

**Setting up Linux server as a router
and
Hello packets**

Preparing the environment

For this presentation, we will need a Kernel Virtual Machine (KVM).

This is the full virtualization solution for Linux. I have made this choice, mainly because the project will involve more than one terminal sessions to be accessed. This also makes possible ssh connections from physical machine to virtual machines.

As virtual environment, Ubuntu 15.10 (Willy Werewolf) server has been chosen to be set up as a router, with IP 192.168.122.129

Let's make ssh connection possible.

As you can probably expect, KVM can manage all the virtual machines from command lines as well.

From the core machine (the one on which the KVM is installed - in our case Ubuntu 14.04-), check the status of the virtual machine:

```
root@tron-X200CA:~# virsh list
  Id    Name                           State
-----
  2     Rinzler                         running

root@tron-X200CA:~#
root@tron-X200CA:~#
```

Log into the Virtual Machine, and define a tty file in order to use it as a serial console. You will have to write a script under **/etc/init/**

For this case, **tty0.conf** file has been created, with the following content:

```
#ttyS0 - getty
#
#
#

start on stopped rc RUNLEVEL=[2345]
stop at runlevel [!2345]

respawn
exec /sbin/getty -L 38400 ttyS0 vt102
~
~
~
~
~
```

Now, go back to the core machine and to reload libvirt and restart the virtual machine by using, with sudo rights, the below commands:

```
root@tron-X200CA:~#  
root@tron-X200CA:~# sudo reload libvirt-bin  
root@tron-X200CA:~#  
root@tron-X200CA:~# sudo virsh shutdown Rinzler  
Domain Rinzler is being shutdown  
  
root@tron-X200CA:~#  
root@tron-X200CA:~#  
root@tron-X200CA:~# sudo virsh start Rinzler  
Domain Rinzler started  
  
root@tron-X200CA:~# █
```

You should be able to access the VM via ssh:

```
tron@tron-X200CA:~$ ssh rinzler@192.168.122.129  
The authenticity of host '192.168.122.129 (192.168.122.129)' can't be established.  
ECDSA key fingerprint is ec:8c:16:29:24:87:28:95:61:d8:0c:93:40:eb:5f:e8.  
Are you sure you want to continue connecting (yes/no)? yes  
Warning: Permanently added '192.168.122.129' (ECDSA) to the list of known hosts.  
rinzler@192.168.122.129's password:  
Welcome to Ubuntu 15.10 (GNU/Linux 4.2.0-16-generic i686)  
  
* Documentation:  https://help.ubuntu.com/  
  
New release '16.04 LTS' available.  
Run 'do-release-upgrade' to upgrade to it.  
  
Last login: Mon May  2 02:50:05 2016 from 192.168.122.1  
rinzler@Rinzler:~$  
rinzler@Rinzler:~$ █
```

As a next step, towards the router set-up, I have decided to create a virtual ethernet interface, that will represent the router's interface.

To create this interface work, we must load the dummy kernel module.

```
root@Rinzler:~#  
root@Rinzler:~# lsmod | grep dummy  
root@Rinzler:~#  
root@Rinzler:~# modprobe dummy  
root@Rinzler:~#  
root@Rinzler:~# lsmod | grep dummy  
dummy                16384  0  
root@Rinzler:~#
```

After the driver is loaded, create the dummy network interface:

```
root@Rinzler:~#  
root@Rinzler:~# ip link set name virt0 dev dummy0  
root@Rinzler:~#  
root@Rinzler:~# ifconfig virt0  
virt0      Link encap:Ethernet  HWaddr 7a:5d:48:bc:cb:29  
           BROADCAST NOARP  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:0  
           RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Then, create an alias, and offer an IP:

```
root@Rinzler:~#  
root@Rinzler:~# ip addr add 192.168.122.121/24 brd + dev virt0 label virt0:0  
root@Rinzler:~#  
root@Rinzler:~# ifconfig -a virt0:0  
virt0:0    Link encap:Ethernet  HWaddr 7a:5d:48:bc:cb:29  
           inet addr:192.168.122.121  Bcast:192.168.122.255  Mask:255.255.255.0  
           BROADCAST NOARP  MTU:1500  Metric:1
```

Do a final check:

```
root@Rinzler:~# ip a | grep -w inet  
    inet 127.0.0.1/8 scope host lo  
    inet 192.168.122.129/24 brd 192.168.122.255 scope global ens3  
    inet 192.168.122.121/24 brd 192.168.122.255 scope global virt0:0  
root@Rinzler:~#  
root@Rinzler:~#
```

Don't forget to bring the interface up:

```
root@Rinzler:~#  
root@Rinzler:~#  
root@Rinzler:~# ip link set dev virt0:0 up  
root@Rinzler:~#  
root@Rinzler:~#
```

Set up a Linux server as an ospf router

First, install Quagga: `sudo apt-get install quagga`.

Modify the daemons under `/etc/quagga/daemons` file, to their highest priority.

So, in our case, only ospfd and zebra will be set to "yes"

```
root@Rinzler:/etc/quagga# more daemons  
# This file tells the quagga package which daemons to start.  
#  
# Entries are in the format: <daemon>=(yes|no|priority)  
# 0, "no" = disabled  
# 1, "yes" = highest priority  
# 2 .. 10 = lower priorities  
# Read /usr/share/doc/quagga/README.Debian for details.  
#  
# Sample configurations for these daemons can be found in  
# /usr/share/doc/quagga/examples/.  
#  
# ATTENTION:  
#  
# When activation a daemon at the first time, a config file, even if it is  
# empty, has to be present *and* be owned by the user and group "quagga", else  
# the daemon will not be started by /etc/init.d/quagga. The permissions should  
# be u=rw,g=r,o=.  
# When using "vtysh" such a config file is also needed. It should be owned by  
# group "quaggavty" and set to ug=rw,o= though. Check /etc/pam.d/quagga, too.  
#  
# The watchquagga daemon is always started. Per default in monitoring-only but  
# that can be changed via /etc/quagga/debian.conf.  
#  
zebra=yes  
bgpd=no  
ospfd=yes  
ospf6d=no  
ripd=no  
ripngd=no  
isisd=no  
babeld=no  
root@Rinzler:/etc/quagga#
```

Restart Quagga:

```
root@Rinzler:/etc/quagga#  
root@Rinzler:/etc/quagga# /etc/init.d/quagga restart  
[ ok ] Restarting quagga (via systemctl): quagga.service.  
root@Rinzler:/etc/quagga#
```

Now go to /usr/share/doc/quagga/ and look after **examples** folder.

Change to that directory, and for our case we will be needing only **zebra.conf.sample** and **ospfd.conf.sample**

Copy these two files under /etc/quagga

```
cp /usr/share/doc/quagga/zebra.conf.sample /etc/quagga/  
cp /usr/share/doc/quagga/ospfd.conf.sample /etc/quagga/
```

```
root@Rinzler:/usr/share/doc/quagga/examples# ls -ltr  
total 44  
-rw-r--r-- 1 root root 385 May 6 2015 zebra.conf.sample  
-rw-r--r-- 1 root root 126 May 6 2015 vtysh.conf.sample  
-rw-r--r-- 1 root root 390 May 6 2015 ripngd.conf.sample  
-rw-r--r-- 1 root root 422 May 6 2015 ripd.conf.sample  
-rw-r--r-- 1 root root 992 May 6 2015 pimd.conf.sample  
-rw-r--r-- 1 root root 182 May 6 2015 ospfd.conf.sample  
-rw-r--r-- 1 root root 1110 May 6 2015 ospf6d.conf.sample  
-rw-r--r-- 1 root root 805 May 6 2015 isisd.conf.sample  
-rw-r--r-- 1 root root 2801 May 6 2015 bgpd.conf.sample2  
-rw-r--r-- 1 root root 582 May 6 2015 bgpd.conf.sample  
-rw-r--r-- 1 root root 655 May 6 2015 babeld.conf.sample  
root@Rinzler:/usr/share/doc/quagga/examples# cp zebra.conf.sample /etc/quagga/  
root@Rinzler:/usr/share/doc/quagga/examples# cp ospfd.conf.sample /etc/quagga/  
root@Rinzler:/usr/share/doc/quagga/examples#  
root@Rinzler:/usr/share/doc/quagga/examples# ls /etc/quagga/  
daemons debian.conf ospfd.conf.sample zebra.conf.sample  
root@Rinzler:/usr/share/doc/quagga/examples#
```

Modify the files:

You will have to add in ospfd.conf file the details of the future router

```
root@Rinzler:/etc/quagga# more ospfd.conf  
?  
? Zebra configuration saved from vty  
? 2016/05/02 03:19:13  
?  
hostname ospfd  
password [REDACTED]  
log stdout  
?  
?  
?  
interface virt0:0  
?  
interface lo  
?  
router ospf  
network 192.168.122.1/24 area 0.0.0.0  
?  
line vty  
?  
root@Rinzler:/etc/quagga#
```

Modify the zebra.conf file:

```
root@Rinzler:/etc/quagga#  
root@Rinzler:/etc/quagga# more zebra.conf  
hostname Rinzler  
password [REDACTED]  
enable password [REDACTED]  
!  
interface ens3  
ip address 192.168.122.129/24  
!  
root@Rinzler:/etc/quagga#
```

Change the files permissions:

```
chmod 770 /etc/quagga/zebra.conf  
chmod 770 /etc/quagga/ospfd.conf
```

Restart Quagga again:

```
root@Rinzler:/etc/quagga#  
root@Rinzler:/etc/quagga# /etc/init.d/quagga restart  
[ ok ] Restarting quagga (via systemctl): quagga.service.  
root@Rinzler:/etc/quagga#
```

Activate the IP forwarding, to make possible the transfer of packets between the network interfaces of a Linux system.

Restart quagga again.

```
root@Rinzler:/etc/quagga#  
root@Rinzler:/etc/quagga# sudo su -c "echo 1 > /proc/sys/net/ipv4/ip_forward"  
root@Rinzler:/etc/quagga#  
root@Rinzler:/etc/quagga#  
root@Rinzler:/etc/quagga#  
root@Rinzler:/etc/quagga# /etc/init.d/quagga restart  
[ ok ] Restarting quagga (via systemctl): quagga.service.  
root@Rinzler:/etc/quagga#
```

Check from /etc/services if the following entries are present:

```
root@Rinzler:/etc/quagga#  
root@Rinzler:/etc/quagga# more /etc/services | grep zebra  
zebrasrv      2600/tcp      # zebra service  
zebra         2601/tcp      # zebra vty  
ripd          2602/tcp      # ripd vty (zebra)  
ripngd        2603/tcp      # ripngd vty (zebra)  
ospfd         2604/tcp      # ospfd vty (zebra)  
bgpd          2605/tcp      # bgpd vty (zebra)  
ospf6d        2606/tcp      # ospf6d vty (zebra)  
isisd         2608/tcp      # ISISd vty (zebra)  
root@Rinzler:/etc/quagga#  
root@Rinzler:/etc/quagga#
```


Verify that the daemons are running:

```
root@Rinzler:~# netstat -vantu | grep 2604
tcp        0      0 127.0.0.1:2604      0.0.0.0:*           LISTEN
root@Rinzler:~#
root@Rinzler:~#
root@Rinzler:~# netstat -vantu | grep 2601
tcp        0      0 127.0.0.1:2601      0.0.0.0:*           LISTEN
root@Rinzler:~#
root@Rinzler:~#
```

To configure OSPF routing process, connect via telnet:
(Make sure telnet is enabled.)

```
root@Rinzler:~#
root@Rinzler:~#
root@Rinzler:~#
root@Rinzler:~#
root@Rinzler:~# telnet localhost ospfd
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^J'.

Hello, this is Quagga (version 0.99.24.1).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

User Access Verification

Password:
ospf>
ospf> enable
ospf# conf terminal
ospf(config)# hostname Rinzler
Rinzler(config)#
Rinzler(config)# interface virt0:0
Rinzler(config-if)#
Rinzler(config-if)# ip
    ospf OSPF interface commands
Rinzler(config-if)# ip ospf
    authentication      Enable authentication on this interface
```

...and we have a router, now!

Briefly, on OSPF

OSPF gathers link state information from available routers and constructs a topology map of the network. The topology determines the routing table presented to the Internet Layer which makes routing decisions based solely on the destination IP address found in IP packets. OSPF detects changes in the topology, such as link failures, very quickly and converges on a new loop-free routing structure within seconds. It computes the shortest path tree for each route using a method based on Dijkstra's algorithm - a shortest path first algorithm.

OSPF encapsulates its routing messages directly on top of IP as its own protocol type. TCP connections are not used.

OSPF packets on short

Few things on IP Multicast

A typical multicast on an Ethernet network, using the TCP/IP protocol, consists of two parts: Hardware/Ethernet multicast and IP Multicast.

With IP multicasting the hardware multicasting MAC address is mapped to an IP Address. Once Layer 2 (Datalink) picks the multicast packet from the network (because it recognises it, as the destination MAC address is a multicast) it will strip the MAC addresses off and send the rest to the above layer, which is the Network Layer. At that point, the Network Layer needs to be able to understand it's dealing with a multicast, so the IP address is set in a way that allows the computer to see it as a multicast datagram.

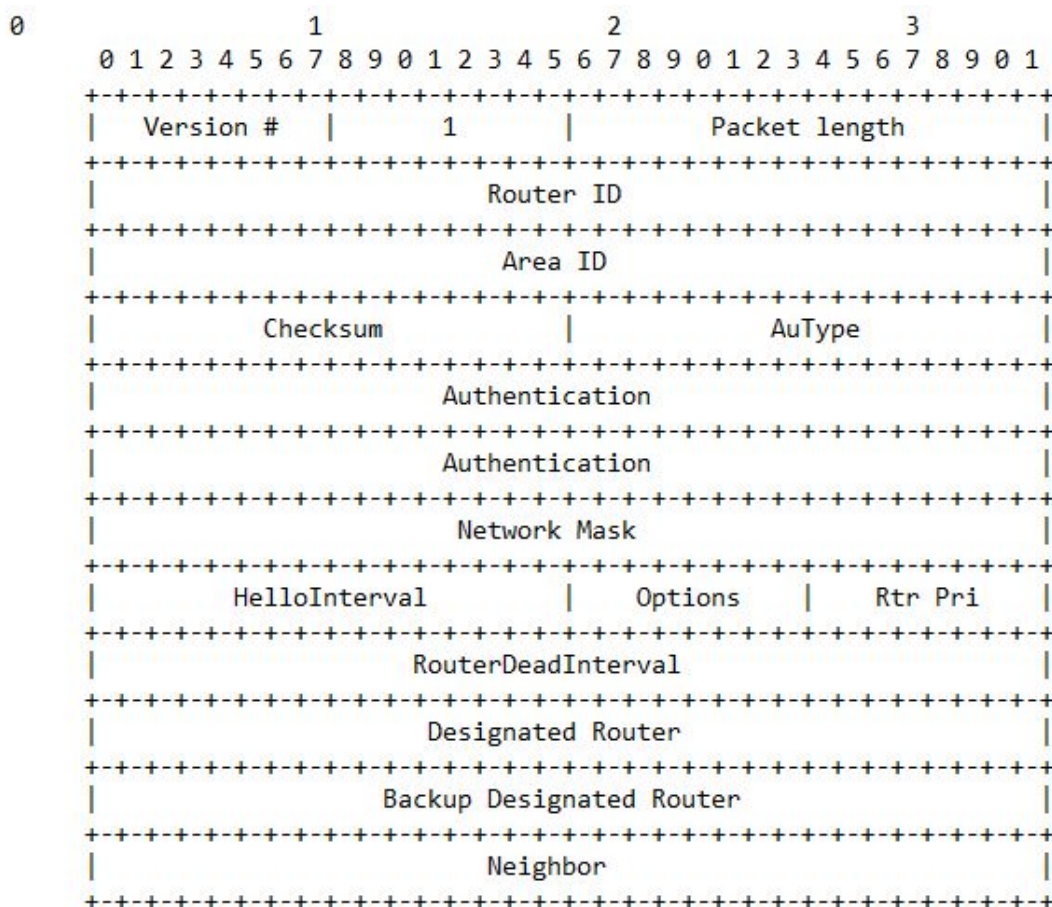
Multicasts are used a lot between routers so they can discover each other on an IP network.

*An Open Shortest Path First (OSPF) router sends a "hello" packet to other OSPF routers on the network. The OSPF router must send this "hello" packet to an assigned multicast address, which is **224.0.0.5**, and the other routers will respond.*

Briefly, on OSPF Hello packets

Hello packet (OSPF Type 1 packet) is used for neighbors discovery, and sent periodically to a multicast group to enable dynamic discovery of nighboring routers.

All routers connected to a common network must agree on certain parameters (Network mask, HelloInterval and RouterDeadInterval). These parameters are included in Hello packets, so that differences can inhibit the forming of neighbor relationships.



Sending OSPF Hello packets

Now that we have our routing process up and running, let's see what we can do from here.

Let's check the traffic of router's interface virt0:0

```
root@Rinzler:/etc/quagga# tcpdump -i virt0:0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on virt0:0, link-type EN10MB (Ethernet), capture size 262144 bytes
22:01:17.170098 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
22:01:27.171286 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
22:01:37.172490 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
22:01:47.173008 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
^C
4 packets captured
4 packets received by filter
0 packets dropped by kernel
root@Rinzler:/etc/quagga#
```

As you can see, we have captured four packets - the router's virt0 interface has 192.168.122.121 IP address and is sending out Hello packet onto multicast address for all adjacent routers.

Now, let's try to "sneak in" few tiny crafted packets and compare results.

```
root@Rinzler:/home/rinzler#
root@Rinzler:/home/rinzler#
root@Rinzler:/home/rinzler# ./packet_raw virt0:0 192.168.122.121 ; date
-----packet sent-----
Mon May  2 23:14:04 EEST 2016
root@Rinzler:/home/rinzler# ./packet_raw virt0:0 1.1.1.1 ; date
-----packet sent-----
Mon May  2 23:14:24 EEST 2016
root@Rinzler:/home/rinzler#
root@Rinzler:/etc/quagga#
root@Rinzler:/etc/quagga# tcpdump -i virt0:0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on virt0:0, link-type EN10MB (Ethernet), capture size 262144 bytes
23:13:42.707590 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
23:13:52.708429 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
23:14:02.708969 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
23:14:04.128028 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 24
23:14:12.709343 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
23:14:22.709968 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
23:14:24.605250 IP 1.1.1.1 > ospf-all.mcast.net: OSPFv2, Hello, length 24
23:14:32.710291 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
23:14:42.711258 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
```

So, in first try, the router's virt0 interface with IP 192.168.122.121 is sending out a Hello packet onto Multicast address (**ospf-all.mcast.net**) for all adjacent routers.

In second try, the router's virt0 interface with IP 1.1.1.1 is sending out a Hello packet onto Multicast address for all adjacent routers.

As well mentioned in the IP Multicasting, in OSPF, the routers send a "hello" packet to an assigned multicast address, which is **224.0.0.5**

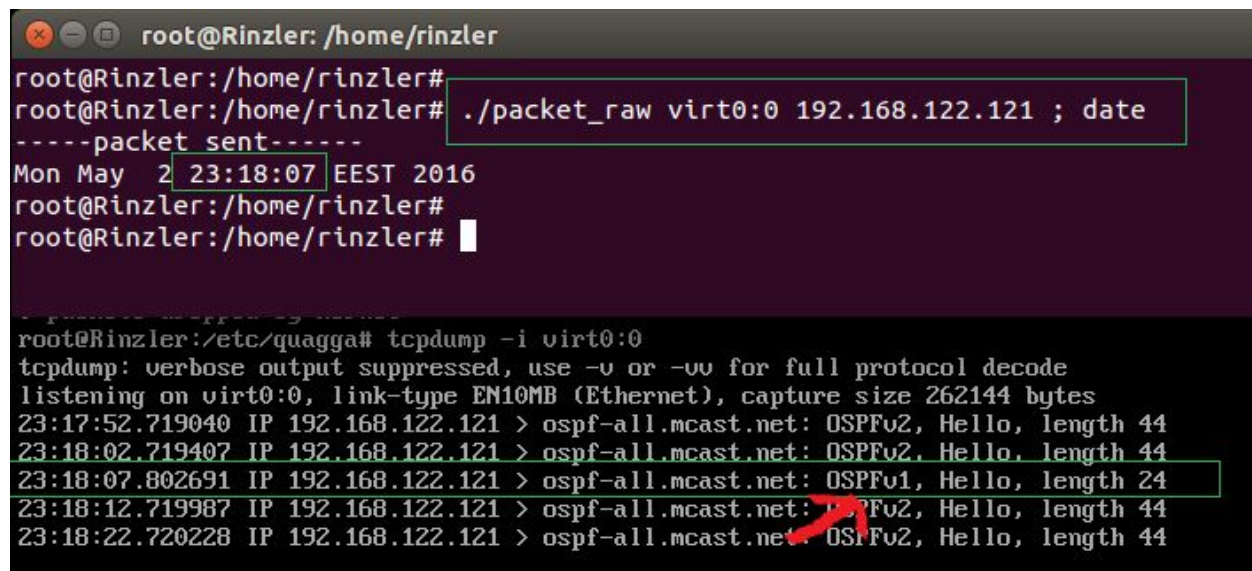
In our case, **ospf-all.mcast.net** is exactly **224.0.0.5**

This is implemented in our code at the packet structure:

```
p.ip.daddr      = inet_addr("0xE0000005");
```

All these details are captured with tcpdump tool, as you can see in the screenshot.

Now, let's change a bit the code, and obtain a packet of OSPF version 1.



```
root@Rinzler: /home/rinzler
root@Rinzler:/home/rinzler#
root@Rinzler:/home/rinzler# ./packet_raw virt0:0 192.168.122.121 ; date
-----packet sent-----
Mon May  2 23:18:07 EEST 2016
root@Rinzler:/home/rinzler#
root@Rinzler:/home/rinzler#

root@Rinzler:/etc/quagga# tcpdump -i virt0:0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on virt0:0, link-type EN10MB (Ethernet), capture size 262144 bytes
23:17:52.719040 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
23:18:02.719407 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
23:18:07.802691 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv1, Hello, length 24
23:18:12.719987 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
23:18:22.720228 IP 192.168.122.121 > ospf-all.mcast.net: OSPFv2, Hello, length 44
```

So, basically, just a very small modification had to be done

from:

```
BYTE ospf[24]={ 0x02,\n                0x01,\n                *****snip*****}
```

to:

```
BYTE ospf[24]={ 0x01,\n                0x01,\n                *****snip*****}
```

...and a bit about the program

The concept is based on raw packets on device level.

Packet sockets are used to receive or send raw packets at the device driver (OSI Layer 2) level. They allow the user to implement protocol modules in user space on top of the physical layer.

Structure **sockaddr_ll** will be used since it represents a device-independent physical-layer address.

Also, structure **struct ifreq**, for low-level access to Linux network devices

*****packet_raw.c*****

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netinet/ip.h>
#include <netpacket/packet.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <getopt.h>
#include <string.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <malloc.h>
#include <net/ethernet.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <sys/types.h>
#include <time.h>
```

```
#define MAX          300
#define ETH_P_ALL    0x0003
#define ETHSIZE      14
#define IPHSIZE      20
#define TCPSIZE      20
```

```
typedef unsigned char  BYTE;
```

```
enum _Boolean_ { FALSE = 0, TRUE = 1 };
```

```
int create_socket(char *device)
```

```
{
```

```

int packet_socket;

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

struct sockaddr_ll rawsll;
memset(&rawsll, 0, sizeof(rawsll));

packet_socket = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));

if(packet_socket == 0)
{
    printf("ERR: socket creation for device: %s\n", device);
    return FALSE;
}

strncpy(ifr.ifr_name, device, sizeof(ifr.ifr_name));

if(ioctl(packet_socket, SIOCGIFINDEX, &ifr) == -1)
{
    printf(" ERR: ioctl failed for device: %s\n", device);
    return FALSE;
}

    rawsll.sll_family      = AF_PACKET;
    rawsll.sll_ifindex     = ifr.ifr_ifindex;
    rawsll.sll_protocol    = htons(ETH_P_ALL);

    if(bind(packet_socket, (struct sockaddr *) &rawsll, sizeof(sll)) ==
-1)
    {
        printf("ERR: bind failed for device: %s\n", device);
        return FALSE;
    }

return packet_socket;
}

```



```

                                0x00, \
                                0x00, \
                                0x00, \
                                0x01, \
                                0xb9, \
                                0xf8, \
                                0x00, \
                                0x01, \
                                0x63, \
                                0x69, \
                                0x73, \
                                0x63, \
                                0x6f, \
                                0x00, \
                                0xb9, \
                                0x00
};
BYTE 12[14] = { 0x01, \
                0x00, \
                0x5e, \
                0x00, \
                0x00, \
                0x05, \
                0xc0, \
                0x01, \
                0x0f, \
                0x78, \
                0x00, \
                0x00, \
                0x08, \
                0x00
};

```

```

struct packet p;
memset(&p, 0x0, sizeof(struct packet));

```

```

p.ip.version    = 4;
p.ip.ihl        = IPHSIZE >> 2;
p.ip.tos        = 0;
p.ip.tot_len    = htons(24+IPHSIZE);
p.ip.id         = htons(rand()%65535);
p.ip.frag_off   = 0;

```

```

    p.ip.ttl      = htons(254);
    p.ip.protocol = 0x59;
    p.ip.saddr    = inet_addr(argv[2]);
    p.ip.daddr    = inet_addr("0xE0000005");
    p.ip.check    = (unsigned short)in_cksum((unsigned
short*)&p.ip,IPHSIZE);

    BYTE buf[MAX];
    memcpy(buf, l2, ETHSIZE);
    memcpy(buf+ETHSIZE, &p.ip, IPHSIZE);
    memcpy(buf+ETHSIZE+IPHSIZE, ospf, 24);

    int sock_fd = create_socket(argv[1]);
    if(!packet_socket) )
        { printf(" no created socket!\n");
          return; }
    write(packet_socket, (BYTE *)buf, ETHSIZE+ntohs(p.ip.tot_len));
    printf("-----packet sent-----\n");
}

```