netkit lab

two-switches

<table>
<thead>
<tr>
<th>Version</th>
<th>2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(s)</td>
<td>G. Di Battista, M. Patrignani, M. Pizzonia, F. Ricci, M. Rimondini</td>
</tr>
<tr>
<td>E-mail</td>
<td><a href="mailto:contact@netkit.org">contact@netkit.org</a></td>
</tr>
<tr>
<td>Web</td>
<td><a href="http://www.netkit.org/">http://www.netkit.org/</a></td>
</tr>
<tr>
<td>Description</td>
<td>experiments with the source address tables of network switches</td>
</tr>
</tbody>
</table>
copyright notice

- All the pages/slides in this presentation, including but not limited to, images, photos, animations, videos, sounds, music, and text (hereby referred to as “material”) are protected by copyright.
- This material, with the exception of some multimedia elements licensed by other organizations, is property of the authors and/or organizations appearing in the first slide.
- This material, or its parts, can be reproduced and used for didactical purposes within universities and schools, provided that this happens for non-profit purposes.
- Information contained in this material cannot be used within network design projects or other products of any kind.
- Any other use is prohibited, unless explicitly authorized by the authors on the basis of an explicit agreement.
- The authors assume no responsibility about this material and provide this material “as is”, with no implicit or explicit warranty about the correctness and completeness of its contents, which may be subject to changes.
- This copyright notice must always be redistributed together with the material, or its portions.
step1 – network topology

all the mac addresses are in the form: 00:00:00:00:XX:YY

ABC are collision domains
step 2 – starting the lab

- the started lab is made up of
  - 3 virtual machines that implement the **pcs**
  - 2 virtual machines that implement the **switches**
    - automatically configured to perform switching
  - all the virtual machines and their network interfaces are automatically configured

```
user@localhost:~$ cd netkit-lab_two-switches
user@localhost:~/netkit-lab_two-switches$ lstart
```
step 3 – configuring network interfaces

- real network interfaces have a wired in mac address
  - the first three bytes make up the Organizationally Unique Identifier (OUI), a sequence that matches the vendor of the nic
  - the remaining three bytes are the interface serial number

- mac address of an interface card manufactured by Asustek inc.:
  
  00:13:D4:AC:55:4E

  oui  serial
step 3 – configuring network interfaces

- Virtual network interfaces are automatically assigned a MAC address.

```
pc:~# ifconfig eth0 14.0.0.2 up
pc:~# ifconfig eth0
eth0 Link encap:Ethernet  HWaddr FE:FD:0E:00:00:02
   inet addr:14.0.0.2  Bcast:14.255.255.255  Mask:255.0.0.0
   inet6 addr: fe80::fcfd:eff:fe00:2/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:1 errors:0 dropped:0 overruns:0 frame:0
      TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:72 (72.0 b)  TX bytes:336 (336.0 b)
pc:~#  
```

- Depending on the version of Netkit in use, the MAC address might be derived from the IP address.
step 3 – configuring network interfaces

- the MAC address of a virtual network interface can be forcibly configured in the following way:

```
switch1:~# ifconfig eth0 up
switch1:~# ifconfig eth0 hw ether 00:00:00:00:01:00
switch1:~# ifconfig eth0
```

```
eth0 Link encap:Ethernet HWaddr 00:00:00:00:01:00
inet6 addr: fe80::fcfd:ff:fe00:0/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:13 errors:0 dropped:0 overruns:0 frame:0
TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:828 (828.0 b) TX bytes:378 (378.0 b)
Interrupt:5
```

```
step 3 – configuring network interfaces

the mac address of a virtual network interface can be forcefully configured in the following way:

```
switch1:~# ifconfig eth0 up
switch1:~# ifconfig eth0 hw ether 00:00:00:00:01:00
switch1:~# ifconfig eth0

eth0    Link encap:Ethernet  HWaddr 00:00:00:00:01:00
        inet6 addr: fe80::fcfd:ff:fe00:0/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500 Metric:1
        RX packets:13 errors:0 dropped:0 overruns:0 frame:0
        TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:828 (828.0 b)  TX bytes:378 (378.0 b)
        Interrupt:5
```

at this point the interface has a default address

```
switch1:~#
```
step 3 – configuring network interfaces

- The MAC address of a virtual network interface can be forcibly configured in the following way:

```
switch1:~# ifconfig eth0 up
switch1:~# ifconfig eth0 hw ether 00:00:00:00:01:00
```

At this point the interface has the desired address.
step 3 – configuring network interfaces

- the mac address of a virtual network interface can be forcibly configured in the following way:

```
switch1:~# ifconfig eth0 up
switch1:~# ifconfig eth0 hw ether 00:00:00:00:01:00
```

notice: • the mac address must be configured after issuing `ifconfig eth0 up`, because this command resets the address to the default value
• a switch is a layer 2 device; therefore, its interfaces do not require an ip address
step 4 – bridging capabilities

- `brctl` allows to check and configure the settings of the bridging capabilities of a virtual machine.

```
switch1:

switch1:~# brctl show
bridge name  bridge id  STP enabled  interfaces
br0          8000.0000000000100  yes  eth0
             8000.00000000000100  yes  eth1
```

```
switch2:

switch2:~# brctl show
bridge name  bridge id  STP enabled  interfaces
br0          8000.0000000000200  yes  eth0
             8000.00000000000200  yes  eth1
```
step 4 – bridging capabilities

- **brctl** allows to check and configure the settings of the bridging capabilities of a virtual machine

```
switch1:~# brctl addbr br0
create a new bridge br0

switch1:~# brctl addif br0 eth0
attach network interfaces to bridge br0

switch1:~# brctl addif br0 eth1

switch1:~# brctl stp br0 on
enable the spanning tree protocol on bridge br0

switch1:~# ifconfig br0 up
enable the bridge
```

- A virtual machine may enable several bridging processes (on different network interfaces)
- Once configured, a bridge is visible as a network interface that must be brought up in order to function properly
step 5 – investigating source address tables

- if the PCs do not generate any traffic, the source address tables only contain information about local ports

---

### switch1

<table>
<thead>
<tr>
<th>port no</th>
<th>mac addr</th>
<th>is local?</th>
<th>ageing timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00:00:00:00:01:00</td>
<td>yes</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>00:00:00:00:01:01</td>
<td>yes</td>
<td>0.00</td>
</tr>
</tbody>
</table>

### switch2

<table>
<thead>
<tr>
<th>port no</th>
<th>mac addr</th>
<th>is local?</th>
<th>ageing timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00:00:00:00:02:00</td>
<td>yes</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>00:00:00:00:02:01</td>
<td>yes</td>
<td>0.00</td>
</tr>
</tbody>
</table>
step 5 – investigating source address tables

- Depending on the configuration, a machine may generate traffic even if not solicited (e.g., broadcast packets).
  - The source address tables of `switch1` and `switch2` may already contain non-local entries.
  - Hard to prevent.
- Ports (= interfaces) are numbered according to the 802.1d standard.
  - The correspondence between kernel interface numbering (ethX) and 802.1d numbering can be obtained by using `brctl showstp`.
### step 5 – investigating source address tables

#### switch1

```plaintext
switch1:~# brctl showstp br0
br0
  bridge id        8000.0000000000100
  designated root  8000.0000000000100
.....
  eth0 (1)
    port id        8001
    state          forwarding
.....
  eth1 (2)
    port id        8002
    state          forwarding
.....
```

#### switch2

```plaintext
switch2:~# brctl showstp br0
br0
  bridge id        8000.0000000000200
  designated root  8000.0000000000100
.....
  eth0 (1)
    port id        8001
    state          forwarding
.....
  eth1 (2)
    port id        8002
    state          forwarding
.....
```
step 6 – evolution of the address tables

- start a sniffer on **pc3**:

  ```
  pc3:~# tcpdump -e -q
  ```

  **dump link-level headers**

  **shorter output**

- generate traffic between **pc2** and **pc3**:

  ```
  pc2:~# ping 10.0.0.3
  PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
  64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=0.237 ms
  64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.184 ms
  64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.182 ms

  --- 10.0.0.3 ping statistics ---
  3 packets transmitted, 3 received, 0% packet loss, time 2004ms
  rtt min/avg/max/mdev = 0.182/0.201/0.237/0.025 ms
  ```
step 6 – evolution of the address tables

pc1
10.0.0.1

10:00

switch1

eth0
01:00

eth1
01:01

switch2

eth1
02:01

eth0
02:00

pc2
10.0.0.2

pc3
10.0.0.3

10:00

20:00

30:00
step 6 – evolution of the address tables

- pc3 sees the traffic exchanged on its collision domain (C)

```
pc3:~# tcpdump -e -q
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
14:32:32.167034 00:00:00:00:20:00 > Broadcast, ARP, length 42: arp who-
has 10.0.0.3 tell 10.0.0.2
14:32:32.167180 00:00:00:00:30:00 > 00:00:00:00:20:00, ARP, length 42: 
arp reply 10.0.0.3 is-at 00:00:00:00:30:00
14:32:32.171178 00:00:00:00:20:00 > 00:00:00:00:30:00, IPv4, length 98:
IP 10.0.0.2 > 10.0.0.3: icmp 64: echo request seq 1
14:32:32.171379 00:00:00:00:30:00 > 00:00:00:00:20:00, IPv4, length 98:
IP 10.0.0.3 > 10.0.0.2: icmp 64: echo reply seq 1
14:32:33.164562 00:00:00:00:20:00 > 00:00:00:00:30:00, IPv4, length 98:
IP 10.0.0.2 > 10.0.0.3: icmp 64: echo request seq 2
.....
```
step 6 – evolution of the address tables

pc1 10.0.0.1
pc2 10.0.0.2
pc3 10.0.0.3

A 10:00
switch1 eth0

B 01:00
switch1 eth1

C 01:01
switch2 eth1

02:01
switch2 eth0

10:00

14:32:32.167034 00:00:00:00:20:00 > Broadcast, ARP, length 42: arp who-has 10.0.0.3 tell 10.0.0.2
step 6 – evolution of the address tables

Address stored in the source address table
step 6 – evolution of the address tables

pc1
10.0.0.1

switch1

switch2

pc2
10.0.0.2

pc3
10.0.0.3

pc3

14:32:32.167180 00:00:00:00:30:00 > 00:00:00:00:20:00, ARP, length 42: arp reply 10.0.0.3 is-at 00:00:00:00:30:00

netkit – [ lab: two-switches ]

last update: May 2007
step 6 – evolution of the address tables

pc1
10.0.0.1

pc2
10.0.0.2

pc3
10.0.0.3

14:32:32.167180 00:00:00:00:30:00 > 00:00:00:00:20:00, ARP, length 42: arp reply 10.0.0.3 is-at 00:00:00:00:30:00
step 6 – evolution of the address tables

14:32:32.171178 00:00:00:00:20:00 > 00:00:00:00:30:00, IPv4, length 98: IP 10.0.0.2 > 10.0.0.3: icmp 64: echo request seq 1
step 6 – evolution of the address tables

pc1
10.0.0.1

switch1
eth0
01:00

switch2
eth1
01:01

pc2
10.0.0.2

pc3
10.0.0.3

pc3
10:00

14:32:32.171379 00:00:00:00:30:00 > 00:00:00:00:20:00, IPv4, length 98: IP 10.0.0.3 > 10.0.0.2: icmp 64: echo reply seq 1

© Computer Networks Research Group Roma Tre

netkit – [ lab: two-switches ]

last update: May 2007
step 6 – evolution of the address tables

switch1

```
switch1:/# brctl showmacs br0
port no mac addr            is local? ageing timer
 1 00:00:00:00:01:00          yes          0.00
 2 00:00:00:00:01:01          yes          0.00
 2 00:00:00:00:20:00          no           1.97
```

switch2

```
switch2:/# brctl showmacs br0
port no mac addr            is local? ageing timer
 2 00:00:00:00:01:01          no           0.59
 1 00:00:00:00:02:00          yes          0.00
 2 00:00:00:00:02:01          yes          0.00
 1 00:00:00:00:20:00          no           0.55
 1 00:00:00:00:30:00          no           0.55
```
step 6 – evolution of the address tables

### switch1

```
switch1:~# brctl showmacs br0
port no  mac addr   is local?  ageing timer
  1  00:00:00:00:01:00  yes       0.00
  2  00:00:00:00:01:01  yes       0.00
  2  00:00:00:00:20:00  no        1.97
```

### switch2

```
switch2:~# brctl showmacs br0
port no  mac addr   is local?  ageing timer
  2  00:00:00:00:01:01  no        0.59
  1  00:00:00:00:02:00  yes       0.00
  2  00:00:00:00:02:01  yes       0.00
  1  00:00:00:00:20:00  no        0.55
  1  00:00:00:00:30:00  no
```

- `this entry is due to packets exchanged for spanning tree calculation`

---

© Computer Networks Research Group Roma Tre
step 6 – evolution of the address tables

- switch2 knows the positions of pc2 and pc3 since it has seen their traffic
- switch1 does not know the position of pc3 since pc3’s traffic has been filtered out by switch2
- the two switches are not aware of pc1
step 7 – filtering in action

- clear the address tables by setting the lifetime (ageing) of the entries to 10 seconds:

  ```
  switch1:
  brctl setageing br0 10
  switch2:
  brctl setageing br0 10
  ```

- after 10 seconds of “silence” only the local interfaces remain in the source address tables.
step 7 – filtering in action

- repeat the **ping** experiment with a 3 seconds interval and place a sniffer on **pc1**:

```
$ pc1:~# tcpdump -e -q
```

```
$ pc2:~# ping -i 3 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=0.237 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.184 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.182 ms
--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.182/0.201/0.237/0.025 ms
```

© Computer Networks Research Group Roma Tre

netkit – [ lab: two-switches ]

last update: May 2007
step 7 – filtering in action

pc1 10.0.0.1

switch1

eth0 01:00
eth1 01:01

switch2

eth1 02:01
eth0 02:00

pc3 10.0.0.3

pc2 10.0.0.2

netkit – [ lab: two-switches ]
step 7 – filtering in action

since the switches filter traffic, only broadcast packets can reach \textbf{pc1}:

```
pc1:~# tcpdump -e -q
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
15:45:50.142942 00:00:00:00:20:00 > Broadcast, ARP, length 42:
  arp who-has 10.0.0.3 tell 10.0.0.2
```
step 7 – filtering in action

- keep the **ping** active and reduce the lifetime of the entries of the source address table:

  ![Switch 1](image1)
  ![Switch 2](image2)

  ```
  switch1:~# brctl setageing br0 1
  switch2:~# brctl setageing br0 1
  ```

- in this way, the entries expire after each echo request has been sent (echo requests are sent every 3 seconds)
  - every time **pc2** generates an echo request:
    - **switch2** does not know about **pc3**, hence performs flooding
    - **switch1** does not know about **pc3**, hence performs flooding
    - as a consequence, **pc1** sees the echo request sent by **pc2**
  - every time **pc3** generates an echo reply:
    - **switch2** knows about **pc2** (thanks to the echo request) and filters traffic
    - as a consequence, neither **switch1** nor **pc1** see the echo reply
    - note that echo replies are sent within the 1 second lifetime
step 7 – filtering in action

- **pc1** only sees the echo requests:

  pc1:~# tcpdump -e -q
  tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
  listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
  16:38:49.305818    00:00:00:20:00 > Broadcast, ARP, length 42: arp who-has
  10.0.0.3 tell 10.0.0.2
  16:38:52.305602    00:00:00:20:00 > 00:00:00:30:00, IPv4, length 98: IP
  10.0.0.2 > 10.0.0.3: icmp 64: echo request seq 2
  16:38:55.322456    00:00:00:20:00 > 00:00:00:30:00, IPv4, length 98: IP
  10.0.0.2 > 10.0.0.3: icmp 64: echo request seq 3
  16:38:58.333206    00:00:00:20:00 > 00:00:00:30:00, IPv4, length 98: IP
  10.0.0.2 > 10.0.0.3: icmp 64: echo request seq 4
  ..... 

- the arp reply sent by **pc3** to **pc2** is filtered because **switch2** knows about **pc2** (thanks to the arp request)

- the first echo request is also filtered because immediately after the arp exchange **switch2** still knows about **pc3**