Understanding Linux Internetworking

In this Paper

Introduction

The Internet: the largest internetwork ever created. In fact, the term *Internet* (with a capital *I*) is just a shortened version of the term *internetwork*, which means multiple networks connected together. Most companies create some form of internetwork when they connect their *local-area network* (LAN) to a *wide area network* (WAN). For IP packets to be delivered from one network to another network, IP routing is used — typically in conjunction with dynamic routing protocols such as OSPF or BGP. You can easily use Linux as an internetworking device and connect hosts together on local networks and connect local networks together and to the Internet.

Here's what you'll learn in this paper:

- The differences between layer 2 and layer 3 internetworking
- How to configure IP routing and bridging in Linux
- How to configure advanced Linux internetworking, such as VLANs, VXLAN, and network packet filtering

To create an internetwork, you need to understand layer 2 and layer 3 internetworking, MAC addresses, bridging, routing, ACLs, VLANs, and VXLAN. We've got a lot to cover, so let's get started!

Layer 2 vs. Layer 3 Internetworking

In the OSI model, layer 1 is the physical layer that includes the physical media used to connect the network. Specifications in this area describe cable qualities and the properties of electrical and optical signals used to move bits around. Examples of layer 1 technologies include Gigabit Ethernet on category 5 cable, 100Gigabit Ethernet on parallel single mode fiber, and 802.11 wireless.

Elsewhere in this Paper: The OSI Model

If you're not sure what is meant by the terms 'Layer 2' and 'Layer 3', please refer to Appendix B for more information.

Above that is layer 2, or the data link layer; Ethernet is a broadly deployed layer 2 protocol. Ethernet networking works to encapsulate data and pass that data in the form of *frames*. Frames leverage the Media Access Control (MAC) addresses. An Ethernet frame includes the MAC address of the destination interface on the target system as well the MAC address of the source interface on the sending system so that the recipient device knows where the frame originated. Every Ethernet device, whether it's installed in a server, a switch, or a router, has a unique MAC address on their local network.

Transparent bridges are layer 2 devices that send all frames received on one port out the other bridge ports, based on knowledge of the frame's destination MAC address. Ethernet switches are multiport network bridges. Multiport network bridges learn of the MAC addresses in the network and intelligently forward frames based on the destination MAC address in the frame.

Layer 2 networking works in one of two ways:

- The device has explicit knowledge of a frame's destination address, and the device sends the frame out on the port where it knows the destination exists.
- In the event that the specific destination is unknown, the device falls back to sending the frame to every node in the layer 2 domain via what is known as a *broadcast*.

Definition: Broadcast Domain

In Ethernet networking, layer 2 broadcasts don't go past routers because that is the boundary of the layer 2 network. Thus, the entire Ethernet network is the broadcast domain because no broadcasts pass the Ethernet LAN.

The problem is that these approaches limit the ability for layer 2 networks alone to operate efficiently beyond relatively small-scale locations and very simple topologies. Layer 2 networks suffer from two major limitations. First, they allow for hosts to send traffic to unknown destinations. This causes broadcasts, which impact every node in the broadcast domain. Many networks have been taken offline due to "broadcast storms," or when many hosts are broadcasting at once. In contrast, layer 3 networks do not allow for unknown communication. If a layer 3 router does not have a route to the destination IP address, it will drop the packet instead of broadcasting like layer 2 does.

Second, layer 2 networks have globally unique MAC addresses that are assigned by the manufacturer. There is no organization to these addresses across manufacturers. If you have servers with Intel and Mellanox network cards, the layer 2 MAC addresses will not have any commonality. Again, when comparing layer 2 MAC addresses to layer 3 IP addresses, companies manually plan IP addressing schemes so that there is a hierarchy to these IP addresses. An office may have all IP addresses within it as part of a single IP subnet, like 10.0.0.0, allowing the company to use a single subnet to represent the entire office. With layer 2 addressing, there is no ability to summarize or aggregate MAC addresses; every unique MAC address must be shared with every host in the layer 2 domain.

When a node sends out an IP packet, it consults its routing and neighbor (ARP) tables and sends the packet to the device most likely to get that packet where it needs to go. If the destination is in the same layer 2 network, an entry in the neighbor (or ARP) table tells the sender how to use layer 2 internetworking. When IP devices need to communicate with other IP-based addresses that are outside of their local layer 2 network, the route table may point to a specific router that will get the packet closer to the destination or fall through to the default gateway, which is then responsible for getting the packet to the destination. If no default route exists and a matching route does not exist, the packet will be dropped.

Layer 2 Internetworking on Linux Systems

Initially, Linux networking was focused on end-node networking and layer 3 internetworking; however, the advent of virtualization and containerization changed that forever. Today's Linux networking stack has rich layer 2 internetworking functionality and continues to evolve at a rapid pace.

Bridging

What do you do when you have two different Ethernet networks that need connecting? Build a bridge! Bridges have traditionally been dedicated hardware devices, but you can easily create a bridge in Linux. For example, when you have a Linux host that has two or more network interfaces, you can create a bridge to pass traffic between these interfaces. You can add two interfaces to a Linux bridge with **`ip link set`** and **`ip link add`** using:

```
david@debian:~$ sudo ip link add br0 type bridge
david@debian:~$ sudo ip link set eth0 master br0
david@debian:~$ sudo ip link set eth1 master br0
```

Here's what is happening:

- The first command, **`ip link add`**, is creating a bridge named br0.
- The two **`ip link set`** commands add the two Ethernet interfaces, eth0 and eth1, to the new bridge resulting in a connection between these two interfaces.

Once a bridge is created, you can view the MAC address table, which shows which ports can reach a specific MAC address, with the **`bridge`** command. The command shown in the example below uses

`fdb show` as its parameter. In this command, fdb stands for forwarding database management, and show is a way for you to see the current contents of this database:

```
david@debian:~$ sudo bridge fdb show
[sudo] password for david:
01:00:5e:00:00:01 dev eth0 self permanent
33:33:00:00:00:01 dev eth0 self permanent
33:33:ff:d0:e8:7e dev eth0 self permanent
01:00:5e:00:00:fb dev eth0 self permanent
33:33:00:00:00:fb dev eth0 self permanent
01:00:5e:7f:ff:fa dev eth0 self permanent
01:00:5e:00:00:01 dev eth1 self permanent
33:33:00:00:00:01 dev eth1 self permanent
01:00:5e:00:00:01 dev eth2 self permanent
33:33:00:00:00:01 dev eth2 self permanent
```

Once the bridge has "bridged," the different Ethernet networks, all the devices on these networks can communicate, at least at layer 2 (see Figure 2).

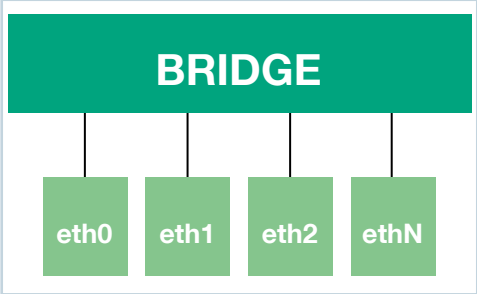


Figure 2. Linux bridge configuration

Spanning Tree

The downside to big networks is that you can accidentally create loops that feed upon themselves and that can ultimately bring the network down. For example, if you accidentally plug one switch port directly into another port on the same switch, you may have created a loop. You can mitigate these loops through the use of *spanning trees*. It's also important to note that layer 3 has a TTL (time to live) field that reduces the impact of loops — packets eventually die and are dropped — while layer 2 does not have a TTL and will loop a frame forever.

A spanning tree is always recommended for any bridge or device configured with a bridge interface to prevent *bridging loops*, reduce broadcast traffic, and provide automatic failover if you have redundant links. You can perform most spanning tree configurations in Linux by using the **`mstpd-ctl`** command, which controls the *multiple spanning tree protocol daemon* (MSTPD).

Bridges bridge frames, and routers route packets. Modern network switches can do a little of both depending on the hardware and software on the device. The really cool thing about Linux is that it can be used to create both layer 2 and layer 3 switches and routers, allowing you to both bridge and route using Linux.

Layer 3 Internetworking View on Linux Systems

The IP protocol is pretty heavily embedded in Linux systems, and it is the primary (and default) way for Linux systems to communicate with the rest of the world, so we'll start with layer 3 internetworking. One interesting thing to note is that the tables, tools, and processes used by end-nodes to reach other end-nodes are exactly the same as those used by routers (layer 3 internetworking devices) to forward packets to end-nodes.

Neighbor Table

When an IP node wants to communicate with a system in the same layer 2 domain, it looks in its neighbor table, or ARP table, to determine how to construct the Ethernet frame. If the desired destination IP address is not in the neighbor table, the node issues an ARP request, which is broadcast to everyone in the layer 2 domain, that asks, "Please tell me the MAC address for the node with IP address X.X.X.X." Assuming the target device is available, the node with that IP address will respond. In Linux, you view (and manipulate) the Neighbor table using the **`ip neighbor show`** command (also known as **`ip neighbor show`**, **`ip neigh show`**, or even just **`ip n s`**):

```
david@debian:~$ ip neigh show
172.20.10.2 dev eth0 lladdr ac:bc:32:9c:a6:3b REACHABLE
172.20.10.1 dev eth0 lladdr 72:70:0d:4c:6b:64 STALE
```

Bridging Loops

Because bridges forward broadcast packets out every port, the broadcast is amplified by both devices when there are multiple paths between two bridges. For example, if two bridges are connected with two links, the first bridge receives a broadcast frame from an attached host. The bridge will take this single frame and send one copy on each link to the other bridge. This second bridge will receive these two broadcast frames, one on each link, and will make new copies, sending them back on each link. This back and forth broadcast replication, known as a "broadcast storm," will continue forever.

Unlike layer 3 packets, layer 2 frames do not possess a TTL field. A packet contains a special field that is set by the host that first created the packet. Each router along the path will decrement this field by 1. If a misconfiguration in the network causes a similar loop, the TTL field will eventually be decremented to 0 and the packet will be dropped. Because a layer 2 frame does not have this field, there is no limit to how many bridges a frame can pass through. Also, because the packet is being bridged and not routed, the TTL field will never be examined by any of the devices and never decremented. The lack of TTL is one of the major problems with layer 2 networks.

The *Spanning Tree Protocol* (STP) does not add a TTL field to the frame, but it will prevent layer 2 loops from forming, preventing the broadcast storm described earlier. Bridges that speak STP will exchange information about the network using *Bridge Protocol Data Units* (BPDUs). Through this BPDU exchange, the bridges will build a loop-free "tree" of the network. In our two-switch example, STP would disable one of the two links and never send traffic over it, until the active link failed

You'll receive a list of IP addresses that have been recently resolved to MAC addresses, their associated MAC addresses, which interface is used to reach the layer 2 network where they can be reached, and the confidence of knowing these IP/MAC address relationships. Typically, the neighbor table is maintained dynamically based on the ARP protocol; however, it can be manually controlled with the **`ip neighbor`** command.

IP Routing

The routing table has knowledge of specific networks, or summaries of networks, that a node can reach. Minimally, each routing table will have a "default route" where the node can send any IP packet that is not in an attached layer 2 network. You can view the routing table with the **`ip route show`** command, like this:

```
david@debian:~$  ip route show
default via 172.20.10.1 dev eth0  proto static  metric 1024
172.20.10.0/28 dev eth0  proto kernel  scope link  src 172.20.10.10
```

Here you can see that the routing table knows that the 172.20.10.0/28 network is a locally attached layer 2 network. The routing table also includes a route to the default gateway (172.20.10.1), which Linux calls "default," that will be used to reach any node that isn't on the local network. If you're used to networking on non-Linux systems, you may have seen a default route expressed as something like 0.0.0.0/0.

Routes can be added or deleted from the routing table in a few different ways:

- By assigning IP addresses to node interfaces
- By manually adding or removing them using the **`ip route`** command
- By dynamically inserting them using routing protocols

For example, to create a static route to router 192.168.1.1 through the eth1 interfaces, you would use the **`ip route`** command, like this:

```
ip route add default via 192.168.1.1 dev eth1
```

However, once the host is restarted, this route disappears because it's not persistent. To make this route persistent, you would edit the /etc/network/interfaces file and, after the network device configuration, add a **`post-up`** command with the same **`ip route`** command so that this static route is added every time the Linux host is restarted or the network interface is brought up. Here's an example of what it might look like in the /etc/network/interfaces file:

```
iface eth1 inet static
address 192.168.1.1
netmask 255.255.255.0
post-up ip route add default via 192.168.1.1 dev eth1
```

The purpose of the **`post-up`** command is to add the default route only after the network interface is brought up.

Definition: Free Range Routing

Free range routing (FRR) is an open-source Linux suite of IP routing protocols that includes BGP, IS-IS, LDP, OSPF, PIM, and RIP. Because it integrates with a wide variety of Linux stacks, FRR has a wide range of use cases including connecting hosts, VMs, and containers to the network, Internet access routers, and Internet peering. Based on the Quagga project, FRR is used by many companies for many use cases around the world.

Virtual LANs (VLANs)

You already know that a LAN is a local area network spanning a relatively small physical area. Building on that concept, a *virtual LAN* (or VLAN) allows LANs to span multiple switches across very large networks while still achieving traffic isolation from other networks. VLANs are used to isolate hosts or applications from each other for the purposes of security, data flow, and scale. Individual interfaces can be a part of one or more VLANs. When they are a part of more than one VLAN, in order to maintain some semblance of sanity, the frames traversing that link are tagged with an *IEEE 802.1Q* tag. These tags are an additional piece of information placed at the front of the frame to identify the VLAN. Interfaces carrying multiple VLANs are often called *trunks.* VLANs are configured using both the **`ip link`** and **`bridge`** Linux commands.

Suppose you want a Linux system to have eth1 in one bridge (VLAN11), eth3 in a second bridge (VLAN12), and eth2 in both (i.e. a tagged trunk). First, we make sure the 802.1Q trunking driver is installed. Then we create a bridge, add the ports to the bridge, and make sure the ports are part of the desired set of VLANs. Notice that both eth1 and eth3 used untagged VLANs. However, per the bridge's configuration, traffic from those ports will be placed onto their configured VLANs, which are VLAN 11 and VLAN 12 in this case. Untagged traffic from the trunk port will be placed into the native VLAN, which is VLAN 1 by default.

If you look at the Ethernet frames, you can't tell that the interfaces are part of a VLAN; however, eth2 is a member of both VLANs, and all frames carry the 802.1Q VLAN tag (shown in Figure 3).

In the following configuration, we ensure that the 802.1q module is loaded, add bridge0 (br0) as the native VLAN (VLAN 1), add the Ethernet interfaces to br0, assign eth1 and eth3 to their respective VLANs (11 and 12), and bring all interfaces up.

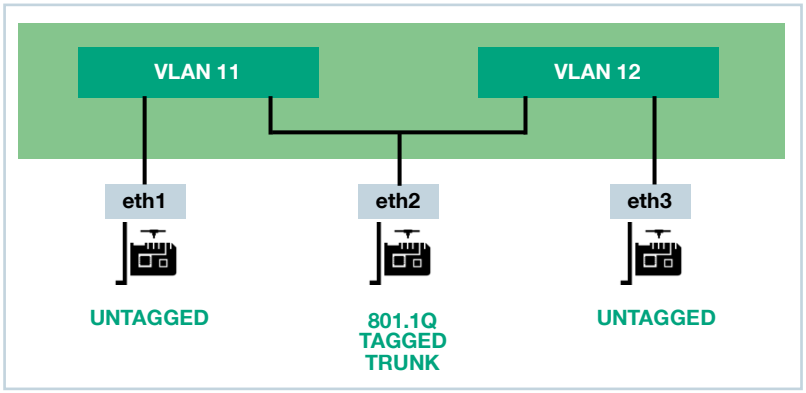


Figure 3. Tagged and untagged VLAN traffic

```
sudo modprobe 8021q
sudo ip link add br0 type bridge vlan_filtering 1
sudo ip link set eth1 master br0
sudo ip link set eth2 master br0
sudo ip link set eth3 master br0
sudo bridge vlan add dev eth1 vid 11 pvid untagged
sudo bridge vlan add dev eth3 vid 12 pvid untagged
sudo bridge vlan add dev eth2 vid 11
sudo bridge vlan add dev eth2 vid 12
sudo ip link set up dev br0
sudo ip link set up dev eth1
sudo ip link set up dev eth2
sudo ip link set up dev eth3
```

To help you better understand the configuration above, there are a few things that you should know:

- In the command **`sudo bridge vlan add dev eth1 vid 11 pvid untagged`**, the *vid* parameter is the VLAN ID. VLAN IDs are used to specify which VLAN the interface is assigned to. The *pvid* parameter specifies the private VLAN ID. In this case, the private VLAN is left untagged.
- In the example above, MAC address tables are per-VLAN. If you were to look at the MAC address table in VLAN 11 and VLAN 12, you'd find that they would be very different. The trunk link would have a combination of MAC addresses from both VLAN 11 and 12.

Much ado about 802.1q

The purpose of the VLAN is to have multiple devices on the same VLAN communicate as if they were the only devices on that network, giving administrators flexibility that they didn't have with physical networks alone. The IEEE networking standard that defines VLANs (or virtual local area networks) is 802.1Q. This standard details how Ethernet frames will have a VLAN tag, or identifier, inserted into them and how Ethernet bridges and switches will handle frames with the VLAN tags. Any Ethernet frame that doesn't have a tag stays on the *native VLAN*, whereas frames that do have tags are only seen by other network devices on that VLAN. Network links between switches that carry multiple VLANs are called *trunk links*

Here are a few useful commands to see what's going on:

- **`bridge link show.`** Check the status of the bridge links
- **`bridge vlan show.`** Check the status of the VLANs traversing the bridge
- **`bridge fdb show.`** View the forwarding database

Overlay Networks with VXLAN

An *overlay network* is essentially a computer network that is built on top of another network. The overlay network is commonly called the "virtual network" that runs on top of an existing "physical network" (and thus, the "overlay" and "underlay" terminology). It's important to note that VLANs discussed in the last section are an example of a virtual network overlay on a L2 network.

VXLAN, or *Virtual eXtensible LAN*, is an overlay network that runs on top of an existing IP network. VXLAN has a number of different use cases, including creating a massively scalable network (up to 16.7 million possible networks) and connecting data centers at layer 2 across a layer 3 network. VXLAN encapsulates frames with layer 3/4 (IP/UDP), sending them over both layer 2 and layer 3 networks. The benefits of VXLAN on layer 2 (IP) networks are global addressing, better scale, more resiliency, and better use of available bandwidth.

The connections between endpoints are called VXLAN tunnels. These VXLAN tunnels are encapsulating traffic as it flows across the network between the VXLAN tunnel endpoints (also called VTEPs, or VXLAN Tunnel EndPoints). The VXLAN encapsulation allows for the transport of traffic over networks that end hosts do not need knowledge of. This means a host could send an ARP request to another host, across the network, through a VxLAN tunnel, and never know about the VxLAN tunnel or the underlay network it travels through.

What is encapsulation?

Encapsulation is when one piece of data or packet on a network is wrapped up in another type of data or network packet. For example, a text file could be encapsulated in an archive file. In networking, encapsulation is used as a means to move traffic that might otherwise not be able to traverse the communications mechanism. For example, you may encapsulate an IP packet encapsulated in an Ethernet frame to move traffic between local hosts, but encapsulation can even happen between the same two protocols. IP could be encapsulated with IP. A common modern-day example of encapsulation is the iSCSI storage protocol. In an iSCSI system, iSCSI commands and a storage payload are encapsulated inside a TCP packet, which is encapsulated inside an IP packet, which is, in turn, encapsulated inside an Ethernet packet. This multi-level encapsulation process enables what would have been local SCSI storage commands to transparently traverse an Ethernet-based TCP/IP network.

VTEPs can be implemented in hardware or software. The configuration of VTEPs and creation of the overlay networks is typically implemented using a commercial controller, such as VMware NSX or Midokura MidoNet, or using a protocol such as BGP EVPN over. However, some people build their own special purpose controllers. Regardless of which of these techniques you use, Linux provides the underlying VTEP building block.

If you have two Linux systems and you want to bridge them with VXLAN, you would install a bridge on both systems, add a local IP address to that bridge, and add a VTEP to that bridge pointing the VTEP to the other Linux host (shown in Figure 4).

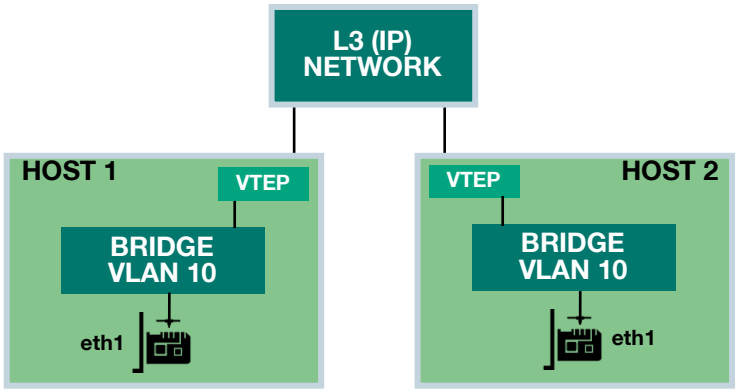


Figure 4. Two Linux hosts connected with VXLAN

Linux System 1

```
sudo ip link add br0 type bridge vlan_filtering 1
sudo ip link add vlan10 type vlan id 10 link bridge protocol none
sudo ip addr add 10.0.0.1/24 dev vlan10
sudo ip link add vtep10 type vxlan id 1010 local 10.1.0.1 remote 10.3.0.1 learning
sudo ip link set eth1 master br0
sudo bridge vlan add dev eth1 vid 10 pvid untagged
```

Linux System 2

```
sudo ip link add br0 type bridge vlan_filtering 1
sudo ip link add vlan10 type vlan id 10 link bridge protocol none
sudo ip addr add 10.0.0.2/24 dev vlan10
sudo ip link add vtep10 type vxlan id 1010 local 10.3.0.1 remote 10.1.0.1 learning
sudo ip link set eth1 master br0
sudo bridge vlan add dev eth1 vid 10 pvid untagged
```

Now these two systems both exist on the 10.0.0.x/24 layer 2 network (via the VXLAN overlay) even though they are connected by a layer 3 IP fabric. It's also worth noting that the hosts are completely isolated from the underlying layer 3 network.

In Summary

After reading this paper, you should now have a good understanding of the basics of Linux internetworking. You learned about layer 2 versus layer 3 networking, bridging, routing, traffic filtering, VXLAN, and more. Hopefully you will spend time learning more about Linux networking administration. Good luck!

Appendix A: The Basics of TCP/IP Addresses

IP version 4 addressing, known as IPv4, uses a 32-bit number to identify every host/device. These addresses are usually written using *dotted decimal*, such as 192.168.192.168. Every device has a configured subnet mask, such as 255.255.255.0, that tells the device which part of the IP address is used to identify the network and which part is the device.

The 32-bit address is broken up into four 8-bit sections called *octets.* For example, the decimal to binary conversation for the above IP address (192.168.192.168) is

11000000 10101000 11000000 10101000.

The conversation of the subnet mask from 255.255.255.0 is

11111111 11111111 11111111 00000000

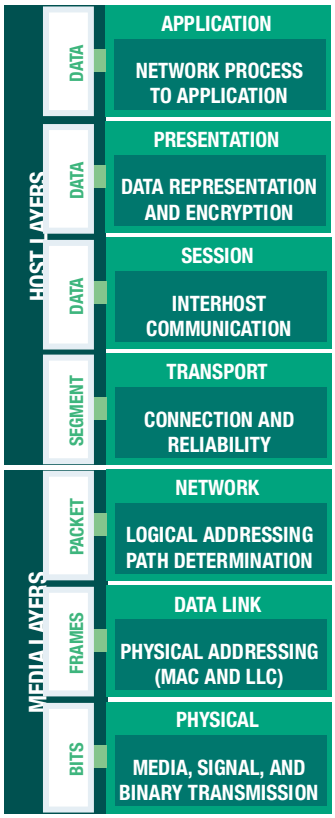
How, exactly, does your networking stack know that 192.168.10.2 is not in the same network as 192.168.192.168 when using a 255.255.255.0 subnet mask? If you've ever wondered how the math works, the magic lies in the use of the bitwise AND operator. In the figure below, you can see that performing a bitwise AND operation between the origination address and the local network's subnet mask results in a calculation that shows that the local network is 192.168.192.0. When a node in this network wants to communicate with the IP address 192.168.10.2, a similar operation is performed on this destination address with the result indicating that the destination address is 192.168.10.0. Because the destination address has been determined to be non-local, this traffic is sent to the local layer 3 device, typically a router, which then forwards the packet to the correct destination network.

Origination Address	192	168	192	168
	11000000	10101000	11000000	10101000
Subnet Mask	255	255	255	0
	11111111	11111111	11111111	00000000
Result	192	168	192	0
	11000000	10101000	11000000	00000000

Destination Address	192	168	10	2
	11000000	10101000	00001010	00000010
Subnet Mask	255	255	255	0
	11111111	11111111	11111111	00000000
Result	192	168	10	0
	11000000	10101000	00001010	00000000

Appendix B: The OSI Model

Before we jump deeply into the networking pool, let's go over the model on which all networking standards are based: the International Organization of Standardization Open Systems Interconnection (ISO OSI) model. This model has been used for decades to describe the networking stack, and it describes the very wires (or lack thereof, in the case of wireless) that transfer data to the applications that operate on the network. This all-encompassing model has driven network development, and most products on the networking market are specifically designed to service one or more layers of the model, which are shown in this callout. In order for an application to "talk" to another application on another machine on the network, that application has to traverse down its own networking stack, ultimately placing its information onto the wireless or media that connects the two machines. The application, however, doesn't need to handle that task itself. It simply presents its data to the next lower layer—the presentation layer—which processes what it gets from the application layer and then sends it down the stack to the session layer and so forth. This is one of the reasons that applications don't need to develop their own communications stacks and can just rely on what is provided to them in the operating system.



Last updated: December 2017