Server Mirroring Report

by:
Jaspal Singh Dhillon (200802016)
Sanihya Kashyap (200802033)

Contents :

Simple Load Balancing cluster using ***pound*** **:**

Load Balancer is a program which is used to distribute the workload evenly across two or more computers, network links, CPUs, hard drives, or other resources in order to get optimal resource utlization, maximize throughput, minimize response time, and avoid overload.

Some Open source load balancers:

- pound
- pen
- varnish
- ha proxy

We tried **pound** which is a reverse proxy, load balancer, and HTTPS front-end for Web browsers.

Pound is

1. *a reverse-proxy*: it passes requests from client browsers to one or more back-end servers.
2. *a load balancer*: it will distribute the requests from the client browsers among several back-end servers, while keeping session information.
3. *an SSL wrappe*r: *Pound* will decrypt HTTPS requests from client browsers and pass them as plain HTTP to the back-end servers.
4. *an HTTP/HTTPS sanitizer*: *Pound* will verify requests for correctness and accept only well-formed ones.
5. *a fail over-server*: should a back-end server fail, *Pound* will take note of the fact and stop passing requests to it until it recovers.
6. *a request redirector*: requests may be distributed among servers according to the requested URL.

But

1. It is not a **Web server**: by itself, *Pound* serves no content - it contacts the back-end server(s) for that purpose.
2. It is not a **Web accelerator**: no caching is done - every request is passed "as is" to a back-end server.

Here, we have put the image files on one server and all the files on other server.

***Host* : 10.3.3.103**
***JPG/GIF Server* : 10.3.3.179**
***HTTP and all other stuff server* : 10.3.3.150**

Basically , when somebody access the 103 server , the files are fetched from 150 while the images are fetched from 179.

Here we are installing pound on the machine:

Commands executed on host :

```
$ yum install pound
$ useradd -s /sbin/nologin -d /root pound
```

Configured /etc/pound.cfg on host as follows :

```
#defining for the user and group pound which gets created in the /etc/passwd file and /etc/group which will
run
user "pound"
Group   "pound"
# used for logging , default is 1 and 0 for no logging
LogLevel 1
# pound will check for the resurrected back-end hosts default is 30 seconds
Alive 30
# option of running the pound either in foreground / background. 1 means background
Daemon 1
ListenHTTP
        Address 10.3.3.103
        Port    80
End
Service
        URL ".*.(jpg|gif)"
        BackEnd
        Address 10.3.3.179
        Port    80
        End
End
Service
        URL ".*"
        BackEnd
        Address 10.3.3.150
        Port    80
        End
End
```

Here, URL is provided by the user which is a regular expression and then it will get redirected to the backend server by the front node that has been exposed. Backend refers to the main machines that are having the data.

Another scenario:
In this scenario, we are sending the http/https connection using the backend servers 179, and 150 considering 150 as the fast server and giving it a priority to send the data more frequently.

```
ListenHTTP
        Address 10.3.3.103
        Port 80
End
ListenHTTPS
        Address 10.3.3.103
        Port 443
        Cert "/etc/pound/server.pem"
End
Service
        BackEnd
                Address 10.3.3.179
                Port 80
        End
        BackEnd
                Address 10.3.3.150
                Port 80
                Priority 3
        End
End
```

Here, Cert refers to the server certificate. The certificate file is required by the HTTPS listeners which contains the Certificate, possibly a certificate chain and the signature of the server. Priority refers to the machine which should be given more priority in terms of accessing. It is ranged from 1 to 9 and default is 5.

There are many scenarios which can be covered using pound as the **sessions** can be on different servers which can be defined on the basis of IP, Basic, header or cookie including the ttl value as well.

Here, we are adding pound to the startup services so that it just starts when the system is booted.

```
$ chkconfig --add pound
$ chkconfig pound on
$ chkconfig httpd off
```

The httpd service is removed from the service because pound will take care of that as it will directly start httpd when it gets start.

The main disadvantage of this is that its just a http/https load balancer as compared to others which are tcp based. Thus pound is built as web server load balancer, it cannot be used for other purposes. Another problem with this one is that it uses multiple threads which behave as processes which uses a lots of resources and thus can be comparatively slow.

==========

High Availability cluster using **heartbeat** for web services

Primary Node : 10.3.3.179
Slave Node : 10.3.3.150 (In case , 179 machine is down , 150 starts serving content)

All commands were executed on both machines.

- First we stop httpd as it is controlled by heartbeat .

$ service httpd stop

- Then we installed heartbeat :

$ yum -y install heartbeat
$ yum -y install heartbeat

It is necessary to execute the above command two times because it gives the following error the first time to try to install on every system for CentOS 5.5 :

The error :
Running Transaction
Installing : heartbeat-pils 1/3
Installing : heartbeat-stonith 2/3
useradd: user hacluster exists
error: %pre(heartbeat-2.1.3-3.el5.centos.x86_64) scriptlet failed, exit status 9
error: install: %pre scriptlet failed (2), skipping heartbeat-2.1.3-3.el5.centos

- Next we created the certificates :

  Authkeys defines authentication keys.
  Several types are available: crc, md5 and sha1. crc is to be used on a secured sub-network (vlan isolation or cross-over cable).
  sha1 offers a higher level of secuirity but can consume a lot CPU resources.
  md5 sits in between. It's not a bad choice.

- /etc/ha.d/authkeys contains pre-shared secrets used for mutual cluster node authentication. It should only be readable by root and follows this format:

  auth *num*
  num algorithm secret

  here:
  * *num* is a simple key index, starting with 1. Usually, one will only have one key in the authkeys file.
  * *algorithm* is the signature algorithm being used. One may use either md5 or sha1; the use of crc (a simple cyclic redundancy check, not secure) is not recommended.

* *secret* is the actual authentication key.

```
$ cat > /etc/ha.d/authkeys << EOF
auth 1
1 crc
EOF
```

```
$ chmod 600 /etc/ha.d/authkeys
```

it is important to change the permission or we will get an error which needs to be corrected unless the permission is set to 600 for the file. The error is because the password or the secret key is stored in text format without any encryption so it should be only read and written by the root and no one else.

- Next we wrote the **ha.cf** (the configuration file) on 10.3.3.150 :

ha.cf contains the main settings like cluster nodes or the communication topology.

```
$ cat >> /etc/ha.d/ha.cf << EOF
# Activate heartbeat 2 configuration
crm on
#debug log
debugfile /var/log/ha-debug
#log file
logfile /var/log/ha-log
#way of output of syslog
logfacility local0
# Heartbeat packets sending frequency
keepalive 2 # specify ms for less than 1 second
#period of time after which node is declared dead
deadtime 30
deadping 40
#send a warning message
# important to adjust the deadtime value
warntime 10
#identical to dead but used for initializing
initdead 60
#udp port to be used
udpport 694
# the interface should be from where the hostpc can talk to the other , not the interface of the other ip
#keeping it unicast, should use bcast if more than two nodes cluster.
ucast eth1 10.3.3.179
#resource switches back when the primary node becomes available
auto_failback on
#cluster node list
node labpc6       # ( result on uname -n ) on each machine
node vm1
# allow to add dynamically a new node to the cluster
autojoin any
# checking the other machine whether its available or not
respawn root /usr/lib64/heartbeat/pingd -m 100 -d 5s -a default_ping_set
```

EOF

For another machine - 10.3.3.179

```
$ cat >> /etc/ha.d/ha.cf << EOF
# Activate heartbeat 2 configuration
crm on
#debug log
debugfile /var/log/ha-debug
#log file
logfile /var/log/ha-log
#way of output of syslog
logfacility local0
# Heartbeat packets sending frequency
keepalive 2 # specify ms for less than 1 second
#period of time after which node is declared dead
deadtime 30
deadping 40
#send a warning message
# important to adjust the deadtime value
warntime 10
#identical to dead but used for initializing
initdead 60
#udp port to be used
udpport 694
# the interface should be from where the hostpc can talk to the other , not the interface of the other ip
#keeping it unicast, should use bcast if more than two nodes cluster.
ucast eth1 10.3.3.150
#resource switches back when the primary node becomes available
auto_failback on
#cluster node list
node labpc6       # ( result on uname -n ) on each machine
node vm1
# allow to add dynamically a new node to the cluster
autojoin any
# checking the other machine whether its available or not
respawn root /usr/lib64/heartbeat/pingd -m 100 -d 5s -a default_ping_set
EOF
```

*In CentOS, when we install hearbeat, the **ha.cf** is available /usr/share/doc/heartbeat-2.1.3/ha.cf the version that we installed on each system.*

The lib64 is very important but we'll find that lib/ is given everywhere on net. This one is for the X86_64 setup.

- Now we start the heartbeat on both servers :

```
$ /etc/rc.d/init.d/heartbeat start
```

- Then we check the ouput of crm_mon -i 3 on both the machines and we got this :

```
[on 150]:
Defaulting to one-shot mode
You need to have curses available at compile time to enable console mode
============
Last updated: Tue Nov  9 04:10:34 2010
```

Current DC: labpc6 (24587ee1-e6cd-4692-8928-e6fca9dc93b4)
1 Nodes configured.
0 Resources configured.
============
Node: labpc6 (24587ee1-e6cd-4692-8928-e6fca9dc93b4): online

[on vm1]
Defaulting to one-shot mode
You need to have curses available at compile time to enable console mode
============
Last updated: Tue Nov 9 04:15:51 2010
Current DC: vm1 (d0390f56-9f3b-45de-a594-60cf9656ca7e)
1 Nodes configured.
0 Resources configured.
============
Node: vm1 (d0390f56-9f3b-45de-a594-60cf9656ca7e): online

Ideally it should have shown two online nodes but showing one only node. The problem was in the line
ucast eth0 ip
The interface should be from where the host pc can talk to the other , not the interface of the other ip. So, it was fixed in this way and again the service was restarted as follows
$ service httpd restart
$ crm_mon -i3

[on vm1]:
[root@vm1 ha.d]# crm_mon
Defaulting to one-shot mode
You need to have curses available at compile time to enable console mode
============
Last updated: Tue Nov 9 05:00:25 2010
Current DC: 10.3.3.179 (d0390f56-9f3b-45de-a594-60cf9656ca7e)
2 Nodes configured.
0 Resources configured.
============
Node: 10.3.3.150 (24587ee1-e6cd-4692-8928-e6fca9dc93b4): online
Node: 10.3.3.179 (d0390f56-9f3b-45de-a594-60cf9656ca7e): online

[on 150]:
[root@10 ~]# crm_mon
Defaulting to one-shot mode
You need to have curses available at compile time to enable console mode
============
Last updated: Tue Nov 9 04:57:02 2010
Current DC: 10.3.3.179 (d0390f56-9f3b-45de-a594-60cf9656ca7e)
2 Nodes configured.
0 Resources configured.
============
Node: 10.3.3.150 (24587ee1-e6cd-4692-8928-e6fca9dc93b4): online
Node: 10.3.3.179 (d0390f56-9f3b-45de-a594-60cf9656ca7e): online

In order to start the heartbeat at startup, we executed
$ chkconfig heartbeat on

crm_mon is used to display nodes status and resources.

- Now adding the resources as for now httpd to the **heartbeat**

$ rm /var/lib/heartbeat/crm/cib.xml.* -f

Replace <resources/> with this in the file **/var/lib/heartbeat/crm/cib.xml**

```
<resources>
        <group id="group_apache">
                <primitive id="ipaddr" class="ocf" type="IPaddr" provider="heartbeat">
                        <instance_attributes id="ia_ipaddr">
                                <attributes>
                                        <nvpair id="ia_ipaddr_ip" name="ip" value="192.168.136.100"/>
                                        <nvpair id="ia_ipaddr_nic" name="nic" value="eth0"/>
                                        <nvpair id="ia_ipaddr_netmask" name="netmask" value="24"/>
                                </attributes>
                        </instance_attributes>
                </primitive>
                <primitive id="apache" class="ocf" type="apache" provider="heartbeat">
                        <instance_attributes id="ia_apache">
                                <attributes>
                                        <nvpair id="ia_apache_configfile" name="configfile"
                        value="/etc/httpd/conf/httpd.conf"/>
                                </attributes>
                        </instance_attributes>
                </primitive>
        </group>
</resources>
```

It's possible to generate the file from a Heartbeat 1 configuration file, haresources located in /etc/ha.d/, with the following command:

$ python /usr/lib/heartbeat/haresources2cib.py > /var/lib/heartbeat/crm/cib.xml

The file can be split in 2 parts: resources and constraints.

Resources

Resources are organized in groups (server1 & 2) putting together a virtual IP address and a service: Apache.
Resources are declared with the <primitive> syntax within the group.
Groups are useful to gather several resources under the same constraints.
The IP1 primitive checks virtual IP 1 is reachable.
It executes OCF type IPaddr script.
OCF scripts are provided with Heartbeat in the rpm packages.
It's also possible to specify the virtual address, the network mask as well as the interface.
Apache resource type is LSB, meaning it makes a call to a startup script located in the usual /etc/init.d.
The script's name in the variable type: type="name".
In order to run with Heartbeat, the script must be LSB compliant. LSB compliant means the script must:

- return its status with “script status”
- not fail “script start” on a service that is already running
- not fail stopping a service already stopped

All LSB specifications can be checked at http://www.linux- ha.org/LSBResourceAgent
The following time values can be defined:

- interval: defines how often the resource’s status is checked
- timeout: defines the time period before considering a start, stop or status action failed

Constraints

2 constraints apply to each group of resources:
The favourite location where resources "should" run.
We give a score of 100 to n1.domain.com for the 1st group.
Hence, if n1.domain.com is active, and option auto_failback is set to "on", resources in this group will always come back there.
Action depending on ping result. If none of the gateways answer ping packets, resources move on another server, and the node goes to standby status.
Score -INFINITY means the the node will never ever accept these resources if the gateways are unreachable.

192.168.136.100 is the virtual ip which is available to be used by the user and the ip 10.3.3.150 and 10.3.3.179 are being used by heartbeat.

We created ***virtual ip*** using apache in which we added the following lines :

```
$ cat  >> /etc/httpd/conf/httpd.conf << EOF
NameVirtualHost 192.168.136.100:80 # virtual ip--> 192.168.136.100 can be anything
<VirtualHost 192.168.136.100:80>
        ServerAdmin admin@machine.com
        DocumentRoot /var/www/html/
        ServerName vip
#       ErrorLog logs/dummy-host.machine.com-error_log
#       CustomLog logs/dummy-host.machine.com-access_log common
</VirtualHost>
EOF
```

Note: The virtual IP that we assigned was not working because the packets were not accepted by the switch. We should have used the ip range specified by the switch which is 10.3.3.0/24. Last time, it worked because the machine on which we were running was itself behaving as a router since we were using 2 nodes as one virtual machine and one physical machine and the virtual machine which was residing on 10.3.3.103 was exposed. So, we had formed a bridge with IP range 192.168.136.0/24 so that’s why it worked in that case. Here, it will definitely work, if we assign the virtual IP in the range 10.3.3.0/24 which are not being used by the systems. Then the switch will forward the packets of the virtual IP.

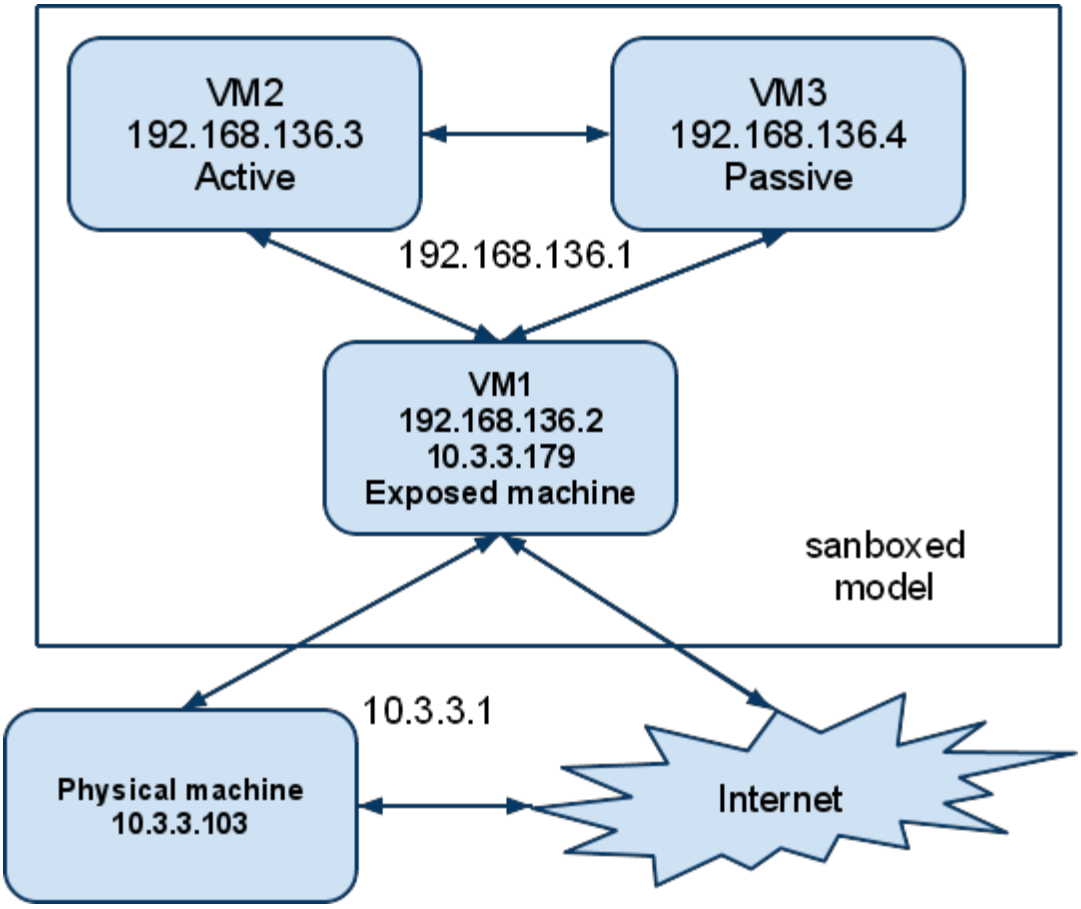
Here we have set up a simple high availability cluster, we can set up another type i.e. **high availability with load balancing**

Here is the link that may be useful
http://www.netexpertise.eu/en/linux/heartbeat-2-howto.html
http://www.drbd.org/users-guide/s-heartbeat-crm.html

==========

Server mirroring using **scapy, xen, wireshark** and **iptables** for TCP/IP packets

Overall Architecture and Idea :
The main idea is to first setup a TCP connection between two machines and then duplicate this connection with a third machine. This becomes more clear after going through the following architecture and diagram :



Here vm stands for virtual machine.
All vms which we set up were installed in only one physical machine with the ip 10.3.3.103.

Setup code for vms :
First we started them on the physical machine :

```
$ virsh start vm1 && virsh start vm2 && virsh start vm3
$ virt-viewer vm1 &
$ virt-viewer vm2 &
$ virt-viewer vm3 &
```

Now, the vm1 was connected to interface xenbr0 to give it a ip on the intranet lan so that it could access internet also.

```
$ virsh attach interface-interface vm1 bridge xenbr0
```

We also enabled packet forwarding in vm1 because we'll be needing this later:

```
$ echo 1 > /proc/sys/net/ipv4/ip_forward
$ echo 1 > /proc/sys/net/ipv4/conf/default/accept_source_root
```

Then we made a internal network for the 3 vms to communicate with each other using the following networking xml script (named as connection.xml) :

```
$ cat >connection.xml << EOF
<network>
   <name> host-only </name>
        <bridge name=”virbr1” />
        <ip address=”192.168.136.1” netmask=”255.255.255.0”>
   </ip>
</network>
EOF
```

Then we attached the 3vms to this network as follows :

```
$ virsh attach-interface vm1 bridge virbr1
$ virsh attach-interface vm1 bridge virbr2
$ virsh attach-interface vm1 bridge virbr3
```

Next we configured the eth0 and eth1 interfaces on vm1 as follows , after which we restarted the network service for the effects to take place :

```
$ cat > /etc/sysconfig/network-scripts/ifcfg-eth1 << EOF
#Xen Virtual Ethernet
DEVICE=eth1
BOOTPROTO=dhcp
ONBOOT=yes
EOF

$ cat > /etc/sysconfig/network-scripts/ifcfg-eth0 << EOF
#Xen Virtual Ethernet
DEVICE=eth0
BOOTPROTO=static
ONBOOT=no
IPADDR=192.168.136.2
NETMASK=255.255.255.0
```

EOF

$ service network restart

Similarly for vm2 :

```
$ cat > /etc/sysconfig/network-scripts/ifcfg-eth0 << EOF
#Xen Virtual Ethernet
DEVICE=eth0
BOOTPROTO=static
ONBOOT=no
IPADDR=192.168.136.3
NETMASK=255.255.255.0
EOF
$ service network restart
```

And lastly vm3 :

```
$ cat > /etc/sysconfig/network-scripts/ifcfg-eth0 << EOF
#Xen Virtual Ethernet
DEVICE=eth0
BOOTPROTO=static
ONBOOT=no
IPADDR=192.168.136.4
NETMASK=255.255.255.0
EOF
$ service network restart
```

Now that the setup is complete , we are going to make a TCP connection from vm1 to vm3 (either by opening vm3's ip in vm1's browser or through ssh) and try to duplicate the connection , that is copy the packets going to vm2 , change the header feilds and send them to vm3 using a tool called scapy.

Note : Scapy will be installed on vm1 only and all related scripts of scapy will run on vm1 only. Vm2 and vm3 are just normal machines.

What is Scapy ?
Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more.

What can Scapy do (What options does it support) ?
It can easily handle most classical tasks like scanning, tracerouting, probing, unit tests, attacks or network discovery (it can replace hping, 85% of nmap, arpspoof, arp-sk, arping, tcpdump, tethereal, p0f, etc.). It also performs very well at a lot of other specific tasks that most other tools can't handle, like sending invalid frames, injecting your own 802.11 frames, combining technics (VLAN hopping+ARP cache poisoning, VOIP decoding on WEP encrypted channel, ...), etc.

Why we used it ?

We used it mainly because of its robustness and the ease of use. As it is a python based program , we didnt have to read huge manpages or how-tos in order to code our 'packet-sniffer-and-copier' . Speed was not a issue for us as we were testing things out and out improving them.

Installing scapy (only on vm1) :

$ wget http://www.secdev.org/projects/scapy/files/scapy-latest.tar.gz
$ tar -xvjf scapy-latest.tar.gz
$ ./configure && make && make install

We also enabled Natting on vm1 since it'll be sending all packets addressed to vm2 , that is vm2 is the server which will handle all incoming requests of vm1. So we added the following two rules :

$ iptables -A PREROUTING -d 192.168.136.2 -p icmp -j DNAT --to-destination 192.168.136.3
$ iptables -I POSTROUTING -s 192.168.136.4 -p icmp -j SNAT --to-source 192.168.136.2

Now the packets which scapy sniffs on the vm1-vm2 connection , copies , changes header feilds and sends to vm3 are absolutely not known to the ip kernel stack of vm1. So if vm3 replies to vm1 , the kernel stack thinks this is not a valid packet and drops it and sends a RST signal to vm3 thereby nullyfing our whole duplicate connection . So we need to make it 'behave' or ignore packets from vm3. This is how we did it :

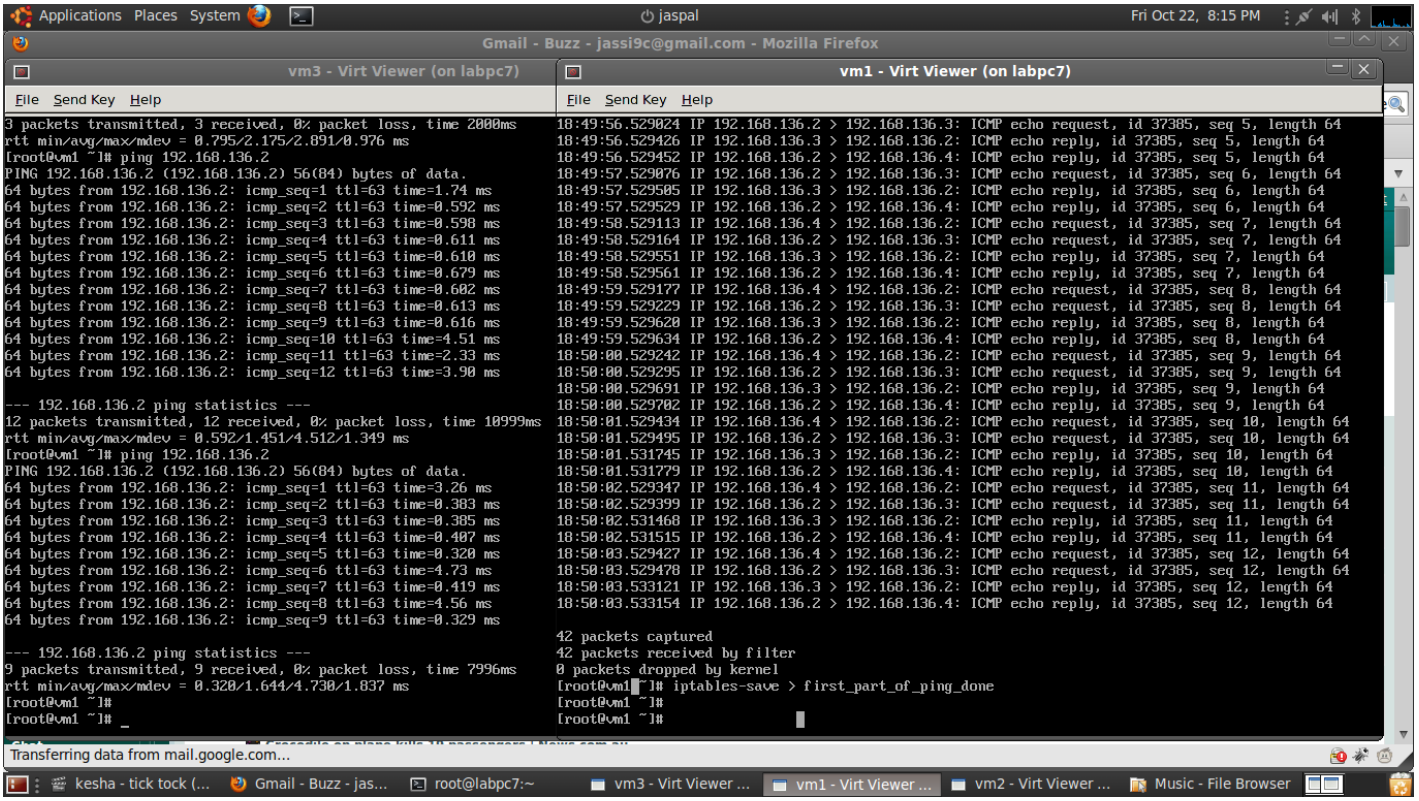
$ iptables -A OUTPUT -p tcp --tcp-flags RST RST -s 192.168.136.2 -d 192.168.136.4 -j DROP
$ iptables -A OUTPUT -s 192.168.136.2 -d 192.168.136.4 -p ICMP --icmp-type port-unreachable -j DROP
Testing ICMP (ping forwarding) and SSH :

First we tested whether packets were indeed being forwarded or not using this idea :
vm3 pings vm1 . Now since vm1 forwards all packets to vm2 , it'll forward this ping. Vm2 finally replies to vm1 and because of natting vm1 replies to vm3.

After that we moved on to ssh by adding these two rules (same as before but without the -p icmp):

$ iptables -A PREROUTING -d 192.168.136.2 -j DNAT --to-destination 192.168.136.3
$ iptables -I POSTROUTING -s 192.168.136.4 -j SNAT --to-source 192.168.136.2

We were successful in doing so and attached is the screenshot depicting the same.



Starting Scapy and sniffing things :

Scapy's interactive shell is run in a terminal session. Root privileges are needed to send the packets, so we're using sudo here :

$ sudo scapy
Welcome to Scapy (2.0.1-dev)
>>>sniff(count=0,filter="tcp",prn=lambda x : x.show())

sniff() is used to sniff packets . Count=0 means sniff indefinitly. filter="tcp" means sniff all tcp protocol packets. Prn is used to tell scapy what to do with a packet. Therefore prn=lambda x: x.show() will show the packets contents as they are captured. Later on we'll modify prn to send the captured packets to a custom function which will copy it and send it to vm3. It can be done like this prn=lambda x : my_function(x)

Moving on to simple TCP Connection :

Below is a simple scapy script running on vm1 which tries to copy the packets and send them to vm3.

```
sq=1                #to store the sequence number of SYN packet going from vm1 to vm2
myack=1             #to store the ack no. of the response to above SYN from vm2
original_tsv=1      #to store the TSV value of the SYN packet
sc=1                #sc stores the response of the vm3 when we duplicate the SYN going to vm2 from vm1 and
                    #send it to vm3
sent=1              #for testing purposes , we wanted to setup just one initial connection.
def sh(packet):     #This function is called in the sniffer part , packet is the set of tuples of sniffed packets
  global sent
                    #packet[0] is the first and only sniffed packet
```

```
p=packet[0][1] #packet[0][1] is the IP Layer in the sniffed packet
q=packet[0][2] #packet[0][2] is the TCP Layer in the sniffed packet
if(p.src=="10.3.3.103" and p.dst=="10.3.3.179"):
        if(q.flags==0x02 and sent==1):
                global original_tsv,myack,sq,sc
                original_tsv=packet[0][2].options[2][1][0]
                packet[0][0].src="00:16:3e:42:21:f5"
                packet[0][0].dst="00:16:3e:d3:8f:71"
                packet[0][1].dst="192.168.136.4"
                packet[0][1].src="192.168.136.2"
                packet[0][1].chksum=None        #None means scapy will calculate the checksum for us
                packet[0][2].chksum=None
                sq=packet[0][2].seq+1
                sc=srp1(packet,iface="eth0")    #send this packet to vm3
                myack=sc.seq+1
                sent=2
        elif(q.flags==0x10 and sent==2):
                packet[0][0].src="00:16:3e:42:21:f5"
                packet[0][0].dst="00:16:3e:d3:8f:71"
                packet[0][1].dst="192.168.136.4"
                packet[0][1].src="192.168.136.2"
                packet[0][1].chksum=None
                packet[0][2].chksum=None
                packet[0][2].seq=sq
                packet[0][2].ack=myack
                packet[0][2].options.remove(packet[0][2].options[2]); #remove the timestamp feild
                qwe = original_tsv+1,sc[2].options[2][1][0]
                new_tuple="Timestamp",qwe
                packet[0][2].options.insert(2,new_tuple) #add the new timestamp feild
                qw=srp1(packet,iface="eth0")
                 sent=3
```

Explanations of some variables are as follows :

p.src : the source of the packet. in this case , 103 is the physical machine.
p.dst : the destination of the packets , in this case 179 is vm1
packet.Checksum=None means that scapy will calculate the chksum for us before sending out the packets.
packet.seq , **packet**.ack mean sequence number and acknowledgement number.
sent variable is used to setup connection just once.

Note : The name of this script was myfunc.py and stored in /usr/local/lib/python2.5/
After starting scapy , we imported this using ,
from myfunc import *

After that , we instructed scapy to call our custom function sh() on each of the captured packets and started testing as follows :
sniff(count=0,filter="tcp",prn=lambda packet : sh(packet))

When we tried to duplicate the packets going from vm1 to vm2 , change header feilds to that

of vm3 , the TCP handshake could not be completed (between vm1 and vm3) because , in the last step , the ACK packet sent by vm1 was not being accepted by vm3. We debuggged a lot ,made the packets identical manually (also made sure the TSV values were what they should be) but could not determine why the hand-shake did not complete.The main problem was that vm3 kept sending the SYN-ACK packets again and again and we were left wondering why isnt our SYN getting accepted. As a result our progress halted there and we decided to experiment other stuff.

Making TCP work by sending payload :
After we presented a demo on what we had done so far , to Saurabh Sir , we understood the problem as to why the other vm kept sending us syn-ack packets. Actually it was requesting further information from us as “payload” . But we were thinking that the ack packet was not getting accepted.

Here is the function which when called will make TCP connection in no time at all :

```
def make_conn_vm():

   ip=IP(src="192.168.136.2",dst="192.168.136.3")
   syn=TCP(sport=11548,dport=80,flags="S",seq=1)
   a=ip/syn
   synack=srp1(a,iface="eth1")

  myack=synack.seq+1
   ack=TCP(sport=11548,dport=80,flags="A",seq=2,ack=myack)
   b=ip/ack
   c=srp1(b,iface="eth1")

payload="stuff"
PUSH=TCP(sport=1154, dport=80, flags="PA", seq=11, ack=my_ack)
send(ip/PUSH/payload)
return synack
```

This is the link that can be used for further reference:
http://www.workrobot.com/sansfire2009/SCAPY-packet-crafting-reference.html

Since a new semester has started ,we can’t add more in the report but will continue it as a side-project with the same zeal and enthusiasm as in the beginning. We will update Sir with our progress.

==========

Finally , We - Jaspal and Sanidhya would like to thank Saurabh Sir for his constant support and his availability at odd times was a real bonus for us.