reads frame over tap descriptor.

For tap adapter in windows, I assign an IP using
"netsh interface ip set address my_tap_iface static 10.0.0.1 255.255.255.0"

The interface is in Enabled(UP) state.

Another thing I do is disable firewall for "my_tap_iface" (Though it's not a better approach to do...)

I found that the application attached with tap interface reads unwanted frames, which I guess should not be destined for tap interface,

Some ARP packets asking 1) who has 10.0.0.1 ? tell 0.0.0.0, 2) who has X.X.X.X ? tell 0.0.0.0, 3) others are related to IPv6 and DHCP protocols.

This things are might happen because I don't have configure tap interface properly.
Can anyone suggest proper configuration steps as described above in the tutorial for linux, and also what are the recommended settings for firewall.

Secondly how to setup NAT iptables rules over windows. like what is similar for the following command in windows:
"iptables -t nat -A POSTROUTING -s 10.0.0.0/24 -o eth0 -j MASQUERADE".

22. *pendrive* says:
June 28, 2013 at 00:02

hi Waldner
First of all, I'm going to thank you for this nice tutorial.
Second of all, I'm gonna implement a udp tunnel based on this tutorial. in the first version, while there is only on client, it works fine. but in a situation that multiple clients want to connect to the server, it can get all from socket and write it to the tun file descriptor, but when comes to get the data from tun_fd, it should make its own choice to send it back to which of the remote clients. the easies way comes to mind, is to create multiple tun_fd for each connection from clients and make a thread for waiting for that tun_fd and each tun_fd is for exactly one client connection.
you mentioned that it is possible to make a tun/tap presistance and with the same owner, connecting to it with every proccess,
I created a tun and setted it presistant. then closed the program and connected to the tun again. So it worked. but all I need is to able to connect multiple times to the same tun from the same program to have multiple tun_fd for each connection. but whenever I call the tun_alloc from my program, it gives me this crap: "ioctl(TUNSETIFF): Device or resource busy"
I tried the forking and pthreading yet still the result is the same.
it is said that the 3.8 kernel has added the multiple queuing capabilites but I have not upgrade to it yet, and am still wondering how openvpn is doing that (have not taken a looked at its sources).
so any thoughts?
thanks

- *waldner* says:
June 28, 2013 at 12:55

  I think at most one program at a time can connect to the tun interface (that is, clone /dev/net/tun to obtain the tun fd). But once that is done, the fd can be accessed by multiple processes (eg children of the original process) or threads. Of course, then you'll have to manage contention yourself.

23. *Griffin* says:
June 27, 2013 at 09:14

Hi Its nice to see something to read abt TUN/TAP. It feels like everybody is using it but nobody cares wat goes on behind the scenes.
Something that i missed in this tutorial and that i wanted to know is ..
I set up a TUN interface using C program and its UP as well...I wanted this interface to be assigned IP(v6) through the SLAAC procedures since thats the behaviour i observed for my eth0 interface.(when eth0 is brought up it sends out router-solicitaions and the procedure for IPv6 acquision completes). Don't TUN/TAP interfaces behave the same way as normal interfaces and can acquire the IP normal procedures? OR the creater takes care of IP allocation on behalf of these TUN/TAP interfaces?

- *waldner* says:
June 27, 2013 at 16:49

SLAAC certainly works with tap interfaces (provided, of course, that you arrange things so that it sees the actual RA frames). I'm not sure about tun, since I've never played with SLAAC + tun. Although netstat shows a tun interface joins the ff02::1 multicast group, it doesn't have a link-local address and it looks like it does not send RD messages; furthermore, as it doesn't have a MAC address, I'm not sure what would happen if it was asked to do SLAAC. More information is welcome.

- *waldner* says:
  June 28, 2013 at 11:48

  So after a few tests, it seems that SLAAC definitely works with tun too. You just need to assign manually a link-local address to it (eg fe80::9999/64), and it will happily start to send RD. Once an RA is received, the interface is autoconfigured with the advertised prefix plus the lower 64 bits of the previously configured link-local address, so in this example if the RA advertises a prefix of 2001:db8::/64, the tun interface ends up configured with 2001:db8::9999/64 after SLAAC.

  - *Griffin* says:
    July 9, 2013 at 06:24

    Thanx for the quick check for me the tun interface does send request to join multicast group but SLAAC is not initiated. i m using centOS 5.x with kernel "2.6.18-348.6.1.el5". could this be the problem?

    - *waldner* says:
      July 9, 2013 at 09:00

      Did you assign a link-local address to the interface? eg ip addr add fe80::4321/64 dev tun0?

      - *Griffin* says:
        July 9, 2013 at 11:05

        this is a snippet of how i m trying to achieve this

```
if( (fd = open("/dev/net/tun", O_RDWR)) < 0 )
{
printf("open failed\n");
return INVALID_HANDLE;
}

memset(&ifr, 0, sizeof(ifr));

/* Flags: IFF_TUN - TUN device (no Ethernet headers)
* IFF_TAP - TAP device
*
* IFF_NO_PI - Do not provide packet information
*/
ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
sprintf(ifr.ifr_name, "%s%d", TAP_IF_NAME_PREFIX, dev_num++);

if( (err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0 ){
printf("ioctl failed: %d\n", errno);
close(fd);
return INVALID_HANDLE;
}

if(ioctl(fd, TUNSETPERSIST, 0) < 0)
{
perror("TUNSETPERSIST");
return INVALID_HANDLE;
}

unsigned char ipv6_linklocal_addr[NUMBER_OF_IPV6_OCTETS] = {0};
if ((sockfd = socket(AF_INET6, SOCK_DGRAM, 0)) ifname);

inaddr = (struct sockaddr_in *) &ifr.ifr_addr;
inaddr->sin_family = AF_INET6;
inaddr->sin_port = 0;

ifr.ifr_flags |= IFF_UP | IFF_RUNNING;
if(ioctl(sockfd, SIOCSIFFLAGS, (void *) &ifr) ifname);
perror("");
return 1;
}

if(ioctl(sockfd, SIOGIFINDEX, (void *) &ifr) ifname);
perror("");
return 1;
}

//since we are configuring this interface for the first time..so first address allocated MUST be
```

```
link-local so append FE80 to this address
memcpy(ipv6_linklocal_addr, netArgs->ipv6_configuration.hostIPv6,
NUMBER_OF_IPV6_OCTETS);
ipv6_linklocal_addr[0] = 0xFE;
ipv6_linklocal_addr[1] = 0x80;
memcpy(&ifr6.ifr6_addr, ipv6_linklocal_addr, NUMBER_OF_IPV6_OCTETS);
ifr6.ifr6_ifindex = ifr.ifr_ifindex;
ifr6.ifr6_prefixlen = 0;

if(ioctl(sockfd, SIOCSIFADDR, (void *) &ifr6) ifname);
perror("");
return 1;
}
```

- *Griffin* says:
  July 10, 2013 at 07:07

  update:
  tried it on debian wheezy and it worked...it has kernel 3.2 i guess..

  - *waldner* says:
    July 13, 2013 at 12:26

    Unfortunately the code snippet you pasted isn't complete and does not compile, but I'm glad to see that it works on recent kernels.

    - *Griffin* says:
      July 15, 2013 at 05:55

      I realized that code snippet was not OK..the copy/paste ate up few words from here and there :(. I wish there was some Attach here so i could attach the file. Thank you for your kind help and research, much much appreciated. :)

24. *wen* says:
    June 12, 2013 at 09:31

    Hi, thanks for your tutorial

    I encountered a strange behavior, as in the link

    http://stackoverflow.com/questions/16915322/delivering-a-packet-through-tun-file-descriptor-leads-to-different-results

    can you help me look at it?

    thanks!

    - *waldner* says:
      June 16, 2013 at 13:17

      It's quite strange, it definitely works using kernel 2.6.32 (debian squeeze), however it doesn't work with 3.9.6 (unless straced, which makes it even stranger).
      This needs further investigation.

      - *waldner* says:
        June 16, 2013 at 15:03

        It even works with wheezy's 3.2.0, and with latest 3.10-rc4. The only distro I have access to where it doesn't work is archlinux with 3.9.6.

        - *wen* says:
          June 16, 2013 at 23:23

          there is a guy who said: "your tun2 is flagged NO-CARRIER and state is down". what did he mean? maybe this is a clue?

          - *waldner* says:
            June 17, 2013 at 09:01

            NO-CARRIER means that your code (which acts like the "wire" for the tun interface) is not running; if your program is not running, it's normal. I suggest you make sure you understand this before doing anything with tun/tap.

25. *einar* says:
    June 10, 2013 at 16:06

    Hi
    I thought someone here might help with my issue.

    I've been trying to use libdnet (libdumbnet in debian) to connect to TUN, since it sets up the IP address and goes "up" in the source code without needing to call "ifconfig tun0 up"/"ip link set tun0 up" on a command line.

My problem is that any received packages to the TUN IP address are dropped, even if the settings for it seem identical to the simpletun settings. The only difference* is that ARP is still on when using libdumbnet->tun_open().

I read the kernel source code for tun, and it can't drop packages - so it seems that they get forwarded to the network driver, and dropped there for some reason. Packets to other IP addresses (e.g. 10.0.0.3-254) just get received (and lost) without increasing the dropped RX package count.

Is it dropping the package because it thinks the IP address is being spoofed?
Does my P-t-P address need to be different from the local TUN address (it's the same in simpletun)? If so, in what range?
Any idea what needs to be done to fix this?

Ip addresses used:
master: tun0 = 10.0.0.1, eth0 = 192.168.0.101
slave: tun0 = 10.0.0.2, eth0 = 192.168.0.102
---
wireshark output - sending and receiving ping requests works, but there's no ping response by either side.
tshark: Lua: Error during loading:
[string "/usr/share/wireshark/init.lua"]:45: dofile has been disabled
Running as user "root" and group "root". This could be dangerous.
Capturing on tun0
0.000000 10.0.0.2 -> 10.0.0.1 ICMP 84 Echo (ping) request id=0x0145, seq=1/256, ttl=64
1.001133 10.0.0.2 -> 10.0.0.1 ICMP 84 Echo (ping) request id=0x0145, seq=2/512, ttl=64
7.965560 10.0.0.1 -> 10.0.0.2 ICMP 84 Echo (ping) request id=0x2505, seq=1/256, ttl=64
9.764407 10.0.0.1 -> 10.0.0.2 ICMP 84 Echo (ping) request id=0x2506, seq=1/256, ttl=64
---
ifconfig output - Number of RX packets == dropped on both sides (unless 10.0.0.3 would be pinged).
ARP is still on when using tun_open(), but according to the Linux Device Drivers book that flag is ignored for point-to-point connections
ifconfig output for the slave - simpletun works with these same settings (I tried to turn ARP on manually for Simpletun, and it still worked fine)...
tun0 Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
inet addr:10.0.0.2 P-t-P:10.0.0.2 Mask:255.255.255.0
UP POINTOPOINT RUNNING MULTICAST MTU:1500 Metric:1
RX packets:2 errors:0 dropped:2 overruns:0 frame:0 # after 2 ping requests from 10.0.0.1
TX packets:2 errors:0 dropped:0 overruns:0 carrier:0 # after 2 ping requests to 10.0.0.1
collisions:0 txqueuelen:500
RX bytes:252 (252.0 B) TX bytes:168 (168.0 B)

*) libdumbnet also has a small bug/feature in the tun_send() code that returns 4 bytes more than tun_recv() (because the IFF_NO_PI flag is unset) - I decrement that number after the call, so even code that is effectively identical to simpletun has this problem with dropped packets.

- *waldner* says:
  June 11, 2013 at 09:32

  I've never used libdumbnet, I might look into it when I have some time. Meanwhile, let's see if someone can help.

  - *einar* says:
    June 11, 2013 at 11:31

    Libdnet/libdumbnet has a really simple wrapper for TUN, but does other things as well.

    The only relevant question is how to set the point-to-point address. According to ifconfig, simpletun uses the same one for both interface address and point-to-point address, but that doesn't seem to work in my case.
    I set the server IP= "10.0.0.1/24" and p-t-p="10.0.0.1/24".
    For the client, it's "10.0.0.2/24" and p-t-p="10.0.0.2/24".

    Libdnet/libdumbnet has a special intf_entry struct to set up the interface (my main reason for using it), but I'm probably not using it right.
    It sets only these variables before calling intf_set(), which is the equivalent of "sudo ip ..."
    strlcpy(ifent.intf_name, tun->ifr.ifr_name, sizeof(ifent.intf_name)); /* the name ("tun%d") */
    ifent.intf_flags = INTF_FLAG_UP|INTF_FLAG_POINTOPOINT; /* P-t-p interface and starts UP */
    ifent.intf_addr = *src; /* interface address+bitmask - converted from string */
    ifent.intf_dst_addr = *dst; /* point-to-point dst+bitmask - converted from string */
    ifent.intf_mtu = mtu; /* interface MTU - set to 1500 */

    It sets the IFF_NO_PI flag off, so 4 bytes of package info are sent and received before the buffer, but the library takes care of adding and removing them without the user ever knowing about them.

    - *waldner* says:
      June 11, 2013 at 12:09

      Simpletun does not set the POINTOPOINT flag. Without looking, have you tried setting the point-to-point

address to that of the other peer (eg 10.0.0.2 on the server and 10.0.0.1 on the client)?

- *einar* says:
  June 11, 2013 at 17:13

  Note that the ifent structure is handling stuff usually done by ifconfig/ip, so the INTF_FLAG_POINTOPOINT flag has no relation to the IFF_POINTTOPOINT flag, which is not used. Only IFF_TUN is set for the tun connection.

  I have tried many combinations of IP addresses, e.g. from the Server side (10.0.0.1) I have tried putting in the client ip (10.0.0.2), a different IP (10.0.0.10 and 10.0.0.254), the eth0 IP (192.168.x.y), and even 127.0.0.1...

  I just thought the experts here might have an "obvious" solution for me.

  - *einar* says:
    June 12, 2013 at 08:59

    I decided that maybe libraries might have bugs, so I tried to see what happens if I add the IFF_NO_PI flag to the libdnet source, and compiled it like that.

    Ping and ssh work right away - with point-to-point IP address == tun IP address.

    The whole changes I did....

    @@ -45,7 +45,7 @@
    - tun->ifr.ifr_flags = IFF_TUN;
    + tun->ifr.ifr_flags = IFF_TUN | IFF_NO_PI;

    @@ -86,7 +86,7 @@
    - return (writev(tun->fd, iov, 2));
    + return (write(tun->fd, buf, size)); // 4 bytes longer than the result of (writev(...))

    @@ -100,7 +100,7 @@

    - return (readv(tun->fd, iov, 2) - sizeof(type));
    + return (read(tun->fd, buf, size)); // same length as the result of (readv(...)-sizeof(type))

    Sending this 4 byte packet info seems to mess something up and cause packets to be dropped...
    I guess I just need to contact the packet maintainer instead of ranting in the comment section of 3 year old blog posts.

    - *waldner* says:
      June 12, 2013 at 18:06

      Well, thanks for sharing your findings.

26. *Neha* says:
    June 3, 2013 at 13:12

    Hi,

    I am using the tun interface to intercept packets sent to destinations in a particular address range. I then modify the destination address and send the packets out into the network.

    The packets are successfully being directed to tun and being read off /dev/net/tun0. However, after making the change to the destination address, when I write back the packet to /dev/net/tun0, the packet disappears into a blackhole instead of going to the destination across a network. tcpdump shows the packet being written to tun0 without any errors, but that's the last I see of the packet. tun is set up in pointopoint mode with the address 10.0.0.1.

    Even if the destination address is the local machine, the packets get lost.
    However, if I change the source IP and replace it with IP of some remote m/c it works fine.

    Is there some special route configuration that I need to do for outgoing packets? I'm essentially trying to use tun like a raw socket here to send outgoing IP packets without any openvpn encapsulation. Is this a legitimate usage of tun at all? Any help or pointers will be appreciated.

    PS: I have disabled rp_filter settings.
    sysctl -w net.ipv4.conf.eth0.rp_filter=0
    sysctl -w net.ipv4.conf.tun0.rp_filter=0

    And forwading is enabled
    echo 1 > /proc/sys/net/ipv4/ip_forward

    - *waldner* says:
      June 3, 2013 at 20:12

      I'm not sure I understand what you're trying to do. There's no such thing as /dev/net/tun0. Anyway, although I

don't see the point in reading packets from tun and writing them back again, if you change even a single bit you have to recalculate all the relevant checksums. Are you sure you're doing it correctly?
Also to you should also set net.ipv4.conf.all.rp_filter to 0, since "The max value from conf/{all,interface}/rp_filter is used when doing source validation on the {interface}", so if you have nonzero in "all" it will override the interface value. Without knowing exactly what you're doing it's difficult to say more.

- *Neha* says:
  June 4, 2013 at 07:05

  Hi Waldner,

  Thanks a lot for immediate response.
  /dev/net/tun0 was a typo.I would like to aplogize for that. What I have used is actually /dev/net/tun.

  >Anyway, although I don't see the point in reading packets from tun and writing
  >them back again, if you change even a single bit you have to recalculate all the > relevant checksums. Are you sure you're doing it correctly?

  Reading packets and writing them back is my requirement..I have recalculated all the relevant checksums.

  Also Setting net.ipv4.conf.all.rp_filter = 0 as suggested by you didnt work.

  I have tried following cases:

  Case 1 :
  src IP - 172.26.192.150 (local machine)
  dest IP - 172.26.192.128 (remote machine)
  In the case the packet gets lost after written to TUN

  Case 2:
  If I change the src IP of the intercepted packet,recalculate the required checksum and re-inject that packet, the packet gets forwarded to the proper interface.
  src IP - Changed from 172.26.192.150 (local machine) to 172.26.192.149 (remote machine)
  dest IP - 172.26.192.128 (remote machine)
  This condition works.

  Do I have to change any other kernel parameters to make this work?
  Kernel details : 3.1.0-7.fc16.i686.PAE

  Any pointers would be greatly appreciated.Hoping for a response.

  - *waldner* says:
    June 4, 2013 at 13:22

    I suppose you should usa a different IP address as source, if 172.26.192.150 is already assigned to the local machine. This comment in ipv4/route.c in the kernel makes it quite explicit what happens if an incoming packet has a local source IP address:

    ```
    /*
     *      NOTE. We drop all the packets that has local source
     *      addresses, because every properly looped back packet
     *      must have correct destination already attached by output routine.
     *
     *      Such approach solves two big problems:
     *      1. Not simplex devices are handled properly.
     *      2. IP spoofing attempts are filtered with 100% of guarantee.
     *      called with rcu_read_lock()
     */
    ```

    So I guess this is the problem you're seeing, since if you change the source IP it works. Perhaps you could try enabling log_martians and see if some message is produced.

    EDIT: after a few tests with scapy, this seems indeed to be the case. If you enable log_martians, you should be getting a message similar to

    ```
    IPv4: martian source x.x.x.x from 172.26.192.150, on dev tun0
    ```

    in the logs.

    - *Neha* says:
      June 5, 2013 at 09:58

      Hi Waldner,

      I got the same message when I enabled log_martians .

      Thanks a lot for the info.

27. *Ronex* says:
    June 1, 2013 at 07:42

    Hi Waldner,

Thanks for a such a good tutorial,

I need to connect TAP interface with physical device (eth0), to communicate with the internet.

I am able to access Tap interface,
Assigned an IP to the TAP interface using,

** #ip addr add 10.0.0.1/24 dev tap0 ---> Does it means ? **

"what I understand here is,
that it assigns an IP address 10.0.0.1 to the tap0 interface and
it also specifies subnet mask for tap0 is 255.255.255.0,

For kernel it's indication that Ethernet frames with destination
IP address ranges from 10.0.0.0 to 10.0.0.254 (except 10.0.0.1)should be destined(or forwarded)
to the tap0 interface where user application attahched to tap0 can read those frames."

As I am new to the networking stuff,I couldn't understand why we are not able to capture frames for 10.0.0.1 IP ?
"Though it explained as Linux kernel has ping server which respond to that" Can u clarify this more.

====== Main thing I need to do is ======

I need to communicate with internet using the tap0 interface, with the help of Host ethernet(eth0) port.

Here, I am creating Ping ICMP request frame for IP address say `www.yahoo.com` and
write it over the tap0, and I wanted to capture ping ICMP reply of the same over
the TAP interface over `wireshark`, One application is attached with TAP to read and write ethernet frames.

The same thing I need to implement for LAN IP addresses.
like pinging IP address of LAN network, and having ping reply.

If I am not wrong I need to some how forward/route the Ethernet frame
created by User application attached to TAP interface, through the Host Ethernet
port(eth0) which is connected to the internet,
And the received ping icmp response over eth0 should be forwarded to TAP
interface IP address(i.e. consider 10.0.0.1)

Kindly explain the necessary things I need to do to achieve this ? If it is routing or gateway related setup how it could be done ?

And what should be the content of frame which I create for ICMP request ? Destination and source MAC as well as Destination and Source IP ?

Any help will greatly appreciated. Hoping for the early response.

- *waldner* says:
  June 1, 2013 at 11:33

  Regarding why you can't see traffic for 10.0.0.1, it's the same thing that happens with normal interfaces. For example, if you ping the IP address of your eth0 ethernet interface, and run tcpdump at the same time on the interface, you won't see any packet. This is because when an interface is configured with an IP address, a route to that IP address is added in the special "local" routing table, which tells the kernel that the destination is local. You can see this special route (among others) by doing "ip route show table local".
  If you remove this special local route, the kernel will no longer know that the destination is local, and will indeed send packets out the interface, even those destined for the local IP address.
  However note that removing the special local route pointing to the interface's IP address has other adverse consequences (for example the kernel will start sending out ARP requests for the IP address even if it's on a local interface), so you should do it only for testing purposes and then restore it.
  Note that even when the local route is present (ie, the normal situation), you will be able to see traffic for local addresses on the loopback interface (lo), so if you ping the tap interface IP address and run tcpdump on lo, you will see the packets.

  Regarding your application, you should be able to accomplish your goal by using the normal Linux facilities, that is, enable routing (by writing 1 in the special file /proc/sys/net/ipv4/conf/all/forwarding) and probably using a simple iptables rule to NAT the traffic leaving the local network. So your application creates an ethernet frame containing a ping packet (ICMP echo request) destined for the IP address of yahoo.com. Since you're using a tap interface, this should be a complete ethrnet frame, with ethernet header, IP header, and ICMP header and payload. It's entirely up to you to build the frame correctly.

  Assuming a point-to-point setup for your tap interface (ie, not bridged), the source MAC address of the frame can be arbitrary (as long as it's valid), while the destination MAC should be that of the tap interface (so the kernel will pick it up). Let's call these addresses $sourceMAC and $destMAC. $sourceMAC is arbitrary (but you have to remember it, as we'll see), while $destMAC is the MAC address of the tap interface. At the IP level, the source IP (let's call it $sourceIP) can be any IP in the range configured for the tap interface, let's assume 10.0.0.2, while the destination IP ($destIP) should be that of yahoo.com (in case it resolves to many IP addresses, you - or your resolver routines - will have to choose one of them). For the ICMP stuff, you have to build a correct ICMP header with ICMP type 8 and code 0, and the payload can be more or less arbitrary, as long as its length is consistent with

the length you declared in the IP header (normally, it would be ICMP payload length + 8 bytes ICMP header + 20 bytes IP header, but it's not always true). See here for some informtion: http://en.wikipedia.org/wiki/Ping_%28networking_utility%29.
Note that it's entirely your code's responsibility to create and correctly fill each and every header, field and checksum at the ethernet, IP and ICMP layers. It's a lot of work, and it's very easy to do something wrong. Still, it can be a very useful experience. I'm going to assume that the frame you create is valid, with correct headers and checksums, otherwise the kernel will drop it. If you don't see anything entering the tap interface, that's probably the case.

So once you have built your frame, your code writes it into the tap interface, where the kernel will see it as incoming. Since $destMAC of the frame is that of the tap interface, the kernel will pick up the frame. Then the kernel will look at the IP header and see that $destIP (that of yahoo) is not local, so it will do a routing lookup and send the packet to the interface that has a route to $destIP (I'm assuming it's eth0 in your case). Since $destIP is not in the local LAN, you'll need an iptables rule to rewrite $sourceIP as it leaves eth0, for example

```
iptables -t nat -A POSTROUTING -s 10.0.0.0/24 -o eth0 -j MASQUERADE
```

If all this works out, your packet should indeed reach yahoo, which will reply with an ICMP echo reply packet. With some luck, this reply packet will reach your machines' eth0, where the kernel will see that it's a reply to the ICMP echo request it saw previously, so the NAT will be undone and the destination address will be changed to 10.0.0.2. Since the route to 10.0.0.0/24 points to the tap interface, the kernel will have to forward this ICMP echo reply packet to the tap interface, which is where your application will need to be ready to read it and process it. But wait, it's not that easy: to be able to send the ICMP echo reply to the tap interface, the kernel needs to know the MAC address of 10.0.0.2, so it can correctly build the ethernet header before. **So it's very likely that the kernel will send an ARP packet to the tap interface, asking "who has 10.0.0.2? Tell 10.0.0.1"**. Your application MUST be prepared to read this ARP packet and reply accordingly, in other words, create an ARP reply packet saying "10.0.0.2 is at $sourceMAC" (remember when we said you have to remember $sourceMAC). When you create this ARP reply packet (again with correct headers, checksums etc.), you write it to the tap descriptor, so the kernel will see an ARP reply incoming and see that 10.0.0.2 is at $sourceMAC. Only now will the kernel be able to build the ethernet frame containing the pending ICMP echo reply, so it will do so and send it to the tap interface, where again your application MUST be ready to read it and process it.

And all this work only to process a single ICMP request/reply!

Hope this helps.

- *Ronex* says:
  June 2, 2013 at 07:35

  Hello Waldner,

  Thanks for helping me out on my query and for such a good explanation again, as it has clear the picture. I successfully got the Ping ICMP Reply frame.

  I have implemented code for ARP Reply, in that to generate ARP reply frame I carried out following steps:

  1) change the Destination/Target address of ether_II frame (i.e., replace it with the source Address received in ARP request - MAC address of TAP)

  2) updated the Source MAC address with arbitrary address(A1:B1:C1:D1:E1:F1) which is used before while generating Ping request

  3) In ARP part,
  a) changed the OPER field for ARP Reply(0x0002)
  b) Alter(swap) SRC & DST IP address
  c) Updated DST MAC with the received SRC MAC (i.e MAC addr of TAP)
  d) Updated SRC MAC with(A1:B1:C1:D1:E1:F1)

  I am very thankful to you, For solving my issue,

  I wanted to ask that "Is this the same way we need to enhance the application attached with TAP interface to support other Internet protocols", and In case of physical device eth0 Does it handle through ethernet controller driver so that eth0 supports internet entirely.

  My application would just support ICMP Reply, ARP Reply and retrieving the ICMP request.
  How can I make it better to support other internet protocols in optimised approach.
  Kindly share some optimum way to make it better.

  Kindly share any reference Books I should refer for good understanding in networking, and virtual network interface.

  Thanks Once Again... :)

  - *waldner* says:
    June 2, 2013 at 10:51

    First of all, well done for what you've accomplished so far. In the case of physical device eth0, the ethernet controller driver manages the actual hardware settings (for example, MAC address

(re)programming, interrupt handling, MTU frame setting, card tx/rx queues, and so on); each ethernet card has its own specific way of doing these things, so the controller driver takes care of hiding the details. Everything else (ARP, IP, TCP etc.) is implemented on top of that, in the case of Linux by dedicated kernel modules (you might want to look into net/ipv4 and net/ipv6 in the linux kernel source code). In the case of tap device, the "ethernet controller" is virtual and you don't have to do anything in your application, since the kernel already hands over complete frames and expects complete frames from your application. But again, your application has to implement everything else (ARP, IP, TCP, etc.), although the focus will have to be slightly different from an in-kernel implementation (a bit easier, I'd say).

If you want to go further, there are plenty of sources to learn more about protocol implementation: personally, I would suggest the first two volumes of W. Richard Stevens' TCP/IP illustrated ("TCP/IP Illustrated, Volume 1: The Protocols" and "TCP/IP Illustrated, Volume 2: The Implementation), and Comer's "Internetworking with TCP/IP Volume II: design, implementation and internals". They are not the latest and greatest, but surely can provide a solid foundation and plenty of material to build upon.

Good luck with your projects!

- *Ronex* says:
  June 6, 2013 at 14:30

  Hello Waldner,

  Thanks for wishes. and also thanks for all the previous reply and explanations.

  I would like to request you to kindly share any document which list out all the thing which I need to implement in the Application attached to the tap interface so that it can work as like a real physical device eth0 ?

  It would be good reference as well as good agenda for me. Otherwise I afraid that whether I am going on the right path or wrong.

  - *waldner* says:
    June 6, 2013 at 21:58

    Strictly speaking, the tap interface already works like an ethernet interface, both are managed by the kernel and not by your application (whatever this means), so you don't have to do anything.
    What your code should do depends entirely on your goals; broadly speaking, you use the frame transmission/reception facilities provided by the "physical" adapter (eth0, tap, whatever) to build "something" on top of that. There's no rule that defines what this "something" should be; it could be a simple ping simulator as you did, or it can be a traffic analyzer, or an entire TCP/IP stack, or a VPN ... these are just examples, the possibilities are many; you decide. The tap interface is a tool that you use to implement whatever you want or need. So, sorry but I'm not going to suggest or recommend anything.
    I believe that the Internet provides enough documentation and resources to undertake whatever project you want to pursue, if you are willing to learn and experiment.

28. *sjram* says:
    April 16, 2013 at 12:26

    hi,

    I want to connect a host machine which have eth0 and eth1 interfaces.
    I want to forward whatever ethernet frames comes to eth0 to tap0 and eth1 to tap1.
    I used gnuradio which will receive tap0 and tap1 ethernet frames and modulate and send to remote system via usrp.

    Nut, when I create tap0 and tap1, then create bridge br0 and br1.
    br0 have tap0 and eth0 interfaces attached, br1 have tap1 and eth1 interfaces attached.
    I want to send data from my host to remote host via tap.

    But, when I try to send using iPerf , no data is received in tap interfaces.

    what is my problem, what I need to correct.
    routing already done.

    Thank you

    - *waldner* says:
      April 19, 2013 at 10:13

      Is there something (ie, a program) connected to the tap interface that is in charge of sending the frames to the remote system?

29. *bishneet* says:
    April 5, 2013 at 16:04

    Hi, I have a doubt

I am creating a tap interface using tunctl and then using ssh to connect 2 systems using that tap inteface. Then I am assiging IP to both tap interfaces in client as well as server using ifconfig. But the problem is I am not able to ping both the systems. Output for some coomands which may be of use are:

at client

bishneet@bishneet:~$ sudo ip addr show
1: lo: mtu 16436 qdisc noqueue state UNKNOWN
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
inet6 ::1/128 scope host
valid_lft forever preferred_lft forever
2: eth0: mtu 1500 qdisc pfifo_fast state DOWN qlen 1000
link/ether e8:11:32:01:4a:1e brd ff:ff:ff:ff:ff:ff
3: wlan0: mtu 1500 qdisc mq state UP qlen 1000
link/ether 4c:ed:de:74:b4:01 brd ff:ff:ff:ff:ff:ff
inet 10.154.148.117/22 brd 10.154.151.255 scope global wlan0
inet6 fe80::4eed:deff:fe74:b401/64 scope link
valid_lft forever preferred_lft forever
10: tap2: mtu 1500 qdisc pfifo_fast state UP qlen 500
link/ether 7e:5b:ea:9b:50:37 brd ff:ff:ff:ff:ff:ff
inet 10.0.0.201/8 brd 10.255.255.255 scope global tap2
inet6 fe80::7c5b:eaff:fe9b:5037/64 scope link
valid_lft forever preferred_lft forever

bishneet@bishneet:~$ sudo route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
10.154.148.0 0.0.0.0 255.255.252.0 U 2 0 0 wlan0
169.254.0.0 0.0.0.0 255.255.0.0 U 1000 0 0 wlan0
10.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 tap2
0.0.0.0 10.154.148.1 0.0.0.0 UG 0 0 0 wlan0

bishneet@bishneet:~$ sudo arp -n
Address HWtype HWaddress Flags Mask Iface
10.0.0.101 (incomplete) tap2
10.154.148.1 ether 00:15:c7:62:3c:00 C wlan0

at server side:
bishneet@bishneet:~$ sudo ip addr show
1: lo: mtu 16436 qdisc noqueue state UNKNOWN
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
inet6 ::1/128 scope host
valid_lft forever preferred_lft forever
2: eth0: mtu 1500 qdisc pfifo_fast state UP qlen 1000
link/ether 00:25:64:9c:d2:0c brd ff:ff:ff:ff:ff:ff
inet 128.243.35.15/24 brd 128.243.35.255 scope global eth0
inet6 fe80::225:64ff:fe9c:d20c/64 scope link
valid_lft forever preferred_lft forever
6: tap2: mtu 1500 qdisc pfifo_fast state UP qlen 500
link/ether de:ae:06:db:5b:35 brd ff:ff:ff:ff:ff:ff
inet 10.0.0.101/8 brd 10.255.255.255 scope global tap2
inet6 fe80::dcae:6ff:fedb:5b35/64 scope link
valid_lft forever preferred_lft forever

bishneet@bishneet:~$ sudo route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
128.243.35.0 0.0.0.0 255.255.255.0 U 1 0 0 eth0
169.254.0.0 0.0.0.0 255.255.0.0 U 1000 0 0 eth0
10.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 tap2
0.0.0.0 128.243.35.1 0.0.0.0 UG 0 0 0 eth0

bishneet@bishneet:~$ sudo arp -n
Address HWtype HWaddress Flags Mask Iface
128.243.35.1 ether 00:15:c7:24:4a:c0 C eth0
10.0.0.201 (incomplete) tap2
10.154.148.117 (incomplete) tap2

can you help??

- *waldner* says:
  April 5, 2013 at 16:37

What is the ssh command you're using to connect? See if this helps: http://backreference.org/2009/11/13/openssh-based-vpns/

- *bishneet* says:
  April 5, 2013 at 16:43

  sudo ssh -v -w 0:0 -o Tunnel=Ethernet host@ip

  - *waldner* says:
    April 5, 2013 at 16:48

    So use tcpdump, see where packets are not getting through and work from there.

  - *bishneet* says:
    April 5, 2013 at 16:50

    its working now. I don't know what was the problem though. I did nothing new, it is working now. Thanks

30. *Manjula* says:
    April 4, 2013 at 12:01

    HI Waldner,
    I am running a Ubuntu 12.04 Virtual Machine. I am doing exactly this:
    root@sarumm-Ubuntu:/dev/net# openvpn --mktun --dev tun3
    Thu Apr 4 16:22:04 2013 TUN/TAP device tun3 opened
    Thu Apr 4 16:22:04 2013 Persist state set to: ON
    root@sarumm-Ubuntu:/dev/net# ip link set tun3 up
    root@sarumm-Ubuntu:/dev/net# ip addr add 10.0.0.1/24 dev tun3

    I can see that the interface is up in ifconfig.

    I just opened another terminal and did:
    sarumm@sarumm-Ubuntu:~$ ping 10.0.0.1

    I receive response from kernel.

    sarumm@sarumm-Ubuntu:~$ ping 10.0.0.2

    No response from kernel.

    On ifconfig,

    tun3 Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
    inet addr:10.0.0.1 P-t-P:10.0.0.1 Mask:255.255.255.0
    UP POINTOPOINT 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:100
    RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

    I dont see any change in the Rx bytes. While capturin tun3 on wireshark also, I dont notice anything.

    root@sarumm-Ubuntu:/dev/net# tshark -i tun3
    tshark: Lua: Error during loading:
    [string "/usr/share/wireshark/init.lua"]:45: dofile has been disabled
    Running as user "root" and group "root". This could be dangerous.
    Capturing on tun3

    Guidance please. I am n a tight spot here. Quick replies would greatly help.
    Thanks in advance.

    - *waldner* says:
      April 4, 2013 at 18:02

      If you ping 10.0.0.2, the kernel sends the packet out the tun interface, so there has to be a program connected to it to read those packets and do something with them.

      - *Manjula* says:
        April 5, 2013 at 08:12

        I understand that. But the packets should be atleast getting dropped right?? Wireshark is not capturing those packets at all on tun3.

        - *waldner* says:
          April 5, 2013 at 09:43

          I don't think so. If there's no program connected to the tun interface, the kernels thinks that it's as if the cable is disconnected (no carrier), see eg this quick test I did:

```
# openvpn --mktun --dev tun678
Fri Apr  5 10:33:05 2013 TUN/TAP device tun678 opened
Fri Apr  5 10:33:05 2013 Persist state set to: ON
# ip link set tun678 up
# ip link show dev tun678
16: tun678:  mtu 1500 qdisc pfifo_fast state DOWN mode DEFAULT qlen 100
    link/none
```

I also tested with a real (physical) disconnected interface, and the behavior is the same.

31. *interwebshark* says:
January 18, 2013 at 21:33

Excellent tutorial thanks!

For starters here is a simplified version of what I'm hoping to accomplish with the tun/tap interface in the long run.

- Create a virtual interface (tap) that is assigned an IP address etc.
- Applications will send UDP data to this IP address.
- A userland application will attach to the tap and do some processing of the packets and then forward them on to a non network custom hw device.

From what i've read it seems this is doable using the tun/tap interface. So far i've followed your example and created a tun, bringing it up with IP 10.0.0.1 and tried pinging it, then pinged 10.0.0.2 and that works fine I see the data in my application that reads from the tun device.

So next I thought I would write a simple UDP client which sends a message to 10.0.0.1 and I should see it on the otherside, in the output of the simple program which attaches to and reads from the tun device. However I only see data coming from the tun device if my UDP client sends it to 10.0.0.2 (or anything but 10.0.0.1). While ping makes sense to me, I would expect the UDP to be sent to the tun interface?

I have linked any code yet because, I'm guessing there is some additional configuration step I am not understanding here, maybe I need a route for this? I am just not sure how to set this up and where my assumptions are incorrect.

Any input would be great.

- *waldner* says:
January 19, 2013 at 11:21

  Well, since in the end it's all IP routing, I don't expect UDP to behave differently from ICMP (or TCP, for that matter). The kernel sends packets out of the tun interface if the destination address is not local and the existing routing table tells it to do so. It doesn't matter which upper layer protocol the packet is coming from.

  - *interwebshark* says:
  January 21, 2013 at 18:15

    That makes sense. Guess I'm wondering how to force the kernel into sending data on the wire for this tap interface, when the some local application is sending data to the tap IP address.

    I tried routing traffic through my tap interface by adding a route but that didn't seem to work or I did something wrong. Any suggestions?

    Thanks!

    - *waldner* says:
    January 21, 2013 at 20:45

      I don't think that is easily done, if at all. If the destination address is local, why should the kernel send the packet out on the wire? (note this is not tun/tap specific, it happens with any kind of interface)

      - *interwebshark* says:
      January 21, 2013 at 21:32

        Yeah I would expect the same behavior whether it is virtual or real interface. I see some reference to people doing things with nat's to trick the kernel into sending data out the wire, but it but it looks painful.

        thanks

      - *interwebshark* says:
      January 22, 2013 at 17:50

        FYI

        I ended up getting this to work with a couple different techniques. The simplest was removing the default route for the tun in the local routing table and replacing it with a unicast route. This seems to work for my purpose.

        - *waldner* says:
        January 22, 2013 at 17:56

          Thanks for the update. I didn't know that the local routing table can be modified, though of

course now I don't see why not.

32. *Steve Beaty* says:
December 21, 2012 at 17:02

Greetings!

Thanks for the great writeup! It helped me get a user-mode tunnel running on CentOS 6. Strangely, I can't make it work on Ubuntu 12.04. I create a user-mode tunnel using tunctl, but get a strange error message at run time. Any clues? Thanks!

$ cc simpletun.c

$ sudo tunctl -t tun9 -u beaty
[sudo] password for beaty:
Set 'tun9' persistent and owned by uid 1000

$ ./a.out -i tun9 -s
ioctl(TUNSETIFF): Invalid argument
Error connecting to tun/tap interface tun9!

$ ls -l /dev/net/tun
crw-rw-rwT 1 root root 10, 200 Dec 2 11:31 /dev/net/tun

$ uname -a
Linux emess 3.2.0-34-generic-pae #53-Ubuntu SMP Thu Nov 15 11:11:12 UTC 2012 i686 i686 i386 GNU/Linux

- *waldner* says:
December 22, 2012 at 14:20

Ok, I had to do some tests. First thing I see is that if I use tunctl to create the device, I get the same error you get. But If I create the device using iproute2, it works:

```
root# ip tuntap add dev tun9 mode tun user waldner group waldner
waldner$ simpletun -i tun9 -s
# listening
```

After some investigation, it looks like tunctl creates tap devices only, ie no tun devices, while of course iproute2 allows to specify whether it will be a tun or tap device.

So if you want to use simpletun with a device created with tunctl, you have to tell it to use tap mode:

```
root# tunctl -t tun9 -u waldner -g waldner
Set 'tun9' persistent and owned by uid 1000 gid 1000
waldner$ simpletun -i tun9 -a -s
# listening
```

A quick search shows that apparently there is a later version of tunctl which supports a **-n** switch to create a tun device, and a **-p** switch to create a tap device, tap being the default, but evidently that's not the version shipped with ubuntu.

  - *Steve Beaty* says:
December 31, 2012 at 20:41

This is great info! Thanks much for taking the time to find all this out -- I can make progress again! By (the) way, love \1 name of you site.

33. *RAGHU* says:
December 13, 2012 at 10:00

Hi Waldenr,

Blog is very informative,
I have scenario here, where in I am developing L2 control plane for distributed data plane for a switch,
The control plane would be running Linux on control cards and remaining all the cards have fast path L2 data plane. All data planes will have 4 to 6 ports and each port will be part of bridge.
L2 Control plane protocols are executed on Control cards, BPDUS are received by Data Plane at first and sent back to Control plane card via tcp. Question is whether I can create tap device(bridge) in control plane card and do the bridge state changes in control plane card and will the state change effect the port states in L2 data plane card

Anyways thanks once again for your time .....
Regards
Raghu

- *waldner* says:
December 15, 2012 at 10:46

I'm not sure I understand, but I don't think changes to ports in the control plane can "propagate" via TCP to the data plane, unless you implement it yourself.

34. *Rami Rosen* says:

November 23, 2012 at 10:42

Hi Waldenr,
Thanks for your answer!
I still think there is an error in the text; it says:
If you configure tun77 as having IP address 10.0.0.1/24 and then run the above program while trying to ping 10.0.0.2
(or any address in 10.0.0.0/24 other than 10.0.0.1, for that matter), you'll read data from the device:

```
# openvpn --mktun --dev tun77 --user waldner
Fri Mar 26 10:48:12 2010 TUN/TAP device tun77 opened
Fri Mar 26 10:48:12 2010 Persist state set to: ON
# ip link set tun77 up
# ip addr add 10.0.0.1/24 dev tun77
# ping 10.0.0.1
...

# on another console
$ ./tunclient
Read 84 bytes from device tun77
Read 84 bytes from device tun77
...
```

Now, ping 10.0.0.1 does not give this result, namely on the
other console we will not see traffic.

So I think it should be ping 10.0.0.2 (or any address other than 10.0.0.1, for that matter)

Rgs,
Rami Rosen
http://ramirose.wix.com/ramirosen

- *waldner* says:
  November 23, 2012 at 10:47

  Ah right, **that** is an error, I'm going to fix it now. Thanks, well spotted!

35. *Rami Rosen* says:
    November 23, 2012 at 09:39

    Hi, Waldner,

    Regarding the tunclient.c example above: it seems that there
    is some trivial error which won't let it work; I tried it. it says:

    If you configure tun77 as having IP address 10.0.0.1/24 and
    then run the above program while trying to ping 10.0.0.2
    (or any address in 10.0.0.0/24 other than 10.0.0.1, for that matter),
    you'll read data from the device:

    ```
    # openvpn --mktun --dev tun77 --user waldner
    Fri Mar 26 10:48:12 2010 TUN/TAP device tun77 opened
    Fri Mar 26 10:48:12 2010 Persist state set to: ON
    # ip link set tun77 up
    # ip addr add 10.0.0.1/24 dev tun77
    # ping 10.0.0.1
    ```

    ping to 10.0.0.1 will *not* work, as mentioned above.
    In the example above, it should have been ping to 10.0.0.2
    (or any other address in this subnet, other than 10.0.0.1)

    Rami Rosen

    http://ramirose.wix.com/ramirosen

    - *waldner* says:
      November 23, 2012 at 10:26

      Of course ping to 10.0.0.1 does work (although no packet is sent to the tun interface, as explained). If you want to see traffic going "out" the tun interface, you have to ping any other address in the 10.0.0.0/24 subnet (again, as explained).

36. *Rami Rosen* says:
    November 23, 2012 at 09:25

    Hi,
    >Perhaps the code should check whether the interface exists >before doing anything.
    It is not simple as that. Suppose that the tun kernel code (drivers/net/tun.c) will check and see that the interface does **not** exist. So what ? should it return an error ? I think that it should **not** return an error. The reason is simple:

when you add a device you *also* use TUNSETIFF ioctl, not only in deleting a device.
And when you add a new device, you don't expect it to exist.

Returning an error in the case it exist will avoid creating a new device.

The solution can be change the ioctls in the driver so that there be one
for adding and one for deletion.

BTW, trying to add the same tun device twice is avoided,

ip tuntap add tun0 mode tun
ip tuntap add tun0 mode tun
ioctl(TUNSETIFF): Device or resource busy

Rami Rosen
http://ramirose.wix.com/ramirosen

- *waldner* says:
  November 23, 2012 at 10:32

  I don't understand what you're trying to say. I was replying to your observation that "ip tuntap del" against a non-existent tap interface does not produce an error. So I said that perhaps the "ip" program should check whether the interface exists before trying to delete it; if it doesn't exist, output a message saying that.

  - *Rami Rosen* says:
    November 23, 2012 at 10:52

    Hi Waldner,

    The "ip" which is part of the iproute2, sends an
    ioctl to the tuntap driver for delete and checks the return value. It indeed does not check existence before.

    Usually (I think always, not sure), the iproute2 code (for all other interfaces) does not check existence of interface before removing it.

    It relies on the ioctl, that the kernel will return
    an error if the interface does not exist. But with
    tuntap driver, this is not the case. The driver implementation is a unique one in the tuntap driver,
    as in add/delete we use a pair of identical ioctls,
    TUNSETIFF and TUNSETPERSIST(the second with different values according to the action, add/delete).

    The driver does not give an error in such a case, which is a bit unconventional behavior.

    (I don't know if it is possible to check the existence of tun/tap device, did not look into it).

    Regards,
    Rami Rosen
    http://ramirose.wix.com/ramirosen

    http://ramirose.wix.com/ramirosen

    - *waldner* says:
      November 25, 2012 at 10:00

      This is what "ip" does when deleting a tun/tap interface:

```
static int tap_del_ioctl(struct ifreq *ifr)
{
        int fd = open(TUNDEV, O_RDWR);
        int ret = -1;

        if (fd < 0) {
                perror("open");
                return -1;
        }
        if (ioctl(fd, TUNSETIFF, ifr)) {
                perror("ioctl(TUNSETIFF)");
                goto out;
        }
        if (ioctl(fd, TUNSETPERSIST, 0)) {
                perror("ioctl(TUNSETPERSIST)");
                goto out;
        }
        ret = 0;
 out:
        close(fd);
        return ret;

}
```

      So essentially the TUNSETIFF ioctl creates the interface even if it did not exist before, although this is only for a moment. That's why the following TUNSETPERSIST ioctl does not give an error: because the interface really exists at that instant, although it did not when "ip tuntap del" was invoked. Hence my comment that perhaps the utility could check for the existence of the interface before calling the above function straight away although, on second thought, that would be racy and effectively useless in

practice.

37. *Rami Rosen* says:
November 22, 2012 at 20:27

Hi,
I think there is something which is a bit wrong with the linux tun driver.
I use the recent iproute2 from git.
If I run:
ip tuntap add tun2 mode tun
and by error try to delete a non existant tun interface like:
ip tuntap del tun3 mode tun
I don't get an error.
The same is with tap.

Trying to delete non tun/tap devices, which are non existant interfaces (virtual or not virtual), does give an error.

Delving inside this problem, the issues is not with iproute2 but with the kernel driver, drivers/net/tun.

When you try to delete non existant tun/tap device, what happens is this:
first you have an ioctl of TUNSETIFF.
This same ioctl is used in delete.
So you register a new net device.
Then there is a second ioctl. In case of delete, it is the non persistent ioctl (TUNSETPERSIST).
This ioctl unregisters the tun/tap device.
So no error is returned.

Rgs,
Rami Rosen

- *waldner* says:
November 23, 2012 at 09:01

I haven't seen the code, but I suppose that this is somehow expected. When you are root, connecting to a tun interface (the TUNSETIFF ioctl) also creates it if it doesn't exist. Perhaps the code should check whether the interface exists before doing anything.

38. *Raman* says:
November 6, 2012 at 12:53

Can we assign MAC address to tun virtual interface?

I createa a tun device and when tried to assign it a MAC address using command:-
ifconfig tun0 hw ether 02:00:27:E1:1E:ff

I got the error:-
SIOCSIFHWADDR: Operation not supported

- *waldner* says:
November 6, 2012 at 22:27

Tun devices are purely layer 3, that is, they don't really have a MAC address (both "ip link" and "ifconfig" show that)..
What you want to do works if the device is a tap (layer 2) interface:

```
# ip tuntap add dev tap1 mode tap
# ip link show dev tap1
6: tap1:  mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 500
    link/ether 42:4d:10:e7:21:de brd ff:ff:ff:ff:ff:ff
# ip link set dev tap1 address 00:11:22:33:44:55
# ip link show dev tap1
6: tap1:  mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 500
    link/ether 00:11:22:33:44:55 brd ff:ff:ff:ff:ff:ff
```

39. *Yang* says:
October 18, 2012 at 06:56

Through iptable tracing, the route in tun0 for the incoming traffic from outside or itself looks different. For the ouside traffic, IN=tun0, then after natting, they would go out from eth0. But for my package wrote into tun0, it's different. But since Ashwin's working, mine should work too.

Yang

- *Yang* says:
October 22, 2012 at 00:48

Fixed. an issue about reverse pathing in rhel6. rp_filter better set to be "2". In rhel6, "0","1","2" are implemented into rp_filter than rhel5 or 4. "2" is loose mode and enough for my job.

40. *Yang* says:
October 18, 2012 at 06:26

Hi waldner,

I got a similar problem with Ashwin's. I setup a tun0, write the packages into the device (need to do some natting through iptables), then read them out through file read. By using ipt_LOG, I can see the wrote package is going through raw prerouting and mangle prerouting, but then, nothing happened later. I can't read the the package out and it looks dropped somewhere. The dst IP of the packages is 192.168.10.1 which default route is set to the tun0.

The interesting thing is if the package is coming from the outside with dst IP as 192.168.10.1(such as dns rsp), then they can be routed correctly from eth0 to tun0 through iptables, then I can read them out.

But for the writing internally, it doesn't work. In kernel trace for ipt_LOG, I can see below, but nothing more:

kernel: TRACE: raw:PREROUTING:policy:2 IN=tun0 OUT= MAC= SRC=x.x.x.x DST=192.168.10.1 LEN=276 TOS=0x00 PREC=0x00 TTL=60 ID=50435 PROTO=UDP SPT=53 DPT=60683 LEN=256
kernel: TRACE: mangle:PREROUTING:policy:1 IN=tun0 OUT= MAC= SRC=x.x.x.x DST=192.168.10.1 LEN=276 TOS=0x00 PREC=0x00 TTL=60 ID=50435 PROTO=UDP SPT=53 DPT=60683 LEN=256

Any thought?

- *Yang* says:
  October 18, 2012 at 06:29

  By the way, In ifconfig, the counter of drop and error are all 0. So, it looks not checksum issue.

  Yang

41. *Ashwin* says:
    August 7, 2012 at 00:42

    Thanks a lot for these inputs. I realized that my problem was because of incorrect checksums on the packets I injected. Another problem that I faced was because of large segmentation offloading and checksum offloading. The kernel assumed the device would compute the checksums and handle large segments. However, as my process is responsible for the tunnel device all these forms of offloading failed.

    - *waldner* says:
      August 7, 2012 at 22:13

      Ah well, of course the checksum verification is the very first thing that happens (both for tun and normal interfaces), so if that fails the packet or frame is dropped (although I'd expect it to show up in the error counters for the interface, but I haven't verified this).
      Glad you finally solved it!

42. *Ashwin* says:
    August 1, 2012 at 02:18

    Hi,

    I would like to know if the packets my process writes on the tap device reach the ip stack.

    I am building on top of your simpletun program to create a simple firewall. I am currently forwarding a subset of packets received on my ethernet interface, for example from 192.168.1.0/24, to my tap device. I achieve this by making my tap device IP the default gateway for packets coming from 192.168.1.0/24. Then I am then performing some activities such as filtering packets and performing some deep packet inspection. If the packet satisfies my criteria I modify the ip from 192.168.1.0/24 to 192.168.2.0/24 (and destination from 192.168.2.0/24 to 192.168.1.0/24 for reverse direction) and I writing these packets on the tap device. During the program execution, the packets received on my ethernet interface are being forwarded to my tap device however the packets written on my tap device are not reaching my ethernet interface. My rules in the routing tables are simple. All packets from 192.168.1.0/24 are forwarded to tap0; all packets from 192.168.2.0/24 are forwarded to eth0; all packets to 192.168.2.0/24 are forwarded to tap0; and all packets to 192.168.1.0/24 are forwarded to eth0.

    To summarize, I would like to know if the packets my process writes on the tap device reach the ip stack.

    - *waldner* says:
      August 4, 2012 at 12:10

      I'm not sure how you are setting up this, but you can run tcpdump on each interface to see which packets pass through it. Also you can use set suitable iptables rules to match your traffic and then check the counters to see how many packets matched the rules.

      - *Ashwin Rao* says:
        August 4, 2012 at 19:42

        My question is that if my process writes packets to the file descriptor associated with a tun device, then will these packets be processed by the iptables and the kernel routing tables or will they be dropped if there is no other user space process is responsible for receiving these packets.

        The packets from the kernel to the tun device are being accounted for by the iptables counters and I can see them in the log files. My process listening to the tun device file descriptor can read these packets. Also, when the same process writes some udp or tcp syn packets to the tun device (with a source ip belonging to the

subnet of the tun device and destination say google.com) I can see that they are received by the tun device. ifconfig shows these packets being successfully received by incrementing the RX bytes. Similarly I can see these packets on the tcpdump. However I am not able to see them in any iptables logs INPUT, FORWARD, or OUTPUT. I have disabled rp_filter for each device as well however I cannot see them. I do not know why these packets are dropped.

- *waldner* says:
  August 4, 2012 at 20:06

  If your process writes packets to the tun file descriptor, the kernel will see those packet as INCOMING on the tun interface. That is, the same as if your ethernet adapter received a frame from the wire. As such, those packets will go through the same steps that normal incoming packets follow, including rp_filter, iptables INPUT chain, interface input counter accounting, and the like.
  On the other hand, the packet that the kernel sends to the tun device are OUTGOING packets, and as such they will be accounted and processed (and a program connected to the tun fd will be able to read() them).

  That said, the tun interface isn't different from any other interface. When a tun interface receives a packet, it's like a normal phisical ethernet interface receiving a packet.
  The kernel determines whether the packet is valid, and whether it has to be delivered locally or routed. If it looks like it has to be delivered locally but there's no process to deliver it to, then it's dropped (depending on the actual type of packet, an error may be generated).
  So to make it simple, if your kernel receives a packet from the tun interface (or any interface, for that matter) whose destination IP address matches the IP address of the tun interface, contains a TCP segment with destination port 80, but there's no process "owning" TCP port 80 in the system, then yes, the packet is dropped, and depending on the type of packet, the kernel may send back a TCP RST, or an ICMP error message. Note that (depending on the routing setup) it's quite likely that this error message will go out the same tun interface from which it came in, and your program that is connected to the tun file descriptor should be prepared to catch those packets.

43. *Chris Dew* says:
    July 23, 2012 at 13:15

    Hi, does running code create the device node, or does that need to be done manually?

    http://stackoverflow.com/questions/11612227/tun-program-not-creating-device-node

    - *waldner* says:
      July 23, 2012 at 23:10

      Neither. Network interfaces in Linux don't appear under /dev; the only thing you'll see there is /dev/net/tun, which is the device that should be opened as the first step to create a tun/tap interface. If you run the sample code, you'll be able to see and configure the interface you create by using "ip link" while the program is running; when the program terminates, the interface disappears. Alternatively, the interface can be made persistent, as explained, and in that case it will survive program termination.
      In any case, no device is created under /dev (apart from the already mentioned /dev/net/tun).

44. *Raj* says:
    July 20, 2012 at 18:22

    I have tried your ping program. I'm able to create new tun interface and my program is listening on that descriptor to read data, but it is not receiving any data. On the other end I got the ping reply. Please help to come across.

    I'm using suse 12.1 on VMplayer.

    linux-2m7c:/home/raj/tipctun # ifconfig tun77
    tun77 Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
    inet addr:10.1.1.1 P-t-P:10.1.1.1 Mask:255.255.255.0
    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:100
    RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)

    linux-2m7c:/home/raj/tipctun # route -n
    Kernel IP routing table
    Destination Gateway Genmask Flags Metric Ref Use Iface
    0.0.0.0 155.53.129.254 0.0.0.0 UG 0 0 0 eth0
    10.1.1.0 0.0.0.0 255.255.255.0 U 0 0 0 tun77
    127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo
    155.53.128.0 0.0.0.0 255.255.254.0 U 0 0 0 eth0
    169.254.0.0 0.0.0.0 255.255.0.0 U 0 0 0 eth0

    - *waldner* says:
      July 23, 2012 at 23:13

      It's almost impossible to tell without seeing the output of "ip addr" or "ifconfig" (and "ip route") on both sides,

without knowing how you created the devices, whether there's a firewall, etc. etc.

45. *deepak gunjal* says:
July 19, 2012 at 06:03

Hi,

I am facing a problem using a TUN interface. I am creating a TUN interface tun77 which is a permanent interface tun77 and attaching to it with flags IFF_TUN|IFF_NO_PI. Then i have opened a UDP socket and sending a UDP frame which is being received at the TUN interface. Now i have a valid IP packet containing the UDP frame which i am writing it to the TUN fd available. But when i am writing this packet it is not received at the UDP socket though the socket is listening at the UDP port available in the IP packet which i am sending.

Is that i am doing some wrong?

- *waldner* says:
July 20, 2012 at 12:37

I don't understand. Are you writing back the same UDP packet you receive to the same TUN descriptor? What's the point of doing that?

46. *Ken* says:
July 6, 2012 at 04:37

Hi waldner,

great tutorial, this article really helps me, thanks.

but i'm kind of confused about the tunnel layout created in simpletun tarball:
at the end of this article, it says, "Finally, it's worth noting that if the tunnel connection is over TCP, we can have a situation where we're running the so-called "tcp over tcp"; ",
which implies that the tunnel described above isn't an over-TCP tunnel, right?

but the socket is created as TCP mode, as in "sock_fd = socket(AF_INET, SOCK_STREAM, 0)", so isn't this an over-TCP tunnel?

thanks

PS. i finally suffer from the Animal Captcha cause English is not my mother language.

- *waldner* says:
July 6, 2012 at 09:03

What it means is that the simpletun endpoints are connected over TCP, but of course what they do is just relaying back and forth data coming from the programs that use the "VPN". In turn, these programs can be exchanging data using whatever protocol, in particular they could be using TCP, leading to the "TCP over TCP" scenario (the "inner" TCP used by the programs, over the "outer" TCP used by simpletun). They could also be using UDP or ICMP or other protocols, in which case TCP over TCP does not happen.

47. *Soumen* says:
June 6, 2012 at 11:45

Thanks waldner for this excellent tutorial.

I just have a question.

If we try to create an alias interface using the code given above
(i.e. if we have created "if1" interface and now try to create
"if1:1" using the same procedure) will the fds returned by ioctl
be same for these two interfaces?

48. *Gregory Nietsky* says:
April 17, 2012 at 11:20

Hi there taking this concept to the next stage i have a taploop daemon
that clones and replaces it with a tap device i want to run it as a multi threaded daemon
with a management socket to control it.

ill be able to support VLAN's on the TAP dev where there is no vlan switch on the
network and segregate it based on mac address also enable disable traffic based on time
throughput constraints as with internet cafes chillispot / coova is another good example of
tun used in a captive portal.

to support tap only VLAN's ill be creating tap devices and when the MAC/VID matches
traffic will be written to this tap directly without 802.1Q.

it uses uthash lists from uthash.sourceforge.net

http://pbx.distrotech.co.za/svn/netfilter_session/taploop.c

49. *Erik* says:
March 17, 2012 at 19:24

Hello,

in case of a TAP-Device without IFF_NO_PI, what are the flags/protocoll are for?
The flags are always 0x0000 and the protocol is the same as in the Ethernetframe (offset 12).

How is it possible to set an other MTU?
In case of a TAP-Device, the MTU means the payload of the ethernet frame and the ethernet header (with or without VLAN-Tag) are additional data, correct?
That means my buffer must be at least MTU + 14/18 (without/with VLAN-Tag), correct?

Thanks for your answer.
Erik

- *waldner* says:
March 17, 2012 at 23:54

Hi Erik,

indeed the packet info data seems to be not so useful. The flag is always either 0, or set to TUN_PKT_STRIP (defined to be 1) if the packet will be stripped (that is, if the userspace call read() passes a too small value for the packet length argument), and as you saw the protocol field is set to the same value that is in the skb.

Regarding the MTU: simplifying a bit, the MTU normally indicates the maximum size of the layer-3 payload, that is, an IP packet (including the IP header). If you use a tap device, you are correct that you need a buffer that is 14/18 bytes larger because in that case the buffer will contain a full ethernet frame (IP packet **plus** the ethernet header).

The method to change the MTU is not specific to tun/tap, it's the same for any interface. From the command line, you can use the appropriate option to the **ip** command, for example

```
ip link set dev tap20 mtu 1600
```

From a C program, you have to use the appropriate ioctl() call, with the SIOCSIFMTU flag (man 7 netdevice on Linux, or google will also find you all the details).

  - *Erik* says:
March 19, 2012 at 10:16

Hello,

Thank You for your answer.

I use now a line with:
ifr.ifr_flags = IFF_TAP | IFF_NO_PI;
and it works perfectly for my needs.

I hope the TAP-Device give no ethernet frames that are bigger than the MTU + Ethernet-Header. In the case of an error i think i should be able to detect it by comparing the IP-Packet-Size with the used space in receive buffer (return value of read()).

The setting of the MTU with the SIOCSIFMTU does not work, the return value of ioctl() is always -1.
The same problem is with setting the TAP-Device up, at the moment i do this manually in an additional console with "ifconfig tap0 up" but this is not practicable for the final program.
Do you have any idea?
Goggle has no help for me.

Regards,
Thanks a lot for your help

    - *waldner* says:
March 20, 2012 at 11:56

Hi Erik,

a quick google search finds this sample program which performs most of the operations you describe:

http://linuxgazette.net/149/misc/melinte/udptun.c. Look at the open_tun_iface() function. Here is the associated article with some explanations: http://linuxgazette.net/149/melinte.html

See if that helps.

      - *Erik* says:
March 21, 2012 at 09:40

Hi Waldner,

Yes this helps.
There is a trick: one must open a normal socket (UDP in this case) and use the ioctl syscall with this

socket-file-descriptor and not with the file-descriptor of the TAP-Device but with the ifreq-variable of the TAP-Device. After finish the configuration of the TAP-Device the socket can closed. This trick should be a little bit more highlighted in the description, in my first reading of the source code i do not have seen it.

Now seems my TAP-Device working properly.
Thousand and One Thanks for Your Help.

Regards
Erik

50. *Nicandro* says:
January 30, 2012 at 14:57

Yes I know how tcpdump works :)
As you suggested it is good just for specific case, I want to be able to select traffic for instance even based on the packet length. That s why I need iptables (mangle / nat), because I have more options in order to split the traffic in more interfaces.

Why do I want to do that? Because when too much traffic is coming, tcpdump may be not able to manage it all. So, by launching more processes of it, with different traffic to monitor, less traffic is lost by the kernel.

tcpdump -i tun1
tcpdump -i tun2
(..)
tcpdump -i tunN

I can open more processes on more cores.

So in my opinion what I should do is to split the traffic in more interfaces. In such a way, I can use the same interfaces for other applications (i.e. snort)

What I wanted to know from you is:
having known that the traffic is coming from and going to the real interface eth0, I want to send a copy of it (selected by filter of iptables) to tun1, tun2, .. tunN.

I saw option of ip route 2 (tee, --gw, ..) but they dont work. DO you know easier way to suggest?

Sorry for disturbing,
Thanks a lot for your help

- *waldner* says:
February 3, 2012 at 08:42

I don't know if there's an easy way to do what you want. Perhaps using the TEE target of iptables, but it's just a wild guess.
To be honest, if your traffic is coming from a SPAN port or equivalent (ie, not destined to the machine where tcpdump is running), I think iptables wouldn't even see it.

51. *Nicandro* says:
January 26, 2012 at 11:22

Hello, I have a problem of routing traffic on two virtual interfaces I have created on my machine (CentOs6)

By using tunctl I created two virtual interfaces tap1 and tap2

let s imagine I gave them two different address

tap1: 10.1.1.1 net 255.255.255.0

tap2: 10.1.2.1 net 255.255.255.0

I m receiving traffic on my real interface eth0: 192.168.1.23 net 255.255.255.0

I tried by using brctl and iptables to send some traffic to tap1 and others to tap2.. unfortunately i m not able to get how to do it.

An example of splitting may be.. let s send all the icmp packets to tap2 and the others on tap1. Is it possible somehow? Can you help me with some instructions?

Regards

- *waldner* says:
January 27, 2012 at 12:42

Not sure what you're trying to do with brctl...anyway, this is a routing issue and has nothing to do with tun/tap interfaces.
I think, if I understand you correctly, that you need policy routing rules to route traffic differently based on protocol. In your specific example of tap1 and tap2, you could create a second routing table where traffic is routed

out tap2, then mark ICMP traffic with iptables, and finally add a routing rule which uses the alternate routing table for marked packets. You can find some theory at http://linux-ip.net/html/routing-tables.html and some examples at http://linux-ip.net/html/adv-multi-internet.html and generally googling for "linux policy routing" should turn up something.

- *Nicandro* says:
  January 27, 2012 at 14:21

  mmm what I would like to do is to monitor the traffic with tcpdump for example.
  and when I wanna do that, for sure I need to give them an interface, isnt ?
  Ok, when you check some info into the traffic it s better to reduce it somehow just selecting the one you more need to check. In this way the process is less stressed overall when it receives high rate traffic.
  Because of that I would like to route traffic incoming from eth0 into two virtual interfaces, the place where I attach a monitoring software, as tshark, tcpdump, etc.
  What do you think?

  - *waldner* says:
    January 27, 2012 at 17:48

    Sorry, I don't understand what you're asking. If you're using tcpdump, you can specify a specific interface with -i or the special interface "any" which captures traffic on all interfaces (but not in promiscuous mode).
    If you want to capture only a certain type of traffic, you can specify filters to tcpdump, for example `tcpdump -i eth0 icmp` will capture only ICMP traffic, or `tcpdump -i eth0 tcp port 80` will capture (hopefully) only HTTP traffic, etc. The manual page for **tcpdump**, or **pcap-filter**, provides all the details on the syntax to use for filtering.
    Hope this answers your question.

52. *jselbie* says:
    November 13, 2011 at 05:03

    Thank you for this writeup. It was most helpful.

53. *saha* says:
    October 29, 2011 at 16:13

    Hi waldner,

    Thank you for this tutorial. Let me first describe what I am trying to achieve. I'm capturing packets from pcap, decapsulating my already encapsulated packets and then pushing them back to tap interface. These packets are ethernet packets, with proper header and checksum. The IP address in these packets are same as that of the tap interface. So, I'm expecting that tap will forward the packet up in the layer. These packets do show up in tshark. But the tap interface drops the packets and does not forward them up in the network stack. From your conversation with Irek, it seems that tap considers these packets as outgoing packets and sends to wire, instead of pushing them up in the network stack. Is there a way that I can push the packets up in the layer using tap interface?

    - *waldner* says:
      October 29, 2011 at 22:20

      I'm not sure I understand what you're trying to do. What is your goal?

54. *Euton* says:
    October 27, 2011 at 16:18

    I have a question regarding this tutorial and android. I want to create a tun device on a android tablet and send the layer 2 data to a java program. Do you have a suggestion of how to interface to the tun device from an android app.

    - *waldner* says:
      October 28, 2011 at 13:45

      Sorry no, I don't have any suggestion. I suppose that, since Android is Linux (well, kind of sort of), you may be able to connect to the tun interface as explained in the article or use existing tools, but I have not tried so this is all guessing.

55. *Dan* says:
    August 10, 2011 at 17:03

    Ok, nevermind, I think I was just leaving out the IFF_NO_PI flag because now it seems to be working. Thanks for you help.

56. *Dan* says:
    August 10, 2011 at 15:56

    I've been trying to write ethernet frames to the kernel using a TAP interface, but when I create the tap interface, no /dev/tapX is ever created. If I create one using "mknod /dev/tap0 c 36 16" I get an error message saying "no such device or address". What's going on? How do I insert ethernet frames directly in to the kernel? Where and how should the /dev/tapX device get created because it is not happening at all for me?

    - *waldner* says:

August 10, 2011 at 16:01

I'm not sure where you got the idea that /dev/tapX should be created. No device files exist under linux representing network interfaces, so it's perfectly normal that you don't see a /dev/tap0 device (in the same way that you don't see /dev/eth0, or whatever).

- *Dan* says:
  August 10, 2011 at 16:14

  Well, I got it from the tun/tap txt file that describes how there are two userland application interfaces:
  " - /dev/tunX - character device;
  - tunX - virtual Point-to-Point interface.

  Userland application can write IP frame to /dev/tunX
  and kernel will receive this frame from tunX interface.
  In the same time every frame that kernel writes to tunX
  interface can be read by userland application from /dev/tunX
  device."

  I'm trying to write data to the kernel. Is there a way to do that? It seemed like from the txt file that there are two ways to write to it from a user level application, but I can only find one way (the tunX virtual point-to-point interface, which if I write to it, it goes "out on the wire" and not to the kernel).

  Thanks for your response. I would appreciate any help you could provide.

  - *waldner* says:
    August 10, 2011 at 16:24

    It looks like you're looking at an old version of the documentation (ie for 2.4.x), because in more recent kernels that file does not mention /dev/tunX, see for example the document at http://www.kernel.org/doc/Documentation/networking/tuntap.txt.
    Anyway, to write data to the kernel (although it's not much clear what you mean with that; you're probably trying to do something else, which you don't explain): briefly, as explained in the tutorial, you have to open /dev/net/tun to get a file descriptor which can then be used to send/receive packets to/from the kernel. It's all explained in the article, including sample C code.

    - *Dan* says:
      August 10, 2011 at 16:35

      Ah, I see, I was looking at an old version. I guess the problem I'm having is that when I open the file descriptor as described in the article and write some ethernet frames to it (using a tap interface), the kernel doesn't respond to the frames, like if the ethernet frames are ARP frames asking for the MAC address of the IP assigned to the tap0 interface, if I write them to tap0, I can see them with tcpdump, but the kernel does not send ARP replies back to my user application, like I would expect...

57. *Juergen* says:
    June 28, 2011 at 17:56

    Hello,

    you need also to replace the send() and recv() parts.
    Here is an example, which uses IPv6 multicasts:

```
int make_socket (uint16_t port)
{
  int sock;
  struct sockaddr_in6 name;

  memset(&name,0,sizeof(name));

  /* Create the socket. */
  sock = socket (PF_INET6, SOCK_DGRAM, 0);
  if (sock < 0)
    {
      ERROR_OUTPUT("socket failed: %s",strerror(errno));
      exit (EXIT_FAILURE);
    }

  /* Give the socket a name. */
  name.sin6_family = AF_INET6;
  name.sin6_port = htons (port);
  if (bind (sock, (struct sockaddr *) &name, sizeof (name)) < 0)
    {
      perror ("bind");
      exit (EXIT_FAILURE);
    }

  return sock;
}

[...]
char* remote_ip= "FF02::1";
struct sockaddr_in6 destination_socket;

  tunnel_fd = make_socket(port);
  if(tunnel_fd < 0) exit(EXIT_FAILURE);
```

```
  // initialize the destination socket, which is used to simulate
  // the send path of the lower layer
  memset(&destination_socket,0,sizeof(destination_socket));
  destination_socket.sin6_family = AF_INET6;
  retval = inet_pton(AF_INET6,remote_ip,&destination_socket.sin6_addr);
  destination_socket.sin6_port = htons(port);
  if(netdevicename) {
    // set the IPv6 scope id, if the user has selected a netdevice
    struct ifreq netdevice;
    strncpy(netdevice.ifr_name,netdevicename,IFNAMSIZ);
    // read the interface index
    retval=ioctl(tunnel_fd,SIOCGIFINDEX,&netdevice);
    if(retval < 0) {
      ERROR_OUTPUT("Failed to read the interface index of %s: %s",
                   netdevicename,strerror(errno));
      exit(EXIT_FAILURE);
    }
    destination_socket.sin6_scope_id = netdevice.ifr_ifindex;
  }

[...]
      retval = sendto(tunnel_fd,telegram,
                      telegram_len,0,
                      (struct sockaddr*) &destination_socket,
                      sizeof(destination_socket));
      if(retval < 0) {
        ERROR_OUTPUT("sendto failed: %s",strerror(errno));
        break; // leave the while loop
      }
[...]
      rx_telegram_len=recvfrom(tunnel_fd,rx_telegram,
                                       sizeof(rx_telegram),0,
                                       (struct sockaddr *) &name, &size);
      if(rx_telegram_len < 0) {
        ERROR_OUTPUT("Failed to read datram from lower layer: %s",
                     strerror(errno));
        break; // leave the while loop
      }
```

If you want to use IPv4, then create the socket as you have already posted. But you need also an IPv4 destination socket.

Greetings
Juergen

- *Juergen* says:
  June 28, 2011 at 18:02

  ups,
  the above was meant as a reply to the post of DoDo.

58. *Tarokkk* says:
    June 19, 2011 at 14:49

    Hi! I tested this thing on my PC (Ubuntu 11.04) and worked well. It created a tun interface and logging the incoming packages but when I tried on my router (OpenWRT - Backfire-rc4) nothing happend. I tried to figure it out, but seems that ping -I tun0 send the data, but the program is waiting at nread(). Is anybody have an idea why this isn'T workin?

    - *waldner* says:
      June 19, 2011 at 17:23

      Are you pinging a non-local IP address (ie, one that would cause the kernel to actually send the data out, as opposed to replying directly)?

      - *Tarokkk* says:
        June 19, 2011 at 18:07

        No I'm pinging non loopback address... (I know unix like skip them to send)

        root@OpenWrt:~# ping -I tun0 192.168.2.22
        PING 192.168.2.22 (192.168.2.22): 56 data bytes
        ^C
        --- 192.168.2.22 ping statistics ---
        13 packets transmitted, 0 packets received, 100% packet loss

        ./tuntap
        Waiting for data in

        ifconfig:

        tun0 Link encap:Ethernet HWaddr 56:F3:A8:84:D3:42
        inet addr:192.168.2.2 Bcast:192.168.2.255 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:1 carrier:0
        collisions:0 txqueuelen:500
        RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

- *waldner* says:
  June 19, 2011 at 18:13

  Another guess...firewall rules blocking outgoing packets?

  - *Tarokkk* says:
    June 19, 2011 at 18:31

    This is a test router with no rules...

    iptables -L
    Chain INPUT (policy ACCEPT)
    target prot opt source destination

    Chain FORWARD (policy ACCEPT)
    target prot opt source destination

    Chain OUTPUT (policy ACCEPT)
    target prot opt source destination

    - *Tarokkk* says:
      June 19, 2011 at 19:07

      Another strange issue, that if I create the tun interface with openvpn I can't connect to it. I read similar error up in comments but in a different situation...

      openvpn --mktun --dev tun0 --dev-node /dev/net/tun --user root

      ifconfig tun0 192.168.2.2 netmask 255.255.255.0

      root@OpenWrt:/tmp# ifconfig tun0
      tun0 Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
      inet addr:192.168.2.2 P-t-P:192.168.2.2 Mask:255.255.255.0
      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:100
      RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

      The error:
      ioctl(TUNSETIFF): Invalid argument
      Error connecting to tun/tap interface tun0!

      - *higginse* says:
        October 21, 2011 at 00:22

        I've noticed it doesn't seem to work with kernel 2.6.38-11
        (This worked with 2.6.32-34 - I saw traffic via tshark)

        host$ sudo openvpn --mktun --dev tun3 --user myself
        Thu Oct 20 22:02:50 2011 TUN/TAP device tun3 opened
        Thu Oct 20 22:02:50 2011 Persist state set to: ON
        host$ sudo ip link set tun3 up
        host$ sudo ip addr add 10.0.0.1/24 dev tun3
        host$ ifconfig tun3
        tun3 Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
        inet addr:10.0.0.1 P-t-P:10.0.0.1 Mask:255.255.255.0
        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:100
        RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

        host$ ping 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
        From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
        From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
        From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
        --- 10.0.0.2 ping statistics ---
        5 packets transmitted, 0 received, +3 errors, 100%packet loss, time 4018ms

        - *waldner* says:
          October 21, 2011 at 08:36

          I'm not sure what you are expecting that to do. If you don't connect any program to the tun interface to catch outgoing traffic, packets will be lost.

        - *higginse* says:
          October 21, 2011 at 09:34

Apologies, I didn't state my observation clearly ...
When running the above (which basically matches the example you provided)
the behaviour is different depending on kernel versions.

under kernel 2.6.32
When you start pinging 10.0.0.2 you will see packets on tshark -i tun3

under kernel 2.6.38
when pinging 10.0.0.2 *nothing* appears on tshark -i tun3

Behaviour under 2.6.38 is different, for me at least:)
Is that what should happen.

(I should also thank you for the very useful tutorial btw)

- *waldner* says:
  October 22, 2011 at 17:05

  Thanks, you're correct (I wasn't aware of this). It seems that behavior was changed between 2.6.35 and 2.6.36, specifically by this patch. Basically, earlier a tun device was always up and running from the moment it was created, while now it needs to see some process attached to the special fd to become up (ie, to get the "carrier"). If no process is attached, the test you are running (and which I too run in the article) now fails because the interface is down and packets are dropped and not "transmitted". I've put a note in the article to point out that what is described there works only with kernels < 2.6.36.
  I understand the patch does the right thing, as having a process attached to the tun fd is the equivalent of having the "link up" for a tun interface; however, for the purposes of the article, this is a bit of a loss because the simple test described there to show how the interface works cannot be done anymore.
  Thanks!

- *waldner* says:
  June 19, 2011 at 19:10

  Then I'm out of ideas. Make sure you're checking the result of every system call in the code, there may be a failure somewhere.

  - *Tarokkk* says:
    October 12, 2011 at 11:40

    The invalid flag argument was a mistake (I had different version of this flag in the compiler and in the kernel)But still not working on the router :S

59. *DoDo* says:
    June 9, 2011 at 13:06

    Hello all,

    I'm Trying to let this simple tun work on UDP.
    But I don't get it working.

    What I did is to comment out the listen and accept part of TCP because I want to use UDP.
    I also changed the sock fd to UDP style:
    code:

    ```
    if ( (sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("socket()");
    exit(1);
    }
    ```

    I'm not familiar with the tap interface of Linux so the while loop is a bit fuzzy for me.
    When i compile and start the client and server i see the creation of a UDP socket of 55555 with an established note by netstat.
    Howeven when i send data from the client to the server i see the debug with the amount of bytes to the tap interface and the amount of bytes to the network, this data is not coming to the server.
    When i initiate a ping from the server to the client (not is debug mode) i see text popping up on the screen, this text is the
    datafield of a ICMP packet that i try to send to the client side of the tunnel.
    I'm doing something stupid but i just can't figure it out.....

    Thanks in advance!

    If somebody has an example that would also be very handy!

    Thanks

DoDo

- *waldner* says:
  June 13, 2011 at 16:13

  I have to look into this, as I've never tried to implement the connection over UDP. I'm a bit busy ATM, but I hope I'll be able to look into it soon.

60. *Adrian* says:
    May 17, 2011 at 05:14

    Hi Waldner,

    Thank you for this detailed tutorial :)

    I am trying to create a tunnel from my linux server to my windows client. The problem is I am getting an invalid packet once I write it to the windows tun driver. Wiresharks says that it has a "Bogus IP header length (0, must be at least 20)" I believe it has something to do with. IFF_NO_PI flag. Is the 4 byte tun header needed when writing a data to tun?

    I already done some research and found out that the Win32 tap driver doesn't prepend the 4 byte tun header. Is there anything I should do first to the read packet from win32 tap driver before writing it to a linux tun driver?

    - *waldner* says:
      May 17, 2011 at 18:35

      I'm afraid I have no experience with tun/tap under Windows. Have you tried to use IFF_NO_PI at both sides?

      - *Adrian* says:
        May 23, 2011 at 03:39

        Thanks the problem was solved with setting IFF_NO_PI on the linux server.

        I still had one problem. I already created my server and client that uses udp. the problem is sometimes the data that is sent from client to server is received out of order. I already expected that to happen because of the nature of UDP. I still continued with my UDP server/client thinking that would be fine since I also expected that once the data is written to the tun driver the TCP layer would handle the error correction. And it did, it actually worked but the upload speed of the client is so slow (1-2kb/s) while the download speed is averaging at 1-5mbps.

        Could that be the effect of the retransmission of packets? Do you have any ideas on how to make UDP more reliable?

61. *Irek* says:
    April 21, 2011 at 22:36

    Hi Waldner,

    Thank you for the tutorial. It's very useful.

    I'm having the following problem, and I'm hoping you could help me out. I'm creating a bridge, and then add to it two tap interfaces. No physical interface is added to the bridge. These are the commands:

```
brctl addbr test
ip tuntap add mode tap tap0
ip tuntap add mode tap tap1
ifconfig test up
ifconfig tap0 up
ifconfig tap1 up
brctl addif test tap0
brctl addif test tap1
```

    The problem is that the bridge doesn't seem to work correctly. I sent through tap0 some broadcast frames (WOL frames), and they didn't reach tap1. I was sending packets with:

```
etherwake -b -i tap0 00:00:00:00:00:00
```

    The tshark command for tap0 showed the frames being sent with tap0, but another tshark for tap1 didn't show them.

    Then I added to tap0 the IP address 192.168.10.1/24, and did:

```
arping 192.168.1.2
```

    I saw ARP request broadcast frames on tap0, but they didn't reach tap1.

    This is the output of ifconfig for test, tap0, and tap1 interfaces:

```
root@computer:~# ifconfig test
test Link encap:Ethernet HWaddr 02:07:b1:eb:2c:2a
inet6 addr: fe80::944b:d9ff:fe10:b240/64 Scope:Link
UP BROADCAST RUNNING PROMISC MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
```

```
TX packets:19 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:3377 (3.2 KiB)

root@computer:~# ifconfig tap0
tap0 Link encap:Ethernet HWaddr 02:07:b1:eb:2c:2a
inet addr:192.168.10.0 Bcast:192.168.10.255 Mask:255.255.255.0
inet6 addr: fe80::7:b1ff:feeb:2c2a/64 Scope:Link
UP BROADCAST RUNNING PROMISC MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:278 overruns:0 carrier:0
collisions:0 txqueuelen:500
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

root@computer:~# ifconfig tap1
tap1 Link encap:Ethernet HWaddr b2:ee:2c:f9:d5:0d
inet6 addr: fe80::b0ee:2cff:fef9:d50d/64 Scope:Link
UP BROADCAST RUNNING PROMISC MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:19 overruns:0 carrier:0
collisions:0 txqueuelen:500
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

The output of the brctl:

```
root@computer:~# brctl show
bridge name bridge id STP enabled interfaces
pan0 8000.000000000000 no
test 8000.0207b1eb2c2a no tap0
tap1
root@computer:~# brctl showmacs test
port no mac addr is local? ageing timer
1 02:07:b1:eb:2c:2a yes 0.00
2 b2:ee:2c:f9:d5:0d yes 0.00
```

The output of route:

```
root@computer:/home/iszczesniak# route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
192.168.2.0 0.0.0.0 255.255.255.0 U 2 0 0 eth1
192.168.10.0 0.0.0.0 255.255.255.0 U 0 0 0 tap0
169.254.0.0 0.0.0.0 255.255.0.0 U 1000 0 0 eth1
0.0.0.0 192.168.2.1 0.0.0.0 UG 0 0 0 eth1
```

What am I doing wrong?

Thanks,
Irek

- *waldner* says:
  April 24, 2011 at 16:26

  Hi Irek,

  I think you've got it backwards. The WOL frame is sent to tap0, which means it goes out "on the wire". There, you should have a program attached to the tap descriptor that catches the frame and does something with it. If there is no such program, the frame is dropped. It would be no different if you added, say, eth0 to the bridge and ran "etherwake -i eth0": the frame would be sent out the network card onto the LAN, and would not appear in the bridge. **Incoming** frames, on the other hand, appear in the bridge as appropriate.

  So in other words, for your frame to show on tap1, you have to set up things so that the WOL frame is incoming to the bridge; which means, for example, connect a VM to tap0, and generate the WOL frame *from the VM*. This way, the bridge will see the wOL frame incoming from the tap0 "port", and will broadcast it (or whatever) as appropriate. Also, using "etherwake -i test" (where "test" is the bridge name) should work too.

  - *Irek* says:
    April 24, 2011 at 19:59

    Thanks for your response, Waldner.

    In my original case when in which nothing is connected to the tap0, why is the wire leaving tap0 going to the application that should be connected to the tap0 interface, and not to the bridge? It seems like tap0 has two wires going different ways. So is the configuration something like this?

    bridge
    | ^
    v |
    tap0 |
    intf |
    | |
    v |
    application

    The configuration above seems like a good bet, because when the application is a VM, and when I send frames from the VM to the MAC address of tap0, the frames are received by the "test" bridge.

You are right: when I connect a VM to tap0 and send broadcast frame from the VM, I can see it on tap1. I get it: the "wire" is connected to the NIC on the VM.

You are also right that when I send broadcast frames to my "test" bridge, the frames get to both tap0 and tap1. But why? Interfaces "tap0" and "test" look the same as reported by ifconfig, and yet they are different.

Moreover, why is the MAC address of the bridge the same as the MAC address of the interface last added to it?

Why does a bridge have a MAC address at all? It shouldn't have an address! After all, hardware switches don't have MAC addresses.

- *waldner* says:
  April 24, 2011 at 21:17

  Yes, the situation is how you depict it in the ASCII diagram. When you do "etherwake -i tap0", the frame goes to the application at the bottom. Remember that the bottom part of the diagram represent the "wire" to the kernel. As I said, it would work exactly the same if you did "etherwake -i eth0", except in that case the wire is a real wire (the LAN cable).

  Also, imagine that you had no bridge; doing "etherwake -i tap0" in that situation, and doing the same when tap0 is part of a bridge, must work the same way in both cases (same for eth0, etc). Adding the interface to a bridge should not generally change its semantics.

  Strictly speaking, a bridge does not need to have a MAC address (in fact, I believe most low-end, cheap unmanaged bridges and switches have no MAC address). However, that way your linux box would just sit there moving frames and nothing else, strictly acting as a bridge, not differently from a 15€ switch, which would be a bit of a waste. If you want to use the machine for something else, you want to assign an IP address to the bridge interface, so you will be able to receive and send local traffic in addition to the bridged traffic. Having an IP address, means you also need to have a layer 2 address (ie, a MAC address here). Under Linux, the bridge interface automatically takes the lowest MAC address of all the enslaved interfaces (see this article for more information on the implications of this).

  - *Irek* says:
    April 24, 2011 at 22:19

    What I like is a minimal design, and a bridge doesn't need a MAC address. I understand that a Linux box might offer more than a regular switch, and for that you need a MAC address. But the services should be provided by a new tap interface added to the bridge. I believe that the bridge should not even be shown by ifconfig.

    That's interesting that the bridge takes the lowest MAC address of the bridge interfaces. I wonder why this is needed.

    - *waldner* says:
      April 24, 2011 at 22:35

      Well in that case I think you may want to look into something else, as (as far as I am aware) I don't think Linux offers a way to change the way it currently works (short of modifying the kernel code, of course).

      - *Irek* says:
        April 24, 2011 at 22:45

        Waldner, thanks again for the tutorial and the comments. I think I need to experiment with the bridges more.

62. *sfseeley* says:
    April 14, 2011 at 20:14

    I am having UDP packet loss on the client host somewhere between client application -> client host kernel -> client-host-tun device. These packets are missing from my application that reads from the tun. My tun is configured like so:

    tun1 Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
    inet addr:192.168.13.5 P-t-P:192.168.13.5 Mask:255.255.0.0
    UP POINTOPOINT RUNNING NOARP MULTICAST MTU:6500 Metric:1
    RX packets:8617887 errors:0 dropped:0 overruns:0 frame:0
    TX packets:6127019 errors:0 dropped:20 overruns:0 carrier:0
    collisions:0 txqueuelen:100
    RX bytes:551235232 (525.6 MiB) TX bytes:391798708 (373.6 MiB)

    I am messing with the MTU and txqueuelen options.

    My client application sends a burst of 64 byte UDP packets followed by a sync packet. I am using the sync packet to throttle the amount of outstanding packets in the previous burst. So in effect (or so it seems) I do not think I am over running any queues. I will send a burst of 20 packets, but only read say 12 or 15 (random amount each time) from the tun1 device. Where are they getting dropped?

    I looked at /proc/net/snmp and I can see that all my expected UDP packets are logged here (no errors). Then I look at

the "TX packets and dropped" counts from ifconfig. Usually I do not see any increase in ifconfig's reported dropped, but the TX packet counts do not match that found in /proc/net/snmp.

It seems the UDP packets are dropped at the tun, but not always logged as dropped..

Any ideas?

Thanks,
Steve

- *waldner* says:
  April 17, 2011 at 22:48

  Perhaps it's not related at all, but this line:

  inet addr:192.168.13.5 P-t-P:192.168.13.5

  are you sure it's the way it's supposed to be?

  - *sfseeley* says:
    April 18, 2011 at 16:06

    Thanks for the response Waldner.

    Very interesting. I am not entirely sure about this point-to-point config to be honest. What I am trying to do is configure the tun device to accept any packets headed to 192.168.x.x. For this I created tun using the famous tunctl like this:

    tunctl -u sseeley -n -t tun1
    ifconfig tun1 192.168.13.5/16 txqueuelen 50

    This creates this funny p-t-p link to itself.

    So I just tried:

    ifconfig tun1 192.168.13.5/16 txqueuelen 50 pointopoint 192.168.13.13

    I get the following:

    tun1 Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
    inet addr:192.168.13.5 P-t-P:192.168.13.13 Mask:255.255.0.0
    UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1500 Metric:1
    RX packets:64996 errors:0 dropped:0 overruns:0 frame:0
    TX packets:370163 errors:0 dropped:59 overruns:0 carrier:0
    collisions:0 txqueuelen:50
    RX bytes:2169920 (2.0 MiB) TX bytes:39235816 (37.4 MiB)

    But I still get random amounts of UDP packet loss. I've tried larger MTUs too.

    I expected to be able to configure a tun device to accept this multicast of destination addresses. It seem to allow this since I can send udp packets to many listing servers on different hosts at the other end of my tunnel.. but I get this random UDP packet loss before it even arrives at my origination tun device. I even just tried no multicast (netmask == 255.255.255.255) and it is still the same.

    I feel I am missing something fundamental. I do not think I should be getting UDP packet loss on the client host before packets arrive at tun. Am I wrong to think this?

    Thanks,
    Steve

    - *waldner* says:
      April 24, 2011 at 16:40

      Alright, it seems that the ifconfig output is misleading.
      Ok, so how do you generate the traffic? For packets to be sent "out" the tun1 interface, they should have a destination IP address in the range 192.168.0.0/16 (except, of course, 192.168.13.5 itself).
      Alternatively, the host should have a route to whatever destination address is in the packet, and that route should point to the tun1 interface.
      That way, if you attach an application to the descriptor corresponding to tun1, that application should receive the packets that the kernel routes out tun1.
      So, how does your application connect to the tun1 interface descriptor, and how do you actually read the packets? Also, how does the application-level program generate the UDP datagrams?

63. *Wutiphong* says:
    March 8, 2011 at 20:12

    Very good totorial. Thank you

64. *Rob* says:

February 17, 2011 at 16:33

sorry please disregard the which NIC part.. need to engage brain before keyboard

65. *Rob* says:
February 17, 2011 at 16:26

Hi Waldner, I wonder if you know if Ethernet Frame CRC is handled by the NIC rather than the TAP interface? and if there's a way of specifying which NIC the TAP will attach to as I have multiple cards & want to just see & forward traffic from eth1 not eth0.

Best Regards

Rob

- *waldner* says:
February 17, 2011 at 18:13

  Hi Rob,

  regarding the CRC: it seems it's not part of the frame you get when read()ing the tap descriptor. It's also easy to verify with a simple test (eg, ping): a standard ping under Linux gives you 98 bytes of data from the tap descriptor; of these, 84 are the IP+ICMP stuff, and you have 14 bytes left which must be the ethernet data. Since the source and destination MAC are 6 bytes each, the type/length is another 2 bytes, 6+6+2=14 and thus there's nothing else.

  Regarding traffic: that tap interface doesn't attach to any NIC; it is an interface on its own. You attach to it to read its traffic. What you get when reading the tap interface is the traffic that the kernel decides has to be routed through the tap interface. So, you have to manage this at the kernel/routing level (iptables, iproute2). You want that the kernel only send the traffic you're interested in to the tap interface.

  - *Rob* says:
  February 24, 2011 at 11:57

    Hi Waldner, I've been using the Tap interface for a while now but seem to have hit a little speed bump.
    My Userland API which sends out packets uses 32bit Int values so I adjusted the Tap Read to read 32bit integers.. For some reason when I do a read I am always 8 bytes short in the value returned for the number of bytes read. TCP Dump on another machine shows this so I never get replys to my ICMP ping.. If I add 8 bytes to number of bytes read, the ping packet is valid so the data is there & it all works properly. However.. this is a hack & want to understand what's going on here.
    I am assuming this is an issue with the Linux Read System call & 32bit integers.. have you seen this before? I could keep the rx buffer as char buffer but casting gets messy in C++ since the API takes a structure & a 32bit int pointer to data.

    - *waldner* says:
    February 24, 2011 at 13:49

      Sorry I'm not sure I follow you here. What does it mean "I adjusted the Tap Read to read 32bit integers"? The read() on the tap fd should read an entire packet or frame every time it's called. Not sure how or where 32 or 64-bit integers come in here.

      - *Rob* says:
      February 24, 2011 at 13:55

        What I mean is that originally I had declared the buffer uint8_t buffer[_MAX_MTU_SIZE]; and then called

        bytes_read = read(_this->_tap_file_descriptor,
        _this->_buffer,
        sizeof(buffer);

        but due to the API I need to call requiring packets to be a 32 but integer pointer & a length I adjusted the code so I'm using

        uint32_t buffer[(_MAX_MTU_SIZE/4)+1];

        bytes_read = read(_this->_tap_file_descriptor,
        _this->_buffer,
        sizeof(buffer);

        the read command uses a void* for the buffer so I was assuming I could use a uint32_t buffer..

        but on a standard ping I am missing 8 bytes

        on a ping -s 512
        IP truncated-ip - 456 bytes missing! 192.168.0.77 > 192.168.0.76: ICMP echo request, id 2796, seq 557, length 520

        - *waldner* says:

February 24, 2011 at 14:07

Sorry for being dense. The only API I see you calling there is read(), and read() wants a void * pointer, so it doesn't matter whether it points to an array of n 1-byte elements or an array of n/4 4-byte elements. What's the real reason you're doing that?
Also you're using _this->_buffer but the name of the array is just "buffer" with no underscore. Is that meant to be the same variable?

- *Rob* says:
  February 24, 2011 at 14:14

  that's what I thought...

  the reason is that when I make an Api Call the packet is a structure based on

  int32_t* data;
  int32_t length
  ..etc..etc..
  so in C++ it saves trying to reinterpret cast a 32bit pointer onto an 8 bit Array although I can do that & it does work.. if the buffer is int32_t I can just say
  buffer->data = &_test_buffer[0];
  and be done with it

  - *Darrell* says:
    February 24, 2011 at 15:25

    Rob,

    I think you are running into an alignment problem, and it sounds like you are on a 64-bit machine. This would explain the missing 8 bytes.

    You are better off using dynamic memory allocation, as the memory pointer will be properly aligned:

    int32_t* data;
    data = (int32_t *)malloc(_MAX_MTU_SIZE);

66. *Rob* says:
    February 12, 2011 at 20:42

    Hi Waldner,
    This is a great tutorial given the rather sketchy txt file that comes with the Tap Tun interface and has helped me get a tap interface up and running. I am trying to forward Ethernet frames to an external API which cannot see the linux protocol stack. I am nearly there but wondered if I am missing something. When I read in ethernet frames they do not seem to conform to spec. I see no pre-amble. I am assuming the interface is filtering that out but yet the source and dest mac addresses seem to be in the wrong place. As "read" is a blocking call, when a packet arrives at my tap interface which is set-up as instructed, I wait for the bytes read... for example a simple ICMP ping. I see the correct number of bytes.. I dump the packet buffer to a txt file and expect to at least see a length field at the specified place.. I can see the TCP header starting with the version but it seems to be in the wrong place. Am I missing something. Is there something I am seriously overlooking here? any help would be really appreciated as I am loosing sleep over this.

    Best Regards

    Rob

    - *waldner* says:
      February 13, 2011 at 10:53

      Hi Rob,

      Difficult to tell without seeing the real thing, but in my experience, ethernet frames you read from the tap interface do not have a preamble, if by that you mean the "10101010" etc. pattern used at the physical level in real ethernet networks to synchronize with the start of the frame. The first byte of the frame you read from the tap interface is the first byte of the destination MAC, and the other fields follow.
      Also if the frames you're reading are coming from the external application, make sure that they are correctly formed, that they are of type ethernet II/DIX, and also watch out for added fields like VLAN tags etc. Ah and of course if you can sniff the traffic with Wireshark or tcpdump, that will also be a great help. Finally, make sure that fields longer than one byte are using the correct endianness.

      - *Rob* says:
        February 15, 2011 at 17:11

        Hi Waldner, I worked out that the IFF_NO_PI Flag was not set. I was getting confused since the source MAC address was always 33:33:00:00:00:16 .. but worked out quickly what that was. My biggest problem is now wondering about the read function.. The problem I have is that I am trying to forward all ethernet traffic to an external API of a programmable device. Older Ethernet frames are easy since they have a length field but Ethernet II packets have a type field. I am assuming because the actually "read" command gets executed in Linux kernel space , that the kernel does not interrupt in the middle of a read, so I can safely assume when

the read function returns say 200 as the number of bytes it has read, I can safely assume this is the entire frame.. Or can I? If I can't they are dangerous assumptions and the whole thing can fall over. But apart from that the interface is working

- *waldner* says:
  February 15, 2011 at 19:23

  Glad you sorted it. I didn't think of the NO_PI flag in my previous reply, but yes, that's something that would give you extra stuff at the beginning, well spotted.

  Regarding read(), although I'm not 100% sure, I'm reasonably confident that when reading tun/tap devices you read a whole packet/frame at a time. It's not written anywhere explicitly, but I've never seen otherwise. This is consistent with write() (you must write a whole packet/frame at once otherwise the packet is discarded because it's invalid) and it's also sensible because otherwise you would be forced to peek inside what you've read to determine whether what you got is a complete packet/frame or only a part of it, which would rapidly become complicated to handle correctly.

  Update: it also seems that other programs that use tun/tap devices make the same assumption.

67. *Darrell* says:
    January 28, 2011 at 23:23

    I'm using Debian Squeeze btw

68. *Darrell* says:
    January 28, 2011 at 16:41

    I've had no trouble opening the "/dev/net/tun" device, and creating the tap device. I don't persist the interface, since I only want the interface running when my program is running, and I'm using libevent to process the network packets (as opposed to using select). My purpose for this is basically to take a machine with two network ports (i.e. eth0, eth1), and virtualize them. By setting the physical ports up in promiscuous mode, and clearing all routes and IP addresses (ifconfig flush ethX), and instead assigning those routes and IP addresses to the tapX devices, then I have a situation where I can read every packet coming in the physical interface (using RAW sockets), write them to the TAP interface, and Vice versa. This way, the kernel will drop all the packets from the physical interfaces (after I get them in the raw socket), and I can firewall/view/modify all packets coming in and out of the system in user space. This is just background info, so you can understand where I'm coming from. I wonder if there are better, more standard ways of doing this, but that's not my real question here.

    I use the following code to create my tap interface:

```
int tuntap_init(char *dev, int tun_or_tap) {
struct ifreq ifr;
int tuntap_fd, err;
char *tundev = "/dev/net/tun";

if((tuntap_fd = open(tundev, O_RDWR)) < 0) {
perror("opening /dev/net/tun");
return tuntap_fd;
}

memset(&ifr, 0, sizeof(ifr));

if(tun_or_tap) { //TAP
ifr.ifr_flags = IFF_TAP | IFF_NO_PI;
}
else { //TUN
ifr.ifr_flags = IFF_TUN;
}

/*
If a desired name is given, try that one
otherwise, a default name will be assigned
*/
if(*dev) {
strncpy(ifr.ifr_name, dev, IFNAMSIZ);
}

/*
Setup TUN/TAP device
*/
if((err = ioctl(tuntap_fd, TUNSETIFF, (void *)&ifr)) < 0) {
perror("ioctl(TUNSETIFF)");
return err;
}

}
```

So, the first time I run this, it works great, the tap device is created, I can set up the interfaces the way I want. The problem I have is that once I exit this program that created the tap device (ctrl-c or kill), I cannot ever get it to run again successfully without rebooting. After killing the program, the tap device does disappear from the network devices (as seen by ifconfig) as expected. Running the program a second time, however, generates the following error (note, this program creates tap0 and tap1 devices):

sudo ./taptest -1 eth0 -2 eth1
ioctl(TUNSETIFF): Invalid argument

Once I reboot, this problem goes away, and it works again 1 time. I've tried removing the tun kernel module and re-adding it (rmmod tun...modprobe tun), without any success.

What am I doing wrong?

Thanks!

- *waldner* says:
  January 29, 2011 at 16:32

  Your code is a bit odd. How are you using tuntap_init(), and in particular, its return value? As written, after the function has been called, the program has no way to read/write data from/to the tun interface because no file descriptor is returned. Although it's not certain that this is related to your problem, I would start by adding a

  return tuntap_fd;

  at the end of your code (and making use of it in the caller).

  Then, you may want to copy the interface name into the provided string, so the caller can see which name was chosen by the kernel (if it was asked to do so):

  strcpy(dev, ifr.ifr_name);

  The caller must reserve enough space in the buffer pointed to by "dev".

  Also, depending on what you do afterwards, you may want to close(tuntap_fd) if the ioctl() returns error.

  - *Darrell* says:
    January 30, 2011 at 05:14

    Sorry, somehow the rest of my code was truncated:

    /*
    If the system assigns a different name,
    copy the name back to the dev name buffer
    */
    strcpy(dev, ifr.ifr_name);

    return tuntap_fd;
    }

    So - a little more investigation shows that iproute2 also has the same error:
    #ip tuntap add tap0 mode tap
    ioctl(TUNSETIFF): Invalid argument

    Finally, I was able to get the program working again if:
    ifconfig tap0 down
    ifconfig tap1 down
    rmmod -f tun
    modprobe tun

    Then it works again...

    I discovered that I can create the tap devices using iproute2, and then connect to them later on in my program. Then, I simply leave the tap devices created always and I they persist after my program exits. Would this be a better way of doing this?

    Thanks.

    - *waldner* says:
      January 30, 2011 at 10:55

      Yes, if you have a version of iproute2 recent enough, you can create interfaces with it. "Better" is subjective anyway, and depends on what you need to do. If you plan to use existing tools to connect to the tap interface, then it's probably quicker and easier to also use existing tools to create it (eg iproute2, tunctl, openvpn can all do it), so you don't have to bother with writing code yourself. If instead you need to do something special or specific to your task that cannot be accomplished with existing tools, then of course writing your own code (C or otherwise) is the way to go. Another reason for writing your own code is to learn and understand better how things work.

69. *Andrew Woods* says:
January 20, 2011 at 16:41

This is a noob question, but can you not use sysfs instead of ioctl to write to the tun/tap devices?

- *waldner* says:
January 20, 2011 at 22:32

Strictly speaking, to write you use write(); ioctl() is used to set certain operating parameters on the interface. As far as I know, you cannot set those parameters via sysfs/procfs, but new information is always welcome.

- *waldner* says:
January 20, 2011 at 22:50

It seems I spoke too soon. There are indeed a number of sysfs entries for a tun/tap interface (and for any interface, for that matter). For example:

```
# ls -l /sys/class/net/tap100/
total 0
-r--r--r-- 1 root root 4096 Jan 20 22:16 addr_assign_type
-r--r--r-- 1 root root 4096 Jan 20 22:16 address
-r--r--r-- 1 root root 4096 Jan 20 22:16 addr_len
-r--r--r-- 1 root root 4096 Jan 20 17:13 broadcast
-r--r--r-- 1 root root 4096 Jan 20 22:16 carrier
-r--r--r-- 1 root root 4096 Jan 20 22:16 dev_id
-r--r--r-- 1 root root 4096 Jan 20 22:16 dormant
-r--r--r-- 1 root root 4096 Jan 20 22:16 duplex
-r--r--r-- 1 root root 4096 Jan 20 22:16 features
-rw-r--r-- 1 root root 4096 Jan 20 22:16 flags
-r--r--r-- 1 root root 4096 Jan 20 22:16 group
-rw-r--r-- 1 root root 4096 Jan 20 22:16 ifalias
-r--r--r-- 1 root root 4096 Jan 20 22:16 ifindex
-r--r--r-- 1 root root 4096 Jan 20 22:16 iflink
-r--r--r-- 1 root root 4096 Jan 20 22:16 link_mode
-rw-r--r-- 1 root root 4096 Jan 20 22:16 mtu
-r--r--r-- 1 root root 4096 Jan 20 22:16 operstate
-r--r--r-- 1 root root 4096 Jan 20 22:16 owner
drwxr-xr-x 2 root root    0 Jan 20 22:16 power
-r--r--r-- 1 root root 4096 Jan 20 22:16 speed
drwxr-xr-x 2 root root    0 Jan 20 22:16 statistics
lrwxrwxrwx 1 root root    0 Jan 20 17:13 subsystem -> ../../../../class/net
-r--r--r-- 1 root root 4096 Jan 20 22:16 tun_flags
-rw-r--r-- 1 root root 4096 Jan 20 22:16 tx_queue_len
-r--r--r-- 1 root root 4096 Jan 20 22:16 type
-rw-r--r-- 1 root root 4096 Jan 20 17:16 uevent
```

however only few of them are writable. Some of those can also be set using iproute2 (mtu, tx_queue_len). So I think it can still be said that ioctl() are needed.

70. *Andrew Woods* says:
January 10, 2011 at 12:00

Very nice tutorial - Thanks

71. *stefan* says:
January 9, 2011 at 12:52

Thank you, this is a great tutorial.

I came accross it while trying to understand kvm networking. Helps a lot.

72. *Actionscript* says:
December 15, 2010 at 06:52

Could you advice I got nothing from

char buffer[];

nread = read(tun_fd,buffer,sizeof(buffer));

printf("Read %d bytes from device %s\n", nread, tun_name);

printf("Buffer Length: %d", sizeof(buffer));
printf("DATA: %s\n", buffer);

73. *Actionscript* says:
December 14, 2010 at 11:36

I am a newbie, when I run the code copy from simpletun.c, get this error, can I have some hints? Thanks.

if((err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0) {
close(fd);

printf("error: %s \n", strerror(errno));
return err;

error: Operation not permitted

- *waldner* says:
  December 14, 2010 at 17:57

  To run that code, you have to either be root, or the interface must exist already and owned by the user you're running as.

  - *Actionscript* says:
    December 14, 2010 at 18:05

    I got that once posted the comment, thanks!

74. *dgen* says:
    October 30, 2010 at 20:37

    I have a tun which I use to forward packets to some other nodes (there is not a tunnel among them). The packets that send through the tun interface are received from the service but I cannot received the responses in the tun. Do you have any solution ?

    - *waldner* says:
      October 31, 2010 at 10:24

      You provide very little information, so it's difficult to tell. Check that the node running the service has a route back to the tun node. Also, what does it mean "there is not a tunnel"?

75. *Dan* says:
    September 7, 2010 at 22:24

    Hi, Waldner,

    This is an excellent tutorial!

    I have a question about using tap interfaces. I'm trying to set up a tap interface on a host, where one end will be the network interface for a virtual machine. On the other side of the interface, I want to read all the outgoing VM traffic and send it to another host (or hosts) which also have VM's and taps running.

    I've had limited success so far: I successfully capture the outgoing VM network traffic. However, it seems as though the host side of the tap interface is handling the traffic as well, and will even respond to the VM via the tap interface. I don't want it to do this! How do I disable the host's network stack for the tap interface?

    Thanks,
    Dan

    - *waldner* says:
      September 8, 2010 at 10:26

      Hi Dan,

      how did you set it up? You didn't provide much detail, so there are just wild guesses:

      Many virtualization platforms like KVM/virt-manager set up things so the communication between the VM and the host is via a tap interface. In those cases, the virtualization code attaches to the tap interface to relay the packets from/to the VM, so the host sees them as incoming/outgoing on the tap interface. If you're also trying to attach to the tap interface at the same time, I wouldn't be surprised (although I haven't tried) to see a race condition between your code and the virtualization code, where you step on each other's toes and end up reading only part of the data each, depending on who issues the read() call first. In this situation, packets that are not caught by your code are obviously caught by the virtualization code, which sends them to the host where they are processed.

      If your code is the only entity attached to the interface and there is no contention, then you are somehow sending the packets to the tap interface thus making them visible to the host (as opposed to blocking them and sending them somewhere else, eg to another host via the network).

      Finally, consider whether what you're trying to do could not be accomplished using the standard available tools (iproute2, iptables) without the need to write ad-hoc code.

      - *waldner* says:
        September 8, 2010 at 20:30

        Ok, I actually tried my first guess above, and if I try to connect to a tap interface that is in use (by a KVM virtual machine) I get **ioctl(TUNSETIFF): Device or resource busy**, so that hypothesis can be ruled out.

        Still, it's difficult to tell what your problem may be without more information.

76. *ted* says:
    September 7, 2010 at 10:48

    On Mac OS X there is a Kernel extensions to create virtual network interfaces: TunTap.

    http://tuntaposx.sourceforge.net

    - *Stef* says:

March 30, 2011 at 17:56

Another nice multipurpose network tool: "socat". It's possible to create the simpletun-test with this tool. socat is a relay for bidirectional data transfer between two independent data channels. It has many options... :-)

http://www.dest-unreach.org/socat/

http://www.dest-unreach.org/socat/doc/socat.html#EXAMPLES

- *waldner* says:
  March 30, 2011 at 18:44

  Yes, in fact I use socat all the time, and setting up a simpletun-like connection is a matter of one line of code on the client and one on the server (with the additional benefit that the server can be made forking so it doesn't terminate when the client disconnects, SSL can be added to the mix etc.). Just as an example with SSL and a self-signed cert (10.100.1.131:4444 is the address and port where the server is listening):

  ```
  server# socat TUN:172.16.44.1/24,iff-up,tun-name=ssl0,tun-type=tun \
  OPENSSL-
  LISTEN:4444,pf=ip4,cipher=HIGH,method=TLSv1,verify=1,cert=selfsign.pem,key=selfsign.key,cafile=selfsign.pem,fork,su=nobody

  client# socat TUN:172.16.44.2/24,iff-up,tun-name=ssl0,tun-type=tun \
  OPENSSL:10.100.1.131:4444,cipher=HIGH,method=TLSv1,verify=1,cert=selfsign.pem,key=selfsign.key,cafile=selfsign.pem,su=nobody
  ```

  That's pretty much it, one may of course want to add routes etc.

  Of course, as much as I like it, using socat this way doesn't give the same insight on the internal workings of a tun/tap interface.

77. *John* says:
    June 7, 2010 at 15:38

    Excellent tutorial.

    Has anyone come across this problem though, all the above works perfectly for me on Debian Etch 2.6.18-6-686, but on Debian Lenny 2.6.31 write always returns -1

    E.g i can open a tap interface fine, do the ioctl without error, I can even read packets from the tap interface, however write always returns -1 errno 22 (invalid argument)

78. *Juergen* says:
    April 16, 2010 at 18:19

    Hello,

    regarding IPv6, there is a difference between Distributions or (kernel) configurations:

    Start simpletun on the first computer:
    ./simpletun -i tun0 -s &
    ifconfig tun0 up
    ifconfig tun0 add fe80::1234/64

    And on the second computer:
    ./simpletun -i tun0 -c computer1 &
    ifconfig tun0 up
    ifconfig tun0 add fe80::5678/64
    ping6 -I tun0 fe80::1234

    I have run this on my 2 Debian PCs. There it worked as expected: The pings were answered through the tun device.
    But then run this also on 2 SUSE PCs. On the SUSE PCs the IPv6 pings were transmitted through the tun device,
    as it could be seen by wireshark. But the kernel has otherwise ignored the IPv6 pings!

    So there must be a difference in kernel configuration between Debian and SUSE, which is related to this.
    Has anyone an idea, what parameter / compile time configuration pevents the kernel from handling IPv6 packets
    received at the tun device (but not on eth)?

    Greetings
    Juergen

    - *waldner* says:
      April 16, 2010 at 20:26

      Hi Juergen,

      I don't have a SUSE system to test. For what it's worth, it works for me on Debian, Gentoo and Arch Linux. The only thing I can think of is to check that there are no firewall rules on the SUSE boxes that block the ICMPv6 packets in either direction (check with **ip6tables-save**).
      I'm assuming that the basic simpletun IPv4 connection is successful? If the other end's firewall is dropping IPv4 packets, it may look like the peers are connected whereas they are not. Try running simpletun with -d to get debug messages and make sure that the two peers are able to connect successfully.

- *Juergen* says:
  April 16, 2010 at 20:50

  Hello Waldner,

  with the -d option and also with wireshark I saw that the IPv6 and the IPv4 packets were transmitted through the tun device tunnel.
  The SUSE kernel responded to the IPv4 pings but not to the IPv6 pings.
  But the SUSE kernel responded to IPv6 pings transmitted through the ethernet device.
  I have also tried with an embedded linux 2.6.15 kernel without firewall. There I saw the same behavior.
  Therefore I don't believe that this is caused by firewall.

  Greetings
  Juergen

  - *waldner* says:
    April 16, 2010 at 21:01

    Hi Juergen,

    ok, fair enough. I'm a bit out of ideas here; just out of curiosity, have you tried to
    1) use tap instead of tun and
    2) use non-link-local addresses (eg something in the 2000::/3 range)?

    - *Juergen* says:
      April 16, 2010 at 21:17

      Hello Waldner,

      1: Not yet
      2: I have tried with an fc00:: prefix.

      Now I have found this old bug for report for freeBSD with a patch for the tun device
      http://www.mail-archive.com/freebsd-net@freebsd.org/msg05969.html

      So I will take a look into the kernel sources.

      Greetings
      Juergen

      - *Juergen* says:
        April 16, 2010 at 21:59

        Hello,

        I have just found in the linux git history that there was a bug in the tun driver:

        commit f271b2cc78f09c93ccd00a2056d3237134bf994c
        Author: Max Krasnyansky
        Date: Mon Jul 14 22:18:19 2008 -0700

        tun: Fix/rewrite packet filtering logic

        Please see the following thread to get some context on this
        http://marc.info/?l=linux-netdev&m=121564433018903&w=2

        Basically the issue is that current multi-cast filtering stuff in
        the TUN/TAP driver is seriously broken.
        Original patch went in without proper review and ACK. It was broken and
        confusing to start with and subsequent patches broke it completely.
        To give you an idea of what's broken here are some of the issues:

        - Very confusing comments throughout the code that imply that the
        character device is a network interface in its own right, and that packets
        are passed between the two nics. Which is completely wrong.

        - Wrong set of ioctls is used for setting up filters. They look like
        shortcuts for manipulating state of the tun/tap network interface but
        in reality manipulate the state of the TX filter.

        - ioctls that were originally used for setting address of the the TX filter
        got "fixed" and now set the address of the network interface itself. Which
        made filter totaly useless.

        - Filtering is done too late. Instead of filtering early on, to avoid
        unnecessary wakeups, filtering is done in the read() call.

        The list goes on and on :)

So I have to replace the kernel.

Greetings
Juergen

- *Matthew* says:
  October 12, 2011 at 02:56

  Juergen,

  I posted a message describing a similar problem, but wanted to reply directly in hopes this reaches you. Posted below:

  "I'm using an embedded build of kernel 2.6.35. My application sits between the tun and a serial port, mostly just passing IPv6 packets back and forth. When I run a ping6, I see the echo reply and write it to the tun; the kernel increments bytes rx'd on the tun etc., however ping6 *usually* doesn't get the data. I say usually, because there is one exception: the first ping6 works; all subsequent pings fail. Anyone seen this, or have a guess as to what the issue would be?

  Juergen - Were you able to resolve the issue you were seeing? "

  Thanks in advance - Matthew

  - *Juergen* says:
    October 12, 2011 at 17:45

    Hello Matthew,

    yes I have solved the problem by porting back the bugfix to the older kernel.
    But kernel 2.6.35 should already contain this bugfix.

    I guess that your problem could be related to the configuration of the tun/tap device
    -> verify with ifconfig
    Or there goes something wrong related to IPv6 neighborhoud discovery or something related.
    -> verify with tcpdump

    Greetings
    Juergen

79. *saime* says:
    April 14, 2010 at 22:46

    Hi Waldner,

    Jpcap also looks to me the better option even not to read from c file but directly reading packets from network or a tun-interface. Nevertheless direct from a network will be better.

    I was wondering about something, how can I write exactly bytes what I read from tun-interface. I mean as a matter of fact ping packet should be 84 bytes but entire buffer with 1500 characters is filled. Should it not be that only 84 char of the buffer should be filled?

    Second problem I have with named pipe. but I dont if you know about it as java code is keep on reading though I have stopped c program to write into pipe. (it is kind of weird for me because as far as I know it should be a FIFO)

    Thanks in advance and thanx for Jpcap library info.

    - *waldner* says:
      April 15, 2010 at 19:46

      Hi Saime,

      I'm not sure what you mean "reading directly from a network". Isn't that what you do when you use Wireshark or your own pcap/Jpcap program to sniff traffic passing through the interface? That should do what you want.
      As for the buffer filling, if you're talking about writing packets to the pcap output file, as documented in the Wireshark wiki page you only have to write the actual packet bytes in the file, with no padding; but you should also write the correct packet length in the packet header (again refer to that page for the details). However, if you use a library like Jpcap, it should have functions that automatically take care of writing the data to the output file in the correct format; for Jpcap, see for example the **writePacket** method in the **JpcapWriter** class. See also the good tutorial on the Jpcap web site.

80. *saime* says:
    April 13, 2010 at 18:39

    Hi Waldner,

    I was wondering about the format of the bytes.

Now I am able to setup tun, collect the data via C program and send it to the named pipe and retrieve it from java code at the other end of the pipe. But I was wondering about the format. Do I need to change the format? and besides that, how can I represent this data in the way that the wireshark or other network analyzers display, Any hing?

thanks in advance.
br,
Saime

- *waldner* says:
  April 13, 2010 at 23:07

  Hi Saime,

  Wireshark wants its data in **pcap** format. This is *not* just a raw dump of the data; it contains additional information. The format is documented for example here in the Wireshark wiki.
  Of course, a pcap file is also what you get by default if you run Wireshark directly on the interface while traffic is flowing, and then save it (File -> Save...).
  In case you want to write the pcap file yourself, you can use libpcap, which is the low-level library upon which tcpdump, Wireshark and other network analyzers are built. There are some Java wrappers to use libpcap from Java: see for example here and here. Both have tutorial and examples, so you should be able to write your own Java code to capture the packets and save them in pcap format.

- *Euton* says:
  December 8, 2011 at 23:32

  I am trying to do something similar. Can you describe how you were able to create a tun and retrieve data to java code.

  - *waldner* says:
    December 9, 2011 at 09:26

    I didn't. The code in the article is C code. To work with tun devices with java, you may check out these urls:

    http://www.bmsi.com/java/posix/index.html To use POSIX system calls with Java
    http://p2pvpn.org/ A VPN using tun/tap devices written in Java
    http://www.koders.com/java/fid6C0CBC76450F649DE9081FE8596BB50FEC023D88.aspx Sample Java code from the P2PVPN application.

- *Euton* says:
  December 9, 2011 at 11:10

  I am attempting to use java native code to have java code attach to a tun device that what setup with the above C code. Can you describe how you were able to create a tun and retrieve data to java code.

  Sorry Waldner, the question was meant for Saime.

81. *saime* says:
    March 31, 2010 at 17:30

    Thanx Waldner,

    It works, though I could also corrected error while using #include before I got this reply.

    But solution you proposed is also work perfectly.

    I have one more question, if you can help me with it. Actually I wanna read these packets from java. Is it any possibility to read TUN packets via java. As I can see from tun_alloc call I am getting descriptor but it is C-compatible.

    Other solution for me would be call this function from java via JINI and then write it into one file and then again read it from Java.

    But is there any easier solution.

    Thanks in advance.

    - *waldner* says:
      March 31, 2010 at 18:34

      Hi saime,

      I'm not really able to help you much with Java. I think there are some libraries floating around to access POSIX syscalls from Java (never tried myself, so I can't really speak); for example this http://www.bmsi.com/java/posix/index.html seems to be reasonably recent; others are old and unmaintained, you can still find them if you google a bit.

      If such a library is not good enough for you, then I'm afraid JNI is your best bet.

    - *Stef* says:
      March 30, 2011 at 18:02

Using TAP with Java is indirectly possible, look at the source of this project:

http://p2pvpn.org/

P2PVPN is a Java program and uses BitTorrent-Trackers for finding other VPN-clients (only invited users get the keys to join such a VPN). Cool idea, never tested this program.

82. *saime* says:
March 31, 2010 at 13:07

Dear Waldner!

I am facing following problem while compiling.

/usr/include/linux/if.h:165: error: field 'ifru_addr' has incomplete type
/usr/include/linux/if.h:166: error: field 'ifru_dstaddr' has incomplete type
/usr/include/linux/if.h:167: error: field 'ifru_broadaddr' has incomplete type
/usr/include/linux/if.h:168: error: field 'ifru_netmask' has incomplete type
/usr/include/linux/if.h:169: error: field 'ifru_hwaddr' has incomplete type

Please let me know if you know the error, and how can I get rid of it.

- *waldner* says:
March 31, 2010 at 14:33

Hi saime,

try replacing

#include <linux/if.h>

with

#include <net/if.h>

and see if that helps.