BSc in Telecommunications Engineering

TEL3214

Computer Communication Networks

Lecture 05
Switching

Eng Diarmuid O'Briain, CEng, CISSP

Department of Electrical and Computer Engineering,
College of Engineering, Design, Art and Technology,
Makerere University

Table of Contents

Illustration Index

1. Bridging and Switching

A bridge is a device used to connect two or more Local Area Networks (LAN) that use identical LAN (Medium Access Control (MAC) Layer) protocols. The bridge acts as an address filter, picking up frames from one LAN segment (collision domain) that are intended for a destination on another LAN segment, and passing those frames on. The bridge does not modify the contents of the frames and does not add anything to the frame. The bridge operates at layer 2 of the OSI model.

The original concept of a bridge was a device that would interface similar LAN segments and would *filter* and *forward* transmissions (pass those which were for address not on the source segment, and not pass those whose destination address is on the source segment). So bridges maintain tables of addresses associated with each port on the bridge. The IEEE 802.1 standard defines bridges, often called "Ethernet bridges" or transparent bridges, and some vendors call them *Ethernet switches*.

In the simplest terms, a bridge *forwards* (sends) frames between LAN segments that are attached to its ports using information it finds in the OSI Model Layer 2, the Data Link Layer (actually the IEEE 802.3 MAC layer addressing), of a frame. It ignores the other layers of the OSI Model. In other words, it looks at the destination address field, compares the address to its address tables for all its ports. If it finds the address associated with a port, it sends the frame out on that port. If it does not find an address, it sends the frame out on all ports.

In a multi-bridge IEEE LAN environment, the bridges usually communicate with each other using the IEEE 802.1d *Spanning Tree Algorithm* (STA) protocol or other protocol. Bridges have a problem when an address is unknown the frame is forwarded to all ports off the bridge. This could cause address table problems and frame propagation in multi-LAN environments without the spanning tree algorithm capability.

In pure bridges, there are 2 types of transparent (with or without Spanning Tree) bridging and Source Routing bridging. Bridges are most effective when there are few links in a network. Larger networks usually use Routers where links are numerous. Transparent bridges are normally *connectionless* switching devices, which means they themselves do not help maintain connections in the network. Transparent bridges just send frames or frames out a port, they do not route them to another device.

1.1 Why use Bridges

- Limit number of stations (contention) transmitting on specific segments.
- Limit Size of LANs.
- Limit volume of traffic (bandwidth).
- Reduces traffic across segments of a single LAN.
- Connect multiple local LANs into a single network at a local level.

2. Switches

A switch is a device designed to segment LANs with one idea in mind, increase the bandwidth. This differs from a bridge or router whose purpose is to limit the amount of traffic flowing between LANs (a LAN will be sometimes referred to as a collision domain).

The Layer 2 (L2) Switch interconnects LAN segments. Traffic between the LAN segments will be switched at near *wire speed*.

A bridge normally will have 2 or 3 ports, where a switch will have 4, 6, or more ports for attaching separate LANs or collision domains. 10/100/1000 Mb/s switches have two or more 1000 Mb/s and 4 or more 10/100 Mb/s ports. Consequently, collision domains can have more segmentation than with a bridge.

2.1 Why Switching

- Switches operate at Layer 2 of the OSI Model.
- Switching is and advance in bridging technology.
- Switches forward frames based on the MAC layer address (the actual Network Interface Card (NIC) address).
- Switches forward frames with very low delay time (wire speed).
- Switches, in most cases, use the IEEE 802.1d Spanning Tree Protocol (STP) or IEEE 802.1w Rapid Spanning Tree Protocol (RSTP) allowing for redundant switches in the network.
- Switches will forward *broadcast* traffic to all LANs attached to them.

2.2 When is switching used

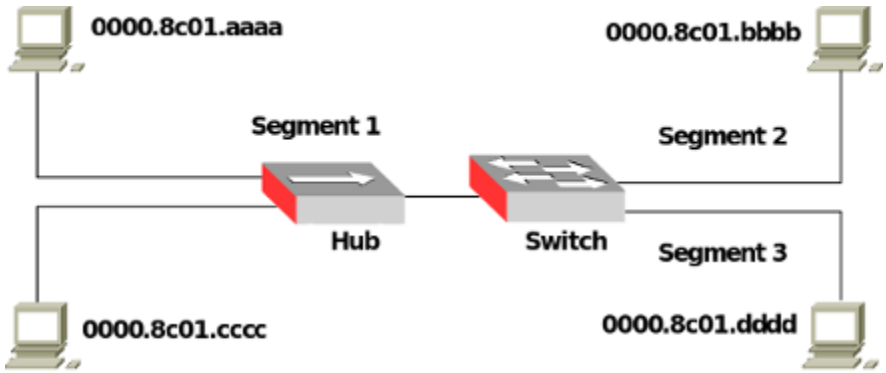
Switching is used when segmentation/connection of several LAN segments is required with increased bandwidth. If security among the LANs is not a significant issue then a switch can be used rather than a router if the following services are not required;

- Support redundant paths.
- Have intelligent frame forwarding.
- Connect to a WAN.

3. Transparent Bridging

L2 Switches and Transparent Bridges use transparent bridging to create their address lookup tables. Transparent bridging allows a switch to learn everything it needs to know about the location of nodes on the network without the network administrator having to statically add entries. Transparent bridging consists of five parts or steps:

- Learning
- Flooding
- Filtering
- Forwarding
- Ageing



Segment 1	0000.8c01.aaaa	0000.8c01.cccc
Segment 2	0000.8c01.bbbb	
Segment 3	0000.8c01.dddd	

Illustration 1: Transparent Bridging

The switch is added to the network, and the various segments are plugged into the switch's ports. A host with the MAC *0000.8c01.aaaa* (aaaa) on the first segment sends data to a host *0000.8c01.bbbb* (bbbb) on another segment 2.

The switch gets the first frame of data from *aaaa*. It reads the MAC address and saves it to the lookup table for Segment 1. The switch now knows where to find *aaaa* any time a frame is addressed to it. This process is called *learning*.

Since the switch does not know where *0000.8c01.bbbb* (bbbb) is, it sends the frame to all of the segments except the one that it arrived on (Segment 1). When a switch sends a frame out to all segments to find a specific node, it is called *flooding*.

The host *bbbb* gets the frame and sends a frame back to *aaaa* in acknowledgement. The frame from *bbbb* arrives at the switch. Now the switch can add the MAC address of *0000.8c01.bbbb* to the lookup table for Segment 2. Since the switch already knows the address of *aaaa*, it sends the frame directly to it. Because *aaaa* is on a different segment than *bbbb*, the switch must connect the two segments to send the frame. This is known as *forwarding*.

The next frame from *aaaa* to *bbbb* arrives at the switch. The switch now has the address of *bbbb* in its tables, so it *forwards* the frame directly to *bbbb*. *0000.8c01.cccc* (cccc) sends information to the switch for *aaaa*. The switch looks at the MAC address for *cccc* and adds it to the lookup table for Segment 1. The switch already has the address for *aaaa* and determines that both nodes are on the same segment, so it does not need to connect Segment 1 to another segment for the data to travel from *cccc* to *aaaa*. Therefore, the switch will ignore frames travelling between nodes on the same segment. This is *filtering*.

Learning and flooding continue as the switch adds nodes to the lookup tables. Most switches have plenty of memory in a switch for maintaining the lookup tables; but to optimise the use of this memory, they still remove older information so that the switch doesn't waste time searching through stale addresses. To do this, switches use a technique called *ageing*. Basically, when an entry is added to the lookup table for a node, it is given a time-stamp. Each time a frame is received from a node, the time-stamp is updated. The switch has a user-configurable timer that erases the entry after a certain amount of time with no activity from that node. This frees up valuable memory resources for other entries. As can be seen, transparent bridging is a great and essentially maintenance-free way to add and manage all the information a switch needs.

In the example, two nodes share segment 1, while the switch creates independent segments for *bbbb* and *dddd*. In an ideal LAN-switched network, every node would have its own segment. This would eliminate the possibility of collisions and also the need for filtering.

3.1 Address Resolution

To allow forwarding and filtering of frames at wire speed, LAN switches should be able to decode MAC addresses very quickly. Since Central Processing Unit (CPU) based lookups are expensive, hardware solutions may be used. Switches maintain address tables just like transparent bridges. They learn the addresses of their neighbours, and when a frame is to be forwarded, they first look up the address table and broadcast only if no entry corresponding to that destination is found. Stations that have not transmitted recently are aged out. This way a small address table can be maintained and the switch can relearn if a station starts transmitting again.

3.2 Broadcast Storm

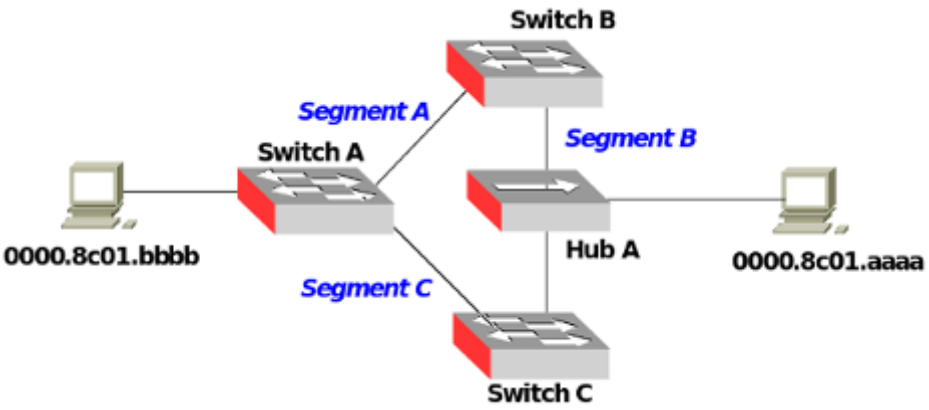


Illustration 2 Broadcast storm

In the example shown in the diagram, even if one of the switches fails, the network will continue to function. The loop provides redundancy, effectively eliminating a single point of failure. However it introduces a new problem. With all of the switches now connected in a loop, a frame from a node could quite possibly come to a switch from two different segments. For example, imagine that *0000.8c01.bbbb* (bbbb) is connected to Switch A, and needs to communicate with *0000.8c01.aaaa* (aaaa) on Segment B. Switch A does not know who *aaaa* is, so it floods the frame.

The frame travels via Segment A or Segment C to the other two switches (B and C). Switch B will add *bbbb* to the lookup table it maintains for Segment A, while Switch C will add it to the lookup table for Segment C. If neither switch has learned the address for *aaaa* yet, they will flood Segment B looking for *aaaa*.

Each switch will take the frame sent by the other switch and flood it back out again immediately, since they still don't know who *aaaa* is. Switch A will receive the frame from each segment and flood it back out on the other segment. This causes a broadcast storm as the frames are broadcast, received and rebroadcast by each switch, resulting in potentially severe network congestion.

4. Spanning Tree Protocol

To prevent broadcast storms and other unwanted side effects of looping, Digital Equipment Corporation (DEC) created the Spanning Tree Protocol (STP), which has been standardised as the IEEE 802.1d specification by the IEEE. Essentially, a spanning tree uses STA, which senses that the switch has more than one way to communicate with a node, determines which way is best and blocks out the other path(s). It also keeps track of the other path(s), just in case the primary path is unavailable.

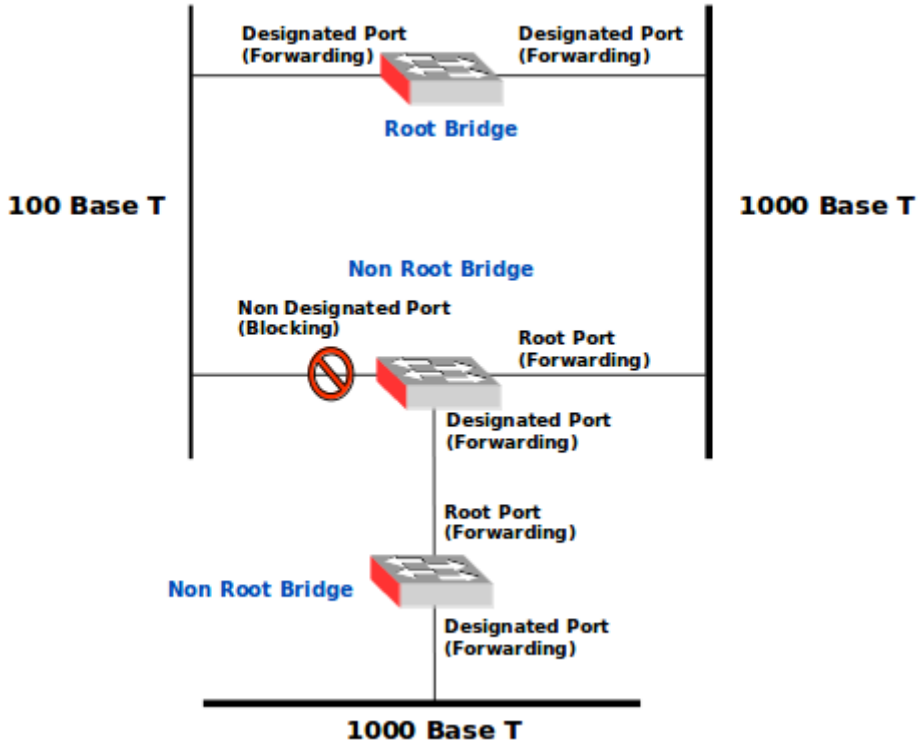


Illustration 3: Spanning Tree Protocol (STP)

Each switch is assigned a group of IDs, one for the switch itself and one for each port on the switch. The switch's identifier, called the Bridge ID (BID), is 8 bytes long and contains a bridge priority (2 bytes) along with one of the switch's MAC addresses (6 bytes). Each port ID is 16 bits long with two parts: a 6-bit priority setting and a 10-bit port number.

A path cost value is given to each port. The cost is typically based on a guideline established as part of IEEE 802.1d and further enhanced with IEEE 802.1w Rapid STP (RSTP). According to the original specification, cost is 1,000 Mb/s (1 gigabit per second) divided by the bandwidth of the segment connected to the port. Therefore, a 10 Mb/s connection would have a cost of (1,000/10) 100.

To compensate for the speed of networks increasing beyond the Gb/s range, the standard cost has been modified over time. The new values are:

Data rate	STP Cost - 802.1d	RSTP Cost - 802.1w
4 Mb/s	250	5000000
10 Mb/s	100	2000000
16 Mb/s	62	1250000
100 Mb/s	19	200000
1 Gb/s	4	20000
2 Gb/s	3	10000
10 Gb/s	2	2000

Illustration 4: STP Path costs

It should also be noted that the path cost can be an arbitrary value assigned by a network administrator in most switches, instead of one of the standard cost values.

Each switch begins a discovery process to choose which network paths it should use for each segment. This information is shared between all the switches by way of special network frames called Bridge Protocol Data Units (BPDU). The BPDU consists of:

- Root BID
    - This is the BID of the current root bridge.
- Path cost to root bridge
    - This determines how far away the root bridge is. For example, if the data has to travel over three 100 Mb/s segments to reach the root bridge, then the cost is (19 + 19 + 0) 38. The segment attached to the root bridge will normally have a path cost of zero.
- Sender BID
    - This is the BID of the switch that sends the BPDU.
- Port ID
    - This is the actual port on the switch that the BPDU was sent from.

A root bridge is chosen based on the results of the BPDU process between the switches. Initially, every switch considers itself the root bridge. When a switch first powers up on the network, it sends out a BPDU with its own BID as the root BID. When the other switches receive the BPDU, they compare the BID to the one they already have stored as the root BID. If the new root BID has a lower value, they replace the saved one. But if the saved root BID is lower, a BPDU is sent to the new switch with this BID as the root BID. When the new switch receives the BPDU, it realises that it is not the root bridge and replaces the root BID in its table with the one it just received. In this way the switch that has the lowest BID is elected by the other switches as the root bridge.

Based on the location of the root bridge, the other switches determine which of their ports has the lowest path cost to the root bridge. These ports are called root ports, and each switch (other than the current root bridge) must have one.

The switches determine who will have designated ports. A designated port is the connection used to send and receive frames on a specific segment. By having only one designated port per segment, all looping issues are resolved.

Designated ports are selected based on the lowest path cost to the root bridge for a segment. Since the root bridge will have a path cost of *0*, any ports on it that are connected to segments will become designated ports. For the other switches, the path cost is compared for a given segment. If one port is determined to have a lower path cost, it becomes the designated port for that segment. If two or more ports have the same path cost, then the switch with the lowest BID is chosen.

Once the designated port for a network segment has been chosen, any other ports that connect to that segment become non-designated ports. They block network traffic from taking that path so it can only access that segment through the designated port.

Each switch has a table of BPDUs that it continually updates. The network is now configured as a single spanning tree, with the root bridge as the trunk and all the other switches as branches. Each switch communicates with the root bridge through the root ports, and with each segment through the designated ports, thereby maintaining a loop-free network. In the event that the root bridge begins to fail or have network problems, STP allows the other switches to immediately reconfigure the network with another switch acting as Root Bridge. This process gives a company the ability to have a complex network that is fault-tolerant and yet fairly easy to maintain.

5. Configuration of a Bridge interface on GNU/Linux

GNU/Linux through the *bridge-utils* offers the functionality to create an internal Ethernet switch and put selected interfaces into it. Control of the bridge is via the *brctl* command. This command gives the configuration options expected of a typical Ethernet switch. It supports functionality like Spanning Tree Protocol (*STP*). As NTE uses GNU/Linux as the basis for all devices bridging can be achieved through *bridge-utilities*.

5.1 bridge-utils

```
  # brctl --help
  Usage: brctl [commands]
  commands:
        addbr           <bridge>                    add bridge
        delbr           <bridge>                    delete bridge
        addif           <bridge> <device>           add interface to bridge
        delif           <bridge> <device>           delete interface from
bridge
        hairpin         <bridge> <port> {on|off}    turn hairpin on/off
        setageing       <bridge> <time>             set ageing time
        setbridgeprio   <bridge> <prio>             set bridge priority
        setfd           <bridge> <time>             set bridge forward delay
        sethello        <bridge> <time>             set hello time
        setmaxage       <bridge> <time>             set max message age
        setpathcost     <bridge> <port> <cost>      set path cost
        setportprio     <bridge> <port> <prio>      set port priority
        show            [ <bridge> ]                show a list of bridges
        showmacs        <bridge>                    show a list of mac addrs
        showstp         <bridge>                    show bridge stp info
        stp             <bridge> {on|off}           turn stp on/off
```

6. Simple bridged network lab

Build a simple network on the NTE emulator.

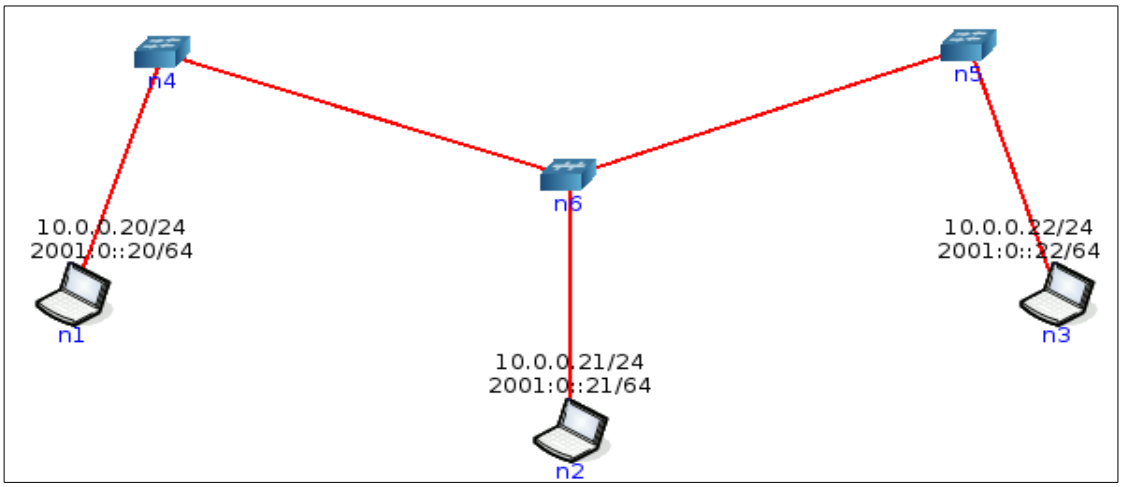


Illustration 5: Simple bridged network

Build as in Illustration 5 with two switches **n5** and **n6**. **n4** is a hub which allows for packet tracing of all traffic between **n5** and **n6**.

Perform a ping from Host **n1** to Host **n3** while monitoring the network with *Tshark* monitoring from Host **n2**. *Tshark* prints a description of the contents of packets on the network interface of **n6** with the description preceded by a time stamp, as hours, minutes, seconds, and fractions of a second since midnight.

```
root@n1:/tmp/pycore.41149/n1.conf# ping -c1 10.0.22
PING 10.0.22 (10.0.0.22) 56(84) bytes of data.
64 bytes from 10.0.0.22: icmp_seq=1 ttl=64 time=0.035 ms

--- 10.0.22 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.035/0.035/0.035/0.000 ms


Capturing on 'eth0'
1 0.000000 10.0.0.20 -> 10.0.0.22  ICMP 98 Echo request  id=0x0018, seq=1/256, ttl=64
2 0.000015 10.0.0.22 -> 10.0.0.20  ICMP 98 Echo reply  id=0x0018, seq=1/256, ttl=64
3 5.002660 00:00:00:aa:00:00 -> 00:00:00:aa:00:02 ARP 42 Who has 10.0.0.22? Tell
10.0.0.20
4 5.002661 00:00:00:aa:00:02 -> 00:00:00:aa:00:00 ARP 42 Who has 10.0.0.20? Tell
10.0.0.22
5 5.002679 00:00:00:aa:00:02 -> 00:00:00:aa:00:00 ARP 42 10.0.0.22 is at
00:00:00:aa:00:02
6 5.002680 00:00:00:aa:00:00 -> 00:00:00:aa:00:02 ARP 42 10.0.0.20 is at
00:00:00:aa:00:00
```

6.1 Reviewing the bridges

```
# brctl show
bridge name     bridge id           STP enabled   interfaces
b.4.26          8000.0a388d18a46b   no            veth2.0.26
                                                  veth4.5.26
                                                  veth4.6.26
b.5.26          8000.1a089586bfc5   no            veth1.0.26
                                                  veth5.4.26
b.6.26          8000.1228663b99b8   no            veth3.0.26
                                                  veth6.4.26
```

6.1.1 Review bridge

Now that the bridge is created review it.

```
root@NTE-i386:~# sudo brctl showmacs b.4.26
port no  mac addr            is local?    ageing timer
   1     0a:38:8d:18:a4:6b   yes             0.00
   2     8a:bb:4a:15:9c:5e   yes             0.00
   3     a2:56:5d:c8:0c:b9   yes             0.00

root@NTE-i386:~# sudo brctl showmacs b.5.26
port no  mac addr            is local?    ageing timer
   1     1a:08:95:86:bf:c5   yes             0.00
   2     1e:80:83:c9:57:3f   yes             0.00

root@NTE-i386:~# sudo brctl showmacs b.6.26
port no  mac addr            is local?    ageing timer
   1     12:28:66:3b:99:b8   yes             0.00
   2     ea:ef:21:4c:67:99   yes             0.00
```

7. Bridge with a loop

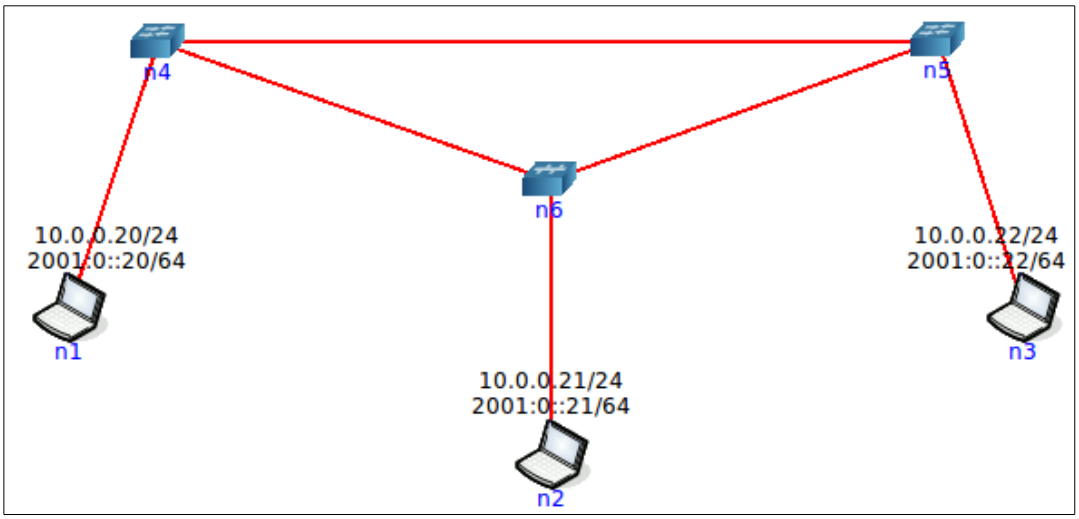


Illustration 6: Bridging loop

So what happens if a loop is introduced and the neetwork is started. Well the system pretty much comes to a halt. Why? because STP is off on each device.

Review the switches.

```
root@NTE-i386:~# brctl show

bridge name     bridge id           STP enabled   interfaces
b.4.29          8000.12f0b2666390   no            veth2.0.29
                                                  veth4.5.29
                                                  veth4.6.29
b.5.29          8000.2243edabfea4   no            veth1.0.29
                                                  veth5.4.29
                                                  veth5.6.29
b.6.29          8000.66d04659b2ec   no            veth3.0.29
                                                  veth6.4.29
                                                  veth6.5.29
```

Enable STP in each of the two switches. (Note: b.45.29 is actually a hub and therefore STP is not relevant, it forwards on all ports anyhow).

```
root@NTE-i386:~# brctl stp b.4.29 on
root@NTE-i386:~# brctl stp b.6.29 on
```

Consider a frame extracted at n2 using Wireshark.

```
Frame: 52 bytes on wire (416 bits), on interface 0
    Interface id: 0 (veth2.0.29)
    Encapsulation type: Ethernet (1)
    Arrival Time: Feb 20, 2016 07:27:05.825613000 GMT
    Epoch Time: 1455953225.825613000 seconds
    Frame Number: 1
    Frame Length: 52 bytes (416 bits)
    Capture Length: 52 bytes (416 bits)
IEEE 802.3 Ethernet
    Destination: Spanning-tree-(for-bridges)_00 (01:80:c2:00:00:00)
        Address: Spanning-tree-(for-bridges)_00 (01:80:c2:00:00:00)
        .... ..0. .... .... .... .... = LG bit: Globally unique address
        .... ...1 .... .... .... .... = IG bit: Group address
(multicast/broadcast)
    Source: 36:ac:5e:6f:00:73
        Address: 36:ac:5e:6f:00:73
        .... ..1. .... .... .... .... = LG bit: Locally administered address
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Length: 38
Logical-Link Control
    DSAP: Spanning Tree BPDU (0x42)
        0100 001. = SAP: Spanning Tree BPDU
        .... ...0 = IG Bit: Individual
    SSAP: Spanning Tree BPDU (0x42)
        0100 001. = SAP: Spanning Tree BPDU
        .... ...0 = CR Bit: Command
    Control field: U, func=UI (0x03)
        000. 00.. = Command: Unnumbered Information (0x00)
        .... ..11 = Frame type: Unnumbered frame (0x03)
Spanning Tree Protocol
    Protocol Identifier: Spanning Tree Protocol (0x0000)
    Protocol Version Identifier: Spanning Tree (0)
    BPDU Type: Configuration (0x00)
    BPDU flags: 0x00
        0... .... = Topology Change Acknowledgment: No
        .... ...0 = Topology Change: No
    Root Identifier: 32768 / 0 / 12:f0:b2:66:63:90
        Root Bridge Priority: 32768
        Root Bridge System ID Extension: 0
        Root Bridge System ID: 12:f0:b2:66:63:90 (12:f0:b2:66:63:90)
    Root Path Cost: 0
    Bridge Identifier: 32768 / 0 / 12:f0:b2:66:63:90
        Bridge Priority: 32768
        Bridge System ID Extension: 0
        Bridge System ID: 12:f0:b2:66:63:90 (12:f0:b2:66:63:90)
    Port identifier: 0x8003
    Message Age: 0
    Max Age: 20
    Hello Time: 2
    Forward Delay: 2
```

7.1 Review bridge

Now that the bridge is created review it.

```
root@NTE-i386:~# sudo brctl showmacs b.4.29
port no  mac addr            is local?    ageing timer
   2     12:f0:b2:66:63:90   yes             0.00
   3     36:ac:5e:6f:00:73   yes             0.00
   1     3e:53:b4:a5:c6:ed   yes             0.00

root@NTE-i386:~# sudo brctl showmacs b.5.29
port no  mac addr            is local?    ageing timer
   1     22:43:ed:ab:fe:a4   yes             0.00
   2     26:ae:6f:53:fc:84   yes             0.00
   1     3e:53:b4:a5:c6:ed   no              0.86
   3     f2:e1:fb:cb:c7:86   yes             0.00

root@NTE-i386:~# sudo brctl showmacs b.6.29
port no  mac addr            is local?    ageing timer
   3     3e:53:b4:a5:c6:ed   no              0.15
   1     66:d0:46:59:b2:ec   yes             0.00
   3     92:55:07:58:7a:03   yes             0.00
   2     b6:c9:d0:0b:4e:f9   yes             0.00
```

8. Switching Lab

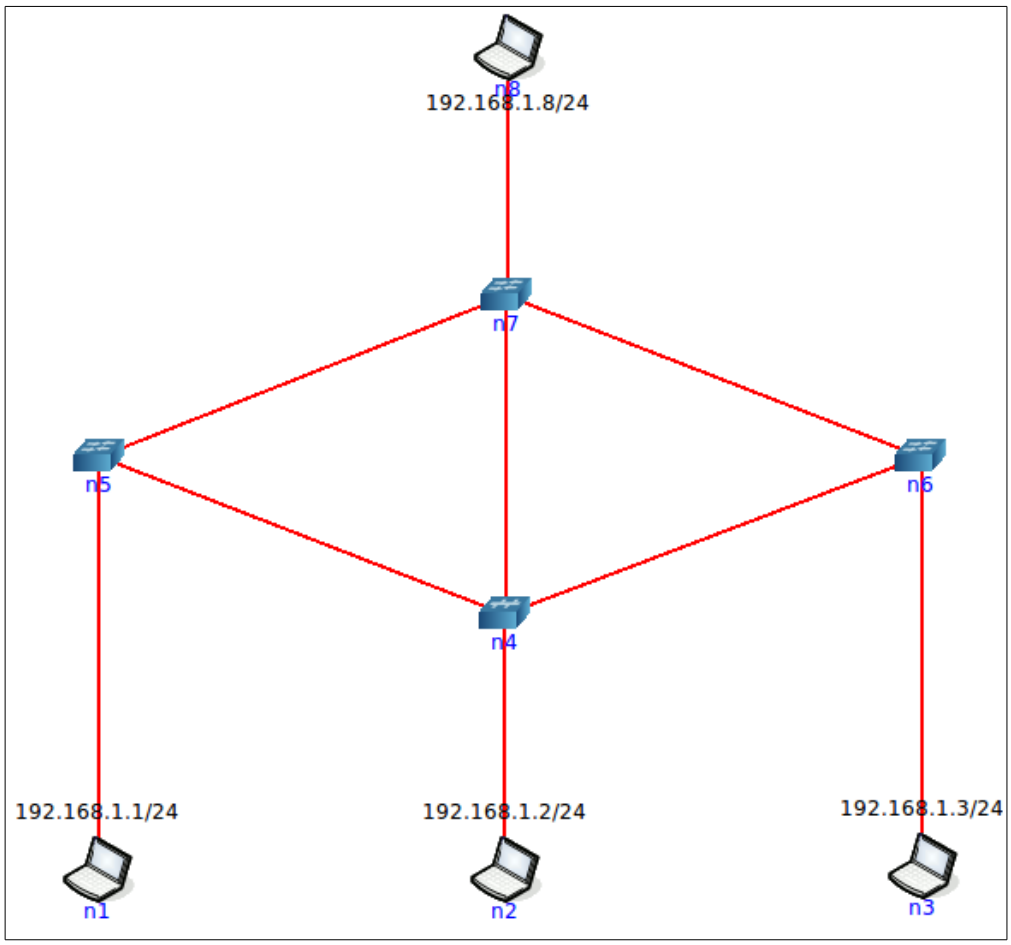


Illustration 7: Switching lab

- Build the network as shown in Illustration 7.
- Enable STP in **n5**, **n6** and **n7**.
- Ping from **n1** to **n2**, **n3** and **n8** to confirm connectivity.
- Use Tcpdump, Tshark and Wireshark to monitor activity on **n2**.

```
root@n1:/tmp/pycore.33410/n1.conf# ping -c1 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.058 ms

--- 192.168.1.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.058/0.058/0.058/0.000 ms
```

```
root@n1:/tmp/pycore.33410/n1.conf# ping -c1 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=0.071 ms

--- 192.168.1.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.071/0.071/0.071/0.000 ms


root@n1:/tmp/pycore.33410/n1.conf# ping -c1 192.168.1.8
PING 192.168.1.8 (192.168.1.8) 56(84) bytes of data.
64 bytes from 192.168.1.8: icmp_seq=1 ttl=64 time=0.065 ms

--- 192.168.1.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.065/0.065/0.065/0.000 ms
```

9. Virtual LANs (VLANs)

A virtual LAN, commonly known as a VLAN, is a logically segmented network mapped over physical hardware. The IEEE 802.1q standard is the predominant protocol.

Early VLANs where often configured to reduce the size of the collision domain in a large single Ethernet segment to improve performance. When Ethernet switches made this a non-issue (because they have no collision domain), attention turned to reducing the size of the broadcast domain at the MAC layer. Another purpose of a virtual network is to restrict access to network resources without regard to physical topology of the network, although the strength of this method is debatable.

VLANs operate at layer 2 of the OSI model. Although often a VLAN is configured to map directly to an IP network, or subnet, which gives the appearance it is involved in layer 3.

Switch to switch links and switch to router links are called trunks. A router serves as the backbone for traffic going across different VLANs.

9.1 Removing the Physical Boundaries

Conceptually, VLANs provide greater segmentation and organisational flexibility. VLAN technology allows network managers to group switch ports and users connected to them into logically defined communities of interest. These groupings can be co-workers within the same department, a cross-functional product team, or diverse users sharing the same network application or software (such as LibreOffice users). Grouping these ports and users into communities of interest, referred to as VLAN organisations, can be accomplished within a single switch, or more powerfully, between connected switches within the enterprise. By grouping ports and users together across multiple switches, VLANs can span single building infrastructures, interconnected buildings, or even Wide Area Networks (WAN). VLANs completely remove the physical constraints of workgroup communications across the enterprise.

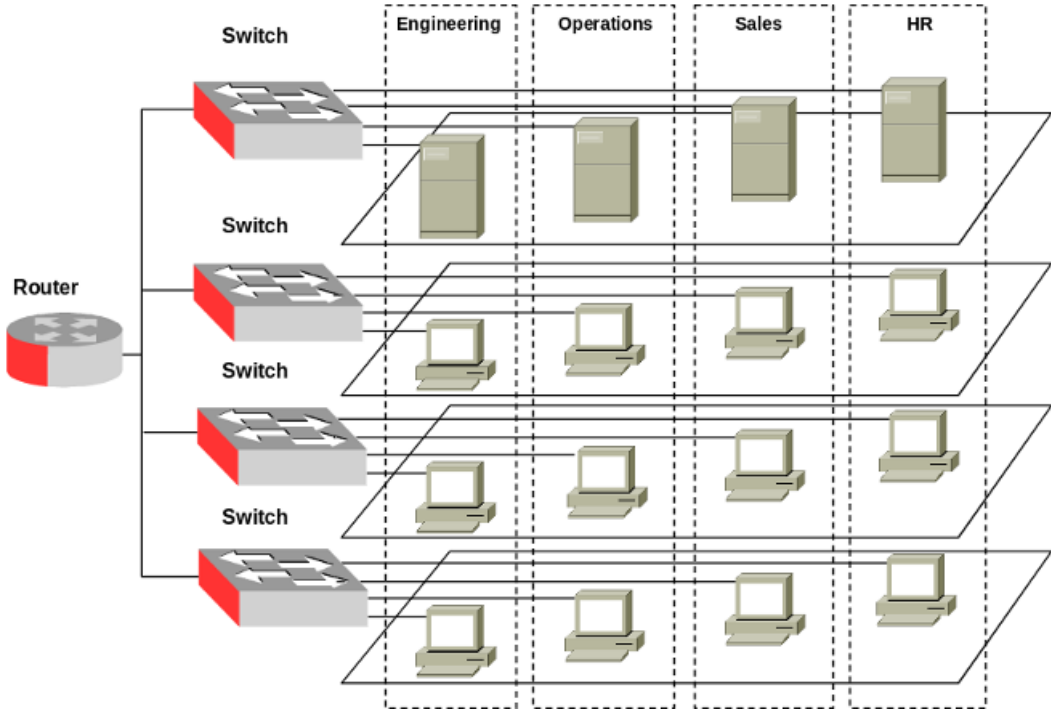
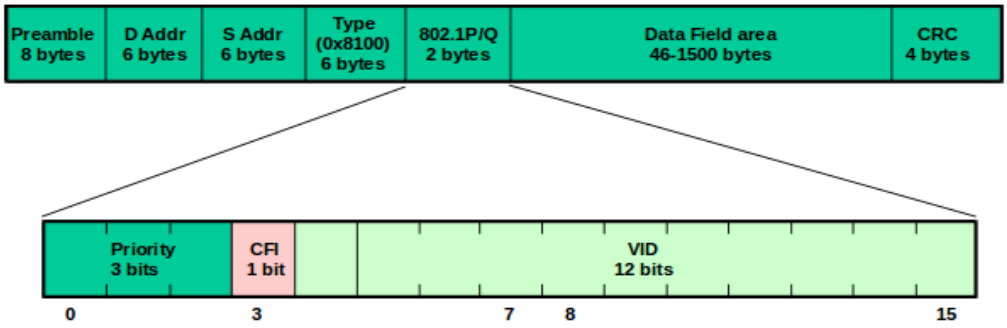


Illustration 8: Virtual LAN

VLANs provide the ability for any organisation to be physically dispersed throughout the company while maintaining its group identity. For example, engineering personnel can be located on the manufacturing floor, in the research and development centre, in the Professional Services demonstration centre, and in the corporate offices, while at the same time all members reside on the same virtual network, sharing traffic only with each other. The graphic above illustrates a typical VLAN architecture that places these employees closer to their assigned areas of management and the people with whom they interact, while maintaining communication integrity within their respective organisation. Today's VLANs better match the way that companies are organised, and allow network managers to more closely align the network to the way that employees work and communicate.

9.2 IEEE 802.1P/Q



CRC - Cyclical Redundancy Check
CFI - Canonical Format Indicator is a single-bit flag, always set to zero for Ethernet switches.
VID - VLAN Identifier 2^{12} = 4096 VLANs. VID 0 is used to identify priority frames and value 4095 (FFF) is reserved, maximum VLAN configurations are 4,094.

Illustration 9: IEEE 802.1P/Q

The IEEE 802.1Q specification is the standard method for inserting VLAN membership information into Ethernet frames. A tag field containing VLAN information can be inserted into an Ethernet frame. If a port has an IEEE 802.1Q compliant device attached (such as another switch), these tagged frames can carry VLAN membership information between switches, thus letting a VLAN span multiple switches.

Note that VLAN functionality is shared in the IEEE 802.1 two bytes 3 priority bits. These 3 bits define 8 classes, the highest priority is 7 for say network-critical traffic such as routing, values 5 and 6 for say delay-sensitive applications such as video and VoIP. The 0 value is used as a best-effort default, invoked automatically when no other value has been set.

The priority function of IEEE 802.1 is known as IEEE 802.1P and the VLAN function as IEEE 802.1Q while combined they are referred to as IEEE 802.1P/Q.

9.2.1 IEEE 802.1Q

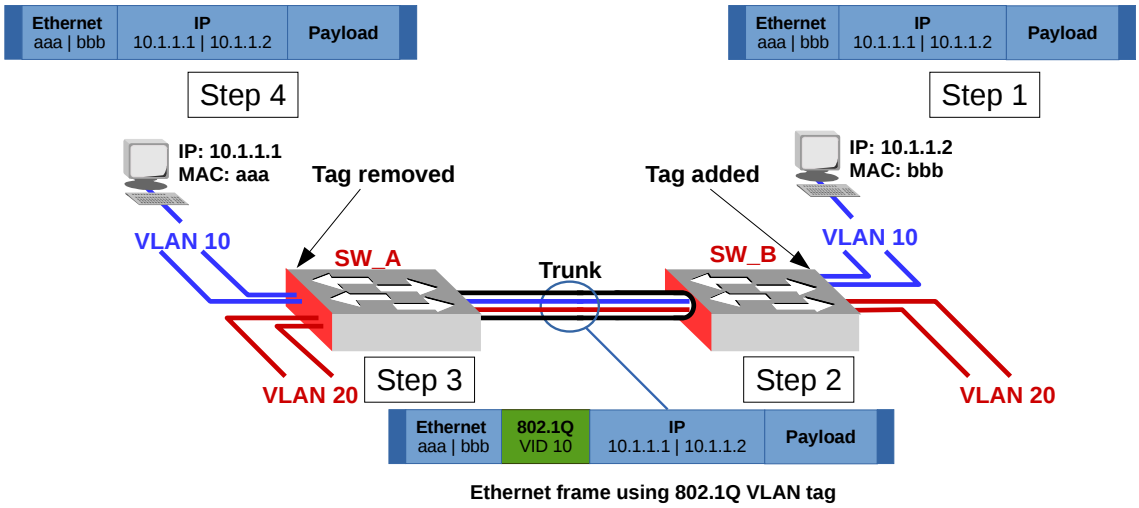


Illustration 10: IEEE 802.1Q

The diagram above shows a frame traversing the VLAN. Step 1 the host with MAC: *bbb* puts a frame on the wire for MAC: *aaa*. The Switch *SW_B* determines the frame belongs to *VLAN 10*, either by the protocol within the frame or in this example the port it is received on. In Step 2 the *SW_B* switch then encapsulates the frame with an IEEE 802.1Q tag and a Frame Check Sequence (FCS), this tag is then used to identify the VLAN the frame is from on all IEEE 802.1Q enabled switches. The tagged frame is then passed on the trunk to the *SW_A* switch. In Step 3 the *SW_A* determines the frame is for *VLAN 10*, removes the tag and puts the frame out the ports associated with *VLAN 10*. Step 4 The workstation with the MAC: *aaa* receives an untagged frame.

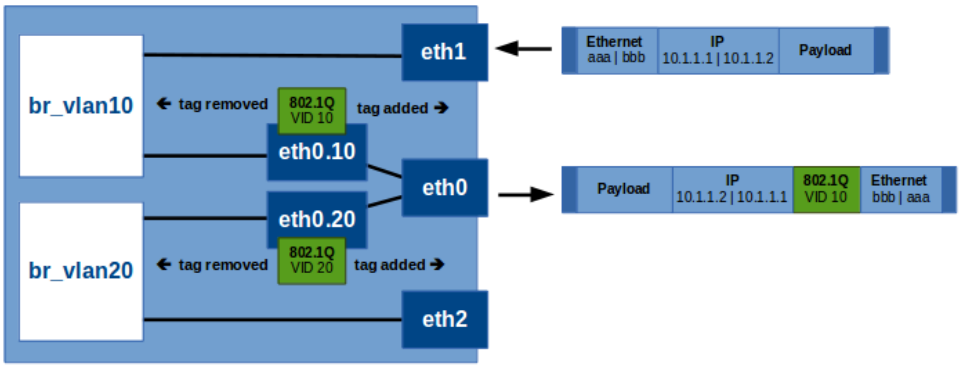


Illustration 11: VLAN tagging

Now dissecting further what happens within the switch. Each VLAN has a bridge configured for it. The interface considered to be the trunk between the switches has sub-interfaces configured, one for each VLAN. These sub-interfaces perform the tagging and untagging. The sub-interfaces are added to their respective bridges. Each other port considered to be an access port is added to the bridge associated with the VLAN for that port. In the example therefore `eth1` is added to the bridge *`br_vlan10`* and *`eth2`* to *`br_vlan20`*.

A frame arrives as `eth1` and is passed to the bridge *`br_vlan10`* and as a result is forwarded to the sub-interface `eth0.10` where an IEEE 802.1Q tag is added to the frame and it is forwarded to the physical interface eth0. The frame leaving `eth0` is therefore tagged.

Here is an example of a frame captured on the wire on a trunk between two switches using IEEE 802.1Q. Note the Ethernet type field has a value of `0x8100` indicating that the next field is IEEE 802.1Q VLAN. This field contains the value *`000000001010 (10)`* which is the VLAN tag and it follows with a type field of `0x0800` indicating that the next field is the IP header.

```
Frame: 102 bytes on wire (816 bits)
Ethernet II, Src: d4:ca:6d:61:dd:89, Dst: 00:12:3f:dc:ab:47
    Destination: 00:12:3f:dc:ab:47
    Source: d4:ca:6d:61:dd:89
    Type: 802.1Q Virtual LAN (0x8100)
802.1Q Virtual LAN, PRI: 0, CFI: 0, ID: 10
    000. .... .... .... = Priority: Best Effort (default) (0)
    ...0 .... .... .... = CFI: Canonical (0)
    .... 0000 0000 1010 = ID: 10
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.10.10.40, Dst: 10.10.10.30
Internet Control Message Protocol
```

9.2.2 IEEE 802.1ad

One of the difficulties presented by IEEE 802.1Q is the fact that tags cannot be *stacked*. Imagine a service provider wants to use VLANs to separate services to different customers. As the provider used VLAN tags for that purpose it prevents the customers using VLANs themselves as IEEE 802.1Q does not support VLANs within VLANs. IEEE 802.1ad is an amendment to the IEEE 802.1Q VLAN standard. It provides for the stacking of VLANs within VLANs which is called names such as provider stacking, stacked VLANs, Q-in-Q or simply QinQ. IEEE 802.ad allows for multiple VLAN headers to be inserted into a single frame an essential capability for implementing Metro Ethernet. QinQ allows multiple VLAN tags in an Ethernet frame; together these tags constitute a tag stack.

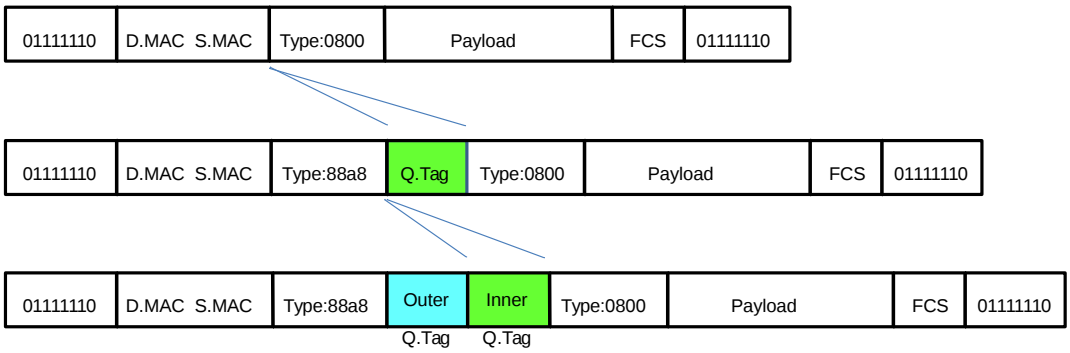


Illustration 12: IEEE 802.1ad

This first example demonstrates the use of IEEE 802.1ad instead of IEEE 802.1Q where there is no stacking of VLANs. Here the Ethernet type field contains *0x88a8* such that the next field is treated as IEEE 802.1ad. Like the earlier example for IEEE 802.1Q this field contains a VLAN ID of *000000001010 (10)* and a type field of `0x0800` to indicate that the next field is the IP header.

```
Frame: 102 bytes on wire (816 bits)
Ethernet II, Src: d4:ca:6d:61:dd:89, Dst: 00:12:3f:dc:ab:47
    Destination: 00:12:3f:dc:ab:47
    Source: d4:ca:6d:61:dd:89
    Type: 802.1ad Provider Bridge (Q-in-Q) (0x88a8)
IEEE 802.1ad, ID: 10
    000. .... .... .... = Priority: 0
    ...0 .... .... .... = DEI: 0
    .... 0000 0000 1010 = ID: 10
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.10.10.40, Dst: 10.10.10.30
Internet Control Message Protocol
```

10. Provider tagging

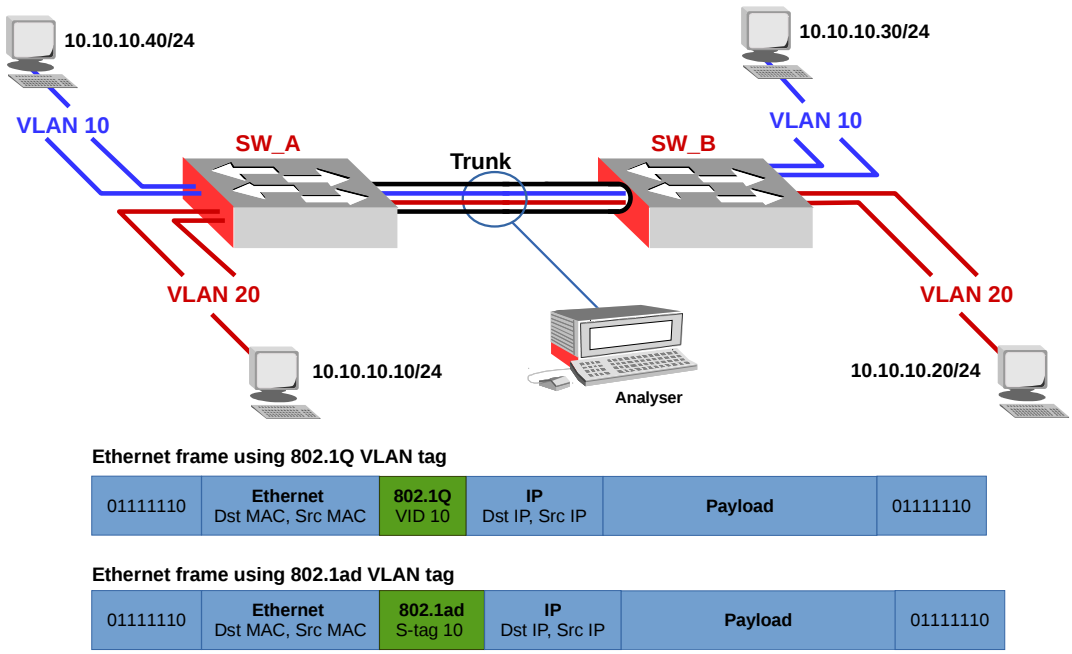


Illustration 13: Provider tagging

Considering the graphic the traffic in the trunks will be treated by *ISP_1* and *ISP_2* as access ports despite they containing VLAN tags already. In fact *ISP_1* and *ISP_2* ignore these tags as they are customer tags (C-tags).

Before forwarding to the other ISP switch each switch adds a provider tag (S-tag) of `1001`. Thus the C-tag is stacked inside the S-tag from the provider. This can be seen by considering the frame capture from the wire between *ISP_1* and *ISP_2* below. In this packet a customer IEEE 802.1Q VLAN is outer labelled with an ISP IEEE 802.1ad (Q-in-Q) S-tag of *001111101001 (1001)*. The Ethernet type field indicates `0x88a8` the next header containing an IEEE 802.1ad tag. This headers type field indicates that the next header is `0x8100` IEEE 802.1Q. This headers type field in turn contains a type field of `0x0800` indicating the next header is the IP header. So in this example a customer IEEE 802.1Q tag is stacked by an IEEE 802.1ad S-tag.

```
Frame: 106 bytes on wire (848 bits)
Ethernet II, Src: d4:ca:6d:61:dd:89, Dst: 00:12:3f:dc:ab:47
    Destination: 00:12:3f:dc:ab:47
    Source: d4:ca:6d:61:dd:89
    Type: 802.1ad Provider Bridge (Q-in-Q) (0x88a8)
IEEE 802.1ad, ID: 1001
    000. .... .... .... = Priority: 0
    ...0 .... .... .... = DEI: 0
    .... 0011 1110 1001 = ID: 1001
    Type: 802.1Q Virtual LAN (0x8100)
802.1Q Virtual LAN, PRI: 0, CFI: 0, ID: 10
    000. .... .... .... = Priority: Best Effort (default) (0)
    ...0 .... .... .... = CFI: Canonical (0)
    .... 0000 0000 1010 = ID: 10
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.10.10.40, Dst: 10.10.10.30
Internet Control Message Protocol
```

Now consider the frame below where the customer tag is also IEEE 802.1ad. The Ethernet type field is `0x88a8` indicating the next header is IEEE 802.1ad Q-in-Q. In this header two tags can be seen, a provider S-tag of *`001111101001 (1001)`* with a customer C-tag of *`000000001010 (10)`*. This headers type field is `0x0800` indicating the next header is the IP header.

```
Frame: 106 bytes on wire (848 bits)
Ethernet II, Src: d4:ca:6d:61:dd:89, Dst: 00:12:3f:dc:ab:47
    Destination: 00:12:3f:dc:ab:47
    Source: d4:ca:6d:61:dd:89
    Type: 802.1ad Provider Bridge (Q-in-Q) (0x88a8)
IEEE 802.1ad, S-VID: 1001, C-VID: 10
    000. .... .... .... = Priority: 0
    ...0 .... .... .... = DEI: 0
    .... 0011 1110 1001 = ID: 1001
    000. .... .... .... = Priority: 0
    ...0 .... .... .... = DEI: 0
    .... 0000 0000 1010 = ID: 10
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.10.10.40, Dst: 10.10.10.30
Internet Control Message Protocol
```

This packet shows a customer IEEE 802.1ad (Q-in-Q) label as an inner C-tag *000000010100 (20)* which is also outer labelled with an ISP IEEE 802.1ad (Q-in-Q) S-tag *001111101001 (1001)*.

```
Frame: 106 bytes on wire (848 bits)
Ethernet II, Src: d4:ca:6d:61:dd:89, Dst: 00:12:3f:dc:ab:47
    Destination: 00:12:3f:dc:ab:47
    Source: d4:ca:6d:61:dd:89
    Type: 802.1ad Provider Bridge (Q-in-Q) (0x88a8)
IEEE 802.1ad, S-VID: 1001, C-VID: 20
    000. .... .... .... = Priority: 0
    ...0 .... .... .... = DEI: 0
    .... 0011 1110 1001 = ID: 1001
    000. .... .... .... = Priority: 0
    ...0 .... .... .... = DEI: 0
    .... 0000 0001 0100 = ID: 20
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.10.10.40, Dst: 10.10.10.30
Internet Control Message Protocol
```

11. VLANs on GNU/Linux

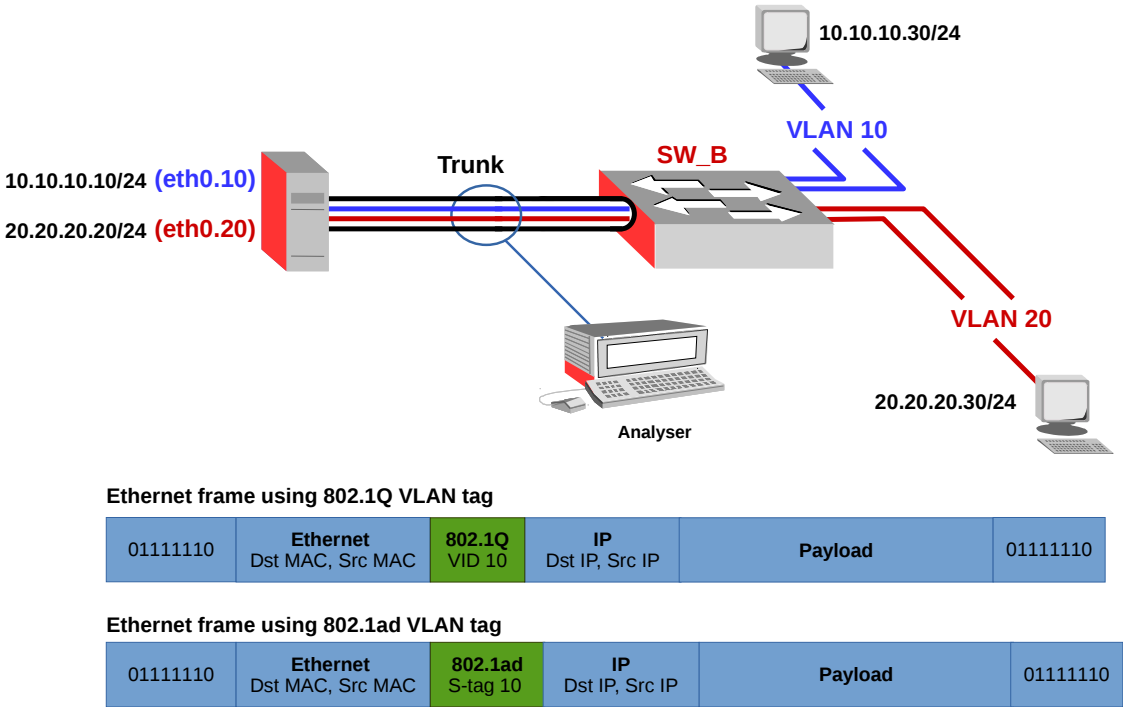


Illustration 14: VLANs on Linux

iproute2 supports IEEE 802.1Q VLAN and IEEE 802.1ad VLAN Stacking. IEEE 802.1Q or IEEE 802.1ad traffic received on the eth0 interface will have the VLAN tag removed and the frame passed to the VLAN interface. Traffic passing out the sub-interface will have the IEEE 802.1Q or IEEE 802.1ad tag added. Create the sub-interfaces with the following commands. These create sub-interfaces for VLAN ID *10* and VLAN ID *20* on the *eth0* interface and gives them labels of *eth0.10* and *eth0.20*.

11.1 VLAN Logical diagram

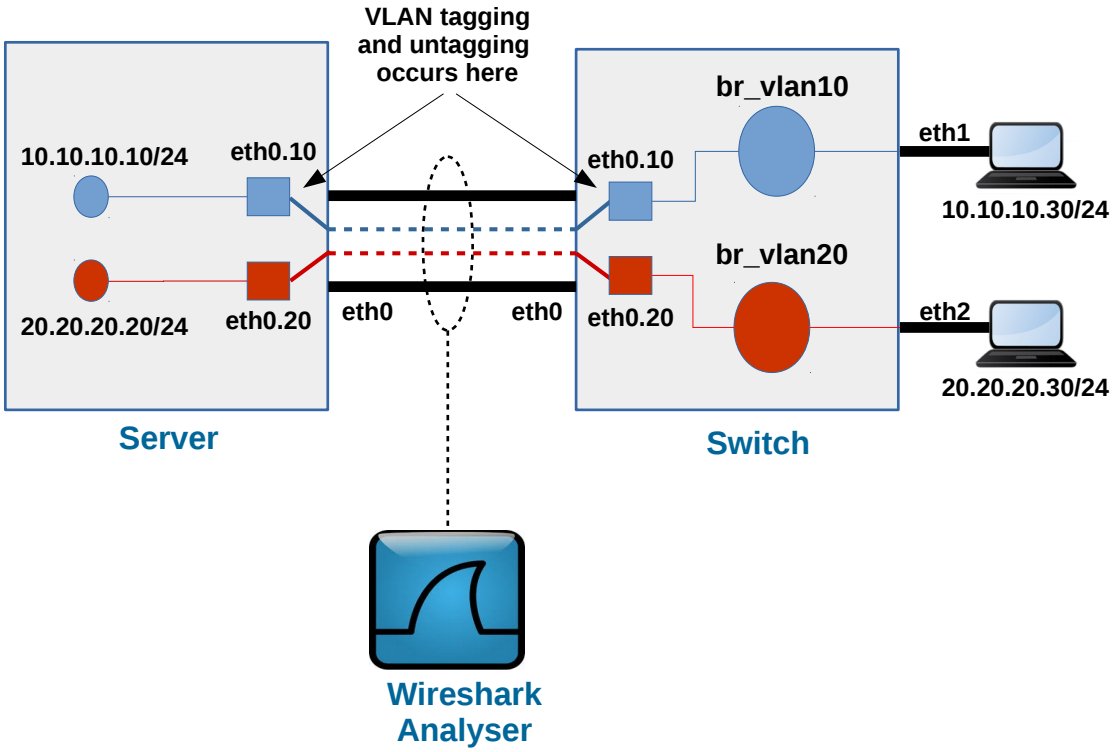


Illustration 15: VLAN logical diagram

Consider the Illustration 15 above. On the left is a server with two sub-interfaces configured on *eth0*, namely *eth0.10* and *eth0.20*. On these sub-interfaces the IP addresses 10.10.10.10/24 and 20.20.20.20/24 are configured respectfully.

On the right is the VLAN switch. It too has two sub-interfaces configured on *eth0*, namely *eth0.10* and *eth0.20*. It also has two bridges created *br_vlan10* and *br_vlan20*. Within *br_vlan10* the sub-interface *eth0.10* and the physical interface *eth1* while in *br_vlan20* is the sub-interface *eth0.20* and the physical interface *eth2*.

Now trace a ping from the workstation 10.10.10.30 to 10.10.10.10 on the server. The ICMP request is passed to interface *eth1* on the switch, it passes through *br_vlan10* and on to sub-interface *eth0.10* where an 802.1q tag of *10* is inserted between the Ethernet and IP headers in the frame. This tag differentiates the frame from frames in the other VLAN. The tagged frame enters *eth0* on the server and due the the 802.1q tag of *10* is passed to sub-interface *eth0.10* where the tag is removed. The frame is passed to the internal layer 3 interface within the server that has IP address 10.10.10.10.

11.2 VLAN Example

The following network was built as shown, except *vlan2* was selected as a server. This is because the detault ethernet bridge in NTE does not support *iproute2*. Right click on the server icon and select ***Configure***. Remove the default IP addresses and change the icon to *lanswitch.gif*. Additionally select ***View > Show > Interface names*** so the interfaces can be associated with their connections.

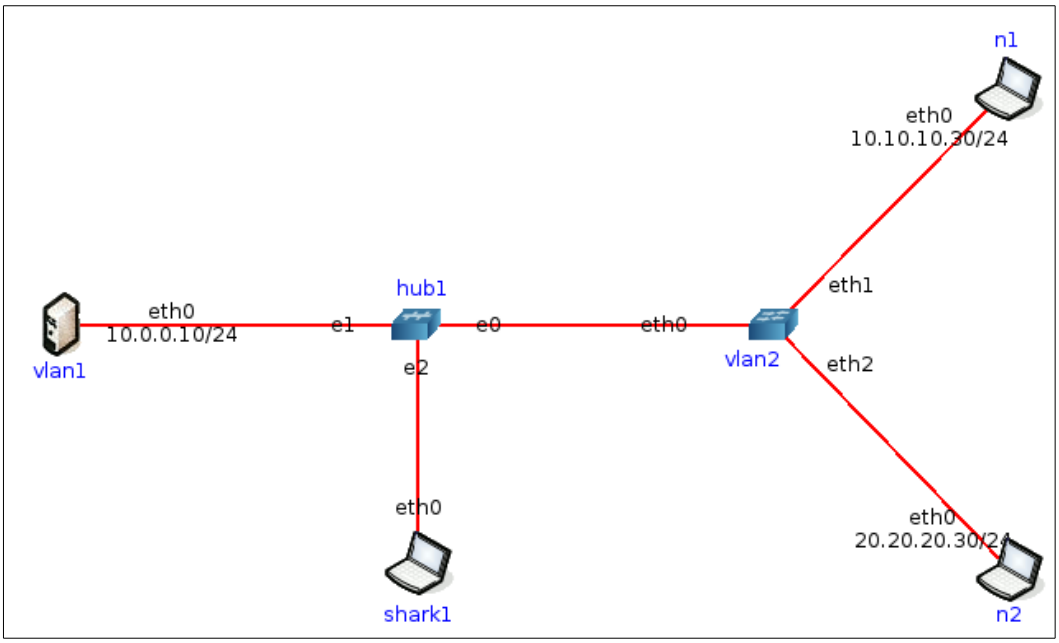


Illustration 16: Example VLANs

11.2.1 Load 802.1q kernel module in NTE

Confirm that the 8021q kernel module is loaded and if not then load it.

```
root@NTE-i386:~# lsmod |grep 8021q
root@NTE-i386:~# modprobe 8021q
root@NTE-i386:~# lsmod |grep 8021q
8021q          18824  0
garp           13025  1 8021q
```

11.2.2 Configure s-vlan1 for VLANs

Check the interfaces on the Server.

```
root@vlan1:~# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000
    link/ether 00:00:00:aa:00:00 brd ff:ff:ff:ff:ff:ff
```

Add sub-interfaces for each VLAN expected on the physical eth0 interface.

```
root@vlan1:~# ip link add link eth0 name eth0.10 type vlan id 10
root@vlan1:~# ip link add link eth0 name eth0.20 type vlan id 20

root@vlan1:~# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000
    link/ether 00:00:00:aa:00:00 brd ff:ff:ff:ff:ff:ff
3: eth0.10@eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default link/ether 00:00:00:aa:00:00 brd ff:ff:ff:ff:ff:ff
4: eth0.20@eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default link/ether 00:00:00:aa:00:00 brd ff:ff:ff:ff:ff:ff
```

Add IP addresses to the sub-interfaces.

```
root@vlan1:~# ip addr add 10.10.10.10/24 dev eth0.10
root@vlan1:~# ip addr add 20.20.20.20/24 dev eth0.20
```

Bring up the interface and its new sub-interfaces.

```
root@vlan1:~# ip link set dev eth0 up
root@vlan1:~# ip link set dev eth0.10 up
root@vlan1:~# ip link set dev eth0.20 up
```

Confirm sub-interfaces are up.

```
root@vlan1:~# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000
    link/ether 00:00:00:aa:00:00 brd ff:ff:ff:ff:ff:ff
3: eth0.10@eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP mode
DEFAULT group default link/ether 00:00:00:aa:00:00 brd ff:ff:ff:ff:ff:ff
4: eth0.20@eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP mode
DEFAULT group default link/ether 00:00:00:aa:00:00 brd ff:ff:ff:ff:ff:ff
```

11.2.3 Configure the switch s-vlan2

Check the interfaces on *vlan2*.

```
root@vlan2:~# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000 link/ether 00:00:00:aa:00:02 brd
ff:ff:ff:ff:ff:ff
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000 link/ether 00:00:00:aa:00:03 brd
ff:ff:ff:ff:ff:ff
4: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000 link/ether 00:00:00:aa:00:05 brd
ff:ff:ff:ff:ff:ff
```

Add sub-interfaces on the *vlan2* interface facing *vlan1*.

```
root@vlan2:~# ip link add link eth0 name eth0.10 type vlan id 10
root@vlan2:~# ip link add link eth0 name eth0.20 type vlan id 20
```

Confirm the sub-interfaces were added.

```
root@vlan2:~# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0.10@eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default link/ether 00:00:00:aa:00:02 brd ff:ff:ff:ff:ff:ff
3: eth0.20@eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default link/ether 00:00:00:aa:00:02 brd ff:ff:ff:ff:ff:ff
4: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000 link/ether 00:00:00:aa:00:02 brd
ff:ff:ff:ff:ff:ff
5: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000 link/ether 00:00:00:aa:00:03 brd
ff:ff:ff:ff:ff:ff
6: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000 link/ether 00:00:00:aa:00:05 brd
ff:ff:ff:ff:ff:ff
```

Bring the sub-interfaces to a state of UP.

```
root@vlan2:~# ip link set dev eth0.10 up
root@vlan2:~# ip link set dev eth0.20 up
```

Confirm the sub-interfaces have come UP.

```
root@vlan2:~# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0.10@eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP mode
DEFAULT group default link/ether 00:00:00:aa:00:02 brd ff:ff:ff:ff:ff:ff
3: eth0.20@eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP mode
DEFAULT group default link/ether 00:00:00:aa:00:02 brd ff:ff:ff:ff:ff:ff
4: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000 link/ether 00:00:00:aa:00:02 brd
ff:ff:ff:ff:ff:ff
5: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000 link/ether 00:00:00:aa:00:03 brd
ff:ff:ff:ff:ff:ff
6: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000 link/ether 00:00:00:aa:00:05 brd
ff:ff:ff:ff:ff:ff
```

Create bridges for each VLAN.

```
root@vlan2:~# brctl addbr br_vlan10
root@vlan2:~# brctl addbr br_vlan20
```

Add the sub-interfaces and the appropriate interfaces on *vlan2* to their respective bridges.

```
root@vlan2:~# brctl addif br_vlan10 eth0.10 eth1
root@vlan2:~# brctl addif br_vlan20 eth0.20 eth2
```

Bring the bridges to a state of UP.

```
root@vlan2:~# ip link set dev br_vlan10 up
root@vlan2:~# ip link set dev br_vlan20 up
```

Confirm that the interfaces, sub-interfaces and bridges are in a state of UP.

```
root@vlan2:~# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0.10@eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP mode
DEFAULT group default link/ether 00:00:00:aa:00:02 brd ff:ff:ff:ff:ff:ff
3: eth0.20@eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP mode
DEFAULT group default link/ether 00:00:00:aa:00:02 brd ff:ff:ff:ff:ff:ff
4: br_vlan10: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP mode DEFAULT
group default link/ether 00:00:00:aa:00:02 brd ff:ff:ff:ff:ff:ff
5: br_vlan20: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP mode DEFAULT
group default link/ether 00:00:00:aa:00:02 brd ff:ff:ff:ff:ff:ff
6: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000 link/ether 00:00:00:aa:00:02 brd
ff:ff:ff:ff:ff:ff
7: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
mode DEFAULT group default qlen 1000 link/ether 00:00:00:aa:00:03 brd
ff:ff:ff:ff:ff:ff
8: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
```

11.2.4 Test the VLAN

Establish a ping from *n2* to *20.20.20.20* which is on *vlan20@eth0* on host *vlan1*. Capture the traffic on *shark1*.

```
Frame: 102 bytes on wire (816 bits), on interface 0
Ethernet II, Src: 00:00:00_aa:00:06, Dst: 00:00:00_aa:00:00
    Destination: 00:00:00_aa:00:00
    Source: 00:00:00_aa:00:06
    Type: 802.1Q Virtual LAN (0x8100)
802.1Q Virtual LAN, PRI: 0, CFI: 0, ID: 20
    000. .... .... .... = Priority: Best Effort (default) (0)
    ...0 .... .... .... = CFI: Canonical (0)
    .... 0000 0001 0100 = ID: 20
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 20.20.20.30, Dst: 20.20.20.20
    Version: 4
    Header Length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-
ECT)
    Total Length: 84
    Identification: 0x5b1c (23324)
    Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
    Header checksum: 0x8f33
    Source: 20.20.20.30
    Destination: 20.20.20.20
Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0xf83a [correct]
    Identifier (BE): 20 (0x0014)
    Identifier (LE): 5120 (0x1400)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
    [Response frame: 2]
    Timestamp from icmp data: Feb 20, 2016 08:55:17.339026000 GMT
    Data (48 bytes)
```

11.3 IEEE 802.1ad support on GNU/Linux

Support for IEEE 802.1ad was incorporated in the GNU/Linux kernel from Kernel version 3.10. Check the kernel version of your system and if less that 3.10, download the latest stable kernel, compile and use it. Also check the version of `iproute*` installed, it needs to be a version 3.10 or higher.

```
# uname -r
3.16.0-4-586

# dpkg -l iproute*

root@NTE-i386:~# dpkg -l iproute*
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/trig-aWait/Trig-pend
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name           Version      Architecture Description
+++-==============-============-============-=================================
ii  iproute        1:3.16.0-2   all          transitional dummy package for ip
ii  iproute2       3.16.0-2     i386         networking and traffic control to
un  iproute2-doc   <none>       <none>       (no description available)
```

Configure interfaces as shown already with the addition of `proto 802.1ad` when creating the sub-interfaces.

```
# ip link add link eth0 name eth0.10 type vlan proto 802.1ad id 10
# ip link add link eth0 name eth0.20 type vlan proto 802.1ad id 20
# ip addr add 10.10.10.10/24 dev eth0.10
# ip addr add 20.20.20.20/24 dev eth0.20
# ip link set dev eth0 up
# ip link set dev eth0.10 up
# ip link set dev eth0.20 up
```

Monitor the trunk link and notice that the GNU/Linux workstation is now terminating directly in the IEEE 802.1ad C-tag interfaces on the `eth0` sub-interfaces.

```
Frame: 102 bytes on wire (816 bits)
Ethernet II, Src: 00:12:3f:dc:ab:47, Dst: d4:ca:6d:61:dd:89
    Destination: d4:ca:6d:61:dd:89
    Source: 00:12:3f:dc:ab:47
    Type: 802.1ad Provider Bridge (Q-in-Q) (0x88a8)
IEEE 802.1ad, ID: 10
    000. .... .... .... = Priority: 0
    ...0 .... .... .... = DEI: 0
    .... 0000 0000 1010 = ID: 10
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.10.10.10, Dst: 10.10.10.30
Internet Control Message Protocol
```

11.4 IEEE 802.1ad support on GNU/Linux as a switch

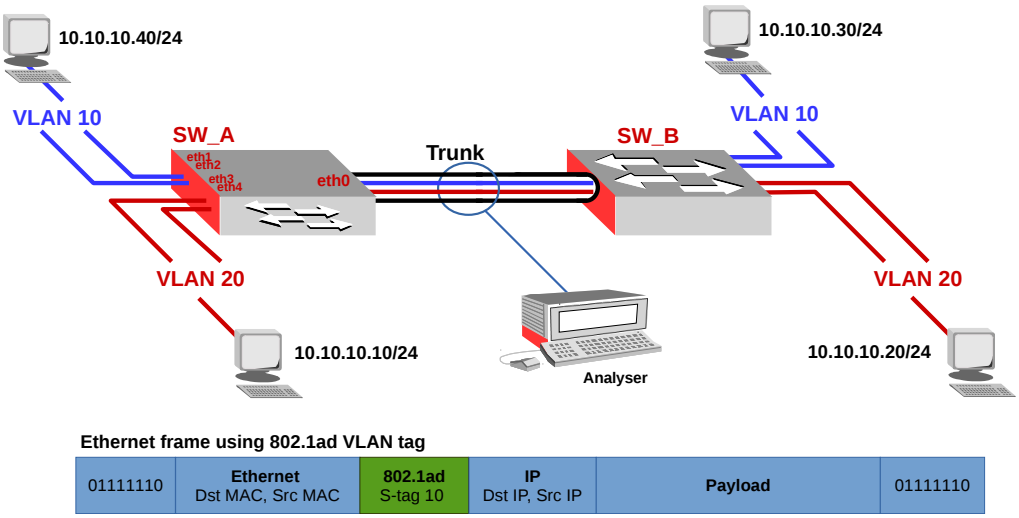


Illustration 17: IEEE 802.1ad on Linux

In this case GNU/Linux workstation will operate as a bridge with a VLAN interface.

Create the VLAN subinterfaces to deal with the incoming trunk interface containing the VLANs on physical interface *eth0*.

```
# ip link add link eth0 name eth0.10 type vlan proto 802.1ad id 10
# ip link add link eth0 name eth0.20 type vlan proto 802.1ad id 20
# ip link set dev eth0 up
# ip link set dev eth0.10 up
# ip link set dev eth0.20 up
```

Bring up the interfaces that connect to the LANs.

```
# ip link set dev eth1 up
# ip link set dev eth2 up
# ip link set dev eth3 up
# ip link set dev eth4 up
```

Create bridges to link the VLANs to their appropriate interfaces.

```
# brctl addbr br_vlan_10
# brctl addbr br_vlan_20
```

Assign interfaces to the various bridges.

```
# brctl addif br_vlan_10 eth0.10 eth1 eth2
# brctl addif br_vlan_20 eth0.20 eth3 eth4
```

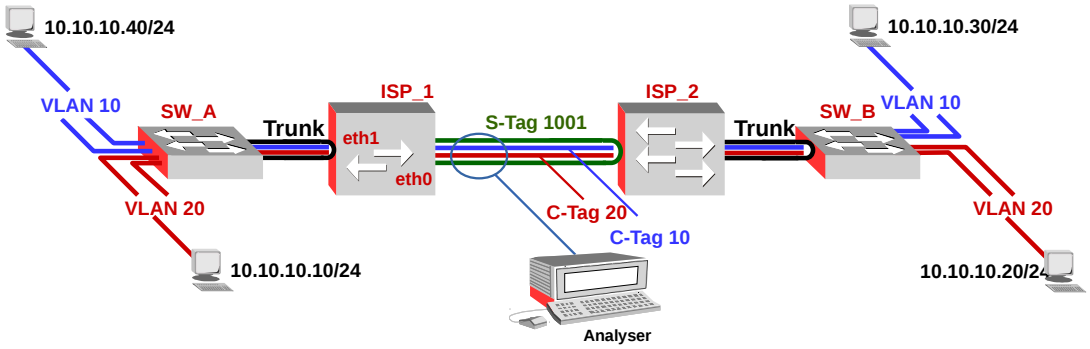
Bring up the bridges.

```
# ip link set dev br_vlan_10 up
# ip link set dev br_vlan_20 up
```

Review the packets on the wire.

```
Frame: 74 bytes on wire (592 bits)
Ethernet II, Src: d4:ca:6d:61:dd:89, Dst: 00:0c:42:8b:73:e4
    Destination: 00:0c:42:8b:73:e4
    Source: d4:ca:6d:61:dd:89
    Type: 802.1Q Virtual LAN (0x8100)
802.1Q Virtual LAN, PRI: 0, CFI: 0, ID: 10
    000. .... .... .... = Priority: Best Effort (default) (0)
    ...0 .... .... .... = CFI: Canonical (0)
    .... 0000 0000 1010 = ID: 10
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.10.10.30, Dst: 10.10.10.40
Internet Control Message Protocol
```

12. GNU/Linux as a Service Provider bridge



Ethernet frame using 802.1ad outer S-tag and 802.1Q inner VLAN tag

01111110	**Ethernet** Dst MAC, Src MAC	**802.1ad** S-tag 1001	**802.1Q** VID 10	**IP** Dst IP, Src IP	**Payload**	01111110

Ethernet frame using 802.1ad outer S-tag and inner C-tag

01111110	**Ethernet** Dst MAC, Src MAC	**802.1ad** S-tag 1001, C-tag 10	**IP** Dst IP, Src IP	**Payload**	01111110

Illustration 18: Service provider bridge

Create a sub-interface on `eth0` to handle the VLAN S-tag *1001* and bring the physical and sub-interface up.

```
# ip link add link eth0 name eth0.1001 type vlan proto 802.1ad id 1001
# ip link set dev eth0 up
# ip link set dev eth0.1001 up
```

Bring up the `eth1` interface which will be connected to the trunk from *SW_A*.

```
# ip link set dev eth1 up
```

Create a bridge *br_vlan_1001* and put the *eth0.1001* sub-interface and *eth1* into it. Then bring the bridge up.

```
# brctl addbr br_vlan_1001
# brctl addif br_vlan_1001 eth0.1001 eth1
# ip link set dev br_vlan_1001 up
```

Now monitor the traffic on the wire between the provider switches. Note the double tag with an S-tag of *1001* and a C-tag of *10*.

```
Frame: 78 bytes on wire (624 bits)
Ethernet II, Src: d4:ca:6d:61:dd:89, Dst: 00:0c:42:8b:73:e4
    Destination: 00:0c:42:8b:73:e4
    Source: d4:ca:6d:61:dd:89
    Type: 802.1Q Virtual LAN (0x8100)
802.1Q Virtual LAN, PRI: 0, CFI: 0, ID: 1001
    000. .... .... .... = Priority: Best Effort (default) (0)
    ...0 .... .... .... = CFI: Canonical (0)
    .... 0011 1110 1001 = ID: 1001
    Type: 802.1ad Provider Bridge (Q-in-Q) (0x88a8)
IEEE 802.1ad, ID: 10
    000. .... .... .... = Priority: 0
    ...0 .... .... .... = DEI: 0
    .... 0000 0000 1010 = ID: 10
    Type: IP (0x0800)
Internet Protocol Version 4, Src: 10.10.10.30, Dst: 10.10.10.40
Internet Control Message Protocol
```

13. VLAN Lab

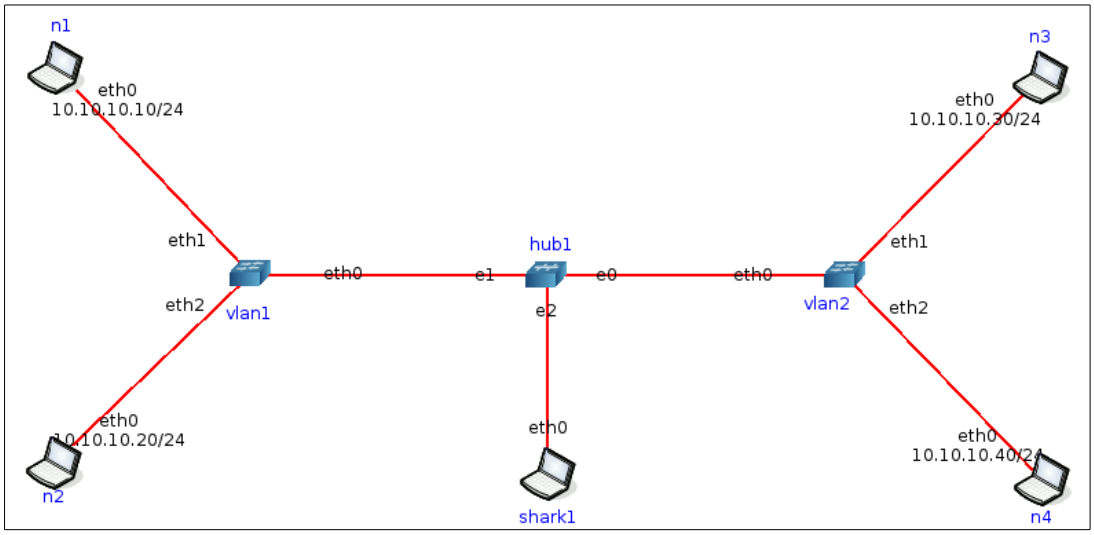


Illustration 19: VLAN Lab

Name	Device	Interface	IPv4	IPv6	Connected to
vlan1	VLAN Switch	eth0			hub1
		vlan10@eth0			vlan2
		vlan20@eth0			vlan2
		eth1			n1
		eth2			n2
vlan2	VLAN Switch	eth0			hub1
		vlan10@eth0			vlan1
		vlan20@eth0			vlan1
		eth1			n3
		eth2			n4
hub1	hub	e0			vlan1
		e1			vlan2
		e2			shark1
n1	host	eth0	10.10.10.70/24	2001:1::70/64	vlan1
n2	host	eth0	10.10.10.80/24	2001:2::80/64	vlan1
n4	host	eth0	10.10.10.40/24	2001:1::40/64	vlan2
n3	host	eth0	10.10.10.50/24	2001:1::50/64	vlan2
shark1	analyser	eth0			hub1

Build the network in Illustration 19 and detailed in the associated table.

Note: that switches vlan1 and vlan2 are servers that have their icons changes to lanswitch.gif and their IP addresses removed. This is to have the required iproute2 support as discussed earlier.

From the hosts *n1, n2, n3* and *n4* use ping and ping6 to check for connectivity between hosts and fill out the truth table below with a checkmark for success and an X for failure.

Are the results as you expect them ? In the space below provide an explaination for your results.

	10.10.10.40/24	**10.10.10.50/24**	**10.10.10.70/24**	**10.10.10.80/24**
10.10.10.40/24				
10.10.10.50/24				
10.10.10.70/24				
10.10.10.80/24				

	2001:1::40/64	**2001:1::50/64**	**2001:1::70/64**	**2001:2::80/64**
2001:1::40/64				
2001:1::50/64				
2001:1::70/64				
2001:2::80/64				

Analyse the packets on the wire at *hub1*, can you confirm the VLAN tagging.

Rebuild the lab and this time configure 802.1ad VLANs.

Instead of

```
# ip link add link eth0 name eth0.10 type vlan id 10
# ip link add link eth0 name eth0.10 type vlan id 10
```

use

```
# ip link add link eth0 name eth0.10 type vlan proto 802.1ad id 10
# ip link add link eth0 name eth0.10 type vlan proto 802.1ad id 10
```

Confirm the frame type on the wire with Wireshark.