Cisco CCNP Routing Study Guide

v1.22 © 2012

Aaron Balchunas
aaron@routeralley.com
http://www.routeralley.com

Foreword:

*This study guide is intended to provide those pursuing the CCNP certification with a framework of what concepts need to be studied. This is **not** a comprehensive document containing all the secrets of the CCNP Routing exam, nor is it a "braindump" of questions and answers.*

*This document is freely given, and can be freely distributed. However, the contents of this document **cannot** be altered, without my written consent. Nor can this document be sold or published without my expressed consent.*

I sincerely hope that this document provides some assistance and clarity in your studies.

Table of Contents

Part I – Addressing

Part II – Basic Routing Concepts

Part III – Dynamic Routing Protocols

Part IV– Advanced Routing Functions

Part I

Addressing

Section 1
- IPv4 Addressing and Subnetting -

Hardware Addressing

A **hardware address** is used to uniquely identify a host *within* a local network. Hardware addressing is a function of the **Data-Link** layer of the OSI model (Layer-2).

Ethernet utilizes the 48-bit **MAC address** as its hardware address. The MAC address is often hardcoded on physical network interfaces, though some interfaces support changing the MAC address using special utilities. In virtualization environments, dynamically assigning MAC addresses is very common.

A MAC address is most often represented in **hexadecimal,** using one of two accepted formats:

00:43:AB:F2:32:13
0043.ABF2.3213

The first six hexadecimal digits of a MAC address identify the *manufacturer* of the physical network interface. This is referred to as the **OUI (Organizational Unique Identifier).** The last six digits uniquely identify the host itself, and are referred to as the **host ID**.

The MAC address has one shortcoming – it contains no *hierarchy.* MAC addresses provide no mechanism to create **boundaries** between networks. There is no method to distinguish one network from another.

This lack of hierarchy poses *significant* difficulties to network scalability. If *only* Layer-2 hardware addressing existed, all hosts would technically exist on the *same* network. Internetworks like the Internet could not exist, as it would be impossible to separate *my* network from *your* network.

Imagine if the entire Internet existed purely as a single Layer-2 switched network. Switches, as a rule, will forward a broadcast out *every* port. With billions of hosts on the Internet, the resulting broadcast storms would be devastating. The Internet would simply collapse.

The scalability limitations of Layer-2 hardware addresses are mitigated using **logical addresses,** covered in great detail in this guide.

Logical Addressing

Logical addressing is a function of the **Network** layer of the OSI Model (Layer-3), and provides a hierarchical structure to separate networks. Logical addresses are never hardcoded on physical network interfaces, and can be dynamically assigned and changed freely.

A logical address contains two components:

- **Network ID** – identifies which network a host belongs to.
- **Host ID** – uniquely identifies the host on that network.

Examples of logical addressing protocols include **Internetwork Packet Exchange (IPX)** and **Internet Protocol (IP)**. IPX was predominantly used on Novell networks, but is now almost entirely deprecated. **IP** is the most widely-used logical address, and is the backbone protocol of the Internet.

Internet Protocol (IP)

In the 1970's, the Department of Defense developed the **Transmission Control Protocol (TCP)**, to provide both Network and Transport layer functions. When this proved to be an inflexible solution, those functions were separated - with the **Internet Protocol (IP)** providing Network layer services, and TCP providing Transport layer services.

Together, TCP and IP provide the core functionality for the **TCP/IP** or **Internet protocol suite**.

IP provides two fundamental Network layer services:

- **Logical addressing** – provides a unique address that identifies both the *host*, and the *network* that host exists on.
- **Routing** – determines the *best path* to a particular destination network, and then *routes* data accordingly.

IP was originally defined in RFC 760, and has been revised several times. IP Version 4 (**IPv4**) was the first version to experience widespread deployment, and is defined in RFC 791. IPv4 will be the focus of this guide.

IPv4 employs a **32-bit address**, which limits the number of possible addresses to 4,294,967,296. IPv4 will eventually be replaced by IP Version 6 (**IPv6**), due to a shortage of available IPv4 addresses. IPv6 is covered in great detail in another guide.

IPv4 Addressing

A core function of IP is to provide logical addressing for hosts. An **IP address** provides a hierarchical structure to both uniquely identify a *host*, and what *network* that host exists on.

An IP address is most often represented in **decimal,** in the following format:

158.80.164.3

An IP address is comprised of four **octets,** separated by periods:

First Octet	Second Octet	Third Octet	Fourth Octet
158	80	164	3

Each octet is an **8-bit** number, resulting in a **32-bit IP address**. The smallest possible value of an octet is *0,* or *00000000* in binary. The largest possible value of an octet is *255*, or *11111111* in binary.

The above IP address represented in binary would look as follows:

First Octet	Second Octet	Third Octet	Fourth Octet
10011110	01010000	10100100	00000011

Decimal to Binary Conversion

The simplest method of converting between decimal and binary is to remember the following table:

128	64	32	16	8	4	2	1

To convert a decimal number of *172* to binary, start with the leftmost column. Since *172* is greater than *128*, that binary bit will be set to *1*. Next, add the value of the next column (*128 + 64 = 192)*. Since *172* is less than *192,* that binary bit will be set to *0.*

Again, add the value of the next column (*128 + 32 = 160)*. Since *172* is greater than *160*, that binary bit will be set to *1.* Continue this process until the columns with binary bits set to *1* add up to 172:

Decimal	128	64	32	16	8	4	2	1
Binary	1	0	1	0	1	1	0	0

Binary to Decimal Conversion

Converting from binary back to decimal is even simpler. Apply the binary number to the conversion table, and then add up any columns with binary bits set to 1.

For example, consider the binary number of *11110001:*

Decimal	128	64	32	16	8	4	2	1
Binary	1	1	1	1	0	0	0	1

By adding *128 + 64 + 32 + 16+ 1,* it can be determined that *11110001* equals *241.*

The Subnet Mask

Part of an IP address identifies the *network.* The other part of the address identifies the *host.* A **subnet mask** is required to provide this distinction**:**

158.80.164.3 255.255.0.0

The above IP address has a subnet mask of *255.255.0.0*. The subnet mask follows two rules:

- If a binary bit is set to a **1** (or *on*) in a subnet mask, the corresponding bit in the address identifies the **network**.
- If a binary bit is set to a **0** (or *off*) in a subnet mask, the corresponding bit in the address identifies the **host**.

Looking at the above address and subnet mask in binary:

```
IP Address:      10011110.01010000.10100100.00000011
Subnet Mask:     11111111.11111111.00000000.00000000
```

The first 16 bits of the subnet mask are set to *1*. Thus, the first 16 bits of the address (*158.80*) identify the *network*. The last 16 bits of the subnet mask are set to *0*. Thus, the last 16 bits of the address (*164.3*) identify the unique *host* on that network.

The network portion of the subnet mask must be **contiguous**. For example, a subnet mask of *255.0.0.255* is not valid.

The Subnet Mask (continued)

Hosts on the same logical network will have *identical* network addresses, and can communicate freely. For example, the following two hosts are on the same network:

Host A:	158.80.164.100 255.255.0.0
Host B:	158.80.164.101 255.255.0.0

Both share the same network address (*158.80*), which is determined by the *255.255.0.0* subnet mask. Hosts that are on *different* networks cannot communicate without an intermediating device. For example:

Host A:	158.80.164.100 255.255.0.0
Host B:	158.85.164.101 255.255.0.0

The subnet mask has remained the same, but the network addresses are now different (*158.80* and *158.85* respectively). Thus, the two hosts are *not* on the same network, and cannot communicate without a **router** between them. **Routing** is the process of forwarding packets from one network to another.

Consider the following, trickier example:

Host A:	158.80.1.1 255.248.0.0
Host B:	158.79.1.1 255.248.0.0

The specified subnet mask is now *255.248.0.0*, which doesn't fall cleanly on an octet boundary. To determine if these hosts are on separate networks, first convert everything to binary:

Host A Address:	10011110.01010000.00000001.00000001
Host B Address:	10011110.01001111.00000001.00000001
Subnet Mask:	11111111.11111000.00000000.00000000

Remember, the **1** (or **on)** bits in the subnet mask identify the *network* portion of the address. In this example, the first *13 bits* (the 8 bits of the first octet, and the first 5 bits of the second octet) identify the network. Looking at only the first 13 bits of each address:

Host A Address:	10011110.01010
Host B Address:	10011110.01001

Clearly, the network addresses are *not* identical. Thus, these two hosts are on separate networks, and require a router to communicate.

IP Address Classes

The IPv4 address space has been structured into several **classes**. The value of the **first octet** of an address determines the class of the network:

Class	*First Octet Range*	*Default Subnet Mask*
Class A	1 - 127	255.0.0.0
Class B	128 - 191	255.255.0.0
Class C	192 - 223	255.255.255.0
Class D	224 - 239	-

Class A networks range from **1** to **127.** The *default* subnet mask is 255.0.0.0. Thus, by *default,* the first octet defines the network, and the last three octets define the host. This results in a maximum of **127** Class A networks, with **16,777,214** hosts per network!

Example of a Class A address:

Address:	64.32.254.100
Subnet Mask:	255.0.0.0

Class B networks range from **128** to **191**. The *default* subnet mask is 255.255.0.0. Thus, by *default,* the first two octets define the network, and the last two octets define the host. This results in a maximum of **16,384** Class B networks, with **65,534** hosts per network.

Example of a Class B address:

Address:	152.41.12.195
Subnet Mask:	255.255.0.0

Class C networks range from **192** to **223.** The *default* subnet mask is 255.255.255.0. Thus, by *default,* the first three octets define the network, and the last octet defines the host. This results in a maximum of **2,097,152** Class C networks, with **254** hosts per network.

Example of a Class C address:

Address:	207.79.233.6
Subnet Mask:	255.255.255.0

Class D networks are reserved for **multicast** traffic. Class D addresses do not use a subnet mask.

CIDR (Classless Inter-Domain Routing)

Classless Inter-Domain Routing (CIDR) is a simplified method of representing a subnet mask. CIDR identifies the number of binary bits set to a **1** (or *on*) in a subnet mask, preceded by a slash.

For example, a subnet mask of *255.255.255.240* would be represented as follows in binary:

11111111.11111111.11111111.11110000

The first 28 bits of the above subnet mask are set to *1*. The CIDR notation for this subnet mask would thus be ***/28***.

The CIDR mask is often appended to the IP address. For example, an IP address of *192.168.1.1* and a subnet mask of *255.255.255.0* would be represented as follows using CIDR notation:

192.168.1.1 /24

Address Classes vs. Subnet Mask

Remember the following three rules:

- The **first octet** on an address dictates the *class* of that address.
- The **subnet mask** determines what part of an address identifies the *network*, and what part identifies the *host*.
- Each class has a ***default*** subnet mask. A network using its default subnet mask is referred to as a **classful network**.

For example, *10.1.1.1* is a Class A address, and its default subnet mask is *255.0.0.0* (*/8* in CIDR).

It is entirely possible to use subnet masks *other* than the default. For example, a Class B subnet mask can be applied to a Class A address:

10.1.1.1 /16

However, **this does not change the class of the above address**. It remains a Class A *address*, which has been subnetted using a Class B *mask*.

Remember, the ***only*** thing that determines the class of an IP address is the first octet of that address. Likewise, the subnet mask is the ***only*** thing that determines what part of an address identifies the network, and what part identifies the host.

Subnet and Broadcast Addresses

On *each* IP network, two host addresses are reserved for special use:

- The **subnet** (or **network**) address
- The **broadcast** address

Neither of these addresses can be assigned to an actual host.

The **subnet** address is used to identify **the network itself**. A routing table contains a list of known networks, and each network is identified by its subnet address. Subnet addresses contain **all 0 bits in the host portion** of the address.

For example, *192.168.1.0/24* is a subnet address. This can be determined by looking at the address and subnet mask in binary:

IP Address:	11000000.10101000.00000001.00000000
Subnet Mask:	11111111.11111111.11111111.00000000

Note that all host bits in the address are set to *0.*

The **broadcast** address identifies *all* hosts on a particular network. A packet sent to the broadcast address will be received and processed by every host on that network. Broadcast addresses contain **all 1 bits in the host portion** of the address.

For example, *192.168.1.255/24* is a broadcast address. Note that all host bits are set to *1:*

IP Address:	11000000.10101000.00000001.11111111
Subnet Mask:	11111111.11111111.11111111.00000000

Broadcasts are one of three types of IP packets:

- **Unicasts** are packets sent from one host to one other host
- **Multicasts** are packets sent from one host to a *group* of hosts
- **Broadcasts** are packets sent from one host to all other hosts on the local network

A router, by default, will **never forward** a multicast or broadcast packet from one interface to another.

A switch, by default, will forward a multicast or broadcast packet **out every port**, except for the port that originated the multicast or broadcast.

Subnetting

Subnetting is the process of creating new networks (or *subnets)* by **stealing bits** from the host portion of a subnet mask. There is one caveat: stealing bits from hosts creates **more** networks but **fewer** hosts per network.

Consider the following Class C network:

192.168.254.0

The default subnet mask for this network is 255.255.255.0. This single network can be segmented, or *subnetted*, into multiple networks. For example, assume a minimum of *10* new networks are required. Resolving this is possible using the following magical formula:

$$2^n$$

The exponent '**n**' identifies the number of bits to steal from the host portion of the subnet mask. The default Class C mask (255.255.255.0) looks as follows in binary:

11111111.1111111.1111111.00000000

There are a total of 24 bits set to *1*, which are used to identify the network. There are a total of 8 bits set to *0*, which are used to identify the host, and these host bits can be *stolen.*

Stealing bits essentially involves changing host bits (set to *0* or *off*) in the subnet mask to network bits (set to *1* or *on*). Remember, network bits in a subnet mask **must always be contiguous** - skipping bits is not allowed.

Consider the result if three bits are stolen. Using the above formula:

2^n = 2^3 = 8 = **8 new networks created**

However, a total of 8 new networks *does not* meet the original requirement of at least 10 networks. Consider the result if four bits are stolen:

2^n = 2^4 = 16 = **16 new networks created**

A total of 16 new networks *does* meet the original requirement. Stealing four host bits results in the following *new* subnet mask:

11111111.11111111.11111111.11110000 = 255.255.255.240

Subnetting (continued)

In the previous example, a Class C network was subnetted to create *16* new networks, using a subnet mask of *255.255.255.240* (or */28* in CIDR). Four bits were stolen in the subnet mask, leaving only four bits for hosts.

To determine the number of hosts this results in, for each of the new 16 networks, a slightly modified formula is required:

$$2^n – 2$$

Consider the result if four bits are available for hosts:

$2^n – 2 \quad = \quad 2^4 – 2 \quad = \quad 16 – 2 \quad =$ **14 usable hosts per network**

Thus, subnetting a Class C network with a /28 mask creates 16 new networks, with 14 usable hosts per network.

Why is the formula for calculating usable hosts $2^n – 2$? Because it is **never possible** to assign a host an address with all *0* or all *1* bits in the *host* portion of the address**.** These are reserved for the subnet and broadcast addresses, respectively. Thus, every time a network is subnetted, useable host addresses are lost.

The 2^n-2 Rule and Subnetted Networks

To avoid confusion, it was historically unacceptable to use the first and last new *networks* created when subnetting, as it is possible for a classful network to have the same subnet and broadcast address as its subnetted networks. This required the $2^n – 2$ formula to also be used when calculating the number of new *networks* created while subnetting.

However, this is **no longer a restriction** for modern equipment and routing protocols. Specifically, on Cisco IOS devices, the following command is now enabled by default:

```
Router(config)#  ip subnet-zero
```

The *ip subnet-zero* commands allows for the use of networks with all ***0*** or all ***1* bits** in the *stolen* network portion of the address. Thus, the formula for calculating the number of new networks created is simply $\mathbf{2^n}$.

Remember though, the formula for calculating usable ***hosts* is always $\mathbf{2^n – 2}$.**

(Reference: http://www.cisco.com/en/US/tech/tk648/tk361/technologies_tech_note09186a0080093f18.shtml)

Determining the Range of Subnetted Networks

Determining the *range* of the newly created networks can be accomplished using several methods. The *long* method involves some binary magic.

Consider the example *192.168.254.0* network again, which was subnetted using a *255.255.255.240* mask:

192.168.254.0:	11000000.10101000.11111110.00000000
255.255.255.240:	11111111.11111111.11111111.11110000

Subnetting stole four bits in the fourth octet, creating a total of *16* new networks. Looking at *only* the fourth octet, the first newly created network is *0000*. The second new network is *0001*. Calculating all possible permutations of the four stolen bits:

Binary	***Decimal***	***Binary***	***Decimal***	***Binary***	***Decimal***
.0000 *xxxx*	.0	.0110 *xxxx*	.96	.1100 *xxxx*	.192
.0001 *xxxx*	.16	.0111 *xxxx*	.112	.1101 *xxxx*	.208
.0010 *xxxx*	.32	.1000 *xxxx*	.128	.1110 *xxxx*	.224
.0011 *xxxx*	.48	.1001 *xxxx*	.144	.1111 *xxxx*	.240
.0100 *xxxx*	.64	.1010 *xxxx*	.160		
.0101 *xxxx*	.80	.1011 *xxxx*	.176		

Note that this equates to exactly *16* new networks. The decimal value represents the first (or the *subnet*) address of each newly created network. To determine the range for the hosts of the *first* new network:

Binary	***Decimal***	***Binary***	***Decimal***	***Binary***	***Decimal***
.0000 0000	.0	.0000 0110	.6	.0000 1100	.12
.0000 0001	.1	.0000 0111	.7	.0000 1101	.13
.0000 0010	.2	.0000 1000	.8	.0000 1110	.14
.0000 0011	.3	.0000 1001	.9	.0000 1111	.15
.0000 0100	.4	.0000 1010	.10		
.0000 0101	.5	.0000 1011	.11		

The binary value has been split to emphasize the separation of the stolen *network* bits from the *host* bits. The first address has all *0* bits in the host portion (*0000*), and is the **subnet address** for this network. The last address has all *1* bits in the host portion, and thus is the **broadcast address** for this network. Note that there are exactly **14 usable addresses** to assign to hosts.

Determining the Range of Subnetted Networks (continued)

Calculating the ranges of subnetted networks can quickly become tedious when using the long binary method. The *shortcut* method involves taking the subnet mask (*255.255.255.240* from the previous example), and subtracting the subnetted octet (*240*) from *256*.

256 – 240 = 16

Assuming *ip subnet-zero* is enabled, the first network will begin at *0.* Then, simply continue adding *16* to identify the first address of each new network:

0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240

Knowing the *first* address of each new network makes it simple to determine the *last* address of each network:

First address of network	0	16	32	48	64	80	96	112	128	144
Last address of network	15	31	47	63	79	95	111	127	143	159

Only the first 10 networks were calculated, for brevity. The first address of each network becomes the **subnet address** for that network**.** The last address of each network becomes the **broadcast address** for that network**.**

Once the first and last address of each network is known, determining the usable range for hosts is straightforward:

Subnet address	0	16	32	48	64	80	96	112	128	144
	1	17	33	49	65	81	97	113	129	145
Usable Range	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕
	14	30	46	62	78	94	110	126	142	158
Broadcast address	15	31	47	63	79	95	111	127	143	159

Hosts on the same network (such as *192.168.254.2* and *192.168.254.14*) can communicate freely.

Hosts on different networks (such as *192.168.254.61* and *192.168.254.66*) require a router to communicate.

Class A Subnetting Example

Consider the following subnetted Class A network: 10.0.0.0 255.255.248.0

Now consider the following questions:

- How many new networks were created?
- How many usable hosts are there per network?
- What is the full range of the first three networks?

By default, the *10.0.0.0* network has a subnet mask of *255.0.0.0*. To determine the number of bits stolen:

255.0.0.0:	11111111.00000000.00000000.00000000
255.255.248.0:	11111111.11111111.11111000.00000000

Clearly, **13 bits** have been stolen to create the new subnet mask. To calculate the total number of new networks:

2^n = 2^{13} = **8192 new networks created**

There are clearly **11 bits** remaining in the host portion of the mask:

$2^n - 2$ = $2^{11} - 2$ = $2048 - 2$ = **2046 usable hosts per network**

Calculating the ranges is a bit tricky. Using the shortcut method, subtract the third octet *(248)* of the subnet mask (*255.255.248.0*) from *256.*

256 – 248 = 8

The first network will begin at *0*, again. **However**, the ranges are spread across multiple octets. The ranges of the first three networks look as follows:

Subnet address	10.0.0.0	10.0.8.0	10.0.16.0
	10.0.0.1	10.0.8.1	10.0.16.1
Usable Range	↕	↕	↕
	10.0.7.254	10.0.15.254	10.0.23.254
Broadcast address	10.0.7.255	10.0.15.255	10.0.23.255

Private vs. Public IPv4 Addresses

The rapid growth of the Internet resulted in a shortage of available IPv4 addresses. In response, a specific subset of the IPv4 address space was designated as *private,* to temporarily alleviate this problem.

A **public address** can be routed on the Internet. Thus, hosts that must be Internet-accessible must be configured with (or *reachable* by) public addresses. Allocation of public addresses is governed by the Internet Assigned Numbers Authority (IANA).

A **private address** is intended for internal use within a home or organization, and can be freely used by anyone. However, private addresses can *never be routed* on the Internet. In fact, Internet routers are configured to immediately drop traffic with private addresses.

Three private address ranges were defined in RFC 1918, one for each IPv4 class:

- Class A - **10.x.x.x /8**
- Class B - **172.16.x.x /12**
- Class C - **192.168.x.x /24**

It is possible to *translate* between private and public addresses, using **Network Address Translation (NAT).** NAT allows a host configured with a private address to be *stamped* with a public address, thus allowing that host to communicate across the Internet. It is also possible to translate multiple privately-addressed hosts to a single public address, which conserves the public address space.

NAT provides an additional benefit – hiding the specific addresses and addressing structure of the internal (or *private*) network.

Note: NAT is *not* restricted to private-to-public address translation, though that is the most common application. NAT can also perform public-to-public address translation, as well as private-to-private address translation.

NAT is only a temporarily solution to the address shortage problem. IPv4 will eventually be replaced with IPv6, which supports a vast address space.

Both NAT and IPv6 are covered extensively in other guides.

Reserved IPv4 Addresses

In addition to the three private IPv4 ranges, several other addresses and ranges are reserved for specific purposes:

- The **0.0.0.0 /0** network is used to identify **all networks,** and is referred to as the **default route.** If a default route exists in a routing table, it will be used only if there is *not* a more *specific* route to a particular destination. Routing and default routes are covered extensively in another guide.

- The **0.0.0.0 /8** range is used to identify hosts on the *local* network. Addresses in this range can only be used as a *source* address. The most commonly used address in this range is **0.0.0.0 /32,** which a host will use when dynamically attempting to learn its IP address via Dynamic Host Configuration Protocol (DHCP). DHCP is covered extensively in another guide.

- The entire **127.x.x.x /8** range is reserved for diagnostic purposes. The most commonly used address in this range is **127.0.0.1**, which identifies the local host, and is referred to as the **loopback** or **localhost** address.

- The **169.254.x.x /16** range is reserved for Automatic Private IP Addressing (APIPA). A host assigns itself an address in this range, if it cannot dynamically obtain an address from a DHCP server.

- The **224.x.x.x – 239.x.x.x** ranges are reserved for **multicast**, and are referred to as **Class D** addresses.

- The **240.x.x.x – 255.x.x.x** ranges are reserved for future and experimental use, and were formerly referred to as **Class E** addresses.

- The **255.255.255.255** address can be used as a broadcast address for the local network.

The IPv4 Header

The IPv4 header is comprised of **12 required fields** and **1 optional field.** The minimum length of the header is **160 bits (20 bytes).**

Field	*Length*	*Description*
Version	4 bits	*Version of IP (in this case, IPv4)*
Internet Header Length	4 bits	*Specifies the length of the IP header (minimum 160 bits)*
DSCP	8 bits	*Classifies traffic for QoS*
Total Length	16 bits	*Specifies the length of both the header and data payload*
Identification	16 bits	*Uniquely identifies fragments of a packet*
Flags	3 bits	*Flags for fragmentation*
Fragment Offset	13 bits	*Identifies the fragment relative to the start of the packet*
Time to Live	8 bits	*Decremented by each router traversed*
Protocol	8 bits	*Specifies the next upper layer protocol*
Header Checksum	16 bits	*Checksum for error checking*
Source Address	32 bits	*Source IPv4 address*
Destination Address	32 bits	*Destination IPv4 address*
Options	Variable	*Optional field for various parameters*

The 4-bit **Version field** is set to a value of *4* for IPv4.

The 4-bit **Internet Header Length field** identifies the length of the IPv4 header, measured in 32-bit *words*. The minimum of length of an IPv4 header is 160 bits, or 5 words (32 x 5 = 160).

The 8-bit **Differentiated Service Code Point (DSCP) field** is used to classify traffic for Quality of Service (QoS) purposes. QoS is covered in great detail in other guides. This field was previously referred to as the Type of Service (ToS) field.

The 16-bit **Total Length field** identifies the total packet size, measured in *bytes,* including both the IPv4 header and the data payload. The minimum size of an IPv4 packet is 20 bytes – essentially a header with no payload. The maximum packet size is **65,535 bytes**.

The IPv4 Header (continued)

Field	*Length*	*Description*
Version	4 bits	*Version of IP (in this case, IPv4)*
Internet Header Length	4 bits	*Specifies the length of the IP header (minimum 160 bits)*
DSCP	8 bits	*Classifies traffic for QoS*
Total Length	16 bits	*Specifies the length of both the header and data payload*
Identification	16 bits	*Uniquely identifies fragments of a packet*
Flags	3 bits	*Flags for fragmentation*
Fragment Offset	13 bits	*Identifies the fragment relative to the start of the packet*
Time to Live	8 bits	*Limits the lifetime of a packet*
Protocol	8 bits	*Specifies the next upper layer protocol*
Header Checksum	16 bits	*Checksum for error checking*
Source Address	32 bits	*Source IPv4 address*
Destination Address	32 bits	*Destination IPv4 address*
Options	Variable	*Optional field for various parameters*

An IPv4 packet that is larger than the **Maximum Transmission Unit (MTU)** size of a link must be **fragmented**. By default, the MTU for Ethernet is **1500 bytes.**

Three fields are used when a packet must be fragmented - the 16-bit **Identification field**, the 3-bit **Flags field**, and the 13-bit **Fragment Offset field**. Each fragment of the packet is marked with the same *Identification* number. The *Fragment Offset* allows the destination host to reassemble the fragments in the proper order.

The *Flags* field dictates two conditions:

- **Don't Fragment (DF)** – indicates the packet cannot be fragmented. If a packet exceeds a link's MTU size and this flag is set, then the packet is dropped. An ICMP error message will then be sent back to the source host.

- **More Fragments (MF)** – all fragments have this bit set to one, *except* for the last fragment, where the bit is set to zero. This allows the destination host to know when it has received all fragments.

The IPv4 Header (continued)

Field	*Length*	*Description*
Version	4 bits	*Version of IP (in this case, IPv4)*
Internet Header Length	4 bits	*Specifies the length of the IP header (minimum 160 bits)*
DSCP	8 bits	*Classifies traffic for QoS*
Total Length	16 bits	*Specifies the length of both the header and data payload*
Identification	16 bits	*Uniquely identifies fragments of a packet*
Flags	3 bits	*Flags for fragmentation*
Fragment Offset	13 bits	*Identifies the fragment relative to the start of the packet*
Time to Live	8 bits	*Limits the lifetime of a packet*
Protocol	8 bits	*Specifies the next upper layer protocol*
Header Checksum	16 bits	*Checksum for error checking*
Source Address	32 bits	*Source IPv4 address*
Destination Address	32 bits	*Destination IPv4 address*
Options	Variable	*Optional field for various parameters*

The 8-bit **Time to Live (TTL) field** limits the lifetime of the packet, preventing it from being endlessly forwarded. When a router forwards a packet, it will decrement the TTL value by one. Once the TTL value reaches zero, the packet is dropped.

The 8-bit **Protocol field** identifies the next upper-layer header, and is covered in the next section.

The 16-bit **Header Checksum field** is used to error-check the IPv4 header. The receiving host will discard the packet if it fails the checksum calculation.

The 32-bit **Source Address field** identifies the *sending* host. The 32-bit **Destination Address field** identifies the *receiving* host. The value of both of these fields can be changed as the packet is forwarded, using NAT.

The variable-length **Options field** provides additional optional IPv4 parameters, outside the scope of this guide.

IPv4 Protocol Numbers

The 8-bit **Protocol field** specifies the next upper-layer header within the data payload of the packet. These upper-layer protocols are identified using **IP Protocol Numbers**.

The following is a list of common IP Protocol Numbers, as assigned by the IANA:

Protocol Number	**Upper-Layer Protocol**
1	ICMP
2	IGMP
6	TCP
9	IGRP
17	UDP
46	RSVP
47	GRE
50	IPSEC ESP
51	IPSEC AH
88	EIGRP
89	OSPF

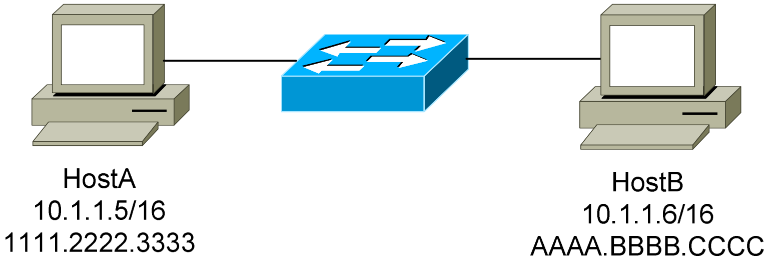
In IPv6, this field is referred to as the **Next Header field.**

(Reference: http://www.iana.org/assignments/protocol-numbers)

Resolving Logical Addresses to Hardware Addresses

A host cannot directly send data to another host's logical address. A destination logical address must be *mapped* to a hardware address, so that the Data-Link layer can package a frame to transmit on the physical medium.

The **Address Resolution Protocol (ARP)** provides this mechanism for IPv4 on Ethernet networks. ARP allows a host to determine the MAC address for a particular destination IP address.



Consider the above diagram. The following demonstrates the steps required for HostA to communicate with HostB:

- First, HostA will determine if the destination IP address of 10.1.1.6 is *itself*. If that address is configured on a local interface, the packet never leaves HostA. In this example, 10.1.1.6 is *not* locally configured on HostA.

- Next, HostA will determine if the 10.1.1.6 address is on the *same network* or *subnet* as itself. HostA consults its **local routing table** to make this determination. In this example, the subnet mask is */16*. Thus, HostA's IP address of 10.1.1.5 and the destination address of 10.1.1.6 are on the same network (*10.1*).

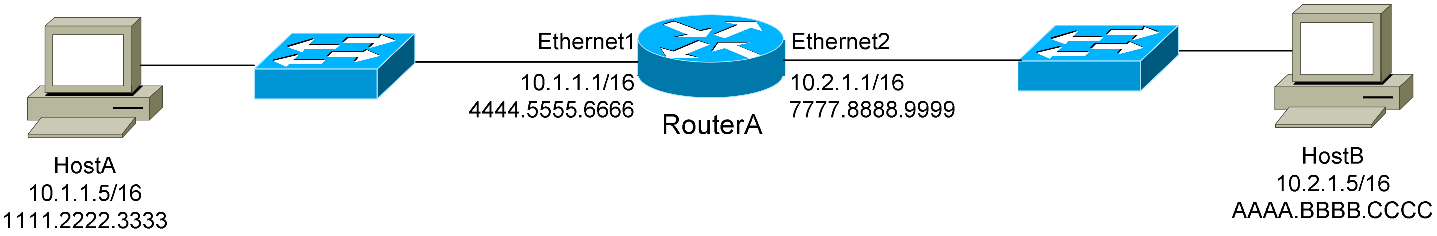
- Because HostA and HostB are on the same network, HostA will then broadcast an **ARP request**, asking for the MAC address of the 10.1.1.6 address.

- HostB responds to the ARP request with an **ARP reply,** containing its MAC address (AAAA.BBBB.CCCC).

- HostA can now construct a Layer-2 frame, with a destination of HostB's MAC address. HostA forwards this frame to the switch, which then forwards the frame to HostB.

Resolving Logical Addresses to Hardware Addresses (continued)

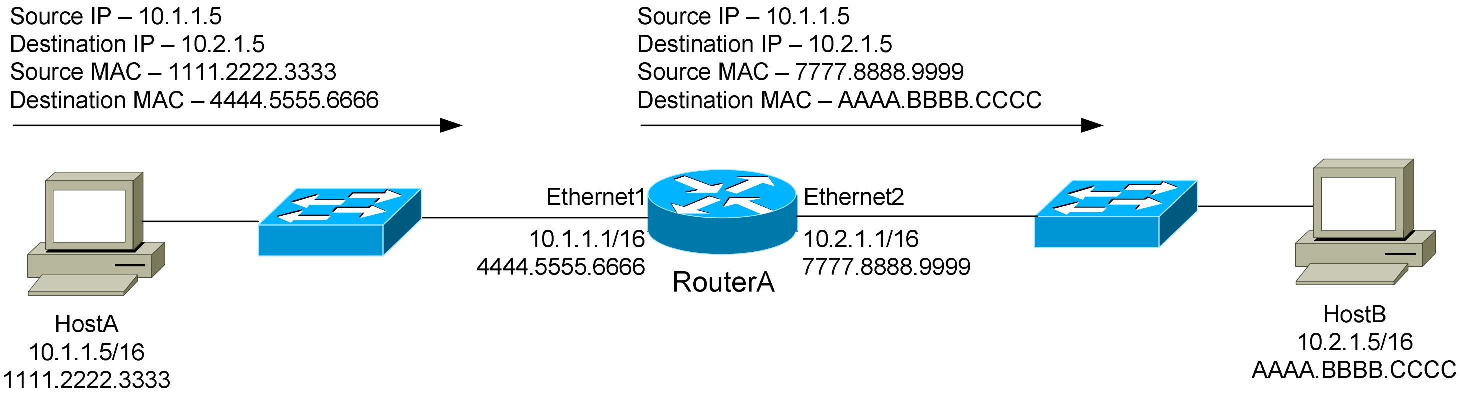
Now consider a slightly modified scenario between HostA and HostB:



- Again, HostA will determine if the destination IP address of 10.2.1.5 is *itself*. In this example, 10.2.1.5 is *not* locally configured on HostA.
- Next, HostA will determine if the 10.2.1.5 address is on the *same network* or *subnet* as itself. In this example, the subnet mask is */16.* Thus, HostA's IP address of 10.1.1.5 and the destination address of 10.2.1.5 are ***not*** on the same network.
- Because HostA and HostB are *not* on the same network, HostA will parse its local routing table for a route to this destination network of 10.2.x.x/16. Hosts are commonly configured with a **default gateway** to reach all other destination networks.
- HostA determines that the 10.1.1.1 address on RouterA is its default gateway. HostA will then broadcast an ARP request, asking for the MAC address of the 10.1.1.1 address.
- RouterA responds to the ARP request with an ARP reply containing its MAC address (4444.5555.6666). HostA can now construct a Layer-2 frame, with a destination of RouterA's MAC address.
- Once RouterA receives the frame, it will parse its own routing table for a route to the destination network of 10.2.x.x/16. It determines that this network is directly attached off of its *Ethernet2* interface. RouterA then broadcasts an ARP request for the 10.2.1.5 address.
- HostB responds to the ARP request with an ARP reply containing its MAC address (AAAA.BBBB.CCCC). RouterA can now construct a Layer-2 frame, with a destination of HostB's MAC address.

Resolving Logical Addresses to Hardware Addresses (continued)

Consider the following example again:



Note that as a packet is *routed,* the source and destination IP address remain unchanged. However, both the source and destination MAC address *did* change.

This is because a MAC address contains no network hierarchy, and thus is only significant on the *local* network. In the above scenario, HostA and HostB could not communicate directly using Layer-2 addressing. At every routed *hop,* the source and destination MAC address are adjusted to reflect the source and destination hosts on the *local* network.

The source and destination IP address will *only* be changed if NAT is used.

The ARP Table

A host can build an **ARP table** that contains a list of IP to MAC address translations. The ARP table is only locally significant to that host. There are two methods to populate an ARP table:

- **Statically**
- **Dynamically**

A **static ARP entry** is created manually on a host, and will remain permanently until purposely removed. More commonly, ARP tables are built **dynamically** by caching ARP replies. Cached entries will eventually be aged out of the ARP table. The aging time will vary depending on the operating system, and can range from several seconds to several hours.

Troubleshooting IP using ICMP

The **Internet Control Message Protocol (ICMP)** is used for a multitude of informational and error messaging purposes.

The following is a list of common ICMP types and codes:

Type	*Code*	*Description*
0	**0**	**Echo Reply**
3	**-**	**Destination Unreachable**
	0	*Network Unreachable*
	1	*Host Unreachable*
	2	*Protocol Unreachable*
	3	*Port Unreachable*
	4	*Fragmentation Needed – Don't Fragment Flag Set*
	6	*Destination Network Unknown*
	7	*Destination Host Unknown*
	9	*Destination Network Administratively Prohibited*
	10	*Destination Host Administratively Prohibited*
5		**Redirect**
8		**Echo**
11		**TTL Exceeded**

The two most common troubleshooting tools that utilize ICMP are:

- **Packet Internet Groper (ping)**
- **Traceroute**

Ping is a core connectivity troubleshooting tool, which utilizes the Echo Request and Echo Reply ICMP messages to determine if an IP address is reachable and responding. Ping will additionally provide the **round-trip time** between the source and destination, usually measured in milliseconds.

Traceroute determines the routing path a packet takes to reach its destination. Traceroute will not only identify each router the packet has been forwarded through, but will also measure the delay experienced at each router *hop*.

Section 2
- IPv6 Addressing -

IPv6 Basics

The most widespread implementation of IP currently is IPv4, which utilizes a 32-bit address. Mathematically, a 32-bit address can provide roughly 4 billion unique IP addresses (2^{32} = 4,294,967,296). Practically, the number of usable IPv4 addresses is much lower, as many addresses are reserved for diagnostic, experimental, or multicast purposes.

The explosive growth of the Internet and corporate networks quickly led to an IPv4 address shortage. Various solutions were developed to alleviate this shortage, including CIDR, NAT, and Private Addressing. However, these solutions could only serve as temporary fixes.

In response to the address shortage, **IPv6** was developed. IPv6 increases the address size to 128 bits, providing a nearly unlimited supply of addresses (340,282,366,920,938,463,463,374,607,431,768,211,456 to be exact). This provides roughly 50 *octillion* addresses *per person alive* on Earth today, or roughly 3.7 x 10^{21} addresses per square inch of the Earth's surface.

(References: http://cc.uoregon.edu/cnews/spring2001/whatsipv6.html; http://en.wikipedia.org/wiki/IPv6)

IPv6 offers the following features:

- **Increased Address Space and Scalability –** *providing the absurd number of possible addresses stated previously.*
- **Simplified Configuration –** *allows hosts to auto-configure their IPv6 addresses, based on network prefixes advertised by routers.*
- **Integrated Security –** *provides built-in authentication and encryption into the IPv6 network header*
- **Compatibility with IPv4 –** *simplifies address migration, as IPv6 is backward-compatible with IPv4*

The IPv6 Address

The IPv6 address is **128 bits**, as opposed to the 32-bit IPv4 address. Also unlike IPv4, the IPv6 address is represented in hexadecimal notation, separate by colons.

An example of an IPv6 address would be:

1254:1532:26B1:CC14:0123:1111:2222:3333

Each "grouping" (from here on called **fields**) of hexadecimal digits is 16 bits, with a total of eight fields. The hexadecimal values of an IPv6 address are **not case-sensitive.**

We can drop any leading zeros in each field of an IPv6 address. For example, consider the following address:

1423:0021:0C13:CC1E:3142:0001:2222:3333

We can condense that address to: 1423:21:C13:CC1E:3142:1:2222:3333

Only leading zeros can be condensed. If we have an entire field comprised of zeros, we can further compact the following address:

F12F:0000:0000:CC1E:2412:1111:2222:3333

The condensed address would be: F12F::CC1E:2412:1111:2222:3333

Notice the double colons (::). We can only condense one set of contiguous zero fields. Thus, if we had the following address:

F12F:0000:0000:CC1E:2412:0000:0000:3333

We could not condense that to: F12F::CC1E:2412::3333

The address would now be ambiguous, as we wouldn't know how many "0" fields were compacted in each spot. Remember that we can only use one set of double colons in an IPv6 address!

The IPv6 Prefix

IPv4 utilizes a *subnet mask* to define the network “prefix” and “host” portions of an address. This subnet mask can also be represented in Classless Inter-Domain Routing (CIDR) format.

IPv6 always use **CIDR** notation to determine what bits **notate the prefix** of an address:

Full Address:	1254:1532:26B1:CC14:123:1111:2222:3333/64
Prefix ID:	1254:1532:26B1:CC14:
Host ID:	123:1111:2222:3333

The */64* indicates that the first 64 bits of this address identify the prefix.

The IPv6 Interface ID and EUI-64 Format

The host portion of an IPv4 address is not based on the hardware address of an interface. IPv4 relies on **Address Resolution Protocol (ARP)** to map between the logical IP address and the **48-bit** hardware **MAC address**.

IPv6 unicasts generally allocate the first 64 bits of the address to identify the network (**prefix**), and the last 64 bits to identify the host (referred to as the **interface ID**). The interface ID *is* based on the interface’s hardware address.

This interface ID adheres to the IEEE **64-bit Extended Unique Identifier** (**EUI-64**) format. Since most interfaces still use the 48-bit MAC address, the MAC must be converted into the EUI-64 format.

Consider the following MAC address: 1111.2222.3333. The first 24 bits, the Organizationally Unique Identifier (OUI), identify the manufacturer. The last 24 bits uniquely identify the host. To convert this to EUI-64 format:

1. The **first 24 bits** of the MAC (the **OUI**), become the first 24 bits of the EUI-64 formatted interface ID.
2. The ***seventh*** bit of the OUI is changed from a “**0**” to a “**1**”.
3. The next 16 bits of the interface ID are **FFFE.**
4. The **last 24 bits** of the MAC (the **host ID**), become the last 24 bits of the interface ID.

Thus, the MAC address 1111.2222.3333 in EUI-64 format would become **1311:22FF:FE22:3333**, which becomes the interface ID.

The IPv6 Address Hierarchy

IPv4 separated its address space into specific **classes.** The class of an IPv4 address was identified by the high-order bits of the first octet:

- **Class A** - (00000001 – 01111111, or 1 - 127)
- **Class B -** (10000000 – 10111111, or 128 - 191)
- **Class C -** (11000000 – 11011111, or 192 - 223)
- **Class D -** (11100000 – 11101111, or 224 - 239)

IPv6's addressing structure is far more scalable. Less than 20% of the IPv6 address space has been designated for use, currently. The potential for growth is enormous.

The address space that *has* been allocated is organized into several types, determined by the high-order bits of the first field:

- **Special Addresses –** addresses begin **00*xx*:**
- **Link Local –** addresses begin **FE8*x*:**
- **Site Local –** addresses begin **FEC*x*:**
- **Aggregate Global –** addresses begin **2*xxx*:** or **3*xxx*:**
- **Multicasts –** addresses begin **FF*xx*:**
- **Anycasts**

(Note: an "x" indicates the value can be any hexadecimal number)

There are **no broadcast addresses** in IPv6. Thus, any IPv6 address that is not a *multicast* is a *unicast* address.

Anycast addresses identify a group of interfaces on multiple hosts. Thus, multiple hosts are configured with an *identical* address. Packets sent to an anycast address are sent to the *nearest* (i.e., least amount of hops) host. Anycasts are indistinguishable from any other IPv6 unicast address.

Practical applications of anycast addressing are a bit murky. One possible application would be a server farm providing an identical service or function, in which case anycast addressing would allow clients to connect to the nearest server.

Special (Reserved) IPv6 Addresses

The first field of a **reserved** or **special** IPv6 address will always begin **00*xx*.** Reserved addresses represent 1/256th of the available IPv6 address space.

Various reserved addresses exist, including:

- **0:0:0:0:0:0:0:0** (or **::**) – is an **unspecified** or **unknown** address. It is the equivalent of the IPv4 0.0.0.0 address, which indicates the absence of a configured or assigned address. In routing tables, the unspecified address is used to identify **all** or **any** possible hosts or networks.
- **0:0:0:0:0:0:0:1** (or **::1**) – is the **loopback** or **localhost** address. It is the equivalent of the IPv4 127.0.0.1 address.

Reserved Addresses - IPv4 and IPv6 Compatibility

To alleviate the difficulties of immediately migrating from IPv4 to IPv6, specific reserved addresses can be used to *embed* an IPv4 address into an IPv6 address.

Two types of addresses can be used for IPv4 embedding, **IPv4-compatible IPv6 addresses**, and **IPv4-mapped IPv6 addresses**.

- **0:0:0:0:0:0:a.b.c.d** (or **::a.b.c.d)** – is an **IPv4-*compatible* IPv6 address**. This address is used on devices that support both IPv4 *and* IPv6. A prefix of /96 is used for IPv4-compatible IPv6 addresses:

  ::192.168.1.1/96

- **0:0:0:0:0:FFFF:a.b.c.d** (or **::FFFF:a.b.c.d) –** is an **IPv4-*mapped* IPv6 address.** This address is used by IPv6 routers and devices to identify *non*-IPv6 capable devices. Again, a prefix of /96 is used for IPv4-mapped IPv6 addresses:

  ::FFFF:192.168.1.1/96

Link-Local IPv6 Addresses

Link-local IPv6 addresses are used only on a single link (subnet). Any packet that contains a link-local source or destination address is *never routed* to another link. Every IPv6-enabled interface on a host (or router) is assigned a link-local address. This address can be manually assigned, or auto-configured.

The first field of a **link-local** IPv6 address will always begin ***FE8x (1111 1110 10).*** Link-local addresses are **unicasts,** and represent 1/1024th of the available IPv6 address space. A prefix of **/10** is used for link-local addresses.

FE80::1311:22FF:FE22:3333/10

There is no hierarchy to a link-local address:

- The first 10 bits are fixed (**FE8**), known as the **Format Prefix (FP).**
- The next 54 bits are set to **0.**
- The final 64 bits are used as the **interface ID**.

Site Local IPv6 Addresses

Site-local IPv6 addresses are the equivalent of "private" IPv4 addresses. Site-local addresses can be routed within a *site* or *organization*, but cannot be globally routed on the Internet. Multiple private subnets within a "site" are allowed.

The first field of a **site-local** IPv6 address will always begin ***FECx (1111 1110 11).*** Site-local addresses are **unicasts,** and represent 1/1024th of the available IPv6 address space.

FEC0::2731:E2FF:FE96:C283/64

Site-local addresses do adhere to a hierarchy:

- The first 10 bits are the fixed FP (**FEC**).
- The next 38 bits are set to **0.**
- The next 16 bits are used to identify the **private subnet ID**.
- The final 64 bits are used as the **interface ID**.

To identify two separate subnets (*1111* and *2222*):

FEC0::1111:2731:E2FF:FE96:C283/64
FEC0::2222:97A4:E2FF:FE1C:E2D1/64

Aggregate Global IPv6 Addresses

Aggregate Global IPv6 addresses are the equivalent of "public" IPv4 addresses. Aggregate global addresses can be routed publicly on the Internet. Any device or site that wishes to traverse the Internet must be uniquely identified with an aggregate global address.

Currently, the first field of an **aggregate global** IPv6 address will always begin **2*xxx* (001).** Aggregate global addresses are **unicasts,** and represent 1/8th of the available IPv6 address space.

2000::2731:E2FF:FE96:C283/64

Aggregate global addresses adhere to a very strict hierarchy:

- The first 3 bits are the fixed FP.
- The next 13 bits are the **top-level aggregation identifier (TLA ID).**
- The next 8 bits are **reserved** for future use.
- The next 24 bits are the **next-level aggregation identifier (NLA ID).**
- The next 16 bits are the **site-level aggregation identifier (SLA ID).**
- The final 64 bits are used as the **interface ID**.

By have multiple **levels**, a consistent, organized, and scalable hierarchy is maintained. High level registries are assigned ranges of TLA IDs. These can then be subdivided in the NLA ID field, and passed on to lower-tiered ISPs.

Such ISPs allocate these prefixes to their customers, which can further subdivide the prefix using the SLA ID field, to create whatever local hierarchy they wish. The 16-bit SLA field provides up to 65535 networks for an organization.

Note: Do not confuse the SLA ID field of a global address field, with a site-local address. Site-local addresses cannot be routed publicly, where as SLA ID's are just a subset of the publicly routable aggregate global address.

Multicast IPv6 Addresses

Multicast IPv6 addresses are the equivalent of IPv4 multicast addresses. Interfaces can belong to one or more multicast **groups**. Interfaces will accept a multicast packet only if they belong to that group. Multicasting provides a much more efficient mechanism than **broadcasting**, which requires that every host on a link accept and process each broadcast packet.

The first field of a **multicast** IPv6 address will always begin **FF*xx* (1111 1111).** The full multicast range is **FF00** through **FFFF**. **Multicasts** represent 1/256th of the available IPv6 address space.

FF01:0:0:0:0:0:0:1

Multicast addresses follow a specific format:

- The first 8 bits **identify the address** as a **multicast** (1111 1111)
- The next 4 bits are a **flag value**. If the flag is set to all zeroes (0000), the multicast address is considered *well-known*.
- The next 4 bits are a **scope value**:
    - 0000 (0) = Reserved
    - 0001 (1) = Node Local Scope
    - 0010 (2) = Link Local Scope
    - 0101 (5) = Site Local Scope
    - 1000 (8) = Organization Local Scope
    - 1110 (e) = Global Scope
    - 1111 (f) = Reserved
- The final 112 bits identify the actual **multicast group**.

IPv4 multicast addresses had no mechanism to support multiple "**scopes.**" IPv6 scopes allow for a multicast hierarchy, a way to *contain* multicast traffic.

* * *

Common IPv6 Multicast Addresses

The following is a list of common, well-known IPv6 multicast addresses:

Node-Local Scope Multicast Addresses

- FF01::1 – All-nodes address
- FF01::2 – All-routers address

Link-Local Scope Multicast Addresses

- FF02::1 – All-nodes address
- FF02::2 – All-routers address
- FF02::5 – OSPFv3 (OSPF IPv6) All SPF Routers
- FF02::6 – OSPFv3 Designated Routers
- FF02::9 – RIPng Routers
- FF02::13 – PIM Routers

Site-Local Scope Multicast Addresses

- FF05::2 – All-routers address

All hosts must join the **all-nodes** multicast group, for both the node-local and link-local scopes. All routers must join the **all-routers** multicast group, for the node-local, link-local, and site-local scopes.

Every site-local and aggregate global address is assigned a **solicited-node multicast** address. This solicited-node address is created by appending the last 24 bits of the interface ID to the following prefix: FF02::1:FF/103.

Thus, if you have a site-local address of:

FEC0::1111:2731:E2FF:FE96:C283

The corresponding solicited-node multicast address would be:

FF02::1:FF96:C283

Solicited-node multicast addresses are most often used for neighbor discovery (covered in an upcoming section in this guide).

Required IPv6 Addresses

At a minimum, each IPv6 interface on a **host** must recognize the following IPv6 addresses:

- The loopback address
- A link-local address
- Any configured site-local or aggregate global addresses
- Any configured multicast groups
- The all-nodes multicast address (both node-local and link-local scopes)
- The solicited-node multicast address for any configured unicast addresses

In *addition* to the above addresses, each IPv6 interface on a **router** must recognize the following IPv6 addresses:

- The subnet-router anycast address
- Any configured multicast groups
- The all-routers multicast address (node-local, link-local, and site-local scopes)

IPv6 Addresses and URLs

IPv6 addresses can also be referenced in **URLs** (Uniform Resource Locator). URL's, however, use the colon to represent a specific TCP "port". This is not an issue with IPv4 addresses, which can easily be referenced using a URL:

http://192.168.1.1/index.html

Because IPv6 fields are separated by colons, the IPv6 address must be placed in brackets, to conform to the URL standard:

http://[FEC0::CC1E:2412:1111:2222:3333]/index.html

The IPv6 Header

The IPv6 header has **8 fields** and is **320 bits** long. It has been considerably streamlined compared to its IPv4 counterpart, which has **12 fields** and is **160 bits** long.

Field	*Length*	*Description*
Version	4 bits	*Version of IP (in this case, IPv6)*
Traffic Class	8 bits	*Classifies traffic for QoS*
Flow Label	20 bits	*Identifies a flow between a source and destination*
Payload Length	16 bits	*Length of data in packet*
Next Header	8 bits	*Specifies the next upper-layer or extension header*
Hop Limit	8 bits	*Decremented by each router traversed*
Source Address	128 bits	*Source IPv6 address*
Destination Address	128 bits	*Destination IPv6 address*

The *Next Header* field is of some importance. This field can identify either the next upper-layer header (for example, UDP, TCP or ICMP), or it can identify a special **Extension Header**, which placed in between the IPv6 and upper layer header.

Several such extension headers exist, and are usually processed in the following order:

- **Hop-by-Hop Options** – *specifies options that should be processed by every router in the path. Directly follows the IPv6 header.*
- **Destination Options –** *specifies options that should be processed by the destination device.*
- **Routing Header** – *specifies each router the packet must traverse to reach the destination (source routing)*
- **Fragment Header** – *used when a packet is larger than the MTU for the path*
- **Authentication Header –** *used to integrate IPSEC Authentication Header (AH) into the IPv6 packet*
- **ESP Header –** *used to integrate IPSEC Encapsulating Security Payload (ESP) into the IPv6 packet*

(Reference: http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122newft/122t/122t2/ipv6/ftipv6o.htm#1004285)

ICMPv6

ICMP Version 6 (**ICMPv6**) is a core component of IPv6. All devices employing IPv6 must also integrate ICMPv6.

ICMPv6 provides many services, including (but not limited to):

- Error Messages
- Informational messages (such as *echo* replies for IPv6 ping)
- MTU Path Discovery
- Neighbor Discovery

There are four key ICMPv6 error messages:

- **Destination Unreachable** (ICMP packet type 1**) –** *indicates that the packet cannot be forwarded to its destination. The node sending this message includes an explanatory code:*
  - **0 -** No route to destination
  - **1** - Access is administratively prohibited
  - **3** - Address unreachable
  - **4** - Port unreachable

- **Packet Too Big (**ICMP packet type 2) – *indicates the packet is larger than the MTU of the link. IPv6* **routers do not fragment** *packets. Instead, the Packet Too Big message is sent to the source (sending) device, which then reduces (or fragments) the size of the packet to the reported MTU. This message is used for* ***Path MTU Discovery (PMTUD)****.*

- **Time Exceeded (**ICMP packet type 3) – *indicates that the hop count limit has been reached, usually indicating a routing loop*

- **Parameter Problem (**ICMP packet type 4**) –** *indicates an error in the IPv6 header, or an IPv6 extension header. The node sending this message includes an explanatory code:*
  - **0 -** Erroneous header field
  - **1 -** Unrecognized next-header type
  - **2 -** Unrecognized IPv6 option

*(*Reference: *http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a0080113b1c.shtml)*

Neighbor Discovery Protocol (NDP) and ICMPv6

The **neighbor discovery protocol (NDP)** provides a multitude of services for IPv6 enabled devices, including:

- Automatic address configuration, and prefix discovery
- Duplicate address detection
- MTU discovery
- Router discovery
- Address resolution

NDP replaces many IPv4 specific protocols, such as DHCP and ARP. NDP utilizes **ICMPv6** to provide the above services.

Periodically, IPv6 routers send out **Router Advertisements (RA's)** to both announce their presence on a link, and to provide auto-configuration information for hosts. This **RA** (ICMP packet type 134) is sourced from the link-local address of the sending router, and sent to the link-scope all-nodes multicast group. The sending router sets a **hop limit** of **255** on a RA; however, the RA packet *must not* be forwarded outside the local link.

Hosts use RA's to configure themselves, and add the router to its local default router list. A host can request an RA by sending out a **Router Solicitation (RS,** ICMP packet type 133) to the link-local all-routers multicast address. A RS is usually sent when a host is *not* currently configured with an IP address.

The RA messages contain the following information for hosts:

- The **router's link-layer address** (to be added to the host's default router list)
- One or more **network prefixes**
- A **lifetime** (measured in seconds) for the prefix(es)
- The link **MTU**

Routers send **Redirect** messages to hosts, indicating a *better* route to a destination. Hosts can have multiple routers in its default router list, but one is chosen as the *true* default router. If this default router deems that another router has a better route to the destination, it forwards the Redirect message to the sending host.

Neighbor Discovery Protocol (NDP) and ICMPv6 (continued)

Neighbor Solicitations (NS's, ICMP packet type 135**)** are sent by hosts to identify the link-layer address of a neighbor, and ensure its reachability. A NS message's source address is the link-local address of the sending host, and the destination is the **solicited-node multicast** address of the destination host.

A neighbor will reply to a NS with a **Neighbor Advertisement (NA,** ICMP packet type 136). This process replaces the Address Resolution Protocol (ARP) used by IPv4, and provides a far more efficient means to learn neighbor address information.

Hosts additionally use the NS messages to **detect duplicate addresses**. Before a host assigns itself an IPv6 address, it sends out a NS to ensure no other host is configured with that address.

Autoconfiguration of Hosts

Hosts can be assigned IPv6 addresses one of two ways: manually, or using autoconfiguration. Hosts learn how to autoconfigure themselves from **Router Advertisements (RA's).**

Two types of autoconfiguration exist, **stateless** and **stateful**.

When using **Stateless Autoconfiguration**, a host first assigns itself a link-local IPv6 address. It accomplishes this by combining the link-local prefix (FE8) with its interface ID (MAC address in EUI-64 format).

The host then sends a **Router Solicitation** multicast to the all-routers multicast address, which provides one or more network prefixes. The host combines these prefixes with its interface ID to create its site-local (or aggregate global) IPv6 addresses.

Stateful Autoconfiguration is used in conjunction with stateless autoconfiguration. Stateful Autoconfiguration utilizes DHCPv6 to provide additional information to the host, such as DNS servers. DHCPv6 can also be used in the event that there is no router on the link, to provide stateless autoconfiguration.

Configuring IPv6 Addresses

IPv6 support is **disabled** by default on Cisco routers, and must be enabled globally:

```
Router(config)#  ipv6 unicast-routing
```

To configure an interface to auto-configure a link-local IPv6 address:

```
Router(config)#  interface e0
Router(config-if)#  ipv6 enable
```

To manually configure a site-local IPv6 address on an interface:

```
Router(config)#  interface e0
Router(config-if)#  ipv6 address FEC0::/64 eui-64
```

The *eui-64* parameter will append interface ID (MAC address in EUI-64 format) to the site-local prefix. Otherwise, we could have specified the full IPv6 address:

```
Router(config-if)#  ipv6 address FEC0::1:1234:23FF:FE21:1212 eui-64
```

Recall that we can configure multiple subnets for our site-local address space:

```
Router(config)#  interface e0
Router(config-if)#  ipv6 address FEC0::2222:0:0:0:0/64 eui-64
```

To configure a router interface to advertise a specific prefix to hosts on the link:

```
Router(config)#  interface e0
Router(config-if)#  ipv6 nd prefix-advertisement 2002:1111::/48 2000 1000 onlink autoconfig
```

The router will *advertise* a *prefix* of *2002:1111::/48* with a **valid lifetime** of *2000* seconds and a **preferred lifetime** of *1000* seconds. The clients will *autoconfig* themselves based on this prefix.

To view IPv6 specific information about an interface:

```
Router#  show ipv6 interface e0
```

To create a static host entry for an IPv6 address:

```
Router(config)#  ipv6 host MYHOST FEC0::1111:2731:E2FF:FE96:C283
```

Configuring IPv6 Static Routes

The syntax to configure an IPv6 static route is simple:

```
Router(config)# ipv6 route FEC0::2222/64 FEC0::1111:3E5F:2E5B:A3D1
```

The above command creates an *ipv6 route* to the *FEC0::2222/64* network, with a next-hop of *FEC0::1111:3E5F:2E5B:A3D1*.

To create an IPv6 default route:

```
Router(config)# ipv6 route ::/0 FE80::2
```

The above command creates an *ipv6* default *route*, with a next hop of *FE80::2*. The *::/0* designation indicates all zeros in the address field, and a mask of zero bits (the **unspecified** address).

To view the IPv6 routing table:

```
Router(config)# show ipv6 route
```

Configuring IPv6 RIPng

A version of RIP for IPv6 was developed called **RIPng** (RIP Next Generation). Functionally, RIPng is the equivalent of RIPv2, with the additional support for IPv6 addresses. However, RIPng is *not* backwards with earlier version of RIP, and does not support IPv4 addressing.

Basic RIPng characteristics:

- Administrative distance of 120
- Maximum hopcount of 16
- Updates are sent every 30 seconds as multicasts

To configure RIPng, we must first enable the RIP process globally:

```
Router(config)#  ipv6 router rip MYPROCESS
```

We are enabling an *ipv6 rip* process called *MYPROCESS.* Next, we must enable RIPng on each participating interface:

```
Router(config)#  interface e0
Router(config-if)#  ipv6 rip MYPROCESS enable
```

RIPng, by default, utilizes **UDP** port **521** and multicast group **FF02::9,** but these parameters can be changed globally:

```
Router(config)#  ipv6 rip MYPROCESS port 555 multicast-group FF02::1111
```

We can adjust RIPng's timers:

```
Router(config)#  ipv6 rip MYPROCESS timers 30 180 180 120
```

In order, the above timers are *update, expire, holddown,* and *garbage-collect*. The above values are default.

To control inbound or outbound RIPng updates, using an access-list:

```
Router(config)#  interface e0
Router(config-if)#  ipv6 rip MYPROCESS input-filter MYACCESSLIST
Router(config-if)#  ipv6 rip MYPROCESS output-filter MYACCESSLIST
```

To view configuration and status information for RIPng:

```
Router#  show ipv6 protocols
Router#  show ipv6 rip
```

Configuring IPv6 OSPF (OSPFv3)

OSPFv2 is a widely used link-state routing protocol in IPv4 environments. To support IPv6, **OSPFv3** was developed. Its function is very similar to OSPFv2.

First, we must first enable the OSPF process globally:

```
Router(config)#  ipv6 router ospf 1
```

The *1* indicates the process ID. Next, we must place the participating interfaces in their appropriate areas:

```
Router(config)#  interface e0
Router(config-if)#  ipv6 ospf 1 area 0

Router(config)#  interface s0
Router(config-if)#  ipv6 ospf 1 area 1
```

Please note: the Router ID for OSPFv3 is still a 32-bit value. Thus, the highest IPv4 loopback address will be chosen first, then the highest IPv4 physical address. If neither exist, a 32-bit Router ID must be manually specified:

```
Router(config)#  ipv6 router ospf 1
Router(config-router)#  router-id 1.1.1.1
```

To create a summarized route on an area boundary:

```
Router(config)#  ipv6 router ospf 1
Router(config-router)#  area range 2001:1111::/48
```

To view configuration and status information for OSPFv3:

```
Router#  show ipv6 ospf neighbor
Router#  show ipv6 ospf interface
```

To clear an OSPFv3 process:

```
Router#  clear ipv6 ospf 1
```

Configuring IPv6 BGP

BGP-4 does not natively support IPv6. Support for IPv6 and other protocols (such as IPX) are included in the **BGP Multi-protocol Extensions**.

Basic BGP configuration using IPv6 is identical to that of IPv4:

```
Router(config)#  router bgp 100
Router(config-router)#  neighbor 2005:2222::1 remote-as 200
```

Notice the use of an aggregate global IPv6 address in the *neighbor* statement.

Additional information is required - we must *activate* the neighbor. This allows the neighbor to share IPv6 routes with the local router:

```
Router(config)#  router bgp 100
Router(config-router)#  address-family ipv6
Router(config-router-af)#  neighbor 2005:2222::1 activate
```

To advertise an IPv6 prefix into BGP:

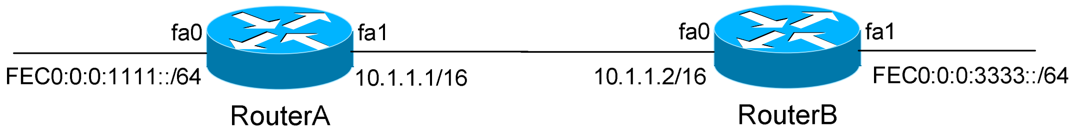
```
Router(config)#  router bgp 100
Router(config-router)#  address-family ipv6
Router(config-router-af)#  network 2005:1111:: /24
```

To view configuration and status information for IPv6 BGP:

```
Router#  show bgp ipv6

Router#  show bgp ipv6 summary
```

(Reference: http://www.cisco.com/en/US/products/sw/iosswrel/ps5187/products_configuration_guide_chapter09186a00801d65f7.html)

Configuring an IPv6 Tunnel



We can configure an IPv6 "tunnel" across an IPv4 link. To accomplish this, we create a virtual tunnel interface on both RouterA and RouterB.

```
RouterA(config)# ipv6 unicast-routing

RouterA(config)# interface fa0
RouterA(config-if)# ipv6 address FEC0:0:0:1111::/64 eui-64

RouterA(config)# interface fa1
RouterA(config-if)# ip address 10.1.1.1 255.255.0.0

RouterA(config)# interface tunnel0
RouterA(config-if)# no ip address
RouterA(config-if)# ipv6 address FEC0:0:0:2222::1/124
RouterA(config-if)# tunnel source fa1
RouterA(config-if)# tunnel destination 10.1.1.2
RouterA(config-if)# tunnel mode ipv6ip
```

Configuration on Router B:

```
RouterB(config)# ipv6 unicast-routing

RouterB(config)# interface fa0
RouterB(config-if)# ip address 10.1.1.2 255.255.0.0

RouterB(config)# interface fa1
RouterB(config-if)# ipv6 address FEC0:0:0:3333::/64 eui-64

RouterB(config)# interface tunnel0
RouterB(config-if)# no ip address
RouterB(config-if)# ipv6 address FEC0:0:0:2222::2/124
RouterB(config-if)# tunnel source fa1
RouterB(config-if)# tunnel destination 10.1.1.1
RouterB(config-if)# tunnel mode ipv6ip
```

We've applied an IPv6 address on the *FEC0:0:0:2222::/124* network. IPv6 traffic can now route across the 10.1.x.x/16 IPv4 network. Any routing protocol configuration for IPv6 should be completed on the tunnel interfaces.

IPv6 Access-Lists

Cisco IOS 12.0(23) or later supports IPv6 access-lists. The configuration is similar to that of IPv4 named access-lists (All IPv6 access-lists are **named**; there are *no* IPv6 numbered access-lists).

```
Router(config)#  ipv6 access-list MYLIST
Router(config-access-list)#  deny ipv6 any 2001:1111::/64
Router(config-access-list)#  permit ipv6 any any

Router(config)#  interface fa0/0
Router(config-if)#  ipv6 traffic-filter MYLIST in
```

Notice the use of a */prefix*, as opposed to a *wildcard* mask.

Also, notice the use of the *ipv6 traffic-filter* command to apply the ACL to the interface, as opposed to *ip access-group*.

Hurray for consistency!

(Reference: *http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122newft/122t/122t2/ipv6/ftipv6c.htm#1064881*)

Section 3
- TCP and UDP -

Transport Layer Protocols

The **Transport layer (OSI Layer-4)** does *not* actually transport data, despite its name. Instead, this layer is responsible for the *reliable* transfer of data, by ensuring that data arrives at its destination error-free and in order.

The Transport layer is referred to as the **Host-to-Host layer** in the Department of Defense (DoD) reference model.

Transport layer communication falls under two categories:

- **Connection-oriented** – requires that a connection with specific agreed-upon parameters be established before data is sent.
- **Connectionless** – requires no connection before data is sent.

Connection-oriented protocols provide several important services:

- **Connection establishment –** connections are established, maintained, and ultimately terminated between devices.
- **Segmentation and sequencing –** data is *segmented* into smaller pieces for transport. Each segment is assigned a *sequence number*, so that the receiving device can reassemble the data on arrival.
- **Acknowledgments –** receipt of data is confirmed through the use of *acknowledgments*. If a segment is lost, data can be retransmitted to guarantee delivery.
- **Flow control** (or **windowing) –** data transfer rate is negotiated to prevent congestion.

The TCP/IP protocol suite incorporates two Transport layer protocols:

- **Transmission Control Protocol (TCP) –** connection-oriented
- **User Datagram Protocol (UDP) -** connectionless

Both TCP and UDP provide a mechanism to differentiate applications running on the same host, through the use of **port numbers.** When a host receives a packet, the port number tells the transport layer which higher-layer application to hand the packet off to.

Both TCP and UDP will be covered in detail in this guide. Please note that the *best* resource on the Internet for TCP/UDP information is the exemplary TCP/IP Guide, found here: http://www.tcpipguide.com/free/index.htm

Port Numbers and Sockets

Both TCP and UDP provide a mechanism to differentiate applications (or **services**) running on the same host, through the use of **port numbers.** When a host receives a segment, the port number tells the transport layer which higher-layer application to hand the packet off to. This allows multiple network services to operate simultaneously on the same logical address, such as a web *and* an email server.

The range for port numbers is **0 – 65535**, for both TCP and UDP.

The combination of the IP address and port number (identifying both the *host* and *service)* is referred to as a **socket**, and is written out as follows:

192.168.60.125:443

Note the colon separating the IP address (*192.168.60.125*) from the port number (*443*).

The first 1024 ports (**0-1023**) have been reserved for widely-used services, and are recognized as **well-known** ports. Below is a table of several common TCP/UDP ports:

Port Number	Transport Protocol	Application
20, 21	TCP	FTP
22	TCP	SSH
23	TCP	Telnet
25	TCP	SMTP
53	UDP or TCP	DNS
80	TCP	HTTP
110	TCP	POP3
443	TCP	SSL
666	TCP	Doom

Ports ranging from **1024 – 49151** are referred to as **registered ports**, and are allocated by the IANA upon request. Ports ranging from **49152 – 65535** cannot be registered, and are considered **dynamic.** A client *initiating* a connection will randomly choose a port in this range as its *source* port (for some operating systems, the dynamic range starts at *1024* and higher).

For a complete list of assigned port numbers, refer to the IANA website:

http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml

Transmission Control Protocol (TCP)

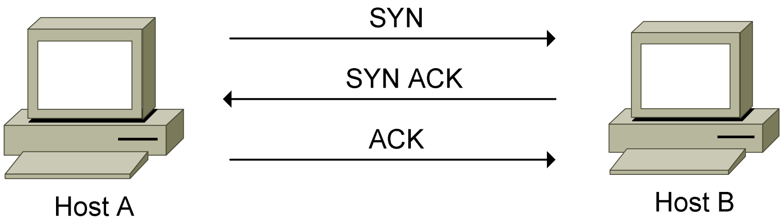
The **Transmission Control Protocol (TCP)** is a **connection-oriented** transport protocol, providing reliable delivery over an Internet Protocol (IP) network. Together, TCP and IP provide the core functionality for the **TCP/IP** or **Internet protocol suite**.

TCP was originally defined in RFC 675, and initially designed to perform *both* Network and Transport layer functions. When this proved to be an inflexible solution, those functions were separated - with IP providing Network layer services, and TCP providing Transport layer services. This separation was formalized in **version 4** of TCP, defined in RFC 793.

Because TCP is connection-oriented, parameters must be agreed upon by both the sending and receiving devices before a connection is established.

Establishing a TCP Connection

TCP employs a **three-way handshake** to form a connection. Control messages are passed between the two hosts as the connection is set up:

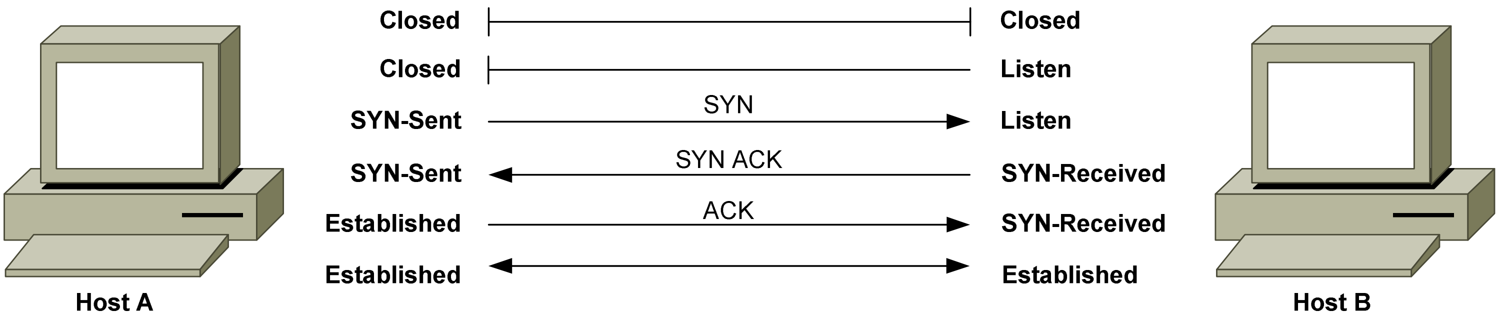


- HostA sends a **SYN** (short for **synchronize)** message to HostB to initiate a connection.
- HostB responds with an **ACK** (short for **acknowledgement)** to HostA's SYN message, and sends its own **SYN** message. The two messages are combined to form a single **SYN+ACK** message.
- HostA completes the three-way handshake by sending an **ACK** to HostB's SYN.

The TCP header contains six different **flags,** including a SYN flag and an ACK flag. Thus, when a particular *type* of message needs to be sent, the appropriate flag is marked as **on,** or changed from a *0* to a *1*. A SYN+ACK message has both flags set to on (*1*).

Establishing a TCP Connection (continued)

As the three-way handshake occurs, the sending and receiving hosts will pass through several **states:**



A **closed** state indicates a complete absence of a TCP connection.

Before a host can accept a request for a TCP connection, the host must enter a **listen** state, also known as a ***passive* open**. For example, a web server will passively *listen* on the HTTP port, waiting for incoming connection requests. A host must listen on each port it wishes to accept connections on.

A host will enter a **SYN-sent** state once it sends a **SYN message** to initiate a connection, also known as an ***active* open**. The sending host will remain in this state as it waits for the remote host's **ACK message.**

The receiving host will respond to the SYN message with a **SYN+ACK message,** and enter a **SYN-received** state**.**

The sending host will respond to the SYN+ACK message with its own **ACK message** and enter an **Established** state. The receiving host will enter an Established state once it receives this final ACK.

An Established state indicates that data transfer can occur. The communication becomes **bidirectional**, regardless of which host initiated the connection.

TCP can support many simultaneous connections, and must track and maintain each connection individually. Connections are identified by the *sockets* of both the source and destination host, and data specific to each connection is maintained in a **Transmission Control Block (TCB).**

(Reference: http://www.tcpipguide.com/free/t_TCPConnectionEstablishmentProcessTheThreeWayHandsh-3.htm)

TCP Segmentation and Sequencing

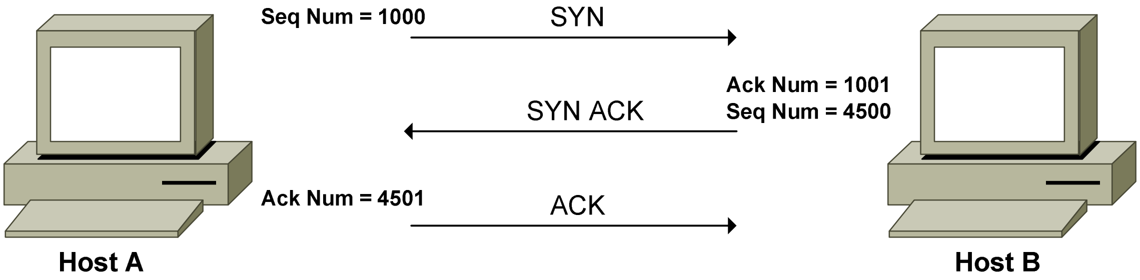
TCP is a **stream-oriented** transport protocol. This allows the application layer to send a continuous stream of unstructured data and rely on TCP to package the data as **segments**, regardless of the amount of data.

TCP will not only segment data into smaller pieces for transport, but will also assign a **sequence number** to each segment. Note though that this sequence number identifies the data (bytes) *within* the segment rather than the segment *itself.*

Sequencing serves two critical purposes:

- It allows the receiving host to reassemble the data from multiple segments in the correct order, upon arrival.
- It allows receipt of data within a segment to be *acknowledged,* thus providing a mechanism for dropped segments to be detected and resent.

When establishing a connection, a host will choose a 32-bit **initial sequence number (ISN).** The ISN is chosen from a randomizing timer, to prevent accidental overlap or predictability.

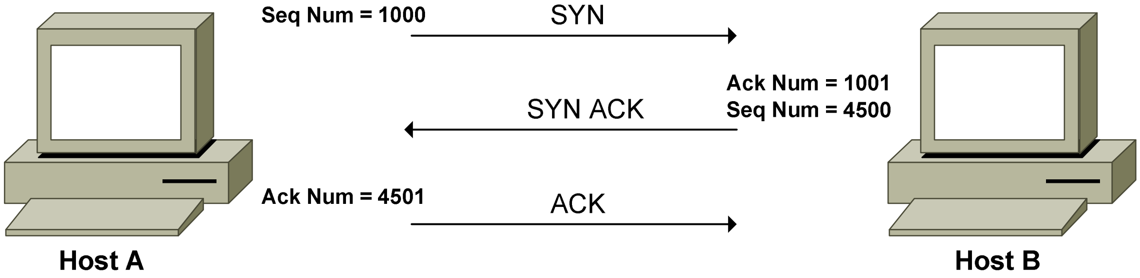


The receiving host responds to this sequence number with an **acknowledgment number**, set to the **sequence number + 1**. In the above example, HostB's acknowledgment number would thus be *1001*.

HostB includes an initial sequence number with *its* SYN message as well – *4500* in the above example. HostA would respond to this sequence number with an acknowledgement number of *4501.*

The TCP header contains both a 32-bit Sequence Number and 32-bit Acknowledgement Number field.

TCP Sliding Window



Once the TCP connection is established, the sequence numbers are used to identify the *data* within the segment. Using the above example again, HostA's first byte of data will be assigned a sequence number *1001*. Note that this is HostB's acknowledgment number, which essentially identifies which byte the receiving host is expecting *next*. HostB's first byte of data will be assigned a sequence number of *4501*.

Note that each *individual* byte of data **is not** assigned a sequence number and acknowledged independently, as this would introduce massive overhead. Instead, data is sequenced and acknowledged in *groups*, dictated by the **TCP window size.** The window size can never exceed the **maximum segment size (MSS)**, which is **536 bytes** by default.

The TCP window size is dictating by the *receiving host*, and informs the sender how many bytes it is permitted to send, before waiting for an acknowledgement. This window size can be dynamically changed to provide a measure of **flow control,** preventing buffer congestion on the receiving host.

A window size of **0** would instruct the sender to send no further data, usually indicating significant congestion on the receiving host.

TCP employs a ***sliding* window mechanism**. Bytes in a sliding window fall into one of four categories:

- Bytes that have *already* been sent *and* acknowledged.
- Bytes that have been sent, but *not acknowledged.*
- Bytes that have *not yet been sent*, but the receiving host *is ready for*.
- Bytes that have *not yet been sent*, and the receiving host *is not ready for.*

(Reference: http://www.tcpipguide.com/free/t_TCPSlidingWindowAcknowledgmentSystemForDataTranspo-5.htm; http://docwiki.cisco.com/wiki/Internet_Protocols#Transmission_Control_Protocol_.28TCP.29)

TCP Sliding Window (continued)

Consider the following conceptual example:

	Byte #	***Category***
	1-50	Bytes sent and acknowledged
TCP Window	51-75	Bytes sent and not yet acknowledged
	76-100	Bytes not sent, receiving host is ready
	101-200	Bytes not sent, receiving host is *not ready*

Several determinations can be made:

- The TCP stream is 200 bytes total.
- The TCP window size is 50 bytes total.
- The sending host can immediately send another 25 bytes of data (bytes 76-100)

Once bytes 51-75 are acknowledged, and bytes 76-100 are sent, the window will *slide* down*:*

	Byte #	***Category***
	1-75	Bytes sent and acknowledged
TCP Window	76-100	Bytes sent and not yet acknowledged
	101-125	Bytes not sent, receiving host is ready
	126-200	Bytes not sent, receiving host is *not ready*

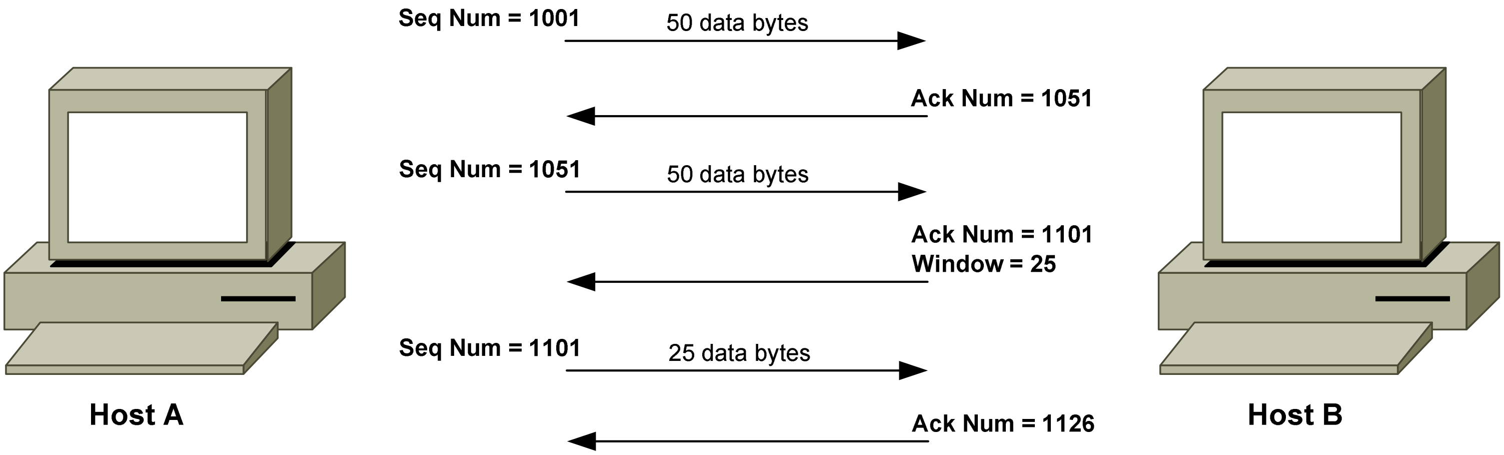
This assumes that that TCP window stays at 50 bytes. Remember that the window size is dictated by the receiving host (in a 16-bit **Window field** in the TCP header), and can be dynamically adjusted.

For efficiency, TCP will generally wait to send a segment until the agreed-upon TCP window size is full. This may not be acceptable for certain types of applications, which may not tolerate this latency.

The TCP header provides a **PSH (Push) flag** to accommodate this, allowing data to be sent *immediately*, regardless if the TCP window has been filled. The PSH flag can be used in conjunction with the **URG (Urgent) flag,** which allows specified data to be prioritized over other data. The URG flag must be used with the **Urgent Pointer field**, which identifies the *last* byte of urgent data, to identify where non-urgent data begins in a segment.

TCP Sliding Window (continued)

How do sequence and acknowledgement numbers fit within the sliding window concept? Consider the following *very* basic example, and assume the TCP connection is already established:



Recall that during the setup of a TCP connection, the acknowledgement number was set to the sequence number + 1. However, during data transfer, the acknowledgement number is used to acknowledge receipt of a *group* of data bytes.

In the above example, the initial TCP window size is set to *50* bytes, and the first byte in the stream is assigned a sequence number of *1001*. HostB acknowledges receipt of these 50 data bytes with an acknowledgement number of *1051* (for the mathematically disinclined, this is 1001 + 50). ☺

Once acknowledged, HostA then sends another 50 bytes of data, identifying the first byte with a sequence number of 1051. HostB acknowledges receipt again, with an ACK number of 1101. However, HostB also adjusts the TCP window size to 25 bytes, perhaps due to congestion.

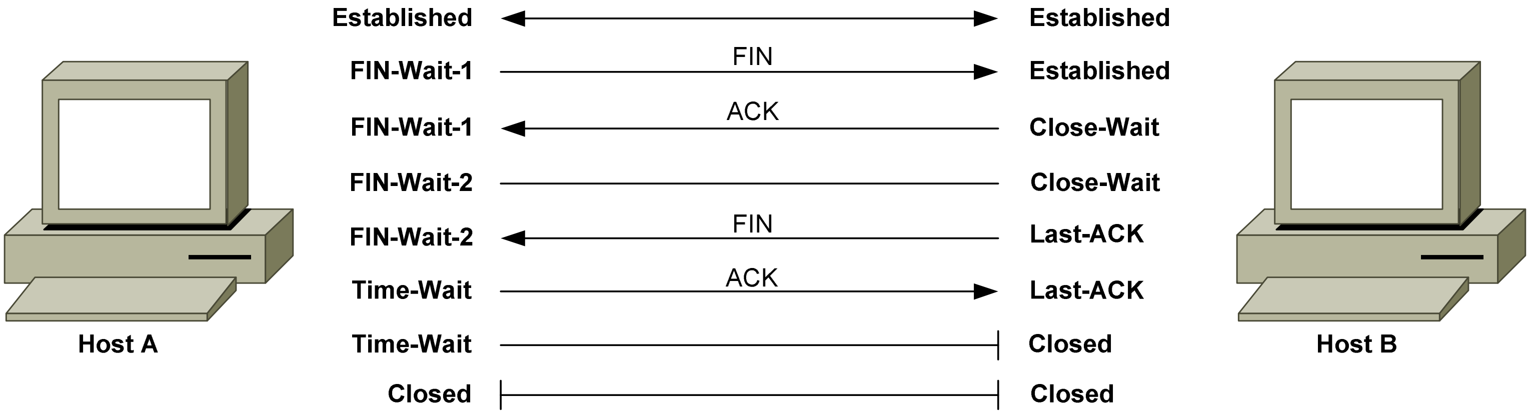
HostA's next segment will thus only contain 25 bytes of data, with a sequence number of 1101. HostB acknowledges these 25 bytes with an ACK number of 1126.

Every time a segment is sent, the sending host starts a **retransmission timer,** dynamically determined (and adjusted) based on the round-trip time between the two hosts. If an acknowledgement is not received before the retransmission timer expires, the segment is resent. This allows TCP to guarantee delivery, even when segments are lost.

Gracefully Terminating a TCP Connection

A TCP connection will remain established until it is purposely **terminated** by either host. The most common reason for connection termination is that both hosts have finished sending data. The termination process is handled separately by each host, allowing both hosts to fully complete data transfer before the connection is terminated.

Hosts can terminate an established TCP connection by sending a message with the **FIN (Finish) flag** set:



Once HostA sends the FIN message, it will enter a **FIN-Wait-1** state, waiting for the FIN to be acknowledged.

HostB responds to the FIN with an ACK message, and enters a **Close-Wait** state, allowing the local application to finish its processes. HostA receives the ACK and enters a **FIN-Wait-2** state, waiting for HostB to send a FIN message of its own, indicating it is safe to close the connection.

HostB sends a FIN message to HostA once the application process is complete, and enters a **Last-ACK** state.

HostA receives the FIN message and responds with an ACK message. HostA then enters a **Time-Wait** state, allowing time for the ACK to be received by HostB.

HostB receives the ACK message and enters a **Closed** state.

HostA's Time-Wait timer expires, and it also enters a Closed state. The connection is now gracefully terminated.

(Reference: http://www.tcpipguide.com/free/t_TCPConnectionTermination-2.htm)

Less than Graceful TCP Connection Termination

A TCP connection can become **half-open,** indicating that one host is an established state while the other is not. Half-open connections can result from **interruption** by an intermediary device (such as a firewall), or from a software or hardware issue.

TCP utilizes the **Reset message,** using the **RST flag**, to address half-open connections. Sending a RST message will force the remote host to reset the TCP connection and return to a *closed* state, or return to a passive *listen* state if the remote host is a server listening on that port.

There are a few scenarios in which a RST might be sent:

- A host receives a TCP segment from a host that it does not have a connection with.
- A host receives a segment with an incorrect sequence or acknowledgement number.
- A host receives a SYN request on a port it is not listening on.

Note on half-open connections: A **SYN flood** is a common denial-of-service attack that sends a large number of TCP SYN messages to a host, while spoofing the source address. The host will respond with an equal number of SYN+ACK messages, and will wait for the final ACK message that never comes. This spawns a large amount of half-open connections, which can prevent the host from responding to legitimate requests.

Modern firewalls can detect SYN flood attacks and minimize the number of accepted half-open connections.

(Reference: http://www.tcpipguide.com/free/t_TCPConnectionManagementandProblemHandlingtheConnec.htm)

The TCP Header

The TCP header is comprised of **12 fields,** and has a minimum size of **160 bits (20 bytes)**:

Field	***Length***	***Description***
Source Port	16 bits	*Source TCP Port*
Destination Port	16 bits	*Destination TCP Port*
Sequence Number	32 bits	*Sequence Number*
Ack Number	32 bits	*Acknowledgement Number*
Data Offset	4 bits	*Indicates where the data begins in a TCP segment*
Reserved	6 bits	*Always set to 0*
Control Bits	6 bits	*URG, ACK, PSH, RST, SYN, and FIN flags*
Window	16 bits	*Used for Flow Control*
Checksum	16 bits	*Used for Error-Checking*
Urgent Pointer	16 bits	*Identifies last byte of Urgent traffic*
Options	Variable	
Padding	Variable	*To ensure the TCP header ends at a 32-bit boundary*

The 16-bit **Source Port field** identifies the application service on the *sending* host. The 16-bit **Destination Port field** identifies the application service on the *remote* host.

The 32-bit **Sequence Number field** is used both during connection establishment, and during data transfer. During connection establishment (SYN message), an *initial sequence number* is randomly chosen. Subsequently, sequence numbers are used to identify data bytes in a stream.

The 32-bit **Acknowledgement Number field**, as its name suggests, is used to acknowledge a sequence number. During connection setup, this is set to the sending host's initial sequence number + 1. During data transfer, this value is used to acknowledge receipt of a group of data bytes.

The 4-bit **Data Offset field** indicates where data begins in a TCP segment, by identifying the number of 32-bit multiples in the TCP header. A TCP header *must* end on a 32-bit boundary.

Following the data offset field is the 6-bit **Reserved** (for future use) **field,** which is always set to zeroes.

The TCP Header (continued)

Field	*Length*	*Description*
Source Port	16 bits	*Source TCP Port*
Destination Port	16 bits	*Destination TCP Port*
Sequence Number	32 bits	*Sequence Number*
Ack Number	32 bits	*Acknowledgement Number*
Data Offset	4 bits	*Indicates where the data begins in a TCP segment*
Reserved	6 bits	*Always set to 0*
Control Bits	6 bits	*URG, ACK, PSH, RST, SYN, and FIN flags*
Window	16 bits	*Used for Flow Control*
Checksum	16 bits	*Used for Error-Checking*
Urgent Pointer	16 bits	*Identifies last byte of Urgent traffic*
Options	Variable	
Padding	Variable	*To ensure the TCP header ends at a 32-bit boundary*

The 6-bit **Control Bits** field contains six 1-bit flags, in the following order:

- **URG (Urgent) –** prioritizes specified traffic.
- **ACK (Acknowledgment) –** acknowledges a SYN or receipt of data.
- **PSH (Push) –** forces an immediate send even if window is not full.
- **RST (Reset) –** forcefully terminates an improper connection.
- **SYN (Synchronize) –** initiates a connection.
- **FIN (Finish) –** gracefully terminates a connection when there is further data to send.

The 16-bit **Window field** identifies the number of data octets that the receiver is able to accept.

The 16-bit **Checksum field** is used for error-checking, and is computed using both the TCP segment and select fields from the IP header. The receiving host will discard the segment if it fails the checksum calculation.

The 16-bit **Urgent Pointer field** is used to identify the *last* byte of prioritized traffic in a segment, when the URG flag is set.

The variable-length **Options field** provides additional optional TCP parameters, outside the scope of this guide.

The variable-length **Padding field** ensures the TCP header ends on a 32-bit boundary, and is always set to zeroes.

User Datagram Protocol (UDP)

The **User Datagram Protocol (UDP)** is a **connectionless** transport protocol, and is defined in RFC 768.

UDP, above all, is *simple.* It provides no three-way handshake, no flow-control, no sequencing, and no acknowledgment of data receipt. UDP essentially forwards the segment and takes no further interest.

Thus, UDP is **inherently unreliable,** especially compared to a connection-oriented protocol like TCP. However, UDP **experiences less latency** than TCP, due to the reduced overhead. This makes UDP ideal for applications that require speed over reliability. For example, DNS primarily uses UDP as its transport protocol, though it supports TCP as well.

Like TCP, UDP does provide basic error-checking using a checksum, and uses port numbers to differentiate applications running on the same host.

The UDP header has only 4 **fields**:

Field	***Length***	***Description***
Source Port	16 bits	*Source UDP Port*
Destination Port	16 bits	*Destination UDP Port*
Length	16 bits	*Length of the header and the data*
Checksum	16 bits	*Used for Error-Checking*

The following provides a quick comparison of TCP and UDP:

TCP	***UDP***
Connection-oriented	Connectionless
Guarantees delivery	Does *not* guarantee delivery
Sends acknowledgments	Does *not* send acknowledgments
Reliable, but slower than UDP	Unreliable, but faster than TCP
Segments and sequences data	Does *not* provide sequencing
Resends dropped segments	Does *not* resend dropped segments
Provides flow control	Does *not* provide flow control
Performs CRC on data	Also performs CRC on data
Uses port numbers	Also uses port numbers

Part II

Basic Routing Concepts

Section 4
- The Routing Table -

Routing Table Basics

Routing is the process of sending a packet of information from one *network* to another *network.* Thus, routes are usually based on the destination network, and not the destination host (host routes *can* exist, but are used only in rare circumstances).

To route, routers build Routing Tables that contain the following:

- The destination network and subnet mask
- The "next hop" router to get to the destination network
- Routing *metrics* and Administrative Distance

The routing table is concerned with two types of protocols:

- A **routed** protocol is a layer 3 protocol that applies logical addresses to devices and routes data between networks. Examples would be IP and IPX.
- A **routing** protocol dynamically builds the network, topology, and next hop information in routing tables. Examples would be RIP, IGRP, OSPF, etc.

To determine the *best* route to a destination, a router considers three elements (in this order):

- **Prefix-Length**
- **Metric** (*within* a routing protocol)
- **Administrative Distance** *(between* separate routing protocols)

Prefix-length is the number of bits used to identify the network, and is used to determine the most *specific* route. A longer prefix-length indicates a more specific route. For example, assume we are trying to reach a host address of 10.1.5.2/24. If we had routes to the following networks in the routing table:

10.1.5.0/24
10.0.0.0/8

The router will do a bit-by-bit comparison to find the most *specific* route (i.e., longest matching prefix). Since the 10.1.5.0/24 network is more specific, that route will be used, ***regardless of metric or Administrative Distance.***

Administrative Distance vs. Metric

A “**metric**” allows a router to choose the best path *within* a routing protocol. Distance vector routing protocols use “**distance**” (usually hop-count) as their metric. Link state protocols utilize some sort of “**cost**” as their metric.

Only routes with the **best metric** are **added to the routing table**. Thus, even if a particular routing protocol (for example, RIP) has four routes to the same network, only the route with the *best* metric (hop-count in this example) would make it to the routing table. If multiple equal-metric routes exist to a particular network, most routing protocols will load-balance.

If your router is running multiple routing protocols, **Administrative Distance** is used to determine which routing protocol to *trust* the most. *Lowest* administrative distance wins.

Again: if a router receives two RIP routes to the same network, it will use the routes’ **metric** to determine which path to use. If the metric is identical for both routes, the router will load balance between both paths.

If a router receives a RIP and an OSPF route to the same network, it will use **Administrative Distance** to determine which routing path to choose.

The Administrative Distance of common routing protocols (remember, lowest wins):

Connected	0
Static	1
EIGRP Summary	5
External BGP	20
Internal EIGRP	90
IGRP	100
OSPF	110
IS-IS	115
RIP	120
External EIGRP	170
Internal BGP	200
Unknown	255

A route with an “unknown” Administrative Distance will never be inserted into the routing table.

Viewing the routing table

The following command will allow you to view the routing table:

```
Router# show ip route
```

```
Gateway of last resort is 192.168.1.1 to network 0.0.0.0

C      192.168.1.0/24 is directly connected, Ethernet0
       150.50.0.0/24 is subnetted, 1 subnets
C      150.50.200.0 is directly connected, Loopback1
C      192.168.123.0 is directly connected, Serial0
C      192.168.111.0 is directly connected, Serial1
R      10.0.0.0 [120/1] via 192.168.123.1, 00:00:00, Serial0
                [120/1] via 192.168.111.2, 00:00:00, Serial1
S*     0.0.0.0/0 [1/0] via 192.168.1.1
```

Routes are labeled based on what protocol placed them in the table:

- C – Directly connected
- S – Static
- S* - Default route
- D - EIGRP
- R – RIP
- I – IGRP
- i – IS-IS
- O - OSPF

Notice the RIP routes contain the following field: **[120/1**]. This indicates both the administrative distance and the metric (the *120* is the AD, and the *1* is the hop-count metric).

To clear all routes from the routing table, and thus forcing any routing protocol to repopulate the table:

```
Router# clear ip route *
```

Choosing the Best Route (Example)

Assume the following routes existed to the following host: 192.168.111.5/24

```
O     192.168.111.0/24 [110/58] via 192.168.131.1, 00:00:00, Serial3
R     192.168.111.0/24 [120/1] via 192.168.123.1, 00:00:00, Serial0
R     192.168.111.0/24 [120/5] via 192.168.5.2, 00:00:00, Serial1
S     192.168.0.0/16 [1/0] via 10.1.1.1
```

We have two RIP routes, an OSPF route, and a Static route to that destination. Which route will be chosen by the router?

Remember the three criteria the router considers:

- **Prefix-Length**
- **Metric**
- **Administrative Distance**

The static route has the lowest administrative distance (**1**) of any of the routes; however, its **prefix-length** is less specific. 192.168.111.0/24 is a more specific route than 192.168.0.0/16. Remember, prefix-length is *always* considered first.

The second RIP route **will not be inserted** into the routing table, because it has a higher metric **(5)** than the first RIP route (**1**). Thus, our routing table will actually look as follows:

```
O     192.168.111.0/24 [110/58] via 192.168.131.1, 00:00:00, Serial3
R     192.168.111.0/24 [120/1] via 192.168.123.1, 00:00:00, Serial0
S     192.168.0.0/16 [1/0] via 10.1.1.1
```

Thus, the true choice is between the OSPF route and the first RIP route. OSPF has the lowest administrative distance, and thus that route will be preferred.

PLEASE NOTE: Calculating the lowest metric route within a routing protocol occurs *before* administrative distance chooses the route it "trusts" the most. This is why the order of the above "criteria" is prefix-length, metric, and *then* administrative distance.

However, the route with the lowest administrative distance is *always* preferred, regardless of metric (assuming the prefix-length is equal). Thus, the metric is *calculated* first, but not *preferred* first over AD.

Section 5
- Classful vs. Classless Routing -

Classful vs Classless routing protocols

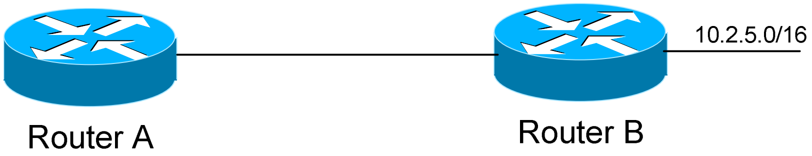
Classful routing protocols do not send subnet mask information with their routing updates. A router running a classful routing protocol will react in one of two ways when receiving a route:

- If the router has a directly connected interface belonging to the same **major network**, it will apply the same subnet mask as that interface.
- If the router *does not* have any interfaces belonging to the same major network, it will apply the *classful* subnet mask to the route.

Belonging to same "major network" simply indicates that they belong to the same "classful" network. For example:

- 10.3.1.0 and 10.5.5.0 belong to the same major network (10.0.0.0)
- 10.1.4.5 and 11.1.4.4 *do not* belong to the same major network
- 192.168.1.1 and 192.168.1.254 belong to the same major network (192.168.1.0)
- 192.168.1.5 and 192.167.2.5 *do not* belong to the same major network

Take the following example (assume the routing protocol is classful):



If Router B sends a routing update to Router A, it will *not* include the subnet mask for the 10.2.0.0 network. Thus, Router A must make a decision.

If Router A has a directly connected interface that belongs to the same major network (10.0.0.0), it will use the subnet mask of that interface for the route. For example, if Router A has an interface on the 10.4.0.0/16 network, it will apply a subnet mask of /16 to the 10.2.0.0 network.

If Router A *does not* have a directly connected interfacing belonging to the same major network, it will apply the classful subnet mask of /8. This can obviously cause routing difficulties.

When using classful routing protocols, the subnet mask must remain consistent throughout your entire network.

Classful vs Classless routing protocols (continued)

Classless routing protocols *do* send the subnet mask with their updates. Thus, Variable Length Subnet Masks (VLSMs) are allowed when using classless routing protocols.

Examples of *classful* routing protocols include RIPv1 and IGRP.

Examples of *classless* routing protocols include RIPv2, EIGRP, OSPF, and IS-IS.

The IP Classless Command

The preceding section described how classful and classless protocols differ when sending *routing updates.* Additionally, the router itself can operate either "classfully" or "classlessly" when actually *routing data*.

When a "classful" router has an interface connected to a major network, *it believes it knows all routes connected to that major network.*

For example, a router may have an interface attached to the 10.1.5.0/24 network. It may also have routes from a routing protocol, also for the 10.x.x.x network.

However, if the classful router receives a packet destined for a 10.x.x.x subnet that is *not* in the routing table, it will *drop* that packet, ***even if there is a default route***.

Again, a classful router believes it knows all possible destinations in a major network.

To configure your router in "classful" mode:

```
Router(config)#  no ip classless
```

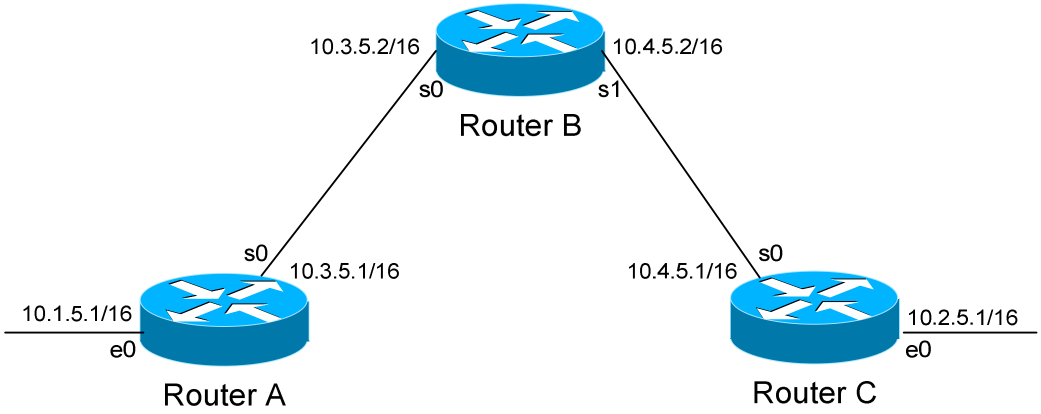
To configure your router in "classless" mode (this is default in IOS 12.0 and greater):

```
Router(config)#  ip classless
```

(Reference: http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a0080094823.shtml)

Limitations of Classful Routing Example

The following section will illustrate the limitations of classful routing, using RIPv1 as an example. Consider the following diagram:



This particular scenario will work when using RIPv1, despite the fact that we've subnetted the major 10.0.0.0 network. Notice that the subnets are contiguous (that is, they belong to the same major network), and use the same subnet mask.

When Router A sends a RIPv1 update to Router B via Serial0, it will not include the subnet mask for the 10.1.0.0 network. However, because the 10.3.0.0 network is in the same major network as the 10.1.0.0 network, it will **not summarize** the address. The route entry in the update will simply state "10.1.0.0".

Router B will accept this routing update, and realize that the interface receiving the update (Serial0) belongs to the same major network as the route entry of 10.1.0.0. It will then apply the subnet mask of its Serial0 interface to this route entry.

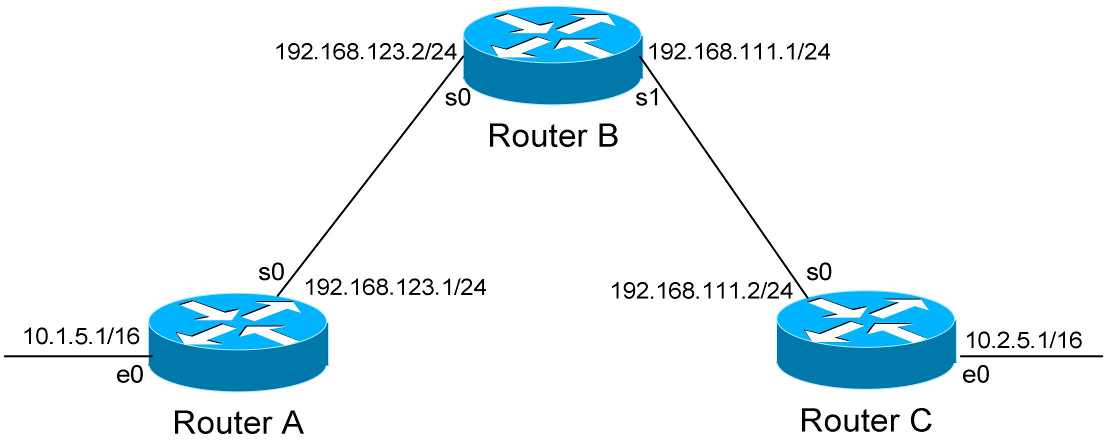
Router C will similarly send an entry for the 10.2.0.0 network to Router B. Router B's routing table will thus look like:

RouterB# *show ip route*

```
Gateway of last resort is not set

       10.0.0.0/16 is subnetted, 4 subnets
C      10.3.0.0 is directly connected, Serial0
C      10.4.0.0 is directly connected, Serial1
R      10.1.0.0 [120/1] via 10.3.5.1, 00:00:00, Serial0
R      10.2.0.0 [120/1] via 10.4.5.1, 00:00:00, Serial1
```

Limitations of Classful Routing Example

Consider the following, slightly altered, example:



We'll assume that RIPv1 is configured correctly on all routers. Notice that our networks are no longer contiguous. Both Router A and Router C contain *subnets* of the 10.0.0.0 major network (10.1.0.0 and 10.2.0.0 respectively).

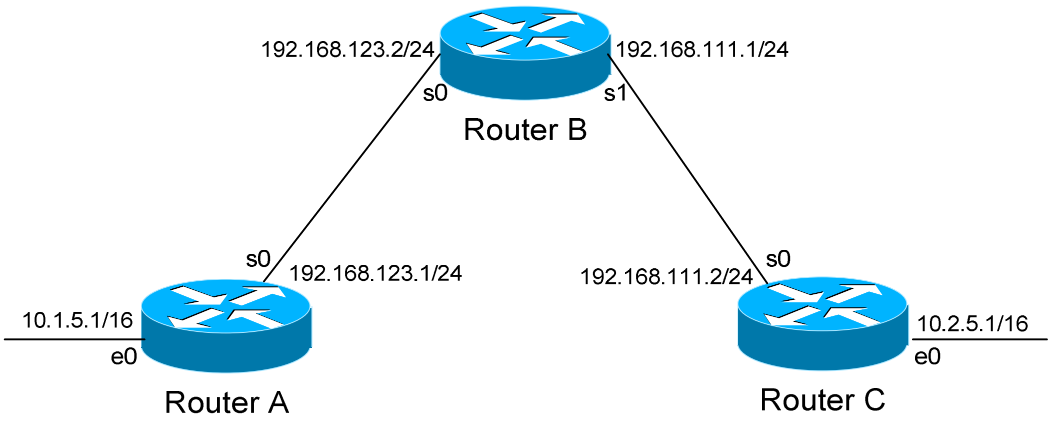
Separating these networks now are two Class C subnets (192.168.123.0 and 192.168.111.0).

Why is this a problem? Again, when Router A sends a RIPv1 update to Router B via Serial, it will not include the subnet mask for the 10.1.0.0 network. Instead, Router A will consider itself a **border** router, as the 10.1.0.0 and 192.168.123.0 networks *do not* belong to the same major network. Router A will **summarize** the 10.1.0.0/16 network to its classful boundary of 10.0.0.0/8.

Router B will accept this routing update, and realize that it does not have a directly connected interface in the 10.x.x.x scheme. Thus, it has no subnet mask to apply to this route. Because of this, Router B will install the summarized 10.0.0.0 route into its routing table.

Router C, similarly, will consider itself a border router between networks 10.2.0.0 and 192.168.111.0. Thus, Router C will *also* send a summarized 10.0.0.0 route to Router B.

Limitations of Classful Routing Example



Router B's routing table will then look like:

RouterB# *show ip route*

```
Gateway of last resort is not set

C     192.168.123.0 is directly connected, Serial0
C     192.168.111.0 is directly connected, Serial1
R     10.0.0.0 [120/1] via 192.168.123.1, 00:00:00, Serial0
               [120/1] via 192.168.111.2, 00:00:00, Serial1
```

That's right, Router B now has two *equal* metric routes to get to the summarized 10.0.0.0 network, one through Router A and the other through Router C. Router B will now *load balance* all traffic to *any* 10.x.x.x network between routers A and C. Suffice to say, this is not a good thing. ☺

It gets better. Router B then tries to send routing updates to Router A and Router C, including the summary route of 10.0.0.0/8. Router A's routing table looks like:

RouterA# *show ip route*

```
Gateway of last resort is not set

C     192.168.123.0 is directly connected, Serial0
       10.0.0.0/16 is subnetted, 1 subnet
C     10.1.0.0 is directly connected, Ethernet0
```

Router A will receive the summarized 10.0.0.0/8 route from Router B, and will reject it. This is because it already has the summary network of 10.0.0.0 in its routing table, and it's directly connected. Router C will respond exactly the same, and the 10.1.0.0/16 and 10.2.0.0/16 networks will never be able to communicate.

Section 6
- Static vs. Dynamic Routing -

Static vs. Dynamic Routing

There are two basic methods of building a routing table:

- **Static Routing**
- **Dynamic Routing**

A **static** routing table is created, maintained, and updated by a network administrator, *manually*. A static route to *every* network must be configured on *every* router for full connectivity. This provides a granular level of control over routing, but quickly becomes impractical on large networks.

Routers will *not* share static routes with each other, thus reducing CPU/RAM overhead and saving bandwidth. However, static routing is *not fault-tolerant*, as any change to the routing infrastructure (such as a link going down, or a new network added) requires manual intervention. Routers operating in a purely static environment cannot seamlessly choose a better route if a link becomes unavailable.

Static routes have an Administrative Distance (AD) of **1**, and thus are always preferred over dynamic routes, unless the default AD is changed. A static route with an adjusted AD is called a **floating static route**.

A **dynamic** routing table is created, maintained, and updated by a *routing protocol* running on the router. Examples of routing protocols include **RIP** (Routing Information Protocol), **EIGRP** (Enhanced Interior Gateway Routing Protocol), and **OSPF** (Open Shortest Path First). Specific dynamic routing protocols are covered in great detail in other guides.

Routers *do* share dynamic routing information with each other, which increases CPU, RAM, and bandwidth usage. However, routing protocols are capable of dynamically choosing a different (or better) path when there is a change to the routing infrastructure.

Do not confuse *routing* protocols with *routed* protocols:

- A ***routed*** protocol is a Layer 3 protocol that applies logical addresses to devices and routes data between networks (such as IP)
- A ***routing*** protocol dynamically builds the network, topology, and next hop information in routing tables (such as RIP, EIGRP, etc.)

Static vs. Dynamic Routing (continued)

The following briefly outlines the advantages and disadvantages of *static* routing:

Advantages of Static Routing

- Minimal CPU/Memory overhead
- No bandwidth overhead (updates are not shared between routers)
- Granular control on how traffic is routed

Disadvantages of Static Routing

- Infrastructure changes must be manually adjusted
- No "dynamic" fault tolerance if a link goes down
- Impractical on large network

The following briefly outlines the advantages and disadvantages of *dynamic* routing:

Advantages of Dynamic Routing

- Simpler to configure on larger networks
- Will dynamically choose a different (or better) route if a link goes down
- Ability to load balance between multiple links

Disadvantages of Dynamic Routing

- Updates are shared between routers, thus consuming bandwidth
- Routing protocols put additional load on router CPU/RAM
- The choice of the "best route" is in the hands of the routing protocol, and not the network administrator

Dynamic Routing Categories

There are two distinct categories of dynamic routing protocols:

- **Distance-vector protocols**
- **Link-state protocols**

Examples of distance-vector protocols include **RIP** and **IGRP**. Examples of link-state protocols include **OSPF** and **IS-IS**.

EIGRP exhibits both distance-vector and link-state characteristics, and is considered a *hybrid* protocol.

Distance-vector Routing Protocols

All **distance-vector** routing protocols share several key characteristics:

- **Periodic** updates of the **full** routing table are sent to routing neighbors.
- Distance-vector protocols suffer from slow convergence, and are highly susceptible to loops.
- Some form of *distance* is used to calculate a route's metric.
- The **Bellman-Ford algorithm** is used to determine the shortest path.

A distance-vector routing protocol begins by advertising directly-connected networks to its neighbors. These updates are sent *regularly* (RIP – every 30 seconds; IGRP – every 90 seconds).

Neighbors will add the routes from these updates to their own routing tables. Each neighbor trusts this information *completely*, and will forward their full routing table (connected *and* learned routes) to every other neighbor. Thus, routers fully (and blindly) rely on neighbors for route information, a concept known as **routing by rumor**.

There are several disadvantages to this behavior. Because routing information is propagated from neighbor to neighbor via periodic updates, distance-vector protocols suffer from slow convergence. This, in addition to blind faith of neighbor updates, results in distance-vector protocols being highly susceptible to routing loops.

Distance-vector protocols utilize some form of **distance** to calculate a route's metric. RIP uses **hopcount** as its distance metric, and IGRP uses a composite of **bandwidth** and **delay**.

Link-State Routing Protocols

Link-state routing protocols were developed to alleviate the convergence and loop issues of distance-vector protocols. Link-state protocols maintain three separate tables:

- **Neighbor table** – contains a list of all neighbors, and the interface each neighbor is connected off of. Neighbors are formed by sending **Hello** packets.
- **Topology table** – otherwise known as the "link-state" table, contains a map of all links within an **area**, including each link's status.
- **Shortest-Path table** – contains the *best* routes to each particular destination (otherwise known as the "routing" table")

Link-state protocols do *not* "route by rumor." Instead, routers send updates advertising the *state* of their *links* (a **link** is a directly-connected network). All routers know the state of all existing links within their **area**, and store this information in a **topology** table. All routers within an area have *identical* topology tables.

The best route to each link (network) is stored in the **routing** (or **shortest-path**) table. If the state of a link changes, such as a router interface failing, an advertisement containing ***only*** **this link-state change** will be sent to all routers within that area. Each router will adjust its topology table accordingly, and will calculate a new *best* route if required.

By maintaining a consistent topology table among all routers within an area, link-state protocols can **converge very quickly** and are **immune to routing loops.**

Additionally, because updates are sent only during a link-state change, and contain *only* the change (and not the full table), link-state protocols are **less bandwidth intensive** than distance-vector protocols. However, the three link-state tables **utilize more RAM and CPU** on the router itself.

Link-state protocols utilize some form of **cost**, usually based on bandwidth**,** to calculate a route's metric. The **Dijkstra formula** is used to determine the shortest path.

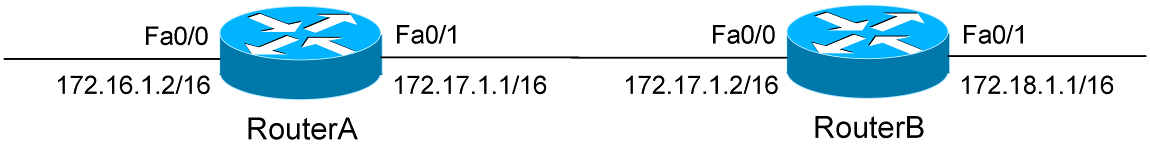
Section 7
- Configuring Static Routes -

Configuring Static Routes

The basic syntax for a static route is as follows:

Router(config)# *ip route [destination_network] [subnet_mask] [next-hop]*

Consider the following example:



RouterA will have the 172.16.0.0/16 and 172.17.0.0/16 networks in its routing table as directly-connected routes. To add a static route on RouterA, pointing to the *172.18.0.0/16* network off of RouterB:

RouterA(config)# *ip route 172.18.0.0 255.255.0.0 172.17.1.2*

Notice that we point to the IP address on RouterB's fa0/0 interface as the *next-hop* address. Likewise, to add a static route on RouterB, pointing to the *172.16.0.0/16* network off of RouterA:

RouterB(config)# *ip route 172.16.0.0 255.255.0.0 172.17.1.1*

To remove a static route, simply type *no* in front of it:

RouterA(config)# *no ip route 172.18.0.0 255.255.0.0 172.17.1.2*

On point-to-point links, an **exit-interface** can be specified instead of a next-hop address. Still using the previous diagram as an example:

RouterA(config)# *ip route 172.18.0.0 255.255.0.0 fa0/1*

RouterB(config)# *ip route 172.16.0.0 255.255.0.0 fa0/0*

A static route using an exit-interface has an Administrative Distance of **0,** as opposed to the default AD of **1** for static routes. An exit-interface is only functional on a point-to-point link, as there is only one possible next-hop device.

Advanced Static Routes Parameters

The Administrative Distance of a static route can be changed to form a **floating static route**, which will only be used if there are no other routes with a lesser AD in the routing table. A floating static route is often used as a *backup* route to a dynamic routing protocol.

To change the Administrative Distance of a static route to *250*:

```
RouterA(config)#  ip route 172.18.0.0 255.255.0.0 172.17.1.2 250
```

Static routes will only remain in the routing table as long as the interface connecting to the next-hop router is up. To ensure a static route remains *permant*ly in the routing table, even if the next-hop interface is down:

```
RouterA(config)#  ip route 172.18.0.0 255.255.0.0 172.17.1.2 permanent
```

Static routes can additionally be used to *discard* traffic to specific networks, by directing that traffic to a virtual *null* interface:

```
RouterA(config)#  ip route 10.0.0.0 255.0.0.0 null0
```

Default Routes

Normally, if a specific route to a particular network does not exist, a router will drop all traffic destined to that network.

A **default route,** or **gateway of last resort**, allows traffic to be forwarded, even without a specific route to a particular network.

The default route is identified by all zeros in both the network and subnet mask (*0.0.0.0 0.0.0.0).* It is the *least* specific route possible, and thus will *only* be used if a more specific route does not exist (hence “gateway of last resort”).

To configure a default route:

```
RouterA(config)#  ip route 0.0.0.0 0.0.0.0 172.17.1.2
```

Advanced default routing is covered in great detail in another guide.

Section 8
- Default Routing -

Default Routing

Normally, if a specific route to a particular network does not exist, a router will drop all traffic destined to that network. A **default route,** or **gateway of last resort**, allows traffic to be forwarded, even without a specific route to a particular network.

The default route is identified by all zeros in both the network and subnet mask (*0.0.0.0 0.0.0.0).* It is the *least* specific route possible, and thus will *only* be used if a more specific route does not exist (hence "gateway of last resort").

To configure a default route:

```
Router(config)#  ip route 0.0.0.0 0.0.0.0 172.17.1.2
```

It is possible to specify an entire **default network** on a Cisco device**:**

```
Router(config)#  ip default-network 172.20.0.0
```

The *172.20.0.0* network must already exist in the routing table (either statically or dynamically), and will be marked as the gateway of last resort.

If IP routing is *disabled* on a Cisco IOS device, the following command will configure a **default-gateway**:

```
Router(config)#  no ip routing
Router(config)#  ip default-gateway 192.168.1.1
```

Essentially, the Cisco router will act as a *host* device, and will perform no routing functions on behalf of other hosts. The router will simply forward its *own* locally-originated traffic to the default-gateway, assuming that traffic is destined for a remote network.

It is possible to generate a default route in most routing protocols (RIP, OSPF, IS-IS, & BGP) using the *default-information originate* command:

```
Router(config)#  router rip
Router(config-router)#  default-information originate
```

(Reference: http://www.cisco.com/warp/public/105/default.html)

Part III

Dynamic Routing Protocols

Section 9
- Routing Information Protocol -

RIP (Routing Information Protocol)

RIP is a standardized Distance Vector protocol, designed for use on smaller networks. RIP was one of the first true Distance Vector routing protocols, and is supported on a wide variety of systems.

RIP adheres to the following Distance Vector characteristics:

- RIP sends out periodic routing updates (every **30 seconds**)
- RIP sends out the full routing table every periodic update
- RIP uses a form of distance as its metric (in this case, **hopcount**)
- RIP uses the Bellman-Ford Distance Vector algorithm to determine the best "path" to a particular destination

Other characteristics of RIP include:

- RIP supports IP and IPX routing.
- RIP utilizes UDP port 520
- RIP routes have an administrative distance of **120**.
- RIP has a maximum hopcount of **15 hops.**

Any network that is 16 hops away or more is considered unreachable to RIP, thus the maximum diameter of the network is 15 hops. A metric of 16 hops in RIP is considered a **poison route** or **infinity metric.**

If multiple paths exist to a particular destination, RIP will load balance between those paths (by default, up to **4**) only if the metric (hopcount) is **equal**. RIP uses a round-robin system of load-balancing between equal metric routes, which can lead to **pinhole congestion**.

For example, two paths might exist to a particular destination, one going through a 9600 baud link, the other via a T1. If the metric (hopcount) is equal, RIP will load-balance, sending an equal amount of traffic down the 9600 baud link and the T1. This will (obviously) cause the slower link to become congested.

RIP Versions

RIP has two versions, **Version 1** (**RIPv1**) and **Version 2** (**RIPv2**).

RIPv1 (RFC 1058) is **classful**, and thus does not include the subnet mask with its routing table updates. Because of this, RIPv1 does not support **Variable Length Subnet Masks (VLSMs**). When using RIPv1, networks must be contiguous, and subnets of a major network must be configured with identical subnet masks. Otherwise, route table inconsistencies (or worse) will occur.

RIPv1 sends updates as **broadcasts** to address 255.255.255.255.

RIPv2 (RFC 2543) is **classless**, and thus *does* include the subnet mask with its routing table updates. RIPv2 fully supports VLSMs, allowing discontiguous networks and varying subnet masks to exist.

Other enhancements offered by RIPv2 include:

- Routing updates are sent via **multicast**, using address **224.0.0.9**
- Encrypted authentication can be configured between RIPv2 routers
- Route tagging is supported (explained in a later section)

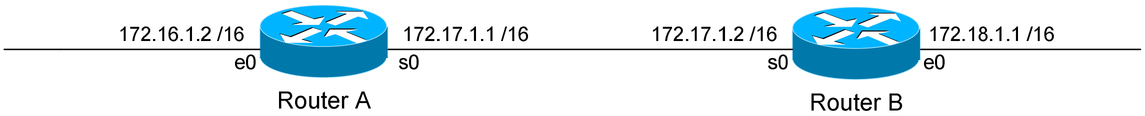
RIPv2 can interoperate with RIPv1. By default:

- RIPv1 routers will sent only Version 1 packets
- RIPv1 routers will receive both Version 1 and 2 updates
- RIPv2 routers will both send and receive only Version 2 updates

We can control the version of RIP a particular interface will "send" or "receive."

Unless RIPv2 is manually specified, a Cisco will default to RIPv1 when configuring RIP.

RIPv1 Basic Configuration



Routing protocol configuration occurs in Global Configuration mode. On Router A, to configure RIP, we would type:

Router(config)# *router rip*
Router(config-router)# *network 172.16.0.0*
Router(config-router)# *network 172.17.0.0*

The first command, *router rip*, enables the RIP process.

The *network* statements tell RIP which networks you wish to advertise to other RIP routers. We simply list the networks that are directly connected to our router. Notice that we specify the networks at their classful boundaries, and we do not specify a subnet mask.

To configure Router B:

Router(config)# *router rip*
Router(config-router)# *network 172.17.0.0*
Router(config-router)# *network 172.18.0.0*

The routing table on Router A will look like:

RouterA# *show ip route*

```
<eliminated irrelevant header>

Gateway of last resort is not set

C     172.16.0.0 is directly connected, Ethernet0
C     172.17.0.0 is directly connected, Serial0
R     172.18.0.0 [120/1] via 172.17.1.2, 00:00:00, Serial0
```

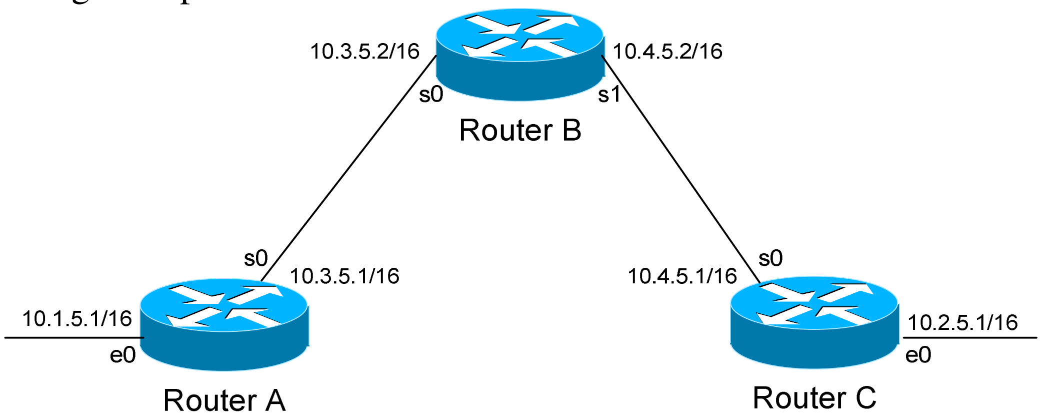
The routing table on Router B will look like:

RouterB# *show ip route*

```
<eliminated irrelevant header>

Gateway of last resort is not set

C     172.17.0.0 is directly connected, Serial0
C     172.18.0.0 is directly connected, Ethernet0
R     172.16.0.0 [120/1] via 172.17.1.1, 00:00:00, Serial0
```

Limitations of RIPv1

The example on the previous page works fine with RIPv1, because the networks are contiguous and the subnet masks are consistent. Consider the following example:



This particular scenario will *still* work when using RIPv1, despite the fact that we've subnetted the major 10.0.0.0 network. Notice that the subnets are contiguous (that is, they belong to the same major network), and use the same subnet mask.

When Router A sends a RIPv1 update to Router B via Serial0, it will not include the subnet mask for the 10.1.0.0 network. However, because the 10.3.0.0 network is in the same major network as the 10.1.0.0 network, it will **not summarize** the address. The route entry in the update will simply state "10.1.0.0".

Router B will accept this routing update, and realize that the interface receiving the update (Serial0) belongs to the same major network as the route entry of 10.1.0.0. It will then apply the subnet mask of its Serial0 interface to this route entry.

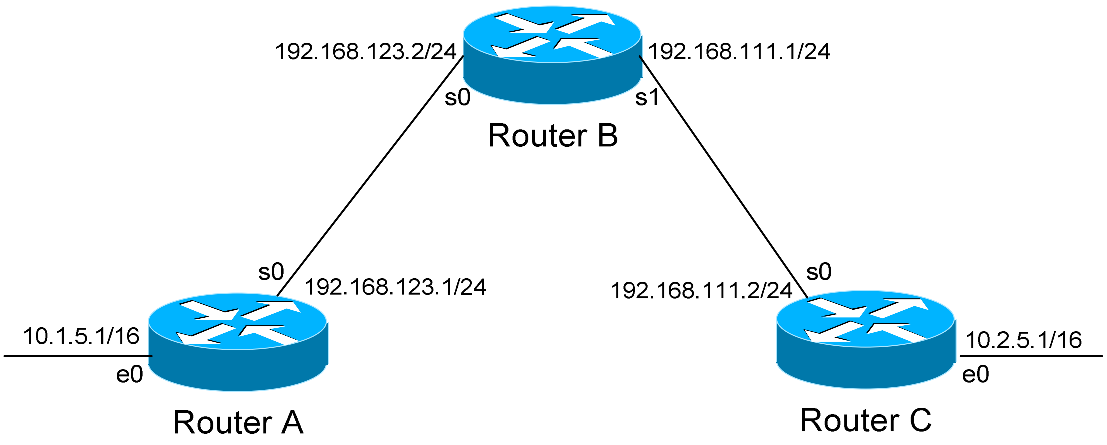
Router C will similarly send an entry for the 10.2.0.0 network to Router B. Router B's routing table will thus look like:

RouterB# *show ip route*

```
Gateway of last resort is not set

       10.0.0.0/16 is subnetted, 4 subnets
C      10.3.0.0 is directly connected, Serial0
C      10.4.0.0 is directly connected, Serial1
R      10.1.0.0 [120/1] via 10.3.5.1, 00:00:00, Serial0
R      10.2.0.0 [120/1] via 10.4.5.1, 00:00:00, Serial1
```

Limitations of RIPv1 (continued)

Consider the following, slightly altered, example:



We'll assume that RIPv1 is configured correctly on all routers. Notice that our networks are no longer contiguous. Both Router A and Router C contain *subnets* of the 10.0.0.0 major network (10.1.0.0 and 10.2.0.0 respectively).

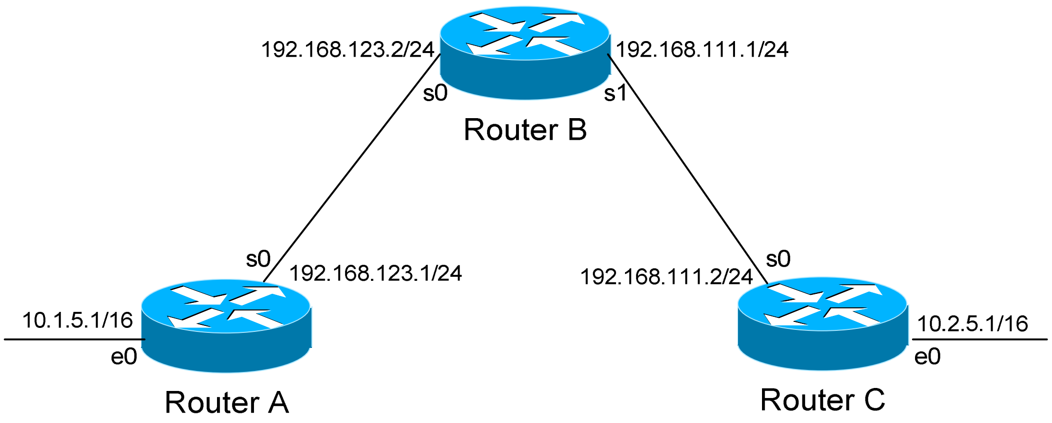
Separating these networks now are two Class C subnets (192.168.123.0 and 192.168.111.0).

Why is this a problem? Again, when Router A sends a RIPv1 update to Router B via Serial, it will not include the subnet mask for the 10.1.0.0 network. Instead, Router A will consider itself a **border** router, as the 10.1.0.0 and 192.168.123.0 networks *do not* belong to the same major network. Router A will **summarize** the 10.1.0.0/16 network to its classful boundary of 10.0.0.0/8.

Router B will accept this routing update, and realize that it does not have a directly connected interface in the 10.x.x.x scheme. Thus, it has no subnet mask to apply to this route. Because of this, Router B will install the summarized 10.0.0.0 route into its routing table.

Router C, similarly, will consider itself a border router between networks 10.2.0.0 and 192.168.111.0. Thus, Router C will *also* send a summarized 10.0.0.0 route to Router B.

Limitations of RIPv1 (continued)



Router B’s routing table will then look like:

RouterB# *show ip route*

```
Gateway of last resort is not set

C     192.168.123.0 is directly connected, Serial0
C     192.168.111.0 is directly connected, Serial1
R     10.0.0.0 [120/1] via 192.168.123.1, 00:00:00, Serial0
               [120/1] via 192.168.111.2, 00:00:00, Serial1
```

That’s right, Router B now has two *equal* metric routes to get to the summarized 10.0.0.0 network, one through Router A and the other through Router C. Router B will now *load balance* all traffic to *any* 10.x.x.x network between routers A and C. Suffice to say, this is not a good thing. ☺

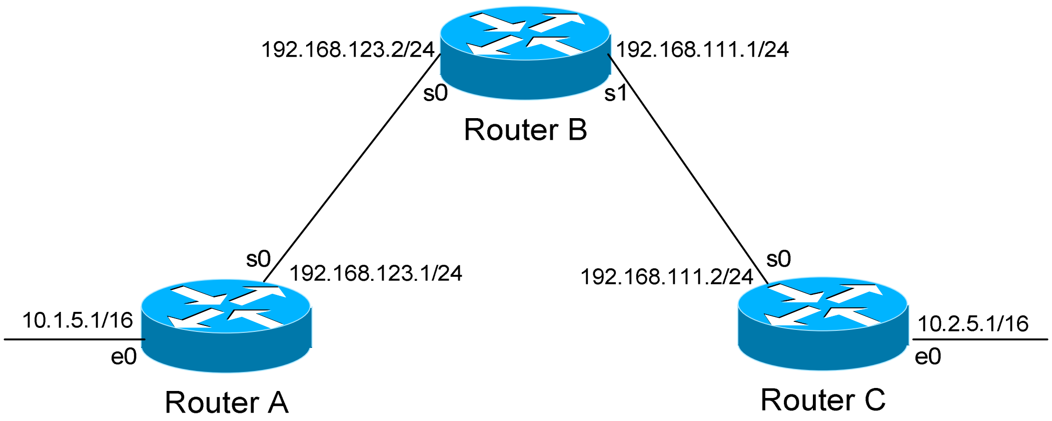
It gets better. Router B then tries to send routing updates to Router A and Router C, including the summary route of 10.0.0.0/8. Router A’s routing table looks like:

RouterA# *show ip route*

```
Gateway of last resort is not set

C     192.168.123.0 is directly connected, Serial0
       10.0.0.0/16 is subnetted, 1 subnet
C     10.1.0.0 is directly connected, Ethernet0
```

Router A will receive the summarized 10.0.0.0/8 route from Router B, and will reject it. This is because it already has the summary network of 10.0.0.0 in its routing table, and it’s directly connected. Router C will respond exactly the same, and the 10.1.0.0/16 and 10.2.0.0/16 networks will *never* be able to communicate.

RIPv2 Configuration



RIPv2 overcomes the limitations of RIPv1 by including the subnet mask in its routing updates. By default, Cisco routers will use RIPv1. To change to Version 2, you must type:

```
Router(config)#  router rip
Router(config-router)#  version 2
```

Thus, the configuration of Router A would be:

```
RouterA(config)#  router rip
RouterA(config-router)#  version 2
RouterA(config-router)#  network 10.0.0.0
RouterA(config-router)#  network 192.168.123.0
```

Despite the fact that RIPv2 is a classless routing protocol, we still specify networks at their classful boundaries, without a subnet mask.

However, when Router A sends a **RIPv2** update to Router B via Serial0, by default it will still **summarize** the 10.1.0.0/16 network to 10.0.0.0/8. Again, this is because the 10.1.0.0 and 192.168.123.0 networks *do not* belong to the same major network. Thus, RIPv2 acts like RIPv1 in this circumstance…

…unless you disable **auto summarization:**

```
RouterA(config)#  router rip
RouterA(config-router)#  version 2
RouterA(config-router)#  no auto-summary
```

The *no auto-summary* command will prevent Router A from summarizing the 10.1.0.0 network. Instead, Router A will send an update that includes both the subnetted network (10.1.0.0) and its subnet mask (255.255.0.0).

RIP Timers

RIP has four basic timers:

Update Timer (default **30 seconds**) – indicates how often the router will send out a routing table update.

Invalid Timer (default **180 seconds**) – indicates how long a route will remain in a routing table before being marked as invalid, if no new updates are heard about this route. The invalid timer will be reset if an update is received for that particular route *before* the timer expires.

A route marked as invalid is *not* immediately removed from the routing table. Instead, the route is marked (and advertised) with a metric of 16, indicating it is unreachable, and placed in a **hold-down** state.

Hold-down Timer (default **180 seconds**) – indicates how long RIP will "suppress" a route that it has placed in a **hold-down** state. RIP will not accept any new updates for routes in a hold-down state, until the hold-down timer expires.

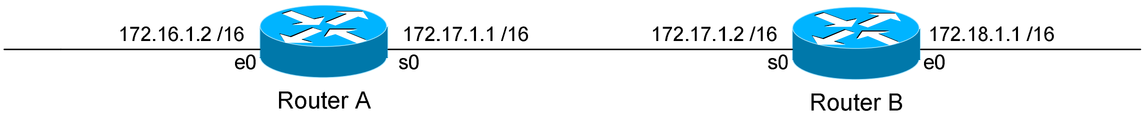
A route will enter a hold-down state for one of three reasons:

- The invalid timer has expired.
- An update has been received from another router, marking that route with a metric of 16 (or *unreachable*).
- An update has been received from another router, marking that route with a *higher* metric than what is currently in the routing table. This is to prevent loops.

Flush Timer (default **240 seconds**) **–** indicates how long a route can remain in a routing table before being flushed, if no new updates are heard about this route. The flush timer runs **concurrently with the invalid timer**, and thus will flush out a route 60 seconds after it has been marked invalid.

RIP timers must be identical on **all** routers on the RIP network, otherwise massive instability will occur.

RIP Timers Configuration and Example



Consider the above example. Router A receives a RIP update from Router B that includes network 172.18.0.0. Router A adds this network to its routing table:

RouterA# *show ip route*

```
Gateway of last resort is not set

C     172.16.0.0 is directly connected, Ethernet0
C     172.17.0.0 is directly connected, Serial0
R     172.18.0.0 [120/1] via 172.17.1.2, 00:00:00, Serial0
```

Immediately, Router A sets an **invalid** timer of 180 seconds and **flush** timer of 240 seconds to this route, which run concurrently. If no update for this route is heard for 180 seconds, several things will occur:

- The route is marked as invalid in the routing table.
- The route enters a **hold-down** state (triggering the hold-down timer).
- The route is advertised to all other routers as unreachable.

The hold-down timer runs for 180 seconds *after* the route is marked as invalid. The router will not accept any new updates for this route until this hold-down period expires.

If no update is heard *at all*, the route will be removed from the routing table once the flush timer expires, which is 60 seconds after the route is marked as invalid. Remember that the invalid and flush timers run concurrently.

To configure the RIP timers:

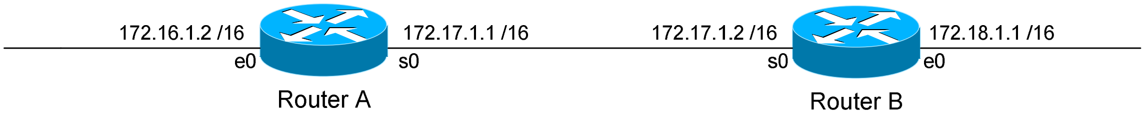
Router(config)# *router rip*
Router(config-router)# *timers basic 20 120 120 160*

The *timers basic* command allows us to change the update (*20*), invalid (*120*), hold-down (*120)*, and flush *(240)* timers. To return the timers back to their defaults:

Router(config-router)# *no timers basic*

RIP Loop Avoidance Mechanisms

RIP, as a Distance Vector routing protocol, is susceptible to loops.



Let's assume no loop avoidance mechanisms are configured on either router. If the 172.18.0.0 network fails, Router B will send out an update to Router A within 30 seconds (whenever its update timer expires) stating that route is unreachable (metric = 16).

But what if an update from Router A reaches Router B *before* this can happen? Router A believes it can reach the 172.18.0.0 network in one hop (through Router B). This will cause Router B to believe it can reach the failed 172.18.0.0 network in **two hops**, through Router A. Both routers will continue to increment the metric for the network until they reach a hop count of **16**, which is unreachable. This behavior is known as **counting to infinity**.

How can we prevent this from happening? There are several loop avoidance mechanisms:

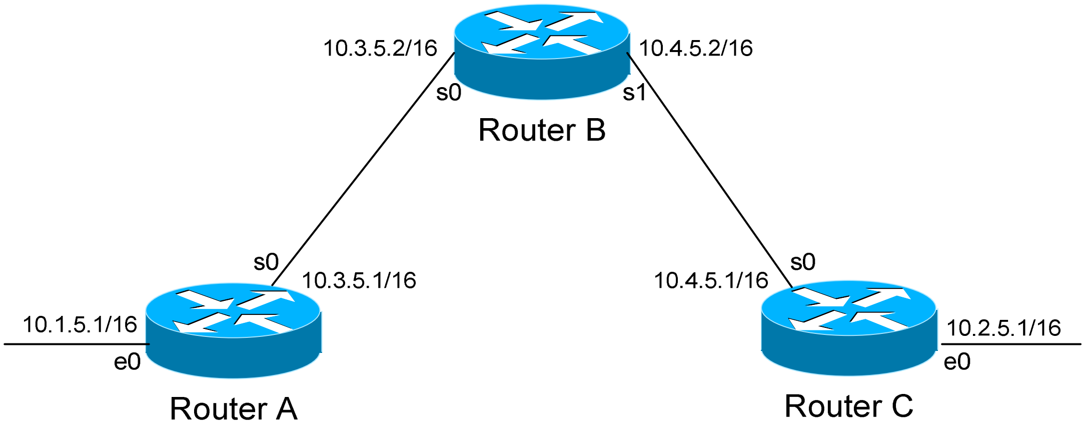
Split-Horizon – Prevents a routing update from being sent out the interface it was received on. In our above example, this would prevent Router A from sending an update for the 172.18.0.0 network *back* to Router B, as it originally learned the route from Router B. Split-horizon is **enabled** by default on Cisco Routers.

Route-Poisoning – Works in conjunction with split-horizon, by **triggering** an automatic update for the failed network, without waiting for the update timer to expire. This update is sent out all interfaces with an infinity metric for that network.

Hold-Down Timers – Prevents RIP from accepting any new updates for routes in a hold-down state, until the hold-down timer expires. If Router A sends an update to Router B with a *higher* metric than what is currently in Router B's routing table, that route will be placed in a hold-down state. (Router A's metric for the 172.18.0.0 network is 1; while Router B's metric is 0).

RIP Passive Interfaces

It is possible to control which router interfaces will participate in the RIP process.



Consider the following scenario. Router C does not want to participate in the RIP domain. However, it still wants to *listen* to updates being sent from Router B, just not *send* any updates back to Router B:

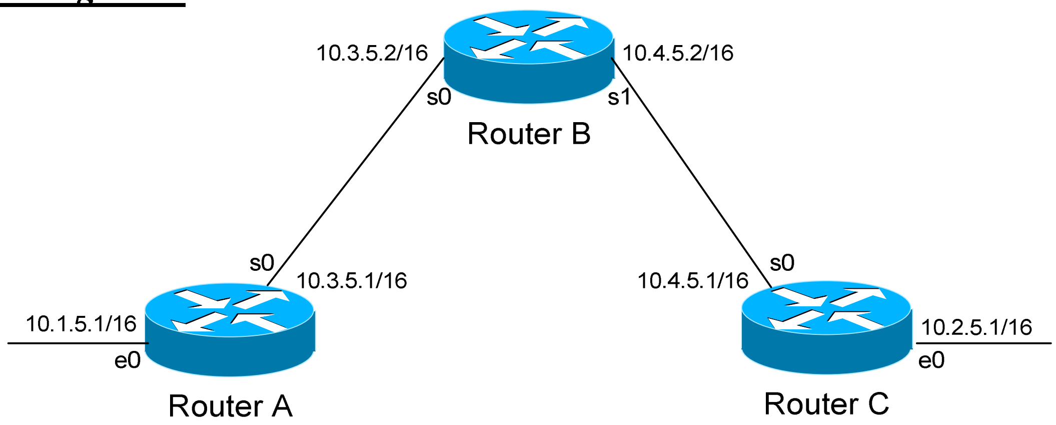
```
RouterC(config)#  router rip
RouterC(config-router)#  network 10.4.0.0
RouterC(config-router)#  network 10.2.0.0
RouterC(config-router)#  passive-interface s0
```

The *passive-interface* command will prevent updates from being **sent** out of the Serial0 interface, but Router C will still **receive** updates on this interface.

We can configure **all** interfaces to be passive using the *passive-interface default* command, and then individually use the *no passive-interface* command on the interfaces we **do** want updates to be sent out:

```
RouterC(config)#  router rip
RouterC(config-router)#  network 10.4.0.0
RouterC(config-router)#  network 10.2.0.0
RouterC(config-router)#  passive-interface default
RouterC(config-router)#  no passive-interface e0
```

RIP Neighbors



Recall that RIPv1 sends out its updates as **broadcasts**, whereas RIPv2 sends out its updates as **multicasts** to the 224.0.0.9 address. We can configure specific RIP *neighbor* commands, which will allow us to **unicast** routing updates to those neighbors.

On Router B:

```
RouterB(config)#  router rip
RouterB(config-router)#  network 10.3.0.0
RouterB(config-router)#  network 10.4.0.0
RouterB(config-router)#  neighbor 10.3.5.1
RouterB(config-router)#  neighbor 10.4.5.1
```

Router B will now unicast RIP updates to Router A and Router C.

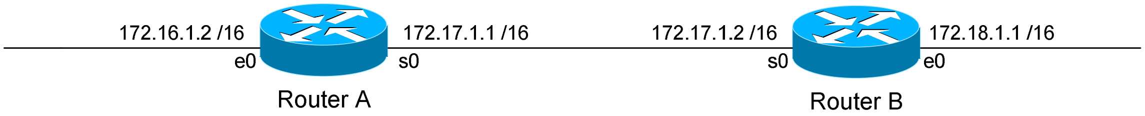
However, Router B will still broadcast (if RIPv1) or multicast (if RIPv2) its updates, in addition to sending unicast updates to its neighbors. In order to prevent broadcast/multicast updates, we must also use **passive interfaces**:

```
RouterB(config)#  router rip
RouterB(config-router)#  passive-interface s0
RouterB(config-router)#  passive-interface s1
RouterB(config-router)#  neighbor 10.3.5.1
RouterB(config-router)#  neighbor 10.4.5.1
```

The *passive-interface* commands prevent the updates from being broadcasted or multicasted. The *neighbor* commands still allow unicast updates to those specific neighbors.

* * *

RIPv2 Authentication



RIPv2 supports authentication to secure routing updates.

The first step is creating a shared authentication *key* that must be identical on both routers. This is accomplished in global configuration mode:

```
RouterA(config)# key chain MYCHAIN
RouterA(config-keychain)# key 1
RouterA(config-keychain-key)# key-string MYPASSWORD

RouterB(config)# key chain MYCHAIN
RouterB(config-keychain)# key 1
RouterB(config-keychain-key)# key-string MYPASSWORD
```

The first command creates a *key chain* called *MYCHAIN*. We must then associate a *key* to our keychain. Then we actually configure the shared key using the *key-string* command.

We then apply our key chain to the interface connecting to the other router:

```
RouterA(config)# interface s0
RouterA(config-if)# ip rip authentication key-chain MYCHAIN

RouterB(config)# interface s0
RouterB(config-if)# ip rip authentication key-chain MYCHAIN
```

If there was another router off of Router B's Ethernet port, we could create a *separate* key chain with a different key-string. Every router on the RIP domain does not need to use the same key chain, only interfaces directly connecting two (or more) routers.

The final step in configuring authentication is identifying which encryption to use. By default, the key is sent in clear text:

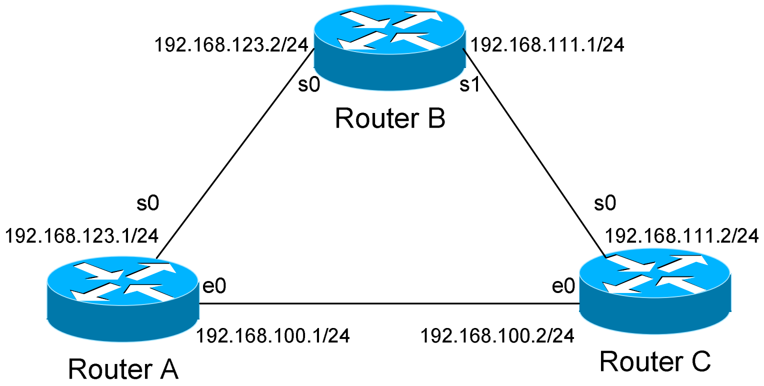
```
RouterA(config)# interface s0
RouterA(config-if)# ip rip authentication mode text
```

Or we can use MD5 encryption for additional security:

```
RouterA(config)# interface s0
RouterA(config-if)# ip rip authentication mode md5
```

Whether text or MD5 is used, it **must** be the same on both routers.

Altering RIP's Metric



Consider the above example. Router B has two paths to get to the 192.168.100.0 network, via Router A and Router C. Because the metric is equal (1 hop), Router B will load balance between these two paths.

What if we wanted Router B to only go through Router A, and use Router C only as a backup? To accomplish this, we can adjust RIP's metric to make one route more preferred than the other.

The first step is creating an access-list on Router B that defines which route we wish to alter:

```
RouterB(config)# ip access-list standard MYLIST
RouterB(config-std-nacl)# permit 192.168.100.0 0.0.0.255
```

Next, we tell RIP how much to **offset** this route if received by Router C:

```
RouterB(config)# router rip
RouterB(config-router)# offset-list MYLIST in 4 s1
```

We specify an *offset-list* pointing to our access list named *MYLIST*. We will increase the routing metric by *4* for that route coming *in*bound to interface *Serial 1*.

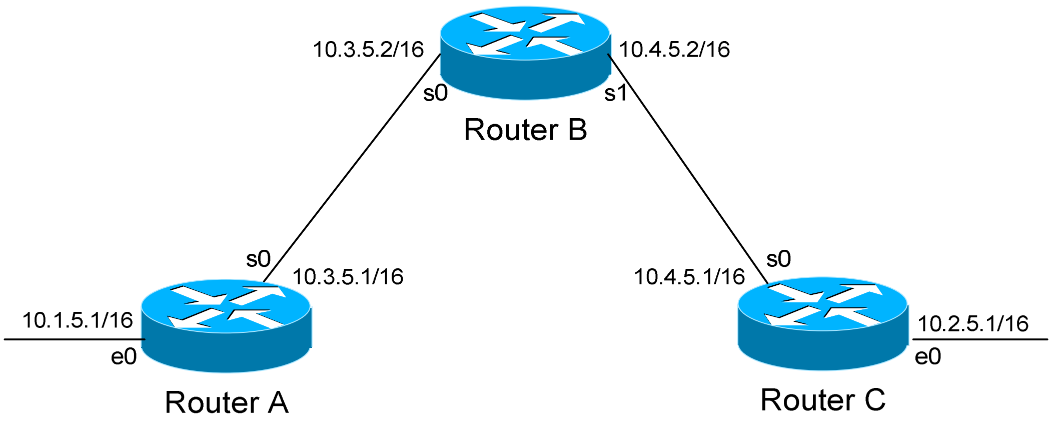
Thus, when Router C sends an update to Router C for the 192.168.100.0 network, Router B will increase its metric of 1 hop to 5 hops, thus making Router A's route preferred.

We could have also configured Router C to **advertise** that route with a higher metric (notice the *out* in the *offset-list* command):

```
RouterC(config)# ip access-list standard MYLIST
RouterC(config-std-nacl)# permit 192.168.100.0 0.0.0.255
RouterC(config)# router rip
RouterC(config-router)# offset-list MYLIST out 4 s0
```

* * *

Interoperating between RIPv1 and RIPv2



Recall that, with some configuration, RIPv1 and RIPv2 can interoperate. By default:

- RIPv1 routers will sent only Version 1 packets
- RIPv1 routers will receive both Version 1 and 2 updates
- RIPv2 routers will both send and receive only Version 2 updates

If Router A is running RIP v1, and Router B is running RIP v2, some additional configuration is necessary.

Either we must configure Router A to send Version 2 updates:

```
RouterA(config)# interface s0
RouterA(config-if)# ip rip send version 2
```

Or configure Router B to accept Version 1 updates.

```
RouterB(config)# interface s0
RouterB(config-if)# ip rip receive version 1
```

Notice that this is configured on an interface. Essentially, we're configuring the version of RIP on a per-interface basis.

We can also have an interface send or receive both versions simultaneously:

```
RouterB(config)# interface s0
RouterB(config-if)# ip rip receive version 1 2
```

We can further for RIPv2 to send broadcast updates, instead of multicasts:

```
RouterB(config)# interface s0
RouterB(config)# ip rip v2-broadcast
```

Triggering RIP Updates

On point-to-point interfaces, we can actually force RIP to only send routing updates if there is a change:

RouterB(config)# *interface s0.150 point-to-point*
RouterB(config-if)# *ip rip triggered*

Again, this is only applicable to **point-to-point** links. We cannot configure RIP triggered updates on an Ethernet network.

Troubleshooting RIP

Various troubleshooting commands exist for RIP.

To view the IP routing table:

Router# *show ip route*

```
<eliminated irrelevant header>

Gateway of last resort is not set

C     172.16.0.0 is directly connected, Ethernet0
C     172.17.0.0 is directly connected, Serial0
R     172.18.0.0 [120/1] via 172.17.1.2, 00:00:15, Serial0
R     192.168.123.0 [120/1] via 172.16.1.1, 00:00:00, Ethernet0
```

To view a specific route within the IP routing table:

Router# *show ip route 172.18.0.0*

```
Routing entry for 172.18.0.0/16
      Known via “rip”, distance 120, metric 1
      Last update from 172.17.1.2 on Serial 0, 00:00:15 ago
```

To debug RIP in real time:

Router# *debug ip rip*

Troubleshooting RIP (continued)

To view information specific to the RIP protocol:

Router# *show ip protocols*

```
Routing Protocol is "rip"
  Sending updates every 30 seconds, next due in 20 seconds
  Invalid after 180 seconds, hold down 180, flushed after 240
  Outgoing update filter list for all interfaces is not set
  Incoming update filter list for all interfaces is not set
  Incoming routes will have 4 added to metric if on list 1
  Redistributing: connected, static, rip
  Default version control: send version 1, receive any version
    Interface             Send  Recv  Triggered RIP  Key-chain
    Ethernet0             1     1 2
    Serial0               1 2   1 2
  Automatic network summarization is in effect
  Maximum path: 4
  Routing for Networks:
    172.16.0.0
    172.17.0.0
  Routing Information Sources:
    Gateway         Distance      Last Update
    172.17.1.2           120      00:00:17
  Distance: (default is 120)
```

This command provides us with information on RIP timers, on the RIP versions configured on each interface, and the specific networks RIP is advertising.

To view *all* routes in the RIP database, and not just the entries added to the routing table:

Router# *show ip rip database*

```
7.0.0.0/8    auto-summary
7.0.0.0/8
    [5] via 172.16.1.1, 00:00:06, Ethernet0
172.16.0.0/16    directly connected, Ethernet0
172.17.0.0/16    directly connected, Serial0
```

Section 10
- Interior Gateway Routing Protocol -

IGRP (Interior Gateway Routing Protocol)

IGRP is a Cisco-proprietary Distance-Vector protocol, designed to be more scalable than RIP, its standardized counterpart.

IGRP adheres to the following Distance-Vector characteristics:

- IGRP sends out periodic routing updates (every **90 seconds**).
- IGRP sends out the full routing table every periodic update.
- IGRP uses a form of distance as its metric (in this case, a composite of **bandwidth** and **delay**).
- IGRP uses the Bellman-Ford Distance Vector algorithm to determine the best “path” to a particular destination.

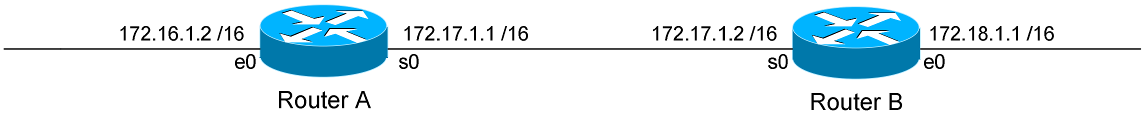
Other characteristics of IGRP include:

- IGRP supports **only IP routing**.
- IGRP utilizes **IP protocol 9**.
- IGRP routes have an administrative distance of **100**.
- IGRP, by default, supports a maximum of **100 hops.** This value can be adjusted to a maximum of **255 hops**.
- IGRP is a **classful** routing protocol.

IGRP uses **Bandwidth** and **Delay of the Line**, by default, to calculate its distance metric. **Reliability**, **Load**, and **MTU** are optional attributes that can be used to calculate the distance metric.

IGRP requires that you include an **Autonomous System (AS)** number in its configuration. Only routers in the same Autonomous system will send updates between each other.

Configuring IGRP



Routing protocol configuration occurs in Global Configuration mode. On Router A, to configure IGRP, we would type:

Router(config)# *router igrp 10*
Router(config-router)# *network 172.16.0.0*
Router(config-router)# *network 172.17.0.0*

The first command, *router igrp 10*, enables the IGRP process. The "*10*" indicates the Autonomous System number that we are using. Only other IGRP routers in Autonomous System *10* will share updates with this router.

The *network* statements tell IGRP which networks you wish to advertise to other RIP routers. We simply list the networks that are directly connected to our router. Notice that we specify the networks at their classful boundaries, and we do not specify a subnet mask.

To configure Router B:

Router(config)# *router igrp 10*
Router(config-router)# *network 172.17.0.0*
Router(config-router)# *network 172.18.0.0*

The routing table on Router A will look like:

RouterA# *show ip route*

```
Gateway of last resort is not set

C     172.16.0.0 is directly connected, Ethernet0
C     172.17.0.0 is directly connected, Serial0
I     172.18.0.0 [120/1] via 172.17.1.2, 00:00:00, Serial0
```

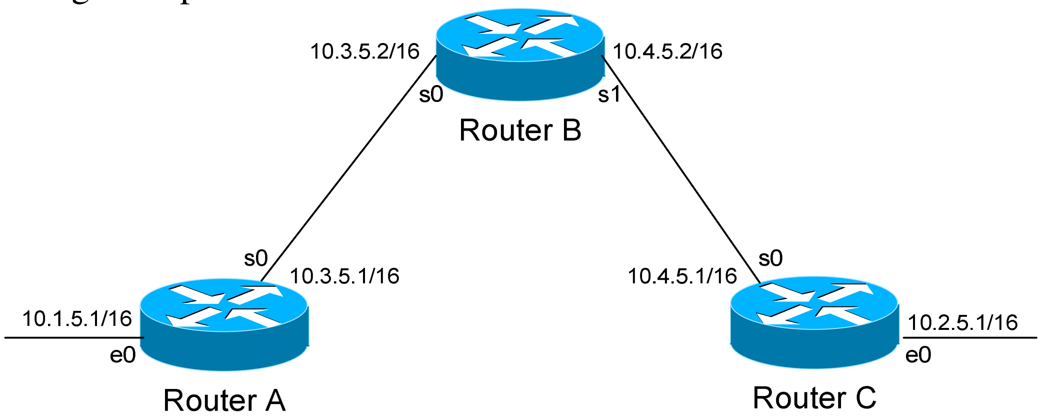
The routing table on Router B will look like:

RouterB# *show ip route*

```
Gateway of last resort is not set

C     172.17.0.0 is directly connected, Serial0
C     172.18.0.0 is directly connected, Ethernet0
I     172.16.0.0 [120/1] via 172.17.1.1, 00:00:00, Serial0
```

Limitations of IGRP

The example on the previous page works fine with IGRP, because the networks are contiguous and the subnet masks are consistent. Consider the following example:



This particular scenario will *still* work when using IGRP, despite the fact that we've subnetted the major 10.0.0.0 network. Notice that the subnets are contiguous (that is, they belong to the same major network), and use the same subnet mask.

When Router A sends an IGRP update to Router B via Serial0, it will not include the subnet mask for the 10.1.0.0 network. However, because the 10.3.0.0 network is in the same major network as the 10.1.0.0 network, it will **not summarize** the address. The route entry in the update will simply state "10.1.0.0".

Router B will accept this routing update, and realize that the interface receiving the update (Serial0) belongs to the same major network as the route entry of 10.1.0.0. It will then apply the subnet mask of its Serial0 interface to this route entry.

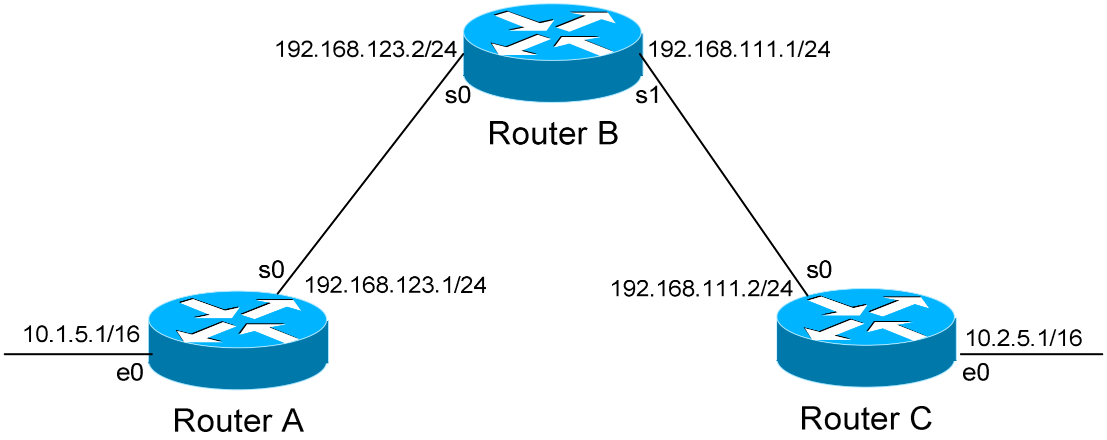
Router C will similarly send an entry for the 10.2.0.0 network to Router B. Router B's routing table will thus look like:

RouterB# *show ip route*

```
Gateway of last resort is not set

       10.0.0.0/16 is subnetted, 4 subnets
C      10.3.0.0 is directly connected, Serial0
C      10.4.0.0 is directly connected, Serial1
I      10.1.0.0 [120/1] via 10.3.5.1, 00:00:00, Serial0
I      10.2.0.0 [120/1] via 10.4.5.1, 00:00:00, Serial1
```

Limitations of IGRP (continued)

Consider the following, slightly altered, example:



We'll assume that IGRP is configured correctly on all routers. Notice that our networks are no longer contiguous. Both Router A and Router C contain *subnets* of the 10.0.0.0 major network (10.1.0.0 and 10.2.0.0 respectively).

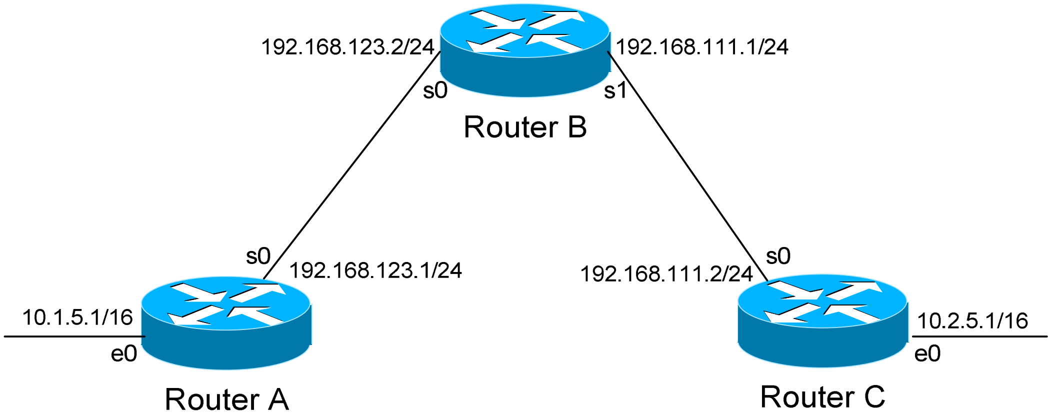
Separating these networks now are two Class C subnets (192.168.123.0 and 192.168.111.0).

Why is this a problem? Again, when Router A sends an IGRP update to Router B via Serial, it will not include the subnet mask for the 10.1.0.0 network. Instead, Router A will consider itself a **border** router, as the 10.1.0.0 and 192.168.123.0 networks *do not* belong to the same major network. Router A will **summarize** the 10.1.0.0/16 network to its classful boundary of 10.0.0.0/8.

Router B will accept this routing update, and realize that it does not have a directly connected interface in the 10.x.x.x scheme. Thus, it has no subnet mask to apply to this route. Because of this, Router B will install the summarized 10.0.0.0 route into its routing table.

Router C, similarly, will consider itself a border router between networks 10.2.0.0 and 192.168.111.0. Thus, Router C will *also* send a summarized 10.0.0.0 route to Router B.

Limitations of IGRP (continued)



Router B's routing table will then look like:

RouterB# *show ip route*

```
Gateway of last resort is not set

C     192.168.123.0 is directly connected, Serial0
C     192.168.111.0 is directly connected, Serial1
I     10.0.0.0 [120/1] via 192.168.123.1, 00:00:00, Serial0
               [120/1] via 192.168.111.2, 00:00:00, Serial1
```

That's right, Router B now has two *equal* metric routes to get to the summarized 10.0.0.0 network, one through Router A and the other through Router C. Router B will now *load balance* all traffic to *any* 10.x.x.x network between routers A and C. Suffice to say, this is not a good thing. ☺

It gets better. Router B then tries to send routing updates to Router A and Router C, including the summary route of 10.0.0.0/8. Router A's routing table looks like:

RouterA# *show ip route*

```
Gateway of last resort is not set

C     192.168.123.0 is directly connected, Serial0
       10.0.0.0/16 is subnetted, 1 subnet
C     10.1.0.0 is directly connected, Ethernet0
```

Router A will receive the summarized 10.0.0.0/8 route from Router B, and will reject it. This is because it already has the summary network of 10.0.0.0 in its routing table, and it's directly connected. Router C will respond exactly the same, and the 10.1.0.0/16 and 10.2.0.0/16 networks will *never* be able to communicate.

IGRP Timers

IGRP has four basic timers:

Update Timer (default **90 seconds**) – indicates how often the router will send out a routing table update.

Invalid Timer (default **270 seconds**) – indicates how long a route will remain in a routing table before being marked as invalid, if no new updates are heard about this route. The invalid timer will be reset if an update is received for that particular route *before* the timer expires.

A route marked as invalid is *not* immediately removed from the routing table. Instead, the route is marked (and advertised) with a metric of 101 (remember, 100 maximum hops is default), indicating it is unreachable, and placed in a **hold-down** state.

Hold-down Timer (default **280 seconds**) – indicates how long IGRP will "suppress" a route that it has placed in a **hold-down** state. IGRP will not accept any new updates for routes in a hold-down state, until the hold-down timer expires.

A route will enter a hold-down state for one of three reasons:

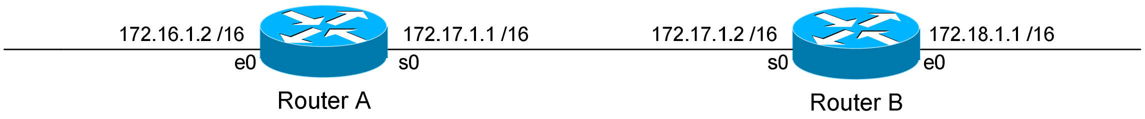
- The invalid timer has expired.
- An update has been received from another router, marking that route with a metric of 101 (unreachable).
- An update has been received from another router, marking that route with a *higher* metric than what is currently in the routing table (this is to prevent loops).

Flush Timer (default **630 seconds**) **–** indicates how long a route can remain in a routing table before being flushed, if no new updates are heard about this route. The flush timer runs **concurrently with the invalid timer**, and thus will flush out a route 360 seconds after it has been marked invalid.

IGRP timers must be identical on **all** routers on the IGRP network, otherwise massive instability will occur.

IGRP Loop Avoidance Mechanisms

IGRP, as a Distance Vector routing protocol, is susceptible to loops.



Let's assume no loop avoidance mechanisms are configured on either router. If the 172.18.0.0 network fails, Router B will send out an update to Router A within 30 seconds (whenever its update timer expires) stating that route is unreachable.

But what if an update from Router A reaches Router B *before* this can happen? Router A believes it can reach the 172.18.0.0 network in one hop (through Router B). This will cause Router B to believe it can reach the failed 172.18.0.0 network in **two hops**, through Router A. Both routers will continue to increment the metric for the network until they reach an infinity hop count (by default, 101). This behavior is known as **counting to infinity**.

How can we prevent this from happening? There are several loop avoidance mechanisms:

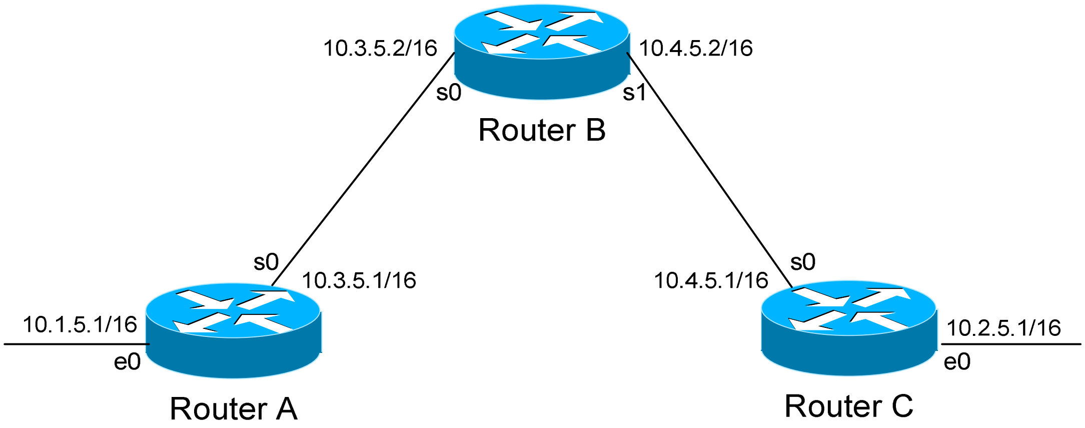
Split-Horizon – Prevents a routing update from being sent out the interface it was received on. In our above example, this would prevent Router A from sending an update for the 172.18.0.0 network *back* to Router B, as it originally learned the route from Router B. Split-horizon is **enabled** by default on Cisco Routers.

Route-Poisoning – Works in conjunction with split-horizon, by **triggering** an automatic update for the failed network, without waiting for the update timer to expire. This update is sent out all interfaces with an infinity metric for that network.

Hold-Down Timers – Prevents IGRP from accepting any new updates for routes in a hold-down state, until the hold-down timer expires. If Router A sends an update to Router B with a *higher* metric than what is currently in Router B's routing table, that route will be placed in a hold-down state.

IGRP Passive Interfaces

It is possible to control which router interfaces will participate in the IGRP process.



Consider the following scenario. Router C does not want to participate in the IGRP domain. However, it still wants to *listen* to updates being sent from Router B, just not *send* any updates back to Router B:

```
RouterC(config)# router igrp 10
RouterC(config-router)# network 10.4.0.0
RouterC(config-router)# network 10.2.0.0
RouterC(config-router)# passive-interface s0
```

The *passive-interface* command will prevent updates from being **sent** out of the Serial0 interface, but Router C will still **receive** updates on this interface.

We can configure **all** interfaces to be passive using the *passive-interface default* command, and then individually use the *no passive-interface* command on the interfaces we **do** want updates to be sent out:

```
RouterC(config)# router igrp 10
RouterC(config-router)# network 10.4.0.0
RouterC(config-router)# network 10.2.0.0
RouterC(config-router)# passive-interface default
RouterC(config-router)# no passive-interface e0
```

* * *

Advanced IGRP Configuration

To change the maximum hop-count to 255 for IGRP:

```
Router(config)#  router igrp 10
Router(config-router)#  metric maximum-hops 255
```

Section 11
- Enhanced Interior Gateway Routing Protocol -

EIGRP (Enhanced Interior Gateway Routing Protocol)

EIGRP is a Cisco-proprietary Hybrid routing protocol, incorporating features of both Distance-Vector and Link-State routing protocols.

EIGRP adheres to the following Hybrid characteristics:

- EIGRP uses **Diffusing Update Algorithm (DUAL)** to determine the best path among all "feasible" paths. DUAL also helps ensure a loop-free routing environment.
- EIGRP will form **neighbor** relationships with adjacent routers in the same **Autonomous System (AS)**.
- EIGRP traffic is either sent as unicasts, or as multicasts on address **224.0.0.10**, depending on the EIGRP packet type.
- Reliable Transport Protocol (RTP) is used to ensure delivery of most EIGRP packets.
- EIGRP routers **do not** send periodic, full-table routing updates. Updates are sent when a change occurs, and include *only* the change.
- EIGRP is a classless protocol, and thus supports VLSMs.

Other characteristics of EIGRP include:

- EIGRP supports IP, IPX, and Appletalk routing.
- EIGRP applies an Administrative Distance of **90** for routes originating *within* the local Autonomous System.
- EIGRP applies an Administrative Distance of **170** for external routes coming from *outside* the local Autonomous System
- EIGRP uses **Bandwidth** and **Delay of the Line,** by default, to calculate its distance metric. It also supports three other parameters to calculate its metric: **Reliability**, **Load**, and **MTU.**
- EIGRP has a maximum hop-count of **224**, though the default maximum hop-count is set to **100**.

EIGRP, much like OSPF, builds three separate tables:

- **Neighbor table** – list of all neighboring routers. Neighbors must belong to the same **Autonomous System**
- **Topology table** – list of *all* routes in the Autonomous System
- **Routing table** – contains the *best* route for each known network

EIGRP Neighbors

EIGRP forms neighbor relationships, called **adjacencies,** with other routers in the same AS by exchanging **Hello** packets. Only after an adjacency is formed can routers share routing information. Hello packets are sent as multicasts to address 224.0.0.10.

By default, on LAN and high-speed WAN interfaces, EIGRP Hellos are sent every **5 seconds**. On slower WAN links (T1 speed or slower), EIGRP Hellos are sent every **60 seconds** by default.

The EIGRP Hello timer can be adjusted on a *per interface* basis:

```
Router(config-if)#  ip hello-interval eigrp 10 7
```

The above command allows us to change the *hello* timer to *7* seconds for Autonomous System *10.*

In addition to the Hello timer, EIGRP neighbors are stamped with a **Hold timer.** The Hold timer indicates how long a router should wait before marking a neighbor inactive, if it stops receiving hello packets from that neighbor.

By default, the Hold timer is **three times** the Hello timer. Thus, on high-speed links the timer is set to **15 seconds**, and on slower WAN links the timer is set to **180 seconds**.

The Hold timer can also be adjusted on a *per interface* basis:

```
Router(config-if)#  ip hold-interval eigrp 10 21
```

The above command allows us to change the *hold* timer to *21* seconds for Autonomous System *10.*

Changing the Hello timer **does not** automatically change the Hold timer. Additionally, Hello and Hold timers ***do not*** need to match between routers for an EIGRP neighbor relationship to form.

(Reference: http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a0080093f07.shtml#eigrp_work)

EIGRP Neighbors (continued)

A **neighbor table** is constructed from the EIGRP Hello packets, which includes the following information:

- The IP address of the neighboring router.
- The local interface that received the neighbor's Hello packet.
- The Hold timer.
- A sequence number indicating the order neighbors were learned.

Adjacencies will not form unless the **primary IP addresses** on connecting interfaces are on the same subnet. Neighbors *cannot* be formed on secondary addresses.

If connecting interfaces are on *different* subnets, an EIGRP router will log the following error to console when a multicast Hello is received:

```
00:11:22: IP-EIGRP: Neighbor 172.16.1.1 not on common
subnet for Serial0
```

Always ensure that primary IP addresses belong to the same subnet between EIGRP neighbors.

To log all neighbor messages and errors to console, use the following two commands:

Router(config)# *router eigrp 10*
Router(config-router)# *eigrp log-neighbor-changes*
Router(config-router)# *eigrp log-neighbor-warnings*

(Reference: http://www.cisco.com/en/US/tech/tk365/technologies_configuration_example09186a0080093f09.shtml)

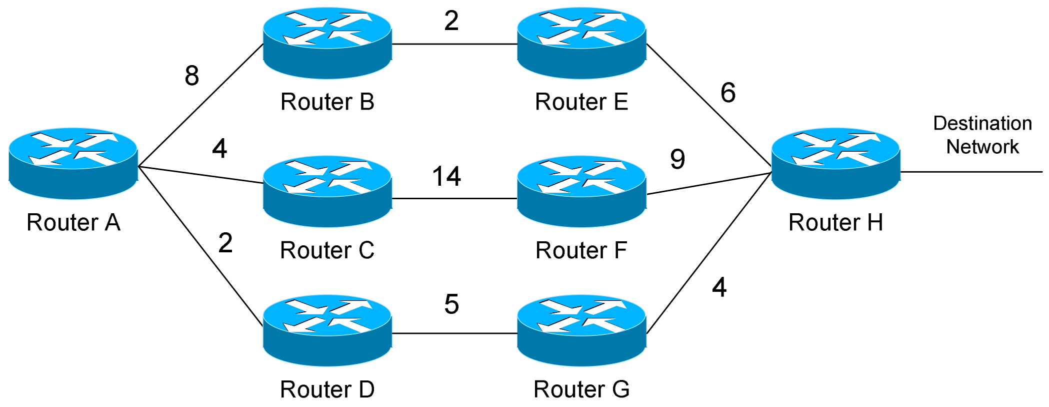
The EIGRP Topology Table

Once EIGRP neighbors form adjacencies, they will begin to share routing information. Each router's update contains a list of all routes known by that router, and the respective metrics for those routes.

All such routes are added to an EIGRP router's topology table. The route with the lowest metric to each network will become the **Feasible Distance (FD)**. The Feasible Distance for each network will be installed into the routing table.

The Feasible Distance is derived from the **Advertised Distance** of the router *sending* the update, and the local router's metric to the advertising router.

Confused? Consider the following example:



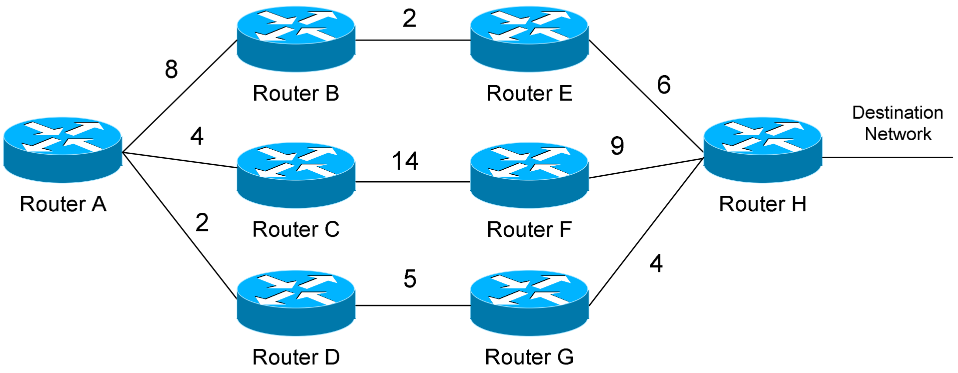
Router A has three separate paths to the Destination Network, either through Router B, C, or D. If we add up the metrics to form a "distance" (the metrics are greatly simplified in this example), we can determine the following:

- Router B's Feasible Distance to the Destination Network is 8.
- Router C's Feasible Distance to the Destination Network is 23.
- Router D's Feasible Distance to the Destination Network is 9.

Router B sends an update to Router A, it will provide an **Advertised Distance** of 8 to the Destination Network. Router C will provide an AD of 23, and D will provide an AD of 9.

Router A calculates the total distance to the Destination network by adding the AD of the advertising router, with its own distance to *reach* that advertising router. For example, Router A's metric to Router B is 8; thus, the *total* distance will be 16 to reach the Destination Network through Router B.

The EIGRP Topology Table (continued)



Remember, however, that Router A's Feasible Distance must be the route with the lowest metric. If we add the Advertised Distance with the local metric between each router, we would see that:

- The route through Router B has a distance of **16** to the destination
- The route through Router C has a distance of **27** to the destination
- The route through Router D has a distance of **11** to the destination

Thus, the route through Router D (metric of 11) would become the **Feasible Distance** for Router A, and is added to the routing table as the *best* route. This route is identified as the **Successor**.

To allow convergence to occur quickly if a link fails, EIGRP includes backup routes in the topology table called **Feasible Successors (FS).** A route will only become a Successor if its Advertised Distance is *less* than the current Feasible Distance**.** This is known as a **Feasible Condition** (**FC**).

For example, we determined that Router A's Feasible Distance to the destination is 11, through Router D. Router C's Advertised Distance is 23, and thus *would not* become a Feasible Successor, as it has a higher metric than Router A's current Feasible Distance. Routes that are not Feasible Successors become route **Possibilities.**

Router B's Advertised Distance is 8, which is less than Router A's current Feasible Distance. Thus, the route through Router B to the Destination Network *would* become a Feasible Successor.

Feasible Successors provide EIGRP with redundancy, without forcing routers to re-converge (thus stopping the flow of traffic) when a topology change occurs. If no Feasible Successor exists and a link fails, a route will enter an **Active** (converging) state until an alternate route is found.

EIGRP Packet Types

EIGRP employs five packet types:

- **Hello packets -** *multicast*
- **Update packets –** *unicast* or *multicast*
- **Query packets** – *multicast*
- **Reply packets** – *unicast*
- **Acknowledgement packets -** *unicast*

Hello packets are used to form neighbor relationships, and were explained in detail previously. Hello packets are always multicast to address 224.0.0.10.

Update packets are sent between neighbors to build the topology and routing tables. Updates sent to *new* neighbors are sent as unicasts. However, if a route's metric is changed, the update is sent out as a multicast to address 224.0.0.10.

Query packets are sent by a router when a Successor route fails, and there are no Feasible Successors in the topology table. The router places the route in an **Active state**, and queries its neighbors for an alternative route. Query packets are sent as a multicast to address 224.0.0.10.

Reply packets are sent in response to Query packets, assuming the responding router has an alternative route (feasible successor). Reply packets are sent as a unicast to the querying router.

Recall that EIGRP utilizes the **Reliable Transport Protocol** (**RTP**) to ensure reliable delivery of most EIGRP packets. Delivery is guaranteed by having packets *acknowledged* using…..**Acknowledgment packets!**

Acknowledgment packets (also known as **ACK's**) are simply Hello packets with no data, other than an acknowledgment number. ACK's are always sent as unicasts. The following packet types employ RTP to ensure reliable delivery via ACK's:

- Update Packets
- Query Packets
- Reply Packets

Hello and Acknowledgments (ha!) packets do not utilize RTP, and thus do not require acknowledgement.

EIGRP Route States

An EIGRP route can exist in one of two states, in the topology table:

- **Active state**
- **Passive State**

A **Passive state** indicates that a route is reachable, and that EIGRP is fully converged. A stable EIGRP network will have all routes in a Passive state.

A route is placed in an **Active state** when the Successor and any Feasible Successors fail, forcing the EIGRP to send out Query packets and re-converge. Multiple routes in an Active state indicate an unstable EIGRP network. If a Feasible Successor exists, a route should *never* enter an Active state.

Routes will become **Stuck-in-Active (SIA)** when a router sends out a Query packet, but does not receive a Reply packet within **three minutes**. In other words, a route will become SIA if EIGRP fails to re-converge. The local router will clear the neighbor adjacency with any router(s) that has failed to Reply, and will place all routes from that neighbor(s) in an Active state.

To view the current state of routes in the EIGRP topology table:

Router# *show ip eigrp topology*

```
IP-EIGRP Topology Table for AS(10)/ID(172.19.1.1)

Codes: P - Passive, A - Active, U - Update, Q - Query, R - Reply,
            r - reply Status, s - sia Status

P 10.3.0.0/16, 1 successors, FD is 2297856
         via 172.16.1.2 (2297856/128256), Serial0
P 172.19.0.0/16, 1 successors, FD is 281600
         via Connected, Serial 1
```

To view only *active* routes in the topology table:

Router# *show ip eigrp topology active*

```
IP-EIGRP Topology Table for AS(10)/ID(172.19.1.1)

Codes: P - Passive, A - Active, U - Update, Q - Query, R - Reply,
            r - Reply status

A 172.19.0.0/16, 1 successors, FD is 23456056 1 replies,
            active 0:00:38, query-origin: Multiple Origins
```

(Reference: http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a008010f016.shtml)

EIGRP Metrics

EIGRP can utilize 5 separate metrics to determine the best route to a destination:

- **Bandwidth** (K1) – Slowest link in the route path, measured in kilobits
- **Load** (K2) – Cumulative load of all outgoing interfaces in the path, given as a fraction of 255
- **Delay of the Line** (K3) – Cumulative delay of all outgoing interfaces in the path in tens of microseconds
- **Reliability** (K4) – Average reliability of all outgoing interfaces in the path, given as a fraction of 255
- **MTU** (K5) – The smallest Maximum Transmission Unit in the path. The MTU value is actually *never* used to calculate the metric

By default, only **Bandwidth** and **Delay of the Line** are used. This is identical to IGRP, except that EIGRP provides a more granular metric by multiplying the bandwidth and delay by 256. Bandwidth and delay are determined by the interfaces that lead towards the destination network.

By default, the full formula for determining the EIGRP metric is:

$$[10000000/\text{bandwidth} + \text{delay}] * 256$$

The bandwidth value represents the link with the *lowest* bandwidth in the path, in kilobits. The delay is the total delay of all outgoing interfaces in the path.

As indicated above, each metric is symbolized with a “K” and then a number. When configuring EIGRP metrics, we actually identify which metrics we want EIGRP to consider. Again, by default, only Bandwidth and Delay are considered. Thus, using on/off logic:

$$K1 = 1, K2 = 0, K3 = 1, K4 = 0, K5 = 0$$

If all metrics were set to “on,” the full formula for determining the EIGRP metric would be:

$$\begin{gathered}[K1 * \text{bandwidth} * 256 + (K2 * \text{bandwidth}) / (256 - \text{load}) \\ + K3 * \text{delay} * 256] * [K5 / (\text{reliability} + K4)]\end{gathered}$$

Remember, the “K” value is either set to on (“1”) or off (“0”).

Configuring EIGRP Metrics

EIGRP allows us to identify which metrics the protocol should consider, using the following commands:

```
Router(config)#  router eigrp 10
Router(config-router)#  metric weights 0 1 1 1 0 0
```

The first command enables the *EIGRP* process for Autonomous System *10*. The second actually identifies which EIGRP metrics to use. The first number (*0*) is for Type of Service, and should always be zero. The next numbers, in order, are K1 (*1*), K2 (*1*), K3 (*1*), K4 *(0)*, and K5 (*0)*. Thus, we are instructing EIGRP to use bandwidth, load, and delay to calculate the total metric, but not reliability or MTU.

Our formula would thus be:

[K1 * bandwidth * 256 + (K2 * bandwidth) / (256 - load) + K3 * delay * 256]

The actual values of our metrics (such as bandwidth or delay) must be configured indirectly. To adjust the bandwidth (in Kbps) of an interface:

```
Router(config)#  int s0/0
Router(config-if)#  bandwidth 64
Router(config-if)#  ip bandwidth-percent eigrp 10 30
```

However, this command does not actually dictate the *physical* speed of the interface. It merely controls how EIGRP *considers* this interface. Best practice is to set the bandwidth to the actual physical speed of the interface. By default, a *serial* interface will have a bandwidth of **1.544 Mbps** (*1544*)**.**

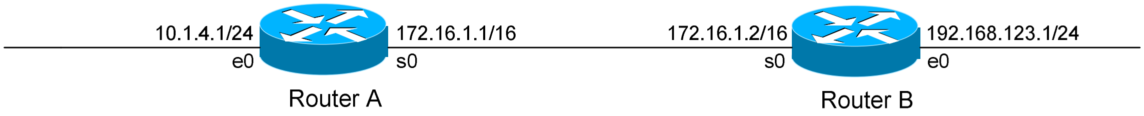
The *ip bandwidth-percent eigrp* command limits the percentage of bandwidth EIGRP can use on an interface. The percentage is based on the configured *bandwidth* value. By default, EIGRP will use **up to 50%** of the bandwidth of an interface. The above command adjusts this to *30%* for Autonomous System *10*. **Percentages over 100%** can be used.

If adjustments to the EIGRP metric need to be made, the delay metric (in tens of microseconds) on an interface should be used:

```
Router(config)#  int s0/0
Router(config-if)#  delay 10000
```

Metric settings must be **identical** on the connecting interfaces of two routers; otherwise they will not form a neighbor relationship.

Configuring Basic EIGRP



Routing protocol configuration occurs in Global Configuration mode. On Router A, to configure EIGRP, we would type:

```
RouterA(config)# router eigrp 10
RouterA(config-router)# network 172.16.0.0
RouterA(config-router)# network 10.0.0.0
```

The first command, *router eigrp 10*, enables the EIGRP process. The *"10"* indicates the Autonomous System number that we are using. The Autonomous System number can range from 1 to 65535.

Only other EIGRP routers in Autonomous System *10* will form neighbor adjacencies and share updates with this router.

The *network* statements serve two purposes in EIGRP:

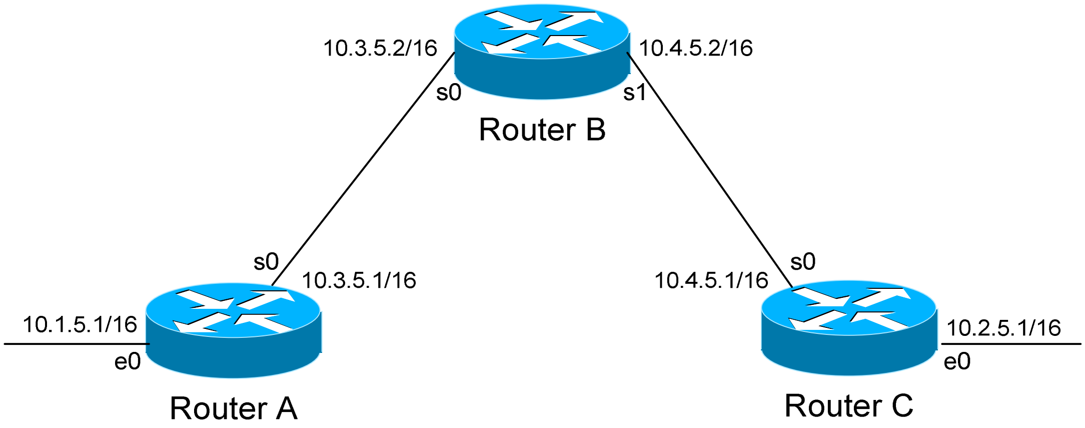
- First, they identify which *networks* you wish to advertise to other EIGRP routers (similar to RIP).
- Second, they identify which *interfaces* on the local router to attempt to form neighbor relationships out of (similar to OSPF).

Prior to IOS version 12.0(4), the *network* statements were classful, despite the fact that EIGRP is a classless routing protocol. For example, the above *network 10.0.0.0* command would advertise the networks of directly-connected interfaces belonging to the 10.0.0.0/8 network and its subnets. It would further attempt to form neighbor relationships out of these interfaces.

IOS version **12.0(4) and later** provided us with more granular control of our *network* statements. It introduced a *wildcard mask* parameter, which allows us to choose the networks to advertise in a classless fashion:

```
RouterA(config)# router eigrp 10
RouterA(config-router)# network 172.16.0.0 0.0.255.255
RouterA(config-router)# network 10.1.4.0 0.0.0.255
```

EIGRP Passive Interfaces



It is possible to control which router interfaces will participate in the EIGRP process. Just as with RIP, we can use the *passive-interface* command.

However, please note that the *passive-interface* command works differently with EIGRP than with RIP or IGRP. EIGRP will no longer form neighbor relationships out of a "passive" interface, thus this command prevents updates from being *sent* or *received* out of this interface:

```
RouterC(config)#  router eigrp 10
RouterC(config-router)#  network 10.4.0.0
RouterC(config-router)#  network 10.2.0.0
RouterC(config-router)#  passive-interface s0
```

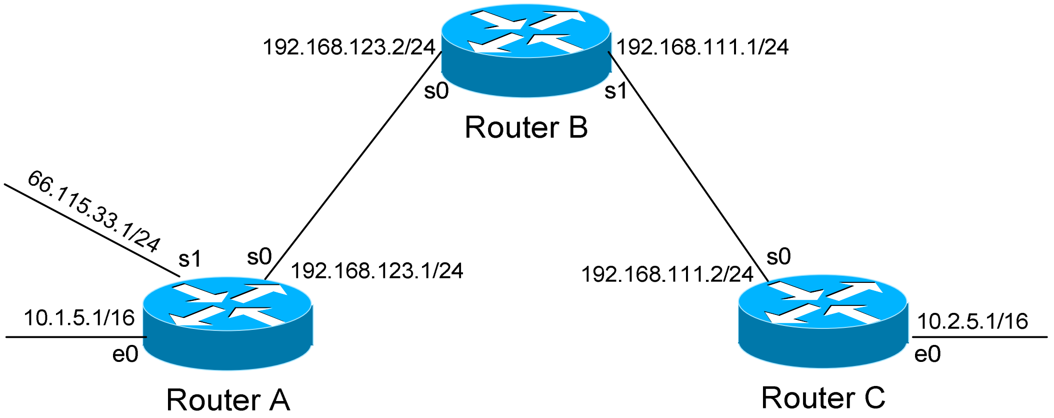
Router C will not form a neighbor adjacency with Router B.

We can configure **all** interfaces to be passive using the *passive-interface default* command, and then individually use the *no passive-interface* command on the interfaces we **do** want neighbors to be formed on:

```
RouterC(config)#  router eigrp 10
RouterC(config-router)#  network 10.4.0.0
RouterC(config-router)#  network 10.2.0.0
RouterC(config-router)#  passive-interface default
RouterC(config-router)#  no passive-interface e0
```

Always remember, that the *passive-interface* command will prevent EIGRP (and OSPF) from forming neighbor relationships out of that interface. **No** routing updates are passed in either direction.

EIGRP Auto-Summarization



EIGRP is a classless routing protocol that supports Variable Length Subnet Masks (VLSMs). The above example would pose no problem for EIGRP.

However, EIGRP will still **automatically summarize** when crossing **major network boundaries**.

For example, when Router A sends an EIGRP update to Router B via Serial0, by default it will still **summarize** the 10.1.0.0/16 network to 10.0.0.0/8. This is because the 10.1.0.0/16 and 192.168.123.0/24 networks *do not* belong to the same major network. Likewise, the 66.115.33.0/24 network will be summarized to 66.0.0.0/8.

An auto-summary route will be advertised as a normal *internal* EIGRP route. The *best* (lowest) metric from among the summarized routes will be applied to this summary route.

The router that *performed* the auto-summarization will also add the summary route to its routing table, with a next hop of the *Null0* interface. This is to prevent routing loops.

This **auto-summarization** can be disabled:

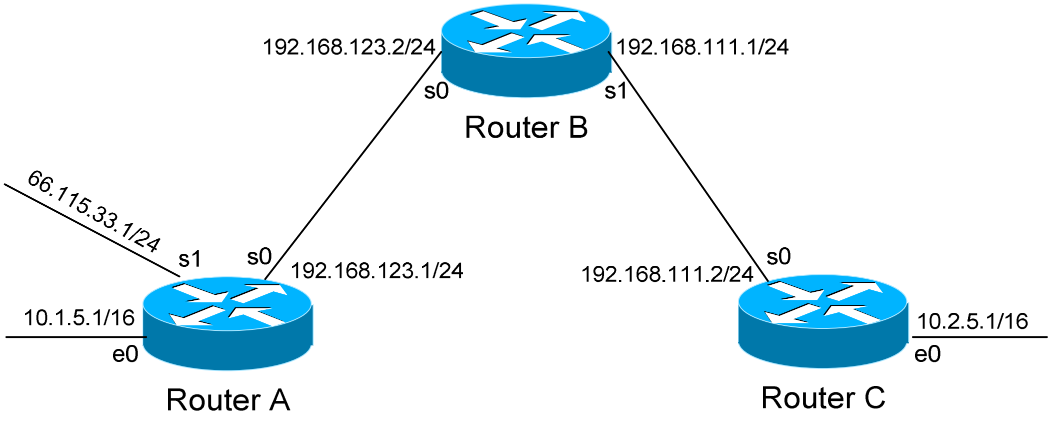
```
RouterA(config)#  router eigrp 10
RouterA(config-router)#  no auto-summary
```

The *no auto-summary* command will prevent Router A from summarizing the 10.1.0.0/16 and 66.115.33.0/24 networks.

(Reference: http://www.cisco.com/en/US/tech/tk365/technologies_white_paper09186a0080094cb7.shtml#summarization)

* * *

EIGRP Manual Summarization



In some instances, it is necessary to *manually* summarize networks.

For example, you may not want certain networks to be auto-summarized, but other specific networks *should* be summarized. In this instance, summarization can be manually applied using the following interface configuration command:

```
RouterA(config)#  int s0
RouterA(config-if)#  ip summary-address eigrp 10 66.0.0.0 255.0.0.0
```

Recall that auto-summarization had been previously disabled on Router A to allow the 10.1.0.0/16 network to be advertised correctly. However, this would also mean that the 66.115.33.0/24 network would *not* be summarized as well.

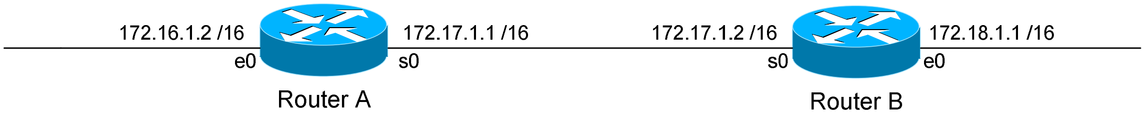
The *ip summary-address* command allows us to **manually** summarize this network. Notice that we configure this on the interface that will be **advertising** this network to the other routers.

The manually-created summary route is *not* advertised as an internal EIGRP route, but instead is classified as an EIGRP summary route. An EIGRP summary route has an Administrative Distance of **5**, as opposed to an AD of **90** for internal routes.

As with auto-summarization, the router performing *manual* summarization will add the summary route to its routing table, with a next hop of the *Null0* interface.

The summary route will only stay in the routing table if a more specific route still exists.

EIGRP Authentication



EIGRP supports authentication to secure routing updates.

The first step is creating a shared authentication *key* that must be identical on both routers. This is accomplished in global configuration mode:

```
RouterA(config)# key chain MYCHAIN
RouterA(config-keychain)# key 1
RouterA(config-keychain-key)# key-string MYPASSWORD

RouterB(config)# key chain MYCHAIN
RouterB(config-keychain)# key 1
RouterB(config-keychain-key)# key-string MYPASSWORD
```

The first command creates a *key chain* called *MYCHAIN*. We must then associate a *key* to our keychain. Then we actually configure the shared key using the *key-string* command.

We then apply our key chain to the interface connecting to the other router:

```
RouterA(config)# interface s0
RouterA(config-if)# ip authentication key-chain eigrp 10 MYCHAIN

RouterB(config)# interface s0
RouterB(config-if)# ip authentication key-chain eigrp 10 MYCHAIN
```

If there was another router off of Router B's Ethernet port, we could create a *separate* key chain with a different key-string. Every router on the EIGRP domain does not need to use the same key chain, only interfaces directly connecting two (or more) routers.

The final step in configuring authentication is identifying which encryption to use. Unlike RIP, EIGRP only supports MD5 encryption:

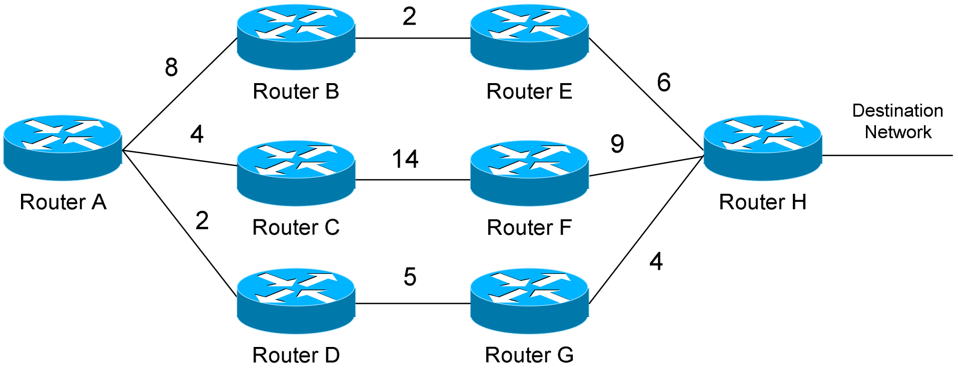
```
RouterA(config)# interface s0
RouterA(config-if)# ip authentication mode eigrp 10 md5
```

Please note that configuring authentication for EIGRP is similar to that of RIP, but there are slight variations in the commands, including the addition of the EIGRP Autonomous System Number.

EIGRP Load-Balancing

By default, EIGRP will automatically load-balance across equal-metric routes (four by default, six maximum). EIGRP also supports load-balancing across routes with an *unequal* metric.

Consider the following example:



Earlier in this section, we established that Router A would choose the route through Router D as its **Feasible Distance** to the destination network. The route through Router B became a **Feasible Successor.**

By default, EIGRP will *not* load-balance between these two routes, as their metrics are different (11 through Router D, 16 through Router B). We must use the *variance* command to tell EIGRP to load-balance across these unequal-metric links:

```
RouterA(config)#  router eigrp 10
RouterA(config-router)#  variance 2
RouterA(config-router)#  maximum-paths 6
```

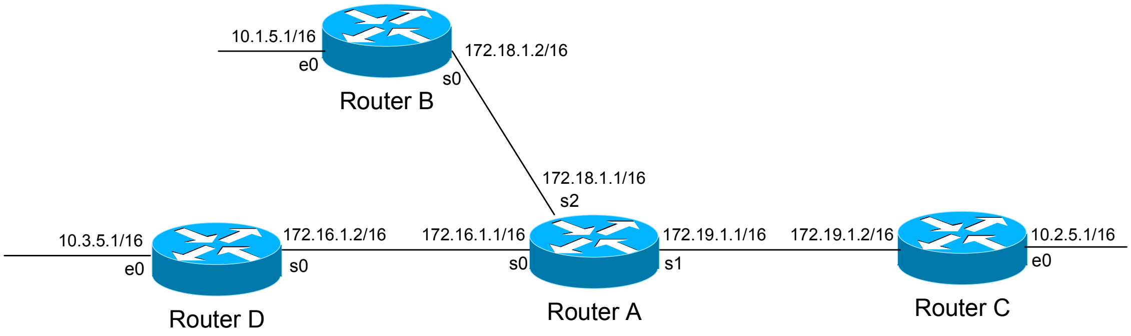
The *variance* command assigns a “multiplier,” in this instance of *2*. We multiply this *variance* value by the metric of our Feasible Distance (2 x 11 = 22). Thus, any Feasible Successors with a metric within twice that of our Feasible Distance (i.e. 12 through 22) will now be used for load balancing by EIGRP.

Remember, only Feasible Successors can be used for load balancing, not Possibilities (such as the route through Router C).

The *maximum-paths* command adjusts the number of links EIGRP can load-balance across.

(Reference: http://www.cisco.com/warp/public/103/19.html)

EIGRP Stubs



Consider the above hub-and-spoke environment. If Router C were to fail, Router A (the *hub* router) would mark the 10.2.0.0/16 route as **Active**, and send out **Query** packets to the *spoke* routers for an alternate path.

However, it is obvious that no other route exists to the 10.2.0.0/16 network. Thus, the *querying* process is a waste of bandwidth and resources.

To prevent unnecessary querying, "spoke" routers in a hub-and-spoke environment can be configured as **Stub** routers. A stub router builds a neighbor adjacency with its hub router(s), and will inform neighbors of its stub status. The stub router will still build the *full* topology table.

However, the stub router will immediately respond to any Query packets with an **Inaccessible** message. Neighbors will eventually stop querying the stub router, which helps EIGRP converge quicker and conserves bandwidth.

Configuration of an EIGRP *stub* is always performed on the **spoke** router:

```
RouterB(config)#  router eigrp 10
RouterB(config-router)#  eigrp stub connected
```

The *eigrp stub* command configures this router as Stub, and supports four possible parameters:

- **Receive-only –** router will not share updates with neighbors
- **Connected –** router will only advertise connected networks
- **Static** – router will only advertise static networks
- **Summary** – router will only advertise summary routes

The *connected* and *static* parameters will only advertise those networks if they have been injected into the EIGRP process, either using *network* statements or using route redistribution. By default, EIGRP stubs will only send **connected** and **summary** routes to neighbors.

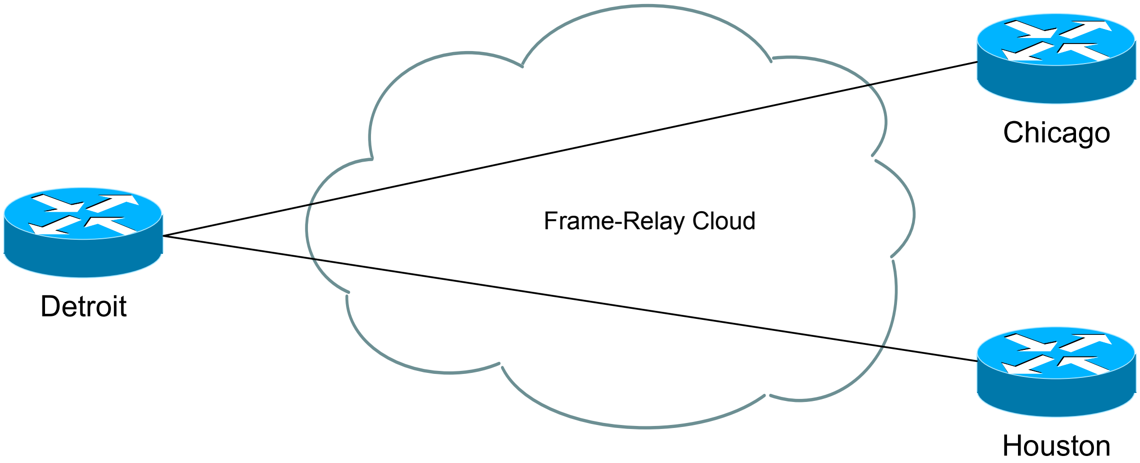
(Reference: http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120s/120s15/eigrpstb.htm)

* * *

EIGRP, Frame-Relay, and Bandwidth

Recall that EIGRP's default bandwidth on serial links is set to 1.544 Mbps (specifically, *1544)*. Bandwidth on LAN interfaces (such as Ethernet) is set to the actual physical speed of the link. For point-to-point PPP or HDLC links, bandwidth should be manually adjusted to the line's physical speed.

Additional considerations exist when using Frame-Relay. Observe the following diagram. Assume the Detroit router's connection into the Frame-Relay cloud is 256 Kbps (shared between the Chicago and Houston PVCs).



Cisco specifies three rules regarding EIGRP over Frame-Relay:

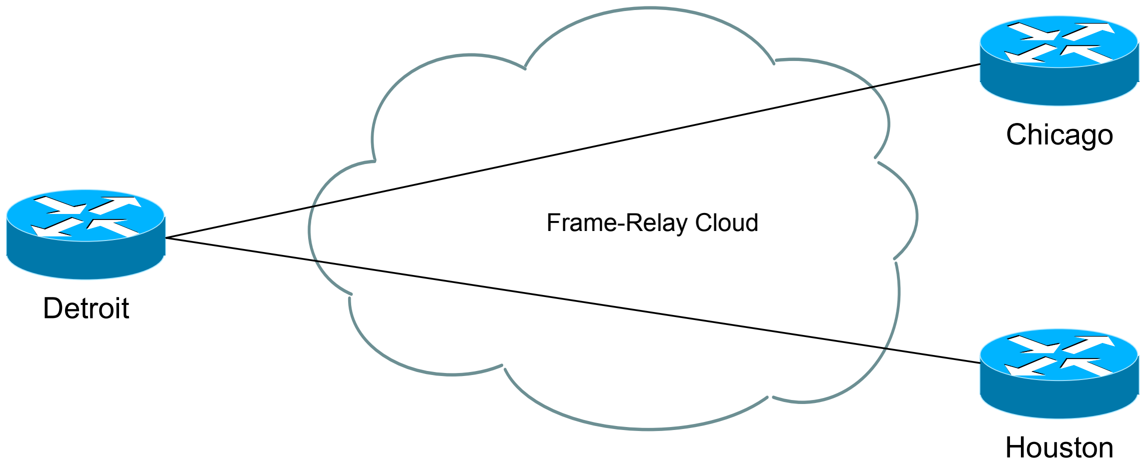
- The configured bandwidth (and the percentage of bandwidth EIGRP can use) for a PVC cannot exceed the bandwidth of the PVC (CIR).
- The bandwidth for EIGRP across *all* PVCs on an interface cannot exceed the physical bandwidth of the interface
- The bandwidth for EIGRP must be identical on both ends of a PVC.

Consider if router Detroit was configured using Frame-Relay point-to-multipoint, using no sub-interfaces. Assume also that no bandwidth command is configured on the physical interface. EIGRP will assume that the bandwidth is evenly split between all PVCs. In the above scenario, EIGRP would assume that each PVC was allocated 128 Kbps.

If the CIR for the PVCs were not equal – say, Detroit to Chicago is 56Kbps, and Detroit to Houston is 256Kbps – the bandwidth should be calculated by multiplying the bandwidth of the *slowest* PVC with the total number of PVCs. In this scenario, the bandwidth should be set to 128Kbps.

(Reference: http://www.cisco.com/warp/public/103/12.html)

EIGRP, Frame-Relay, and Split-Horizon



Observe the above Frame-Relay network. We have two possible configuration options for Detroit:

- Configure frame-relay map statements on the physical interface
- Create separate subinterfaces for each link, treating them as separate point-to-points.

If choosing the latter, EIGRP will treat each subinterface as a separate link, and routing will occur with no issue.

If choosing the former, EIGRP will be faced with a split-horizon issue. Updates from Houston will not be forwarded to Chicago, and visa versa, as split horizon prevents an update from being sent out the link it was received on.

Thus, we must disable split horizon for EIGRP:

```
Detroit(config)#  interface s0/0
Detroit(config-router)#  no ip split-horizon eigrp 10
```

Troubleshooting EIGRP

To view the EIGRP Neighbor Table:

Router# *show ip eigrp neighbor*

```
IP-EIGRP neighbors for process 10
H  Address     Interface  Hold   Uptime    SRTT    RTO   Q      Seq    Type
                                           (sec)   (ms)  Cnt    Num
0  172.16.1.2     S0       13    00:00:53   32     200    0      2
0  172.18.1.2     S2       11    00:00:59   32     200    0      3
```

To view the EIGRP Topology Table, containing all EIGRP route information:

Router# *show ip eigrp topology*

```
IP-EIGRP Topology Table for AS(10)/ID(172.19.1.1)

Codes: P - Passive, A - Active, U - Update, Q - Query, R - Reply,
            r - reply Status, s - sia Status

P 10.3.0.0/16, 1 successors, FD is 2297856
         via 172.16.1.2 (2297856/128256), Serial0
P 172.19.0.0/16, 1 successors, FD is 281600
         via Connected, Serial 1
P 172.18.0.0/16, 1 successors, FD is 128256
         via Connected, Serial 2
P 172.16.0.0/16, 1 successors, FD is 2169856
         via Connected, Serial0
```

To view information on EIGRP traffic sent and received on a router:

Router# *show ip eigrp traffic*

```
IP-EIGRP Traffic Statistics for process 10
  Hellos sent/received: 685/429
  Updates sent/received: 4/3
  Queries sent/received: 0/0
  Replies sent/received: 0/0
  Acks sent/received: 1/2
  Input queue high water mark 1, 0 drops
  SIA-Queries sent/received: 0/0
  SIA-Replies sent/received: 0/0
```

Troubleshooting EIGRP (continued)

To view the bandwidth, delay, load, reliability and MTU values of an interface:

Router# *show interface s0*

```
Serial0 is up, line protocol is up
  Hardware is HD64570
  Internet address is 172.16.1.1/16
  MTU 1500 bytes, BW 1544 Kbit, DLY 20000 usec,
     reliability 255/255, txload 1/255, rxload 1/255

<irrelevant output removed>
```

To view information specific to the EIGRP protocol:

Router# *show ip protocols*

```
Routing Protocol is "eigrp 10"
  Outgoing update filter list for all interfaces is not set
  Incoming update filter list for all interfaces is not set
  Default networks flagged in outgoing updates
  Default networks accepted from incoming updates
  EIGRP metric weight K1=1, K2=0, K3=1, K4=0, K5=0
  EIGRP maximum hopcount 100
  EIGRP maximum metric variance 1
  Redistributing: eigrp 10
  Automatic network summarization is not in effect
  Maximum path: 4
  Routing for Networks:
    172.16.0.0
    172.18.0.0
    172.19.0.0
  Routing Information Sources:
    Gateway         Distance      Last Update
    (this router)         90      00:26:11
    172.16.1.2            90      00:23:49
  Distance: internal 90 external 170
```

This command provides us with information on EIGRP timers, EIGRP metrics, summarization, and the specific networks RIP is advertising.

Troubleshooting EIGRP (continued)

To view the IP routing table:

Router# *show ip route*

```
Gateway of last resort is not set

C     172.16.0.0 is directly connected, Serial0
C     172.19.0.0 is directly connected, Serial1
D     10.3.0.0 [90/2297856] via 172.16.1.2, 00:00:15, Serial0
```

To view a specific route within the IP routing table:

Router# *show ip route 10.3.0.0*

```
Routing entry for 10.3.0.0/16
  Known via "eigrp 10", distance 90, metric 2297856 type internal
  Last update from 172.16.1.2 on Serial 0, 00:00:15 ago
```

To debug EIGRP in realtime:

Router# *debug eigrp neighbors*
Router# *debug eigrp packet*
Router# *debug eigrp route*
Router# *debug eigrp summary*

Section 12
- Open Shortest Path First -

OSPF (Open Shortest Path First)

OSPF is a standardized Link-State routing protocol, designed to scale efficiently to support larger networks.

OSPF adheres to the following Link State characteristics:

- OSPF employs a hierarchical network design using **Areas.**
- OSPF will form **neighbor** relationships with adjacent routers in the same **Area.**
- Instead of advertising the *distance* to connected networks, OSPF advertises the *status* of directly connected **links** using **Link-State Advertisements (LSAs)**.
- OSPF sends updates (LSAs) when there is a change to one of its links, and will *only* send the change in the update. LSAs are additionally refreshed every **30 minutes**.
- OSPF traffic is multicast either to address **224.0.0.5 (**all OSPF routers) or **224.0.0.6** (all Designated Routers).
- OSPF uses the **Dijkstra Shortest Path First** algorithm to determine the shortest path.
- OSPF is a classless protocol, and thus supports VLSMs.

Other characteristics of OSPF include:

- OSPF supports only IP routing.
- OSPF routes have an administrative distance is **110**.
- OSPF uses **cost** as its metric, which is computed based on the bandwidth of the link. OSPF has no hop-count limit.

The OSPF process builds and maintains three separate tables:

- A **neighbor table** – contains a list of all neighboring routers.
- A **topology table** – contains a list of *all* possible routes to all known networks within an area.
- A **routing table** – contains the *best* route for each known network.

OSPF Neighbors

OSPF forms neighbor relationships, called **adjacencies,** with other routers in the same **Area** by exchanging **Hello** packets to multicast address **224.0.0.5**. Only after an adjacency is formed can routers share routing information.

Each OSPF router is identified by a unique **Router ID.** The Router ID can be determined in one of three ways:

- The Router ID can be **manually** specified.
- If not manually specified, the highest IP address configured on any **Loopback interface** on the router will become the Router ID.
- If no loopback interface exists, the highest IP address configured on any **Physical interface** will become the Router ID.

By default, Hello packets are sent out OSPF-enabled interfaces every **10 seconds** for broadcast and point-to-point interfaces, and **30 seconds** for non-broadcast and point-to-multipoint interfaces.

OSPF also has a **Dead Interval**, which indicates how long a router will wait without hearing any hellos before announcing a neighbor as "down." Default for the Dead Interval is **40 seconds** for broadcast and point-to-point interfaces, and **120** seconds for non-broadcast and point-to-multipoint interfaces. Notice that, by default, the dead interval timer is four times the Hello interval.

These timers can be adjusted on a *per interface* basis:

```
Router(config-if)#  ip ospf hello-interval 15
Router(config-if)#  ip ospf dead-interval 60
```

OSPF Neighbors (continued)

OSPF routers will only become neighbors if the following parameters within a Hello packet are identical on each router:

- Area ID
- Area Type (stub, NSSA, etc.)
- Prefix
- Subnet Mask
- Hello Interval
- Dead Interval
- Network Type (broadcast, point-to-point, etc.)
- Authentication

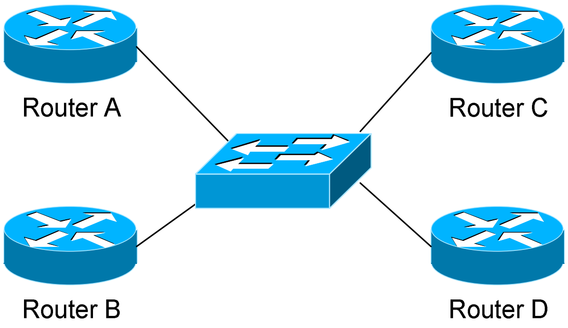
The Hello packets also serve as **keepalives** to allow routers to quickly discover if a neighbor is down. Hello packets also contain a **neighbor field** that lists the Router IDs of all neighbors the router is connected to.

A **neighbor table** is constructed from the OSPF Hello packets, which includes the following information:

- The **Router ID** of each neighboring router
- The current "state" of each neighboring router
- The interface directly connecting to each neighbor
- The IP address of the remote interface of each neighbor

(Reference: http://www.cisco.com/warp/public/104/29.html)

OSPF Designated Routers



In multi-access networks such as Ethernet, there is the possibility of *many* neighbor relationships on the same physical segment. In the above example, four routers are connected into the same multi-access segment. Using the following formula (where "n" is the number of routers):

n(n-1)/2

…..it is apparent that **6** separate adjacencies are needed for a fully meshed network. Increase the number of routers to five, and **10** separate adjacencies would be required. This leads to a considerable amount of unnecessary Link State Advertisement (LSA) traffic.

If a link off of Router A were to fail, it would flood this information to all neighbors. Each neighbor, in turn, would then flood that same information to all other neighbors. This is a waste of bandwidth and processor load.

To prevent this, OSPF will elect a **Designated Router (DR)** for each multi-access networks, accessed via multicast address **224.0.0.6**. For redundancy purposes, a **Backup Designated Router (BDR)** is also elected.

OSPF routers will form adjacencies with the DR and BDR. If a change occurs to a link, the update is forwarded only to the DR, which then forwards it to all other routers. This greatly reduces the flooding of LSAs.

DR and BDR elections are determined by a router's **OSPF priority**, which is configured on a per-interface basis (a router can have interfaces in multiple multi-access networks). The router with the **highest priority** becomes the DR; second highest becomes the BDR. If there is a tie in priority, whichever router has the **highest Router ID** will become the DR. To change the priority on an interface:

```
Router(config-if)#  ip ospf priority 125
```

Default priority on Cisco routers is **1**. A priority of **0** will prevent the router from being elected DR or BDR. **Note:** The DR election process is ***not preemptive***. Thus, if a router with a higher priority is added to the network, it will *not* automatically supplant an existing DR. Thus, a router that should never become the DR should always have its priority set to 0.

OSPF Neighbor States

Neighbor adjacencies will progress through several **states**, including:

Down – indicates that no Hellos have been heard from the neighboring router.

Init – indicates a Hello packet has been heard from the neighbor, but two-way communication has not yet been initialized.

2-Way – indicates that bidirectional communication has been established. Recall that Hello packets contain a *neighbor* field. Thus, communication is considered 2-Way once a router sees its own Router ID in its neighbor's Hello Packet. **Designated** and **Backup Designated Routers** are elected at this stage.

ExStart – indicates that the routers are preparing to share link state information. Master/slave relationships are formed between routers to determine who will begin the exchange.

Exchange – indicates that the routers are exchanging **Database Descriptors (DBDs).** DBDs contain a description of the router's Topology Database. A router will examine a neighbor's DBD to determine if it has information to share.

Loading – indicates the routers are finally exchanging **Link State Advertisements,** containing information about all links connected to each router. Essentially, routers are sharing their topology tables with each other.

Full – indicates that the routers are fully synchronized. The topology table of all routers in the area should now be identical. Depending on the "role" of the neighbor, the state may appear as:

- **Full/DR** – indicating that the neighbor is a Designated Router (DR)
- **Full/BDR** – indicating that the neighbor is a Backup Designated Router (BDR)
- **Full/DROther –** indicating that the neighbor is neither the DR or BDR

On a multi-access network, OSPF routers will *only* form Full adjacencies with DRs and BDRs. Non-DRs and non-BDRs will still form adjacencies, but will remain in a **2-Way State**. This is normal OSPF behavior.

OSPF Network Types

OSPF's functionality is different across several different network topology types. OSPF's interaction with Frame Relay will be explained in another section

Broadcast Multi-Access – indicates a topology where broadcast occurs.

- Examples include Ethernet, Token Ring, and ATM.
- OSPF *will* elect DRs and BDRs.
- Traffic to DRs and BDRs is multicast to 224.0.0.6. Traffic from DRs and BDRs to other routers is multicast to 224.0.0.5.
- Neighbors *do not* need to be manually specified.

Point-to-Point – indicates a topology where two routers are directly connected.

- An example would be a point-to-point T1.
- OSPF *will not* elect DRs and BDRs.
- All OSPF traffic is multicast to 224.0.0.5.
- Neighbors *do not* need to be manually specified.

Point-to-Multipoint – indicates a topology where one interface can connect to multiple destinations. Each connection between a source and destination is treated as a point-to-point link.

- An example would be Point-to-Multipoint Frame Relay.
- OSPF *will not* elect DRs and BDRs.
- All OSPF traffic is multicast to 224.0.0.5.
- Neighbors *do not* need to be manually specified.

Non-broadcast Multi-access Network (NBMA) – indicates a topology where one interface can connect to multiple destinations; however, broadcasts cannot be sent across a NBMA network.

- An example would be Frame Relay.
- OSPF *will* elect DRs and BDRs.
- OSPF neighbors must be *manually* defined, thus All OSPF traffic is unicast instead of multicast.

Remember: on *non-broadcast* networks, neighbors must be **manually specified**, as multicast Hello's are not allowed.

Configuring OSPF Network Types

The default OSPF network type for basic Frame Relay is **Non-broadcast Multi-access Network (NBMA).** To configure manually:

```
Router(config)#  interface s0
Router(config-if)#  encapsulation frame-relay
Router(config-if)#  frame-relay map ip 10.1.1.1 101
Router(config-if)#  ip ospf network non-broadcast

Router(config)#  router ospf 1
Router(config-router)#  neighbor 10.1.1.1
```

Notice that the *neighbor* was manually specified, as multicasting is not allowed on an NBMA. However, the Frame-Relay network can be tricked into allowing broadcasts, eliminating the need to manually specify neighbors:

```
Router(config)#  interface s0
Router(config-if)#  encapsulation frame-relay
Router(config-if)#  frame-relay map ip 10.1.1.1 101 broadcast
Router(config-if)#  ip ospf network broadcast
```

Notice that the *ospf network* type has been changed to *broadcast,* and the *broadcast* parameter was added to the *frame-relay map* command. The neighbor no longer needs to be specified, as multicasts will be allowed out this map.

The default OSPF network type for Ethernet and Token Ring is **Broadcast Multi-Access.** To configure manually:

```
Router(config)#  interface e0
Router(config-if)#  ip ospf network broadcast
```

The default OSPF network type for T1's (HDLC or PPP) and Point-to-Point Frame Relay is **Point-to-Point.** To configure manually:

```
Router(config)#  interface s0
Router(config-if)#  encapsulation frame-relay

Router(config)#  interface s0.1 point-to-point
Router(config-if)#  frame-relay map ip 10.1.1.1 101 broadcast
Router(config-if)#  ip ospf network point-to-point
```

Configuring OSPF Network Types (continued)

The default OSPF network type for Point-to-Multipoint Frame Relay is *still* **Non-broadcast Multi-access Network (NBMA).** However, OSPF supports an additional network type called **Point-to-Multipoint,** which will allow neighbor discovery to occur automatically**.** To configure:

```
Router(config)#  interface s0
Router(config-if)#  encapsulation frame-relay

Router(config)#  interface s0.2 multipoint
Router(config-if)#  frame-relay map ip 10.1.1.1 101 broadcast
Router(config-if)#  ip ospf network point-to-multipoint
```

Additionally, a *non-broadcast* parameter can be added to the *ip ospf network* command when specifying *point-to-multipoint.*

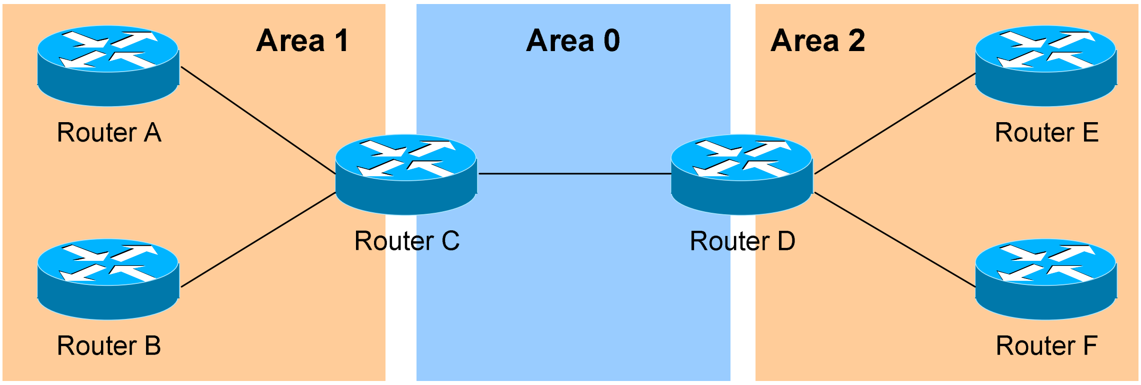
```
Router(config)#  interface s0
Router(config-if)#  encapsulation frame-relay

Router(config)#  interface s0.2 multipoint
Router(config-if)#  frame-relay map ip 10.1.1.1 101
Router(config-if)#  ip ospf network point-to-multipoint non-broadcast

Router(config)#  router ospf 1
Router(config-router)#  neighbor 10.1.1.1
```

Notice the different in configuration. The *frame-relay map* command no longer has the *broadcast* parameter, as broadcasts and multicasts are not allowed on a non-broadcast network.

Thus, in the OSPF router configuration, neighbors must again be manually specified. Traffic to those neighbors will be **unicast** instead of multicast.

OSPF network types must be set identically on two “neighboring” routers, otherwise they will never form an adjacency.

The OSPF Hierarchy



OSPF is a hierarchical system that separates an Autonomous System into individual **areas**. OSPF traffic can either be **intra-area** (within one area), **inter-area** (between separate areas), or **external** (from another AS).

OSPF routers build a **Topology Database** of all **links** within their area, and all routers within an area will have an *identical* topology database. Routing updates between these routers will *only* contain information about links local to their area. Limiting the topology database to include only the local area conserves bandwidth and reduces CPU loads.

Area 0 is required for OSPF to function, and is considered the "**Backbone**" area. As a rule, all other areas must have a connection into Area 0, though this rule can be bypassed using **virtual links** (explained shortly). Area 0 is often referred to as the *transit* area to connect all other areas.

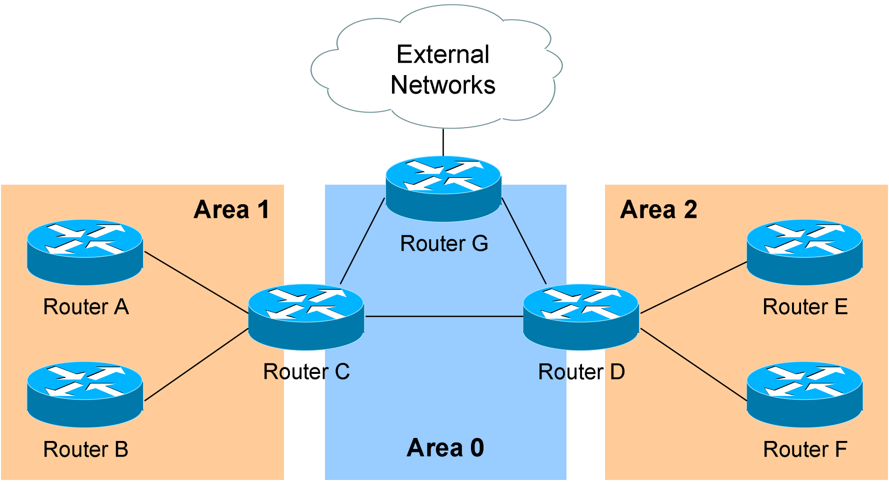
OSPF routers can belong to multiple areas, and will thus contain separate Topology databases for each area. These routers are known as **Area Border Routers (ABRs)**.

Consider the above example. Three areas exist: Area 0, Area 1, and Area 2. Area 0, again, is the backbone area for this Autonomous System. Both Area 1 and Area 2 must directly connect to Area 0.

Routers A and B belong fully to Area 1, while Routers E and F belong fully to Area 2. These are known as **Internal Routers**.

Router C belongs to both Area 0 and Area 1. Thus, it is an **ABR**. Because it has an interface in Area 0, it can also be considered a **Backbone Router**. The same can be said for Router D, as it belongs to both Area 0 and Area 2.

The OSPF Hierarchy (continued)



Now consider the above example. Router G has been added, which belongs to Area 0. However, Router G also has a connection to the Internet, which is outside this Autonomous System.

This makes Router G an **Autonomous System Border Router** (**ASBR**). A router can become an ASBR in one of two ways:

- By connecting to a separate Autonomous System, such as the Internet
- By redistributing another routing protocol into the OSPF process.

ASBRs provide access to *external* networks. OSPF defines two "types" of external routes:

- **Type 2 (E2**) – Includes only the external cost to the destination network. External cost is the metric being advertised from outside the OSPF domain. This is the default type assigned to external routes.
- **Type 1 (E1**) – Includes both the external cost, and the internal cost to reach the ASBR, to determine the total metric to reach the destination network. Type 1 routes are always *preferred* over Type 2 routes to the same destination.

Thus, the four separate OSPF router types are as follows:

- **Internal Routers –** all router interfaces belong to only one Area.
- **Area Border Routers (ABRs) –** contains interfaces in at least two separate areas
- **Backbone Routers –** contain at least one interface in Area 0
- **Autonomous System Border Routers (ASBRs) –** contain a connection to a separate Autonomous System

LSAs and the OSPF Topology Database

OSPF, as a link-state routing protocol, does not rely on *routing-by-rumor* as RIP and IGRP do.

Instead, OSPF routers keep track of the status of **links** within their respective areas. A link is simply a router interface. From these lists of links and their respective statuses, the topology database is created. OSPF routers forward **link-state advertisements (LSAs)** to ensure the topology database is consistent on each router within an area.

Several LSA types exist:

- **Router LSA (Type 1) –** Contains a list of all links local to the router, and the status and "cost" of those links. Type 1 LSAs are generated by all routers in OSPF, and are flooded to all other routers within the local area.

- **Network LSA (Type 2) –** Generated by all Designated Routers in OSPF, and contains a list of all routers attached to the Designated Router.

- **Network Summary LSA (Type 3)** – Generated by all ABRs in OSPF, and contains a list of all destination networks within an area. Type 3 LSAs are sent between areas to allow inter-area communication to occur.

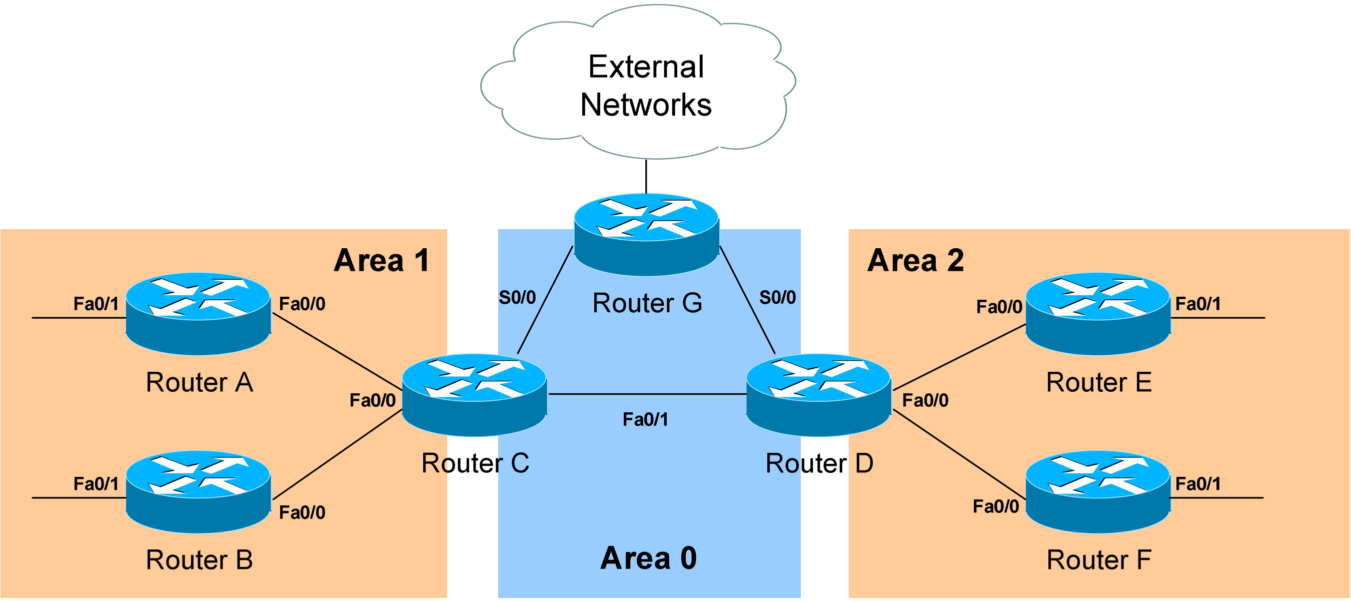
- **ASBR Summary LSA (Type 4)** – Generated by ABRs in OSPF, and contains a *route* to any ASBRs in the OSPF system. Type 4 LSAs are sent from an ABR into its local area, so that Internal routers know how to exit the Autonomous System.

- **External LSA (Type 5)** – Generated by ASBRs in OSPF, and contain routes to destination networks *outside* the local Autonomous System. Type 5 LSAs can also take the form of a **default route** to all networks outside the local AS. Type 5 LSAs are flooded to all areas in the OSPF system.

Multicast OSPF (MOSPF) utilizes a Type 6 LSA, but that goes beyond the scope of this guide.

Later in this section, **Type 7 NSSA External LSAs** will be described in detail.

LSAs and the OSPF Topology Database (continued)



From the above example, the following can be determined:

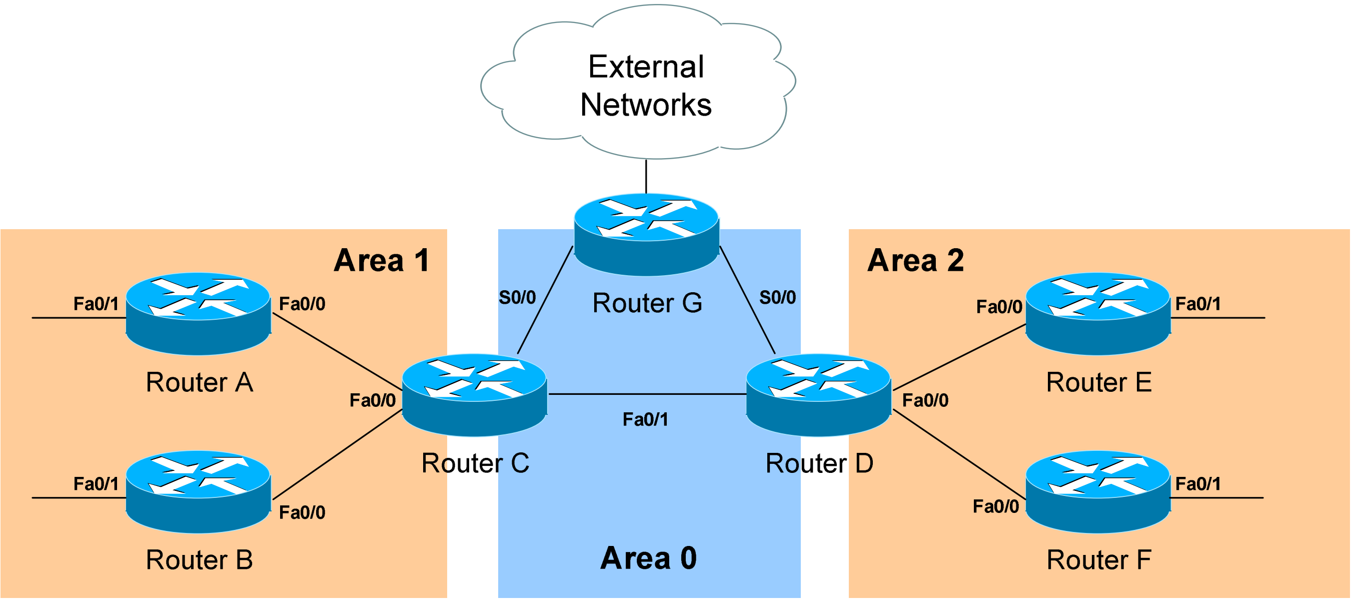
- Routers A, B, E, and F are **Internal Routers.**
- Routers C and D are **ABRs.**
- Router G is an **ASBR.**

All routers will generate **Router (Type 1) LSAs**. For example, Router A will generate a Type 1 LSA that contains the status of links FastEthernet 0/0 and FastEthernet 0/1. This LSA will be flooded to all other routers in Area 1.

Designated Routers will generate **Network (Type 2) LSAs.** For example, if Router C was elected the DR for the multi-access network in Area 1, it would generate a Type 2 LSA containing a list of all routers attached to it.

Area Border Routers (ABRs) will generate **Network Summary (Type 3) LSAs**. For example, Router C is an ABR between Area 0 and Area 1. It will thus send Type 3 LSAs into *both* areas. Type 3 LSAs sent into Area 0 will contain a list of networks within Area 1, including **costs** to reach those networks. Type 3 LSAs sent into Area 1 will contain a list of networks within Area 0, *and* all other areas connected to Area 0. This allows Area 1 to reach any other area, and all other areas to reach Area 1.

LSAs and the OSPF Topology Database (continued)



ABRs will also generate **ASBR Summary (Type 4) LSAs.** For example, Router C will send Type 4 LSAs into Area 1 containing a route to the ASBR, thus providing routers in Area 1 with the path out of the Autonomous System.

ASBRs will generate **External (Type 5) LSAs.** For example, Router G will generate Type 5 LSAs that contain routes to network outside the AS. These Type 5 LSAs will be flooded to routers of all areas.

Each type of LSA is propagated under three circumstances:

- When a new adjacency is formed.
- When a change occurs to the topology table.
- When an LSA reaches its maximum age (every **30 minutes,** by default).

Thus, though OSPF is typically recognized to only send updates when a change occurs, LSA's are still periodically refreshed every 30 minutes.

The OSPF Metric

OSPF determines the best (or *shortest)* path to a destination network using a **cost** metric, which is based on the bandwidth of interfaces. The *total* cost of a route is the sum of all *outgoing* interface costs. Lowest cost is preferred.

Cisco applies default costs to specific interface types:

Type	***Cost***
Serial (56K)	*1785*
Serial (64K)	*1562*
T1 (1.544Mbps)	*64*
Token Ring (4Mbps)	*25*
Ethernet (10 Mbps)	*10*
Token Ring (16 Mbps)	*6*
Fast Ethernet	*1*

On Serial interfaces, OSPF will use the configured *bandwidth* (measured in Kbps) to determine the cost:

```
Router(config)#  interface s0
Router(config-if)#  bandwidth 64
```

The default cost of an interface can be superseded:

```
Router(config)#  interface e0
Router(config-if)#  ip ospf cost 5
```

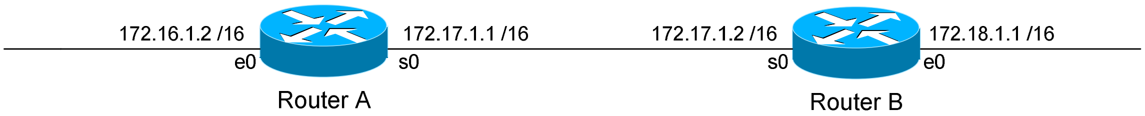
Changing the cost of an interface can alter which path OSPF deems the "shortest," and thus should be used with great care.

To alter how OSPF calculates its default metrics for interfaces:

```
Router(config)#  router ospf 1
Router(config-router)#  ospf auto-cost reference-bandwidth 100
```

The above *ospf auto-cost* command has a value of *100* configured, which is actually the default. This indicates that a 100Mbps link will have a cost of 1 (because 100/100 is 1). All other costs are based off of this. For example, the cost of 4 Mbps Token Ring is 25 because 100/4 = 25.

Configuring Basic OSPF



Routing protocol configuration occurs in Global Configuration mode. On Router A, to configure OSPF:

```
RouterA(config)#  router ospf 1
RouterA(config-router)#  router-id 1.1.1.1
RouterA(config-router)#  network 172.16.0.0 0.0.255.255 area 1
RouterA(config-router)#  network 172.17.0.0 0.0.255.255 area 0
```

The first command, *router ospf 1*, enables the OSPF process. The "*1*" indicates the OSPF process ID, and can be unique on each router. The process ID allows multiple OSPF processes to run on the same router. The *router-id* command assigns a unique OSPF ID of *1.1.1.1* for this router.

Note the use of a wildcard mask instead of a subnet mask in the *network* statement. With OSPF, we're *not* telling the router what networks to advertise; we're telling the router to place certain interfaces into specific areas, so those routers can form neighbor relationships. The wildcard mask *0.0.255.255* tells us that the last two octets can match any number.

The first *network* statement places interface E0 on Router A into Area 1. Likewise, the second *network* statement places interface S0 on Router A into Area 0. The network statement could have been written more specifically:

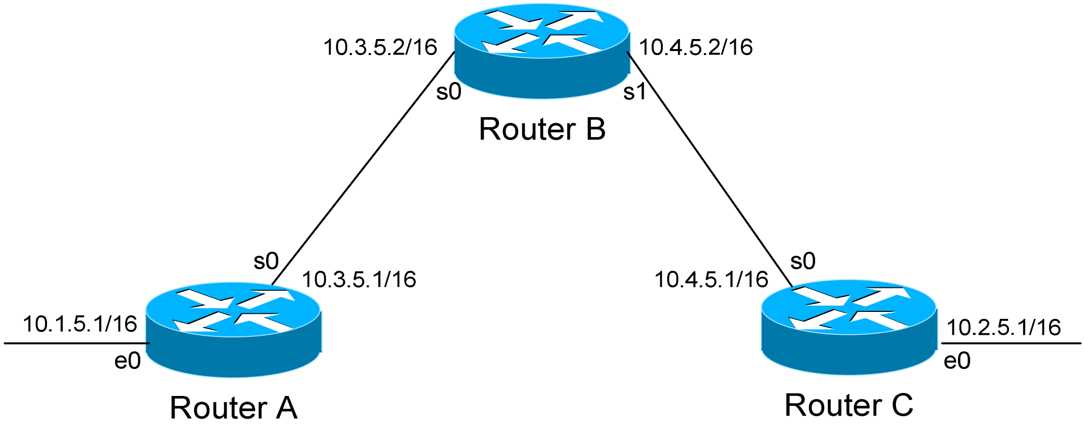
```
RouterA(config)#  router ospf 1
RouterA(config-router)#  network 172.16.1.2 0.0.0.0 area 1
RouterA(config-router)#  network 172.17.1.1 0.0.0.0 area 0
```

In order for Router B to form a neighbor relationship with Router A, its connecting interface must be put in the same Area as Router A:

```
RouterB(config)#  router ospf 1
RouterA(config-router)#  router-id 2.2.2.2
RouterB(config-router)#  network 172.17.1.2 0.0.0.0 area 0
RouterB(config-router)#  network 172.18.1.1 0.0.0.0 area 2
```

If Router B's S0 interface was placed in a different area than Router A's S0 interface, the two routers would never form a neighbor relationship, and never share routing updates.

OSPF Passive-Interfaces



It is possible to control which router interfaces will participate in the OSPF process. Just as with EIGRP and RIP, we can use the *passive-interface* command.

However, please note that the *passive-interface* command works differently with OSPF than with RIP or IGRP. OSPF will no longer form neighbor relationships out of a "passive" interface, thus this command prevents updates from being *sent* or *received* out of this interface:

```
RouterC(config)# router ospf 1
RouterC(config-router)# network 10.4.0.0 0.0.255.255 area 0
RouterC(config-router)# network 10.2.0.0 0.0.255.255 area 0
RouterC(config-router)# passive-interface s0
```

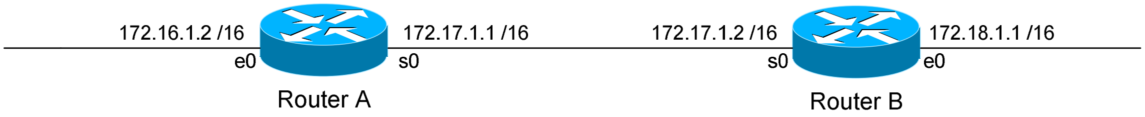
Router C will not form a neighbor adjacency with Router B.

It is possible to configure **all** interfaces to be passive using the *passive-interface default* command, and then individually use the *no passive-interface* command on the interfaces that neighbors **should** be formed on:

```
RouterC(config)# router ospf 1
RouterC(config-router)# network 10.4.0.0 0.0.255.255 area 0
RouterC(config-router)# network 10.2.0.0 0.0.255.255 area 0
RouterC(config-router)# passive-interface default
RouterC(config-router)# no passive-interface e0
```

Always remember, that the *passive-interface* command will prevent OSPF (and EIGRP) from forming neighbor relationships out of that interface. **No** routing updates are passed in either direction.

OSPF Authentication



OSPF supports authentication to secure routing updates. However, OSPF authentication is configured differently than RIP or EIGRP authentication.

Two forms of OSPF authentication exist, using either **clear-text** or an **MD5 hash.** To configure clear-text authentication, the first step is to enable authentication for the area, under the OSPF routing process:

```
RouterA(config)# router ospf 1
RouterA(config-router)# network 172.17.0.0 0.0.255.255 area 0
RouterA(config-router)# area 0 authentication
```

Then, the authentication *key* must be configured on the interface:

```
RouterA(config)# interface s0
RouterA(config-if)# ip ospf authentication
RouterA(config-if)# ip ospf authentication-key MYKEY
```

To configure MD5-hashed authentication, the first step is also to enable authentication for the area under the OSPF process:

```
RouterA(config)# router ospf 1
RouterA(config-router)# network 172.17.0.0 0.0.255.255 area 0
RouterA(config-router)# area 0 authentication message-digest
```

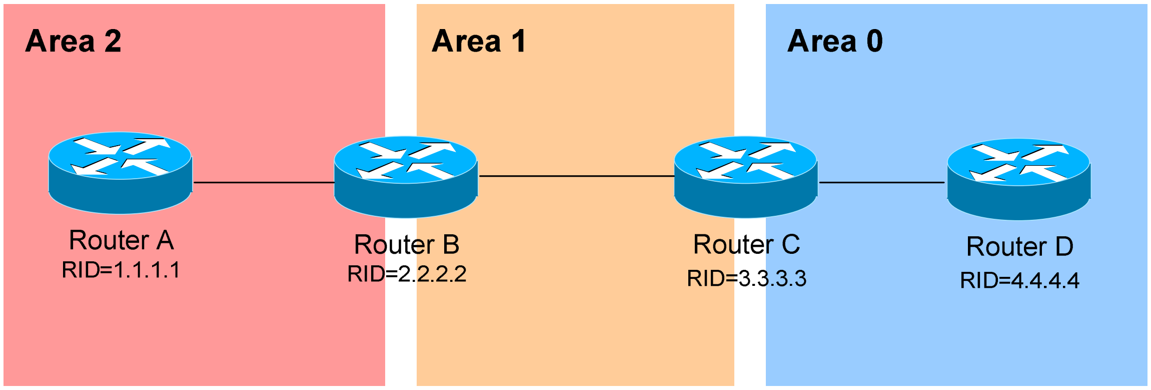
Notice the additional parameter *message-digest* included with the *area 0 authentication* command. Next, the hashed authentication key must be configured on the interface:

```
RouterA(config)# interface s0
RouterA(config-router)# ip ospf message-digest-key 10 md5 MYKEY
```

Area authentication must be enabled on all routers in the area, and the form of authentication must be identical (clear-text or MD5). The authentication keys do *not* need to be the same on every router in the OSPF area, but must be the same on interfaces connecting two neighbors.

Please note: if authentication is enabled for Area 0, the same authentication must be configured on Virtual Links, as they are "extensions" of Area 0.

OSPF Virtual Links



Earlier in this guide, it was stated that all areas must directly connect into Area 0, as a rule. In the above example, Area 2 has no direct connection to Area 0, but must transit through Area 1 to reach the backbone area. In normal OSPF operation, this shouldn't be possible.

There may be certain circumstances that may prevent an area from directly connecting into Area 0. **Virtual links** can be used as a workaround, to *logically* connect separated areas to Area 0. In the above example, a virtual link would essentially create a tunnel from Area 2 to Area 0, using Area 1 a **transit area**. One end of the Virtual Link *must* be connected to Area 0.

Configuration occurs on the **Area Border Routers (ABRs)** connecting Area 1 to Area 2 (Router B), and Area 1 to Area 0 (Router C). Configuration on Router B would be as follows:

```
RouterB(config)#  router ospf 1
RouterB(config-router)#  router-id 2.2.2.2
RouterB(config-router)#  area 1 virtual-link 3.3.3.3
```

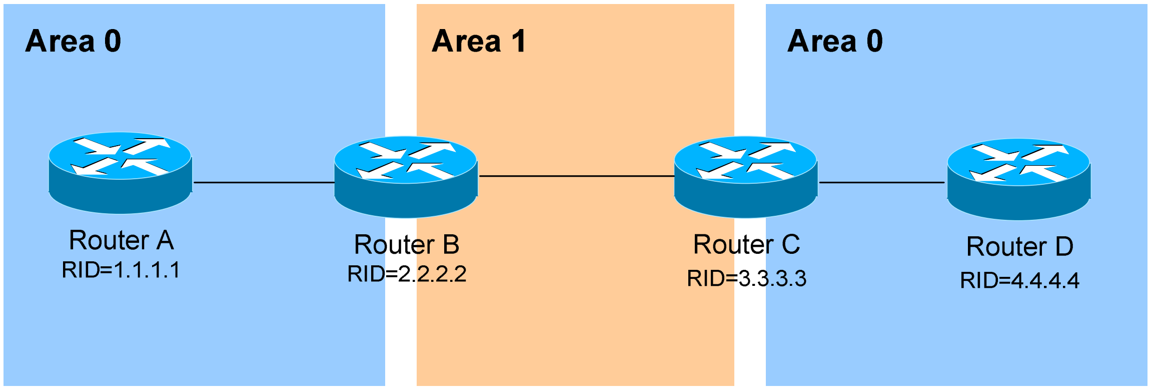
The first command enables the *ospf* process. The second command manually sets the *router-id* for Router B to *2.2.2.2*.

The third command actually creates the *virtual-link*. Notice that it specifies *area 1*, which is the **transit area**. Finally, the command points to the remote ABR's Router ID of *3.3.3.3*.

Configuration on Router C would be as follows:

```
RouterC(config)#  router ospf 1
RouterC(config-router)#  router-id 3.3.3.3
RouterC(config-router)#  area 1 virtual-link 2.2.2.2
```

OSPF Virtual Links (continued)



It is also possible to have two separated (or discontiguous) Area 0's. In order for OSPF to function properly, the two Area 0's must be connected using a **virtual link**.

Again, configuration occurs on the transit area's ABRs:

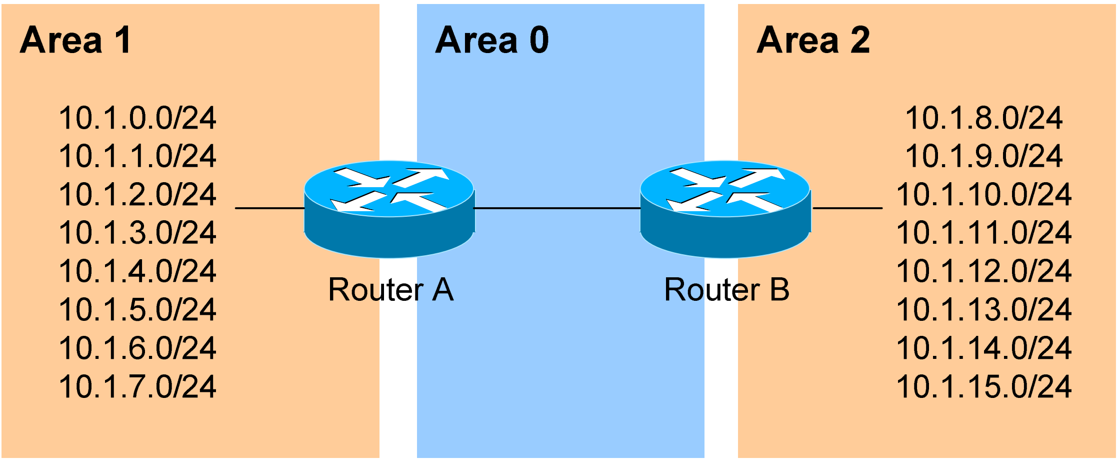
```
RouterB(config)#  router ospf 1
RouterB(config-router)#  router-id 2.2.2.2
RouterB(config-router)#  area 1 virtual-link 3.3.3.3

RouterC(config)#  router ospf 1
RouterC(config-router)#  router-id 3.3.3.3
RouterC(config-router)#  area 1 virtual-link 2.2.2.2
```

Always remember: the area specified in the *virtual-link* command is the **transit** area. Additionally, the transit area ***cannot*** be a stub area.

As stated earlier, if authentication is enabled for Area 0, the same authentication must be configured on Virtual Links, as they are "extensions" of Area 0:

```
RouterB(config)#  router ospf 1
RouterB(config-router)#  area 1 virtual-link 3.3.3.3 message-digest-key 1 md5 MYKEY

RouterC(config)#  router ospf 1
RouterC(config-router)#  area 1 virtual-link 2.2.2.2 message-digest-key 1 md5 MYKEY
```

Inter-Area OSPF Summarization



Consider the above example. OSPF is a classless routing protocol, thus all of the listed networks would be advertised individually. This increases the size of the topology databases and routing tables on routers in the domain, and may be undesirable. Advertising *only* a summary route for inter-area communication can reduce the load on router CPUs.

For example, all of the networks in Area 1 can be summarized as **10.1.0.0/21.** Similarly, all of the networks in Area 2 can be summarized as **10.1.8.0/21.**

Inter-area summarization is configured on **Area Border Routers (ABRs)**. Configuration on Router A would be as follows:

```
RouterA(config)#  router ospf 1
RouterA(config-router)#  network 10.1.0.0 0.0.7.255 area 1
RouterA(config-router)#  area 1 range 10.1.0.0 255.255.248.0
```

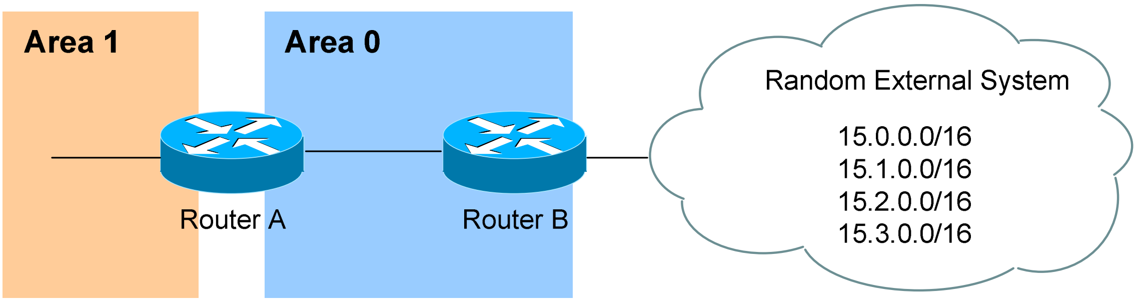
The *network* statement includes all of the 10.1.x.0 networks into Area 1. The *area 1 range* command creates a summary route for those networks, which is then advertised into Area 0, as opposed to each route individually.

Proper design dictates that a static route be created for the summarized network, pointing to the Null interface. This sends any traffic destined *specifically* to the summarized address to the bit-bucket in the sky, in order to prevent routing loops:

```
RouterA(config)#  ip route 10.1.0.0 255.255.248.0 null0
```

In IOS versions 12.1(6) and later, this static route is created automatically.

External OSPF Summarization



Consider the above example. Router B is an Autonomous System Border Router (ASBR). It is possible to redistribute the four "external" networks into the OSPF system. However, a separate route for each network will be advertised.

Again, this is wasteful. The four external networks can be summarized as **15.0.0.0/14**.

External Summarization is configured on ASBRs, and will only summarize external routes learned by route redistribution. Configuration on Router B would be as follows:

```
RouterB(config)# router ospf 1
RouterB(config-router)# summary-address 15.0.0.0 255.252.0.0
```

This summarized route is now propagated to all routers in every OSPF area.

Summarization can be used to filter certain routes (true route filtering is covered in a separate guide). To force OSPF to advertise the 15.0.0.0 and 15.1.0.0 networks as a summarized route, but *not* advertise the 15.2.0.0 and 15.3.0.0 prefixes:

```
RouterB(config)# router ospf 1
RouterB(config-router)# summary-address 15.0.0.0 255.254.0.0
RouterB(config-router)# summary-address 15.2.0.0 255.255.0.0 not-advertise
RouterB(config-router)# summary-address 15.3.0.0 255.255.0.0 not-advertise
```

The first *summary-address* command summarizes the 15.0.0.0/16 and 15.1.0.0/16 networks to 15.0.0.0/15, and advertises the summary as normal in the OSPF domain. The next two *summary-address* commands specifically reference the 15.2.0.0/16 and 15.3.0.0/16 networks, with the *not-advertise* parameter. As implied, these networks will *not* be advertised in OSPF.

OSPF Area Types

In order to control the propagation of LSAs in the OSPF domain, several area **types** were developed.

***Standard Area* –** A "normal" OSPF area.

- Routers within a standard area will share Router (Type 1) and Network (Type 2) LSAs to build their topology tables. Once fully synchronized, routers within an area will all have *identical* topology tables.
- Standard areas will accept Network Summary (Type 3) LSAs, which contain the routes to reach networks in all other areas.
- Standard areas will accept ASBR Summary (Type 4) and External (Type 5) LSAs, which contain the route to the ASBR and routes to external networks, respectively.

Configuration of standard areas is straight forward:

```
Router(config)#  router ospf 1
Router(config-router)#  network 10.1.0.0 0.0.7.255 area 1
```

***Stub Area* –** Prevents external routes from flooding into an area.

- Like Standard areas, Stub area routers will share Type 1 and Type 2 LSAs to build their topology tables.
- Stub areas will also accept Type 3 LSAs to reach other areas.
- Stub areas will ***not accept*** Type 4 or Type 5 LSAs, detailing routes to external networks.

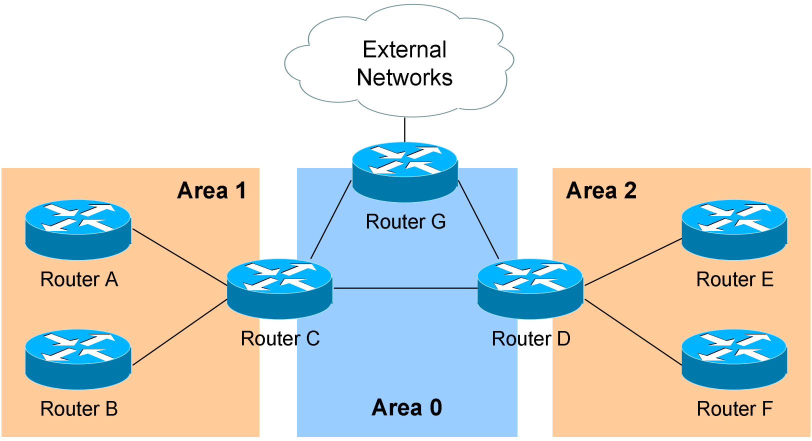
The purpose of Stub areas is to limit the number of LSAs flooded into the area, to conserve bandwidth and router CPUs. The Stub's ABR will *automatically* inject a **default route** into the Stub area, so that those routers can reach the external networks. The ABR will be the *next-hop* for the default route.

Configuration of stub areas is relatively simple:

```
Router(config)#  router ospf 1
Router(config-router)#  network 10.1.0.0 0.0.7.255 area 1
Router(config-router)#  area 1 stub
```

The *area 1 stub* command must be configured on **all** routers in the Stub area. No ASBRs are allowed in a Stub area.

OSPF Area Types (continued)



***Totally Stubby Area* –** Prevents both inter-area *and* external routes from flooding into an area.

- Like Standard and Stub areas, Totally Stubby area routers will share Type 1 and Type 2 LSAs to build their topology tables.
- Totally Stubby areas will ***not accept*** Type 3 LSAs to other areas.
- Totally Stubby areas will also ***not accept*** Type 4 or Type 5 LSAs, detailing routes to external networks.

Again, the purpose of Totally Stubby areas is to limit the number of LSAs flooded into the area, to conserve bandwidth and router CPUs. The Stub's ABR will instead *automatically* inject a **default route** into the Totally Stubby area, so that those routers can reach both inter-area networks and external networks. The ABR will be the *next-hop* for the default route.

Configuration of totally stubby areas is relatively simple:

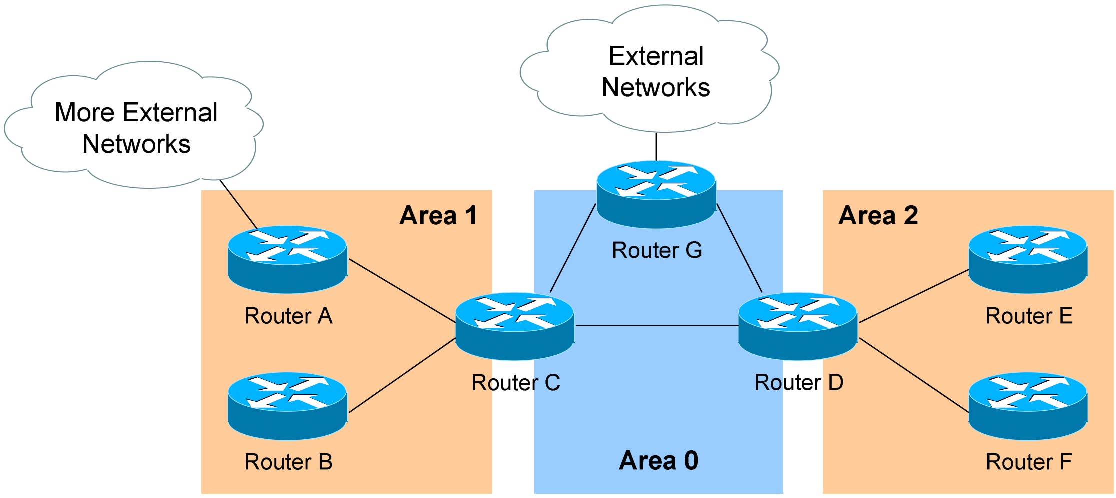
```
Router(config)#  router ospf 1
Router(config-router)#  network 10.1.0.0 0.0.7.255 area 1
Router(config-router)#  area 1 stub no-summary
```

The *area 1 stub no-summary* command is configured only on the **ABR** of the Totally Stubby area; other routers within the area are configured with the *area 1 stub* command. No ASBRs are allowed in a Totally Stubby area.

In the above example, if we were to configure Area 1 as a Totally Stubby area, it would not accept any external routes originating from the ASBR (Router G). It *also* would not accept any Type 3 LSAs containing route information about Area 0 and Area 2. Instead, Router C (the ABR) will inject a default route into Area 1, and all routers within Area 1 will use Router C as their gateway to all other networks.

* * *

OSPF Area Types (continued)



Not So Stubby Area (NSSA) **–** Similar to a Stub area; prevents external routes from flooding into an area, *unless* those external routes originated from an ASBR within the NSSA area.

- Like Standard and Stub areas, NSSA area routers will share Type 1 and Type 2 LSAs to build their topology tables.
- NSSA areas will also accept Network Summary (Type 3) LSAs, which contain the routes to reach networks in all other areas.
- NSSA areas will ***not accept*** Type 4 or Type 5 LSAs, detailing routes to external networks.
- If an ASBR exists *within* the NSSA area, that ASBR will generate **Type 7 LSAs**.

Again, NSSA areas are almost identical to Stub areas. If Area 1 was configured as an NSSA, it would not accept any external routes originating from Router G (an ASBR *outside* Area 1).

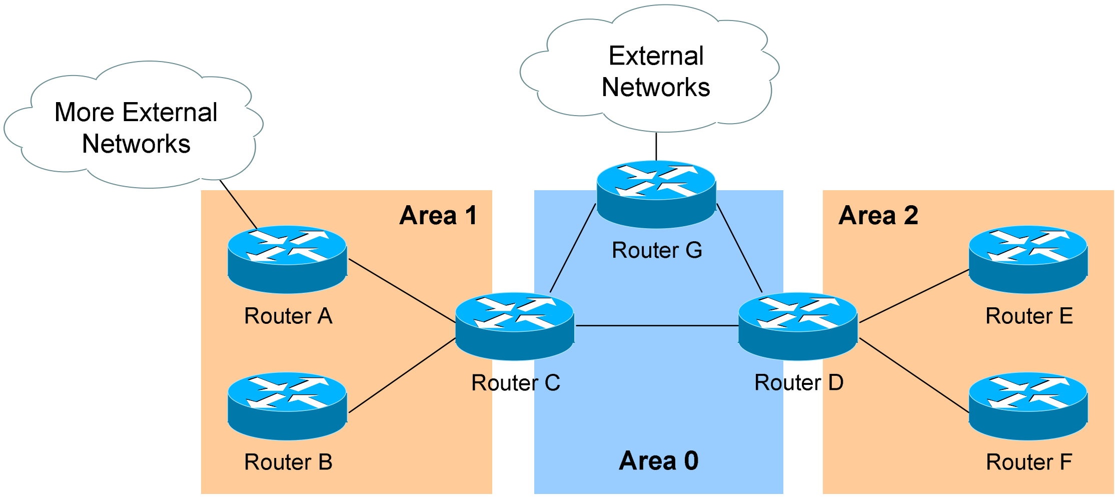
However, Area 1 also has an ASBR *within* the area (Router A). Those external routes will be flooded into Area 1 as **Type 7 LSAs**. These external routes *will not* be forwarded to other areas as Type 7 LSAs; instead, they will be converted into Type 5 LSAs by Area 1's ABR (Router C).

Configuration of NSSA areas is relatively simple:

```
Router(config)#  router ospf 1
Router(config-router)#  network 10.1.0.0 0.0.7.255 area 1
Router(config-router)#  area 1 nssa
```

The *area 1 nssa* command must be applied to **all** routers in the NSSA area.

OSPF Area Types (continued)



***Totally Not So Stubby Area (TNSSA)* –** Similar to a Totally Stubby area; prevents both inter-area *and* external routes from flooding into an area, *unless* those external routes originated from an ASBR within the NSSA area.

- Like Standard and Stub areas, TNSSA area routers will share Type 1 and Type 2 LSAs to build their topology tables.
- TNSSA areas will ***not accept*** Type 3 LSAs to other areas.
- TNSSA areas will ***not accept*** Type 4 or Type 5 LSAs, detailing routes to external networks.
- If an ASBR exists *within* the TNSSA area, that ASBR will generate **Type 7 LSAs**.

With the exception of not accepting inter-area routes, TNSSA areas are identical in function to NSSA areas.

Configuration of TNSSA areas is relatively simple:

```
Router(config)#  router ospf 1
Router(config-router)#  network 10.1.0.0 0.0.7.255 area 1
Router(config-router)#  area 1 nssa no-summary
```

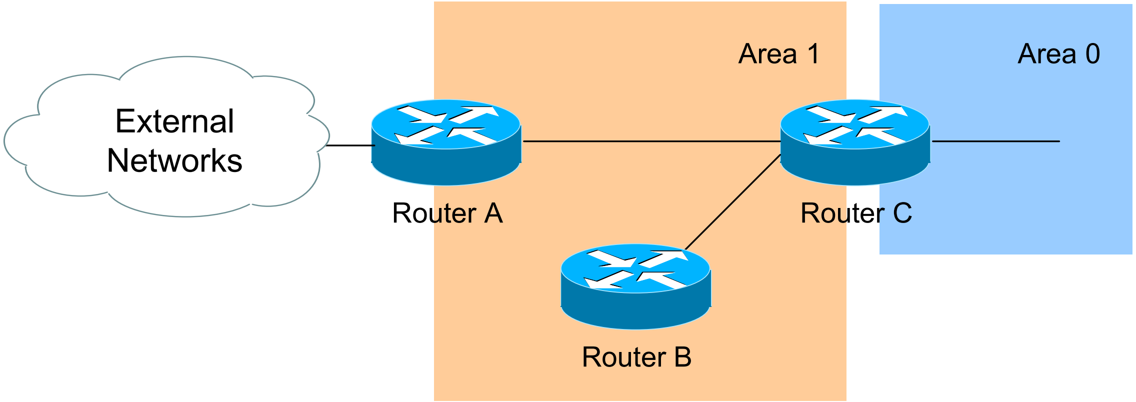
The *area 1 nssa no-summary* command is configured only on the **ABR** of the TNSSA area; other routers within the area are configured with the *area 1 nssa* command.

OSPF and Default Routes

We have learned about four types of OSPF areas:

- *Standard areas*
- *Stub areas*
- *Totally Stubby areas*
- *Not So Stubby areas (NSSA)*

The ABRs and ASBRs of **Standard areas** *do not* automatically generate (or inject) default routes into the area. Consider the following example:



Assume that Area 1 is configured as a Standard area. Router C will forward Type 3 LSAs from all other areas into Area 1, allowing Router A and Router B to reach inter-area networks.

Notice also that Router A is an ASBR, connecting to an external Autonomous System. Thus, Router A will generate Type 5 LSAs, detailing the routes to these external networks.

To additionally force Router A to generate a **default route** (indicating itself as the next hop) for the external networks, and inject this into Area 1. This default route will be advertised as a Type 5 LSA to all other areas:

```
RouterA(config)#  router ospf 1
RouterA(config-router)#  default-information originate
```

Router A *must* have a default route in its routing table in order for the above command to function. Router A's default route would point to some upstream router in the external Autonomous System.

If a default route *does not* exist in its routing table, Router A can still be forced to advertise a default route using the *always* parameter:

```
RouterA(config)#  router ospf 1
RouterA(config-router)#  default-information originate always
```

OSPF and Default Routes (continued)

The ABRs of **Stub** *and* **Totally Stubby** areas *automatically* generate (and inject) a **default route (0.0.0.0/0)** into the area. Routers in Stub areas use this default route to reach *external* networks, while routers in Totally Stubby areas use the default route to reach both *inter-area* and *external* networks.

To control the "cost" metric of the default route in Stub or Totally Stubby areas (configured on the ABR):

```
Router(config)#  router ospf 1
Router(config-router)#  area 1 stub
Router(config-router)#  area 1 default-cost 10
```

The ABRs of **NSSA areas** must be *manually configured* to generate (and inject) a default route into the area:

```
Router(config)#  router ospf 1
Router(config-router)#  area 1 nssa default-information-originate
```

Additionally, the ASBR of an NSSA area can generate and inject a default route. This default route will be advertised as a Type 7 LSA, as Type 5 LSA's are not allowed in NSSAs. The command is no different than injecting a default route from an NSSA ABR:

```
Router(config)#  router ospf 1
Router(config-router)#  area 1 nssa default-information-originate
```

Reference: (*http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a0080094a74.shtml*)

OSPF SPF Timers

To adjust the SPF timers in OSPF:

```
Router(config)#  router ospf 1
Router(config-router)#  timers spf 10 15
```

The *timers spf* command includes two parameters, measured in **seconds**. The first (*10*) indicates the SPF-Delay, or how long the OSPF should wait after receiving a topology change to recalculate the shortest path. The second (*15*) indicates the SPF-Holdtime, or how long OSPF should wait *in between* separate SPF calculations.

The *timers spf* command has actually become deprecated. It has been replaced with:

```
Router(config)#  router ospf 1
Router(config-router)#  timers throttle spf 5 10000 80000
```

The *timers throttle spf* command includes three parameters, measure in **milliseconds.** The first *(5)* indicates how long OSPF should wait after receiving a topology change to recalculate the shortest path. The second *(10000)* indicates the hold-down time, or how long OSPF should wait *in between* separate SPF calculations. If OSPF receives another topology change during the hold-time interval, it will continue to *double* the hold-time interval until it reaches the maximum hold-time *(80000*).

The purpose of the both SPF timer commands is to prevent OSPF from constantly converging, if the network links are “flapping.” The *timers spf* and *timers throttle spf* commands cannot be used together.

Advanced OSPF Configuration

To force the OSPF process to ignore OSPF Multicast (Type 6) LSAs:

```
Router(config)#  router ospf 1
Router(config-router)#  ignore lsa mospf
```

To force an interface to filter all outgoing OSPF LSA’s:

```
Router(config)#  interface e0
Router(config-if)#  ip ospf database-filter all out
```

Loopback interfaces are treated differently than other interfaces, when advertised in OSPF. OSPF will advertise a loopback interface as a specific “host” route (with a mask of /32 or 255.255.255.255). To force OSPF to advertise a loopback interface with its proper subnet mask:

```
Router(config)#  interface loopback0
Router(config-if)#  ip address 10.50.5.1 255.255.255.0
Router(config-if)#  ip ospf network point-to-point
```

Troubleshooting OSPF

To view the OSPF Neighbor Table:

```
Router#  show ip ospf neighbor

Neighbor ID   Pri    State         Dead Time    Address       Interface
7.7.7.7       1      FULL/  -      00:00:36     150.50.17.2   Serial0
6.6.6.6       1      FULL/DR       00:00:11     150.50.18.1   Ethernet0
```

The Neighbor Table provides the following information about each neighbor:

- The ***Router ID*** of the remote neighbor.
- The OSPF ***priority*** of the remote neighbor (used for DR/BDR elections).
- The current neighbor ***state***.
- The ***dead interval*** timer.
- The connecting ***IP address*** of the remote neighbor.
- The local ***interface*** connecting to the remote neighbor.

To view the OSPF topology table:

```
Router#  show ip ospf database

OSPF Router with ID (9.9.9.9) (Process ID 10)

Router Link States (Area 0)

Link ID       ADV Router   Age     Seq#         Checksum   Link count
7.7.7.7       7.7.7.7      329     0x80000007   0x42A0     2
8.8.8.8       8.8.8.8      291     0x80000007   0x9FFC     1

Summary Net Link States (Area 0)

Link ID       ADV Router   Age     Seq#         Checksum
192.168.12.0  7.7.7.7      103     0x80000005   0x13E4
192.168.34.0  7.7.7.7      105     0x80000003   0x345A
```

The Topology Table provides the following information:

- The actual ***link*** (or *route*).
- The ***advertising*** Router ID.
- The link-state ***age*** timer.
- The ***sequence number*** and ***checksum*** for each entry.

(Reference: http://www.cisco.com/en/US/products/sw/iosswrel/ps5187/products_command_reference_chapter09186a008017d02e.html)

Troubleshooting OSPF (continued)

To view the specific information about an OSPF process:

```
Router#  show ip ospf 1

Routing Process "ospf 1" with ID 9.9.9.9
 Supports only single TOS(TOS0) routes
 Supports opaque LSA
 SPF schedule delay 5 secs, Hold time between two SPFs 10 secs
 Minimum LSA interval 5 secs. Minimum LSA arrival 1 secs
 Number of external LSA 0. Checksum Sum 0x0
 Number of opaque AS LSA 0. Checksum Sum 0x0
 Number of DCbitless external and opaque AS LSA 0
 Number of DoNotAge external and opaque AS LSA 0
 Number of areas in this router is 1. 1 normal 0 stub 0 nssa
 External flood list length 0
    Area BACKBONE(0)
        Number of interfaces in this area is 1
        Area has no authentication
        SPF algorithm executed 3 times
        Area ranges are
        Number of LSA 2. Checksum Sum 0xDDEC
        Number of opaque link LSA 0. Checksum Sum 0x0
        Number of DCbitless LSA 0
        Number of indication LSA 0
        Number of DoNotAge LSA 0
        Flood list length 0
```

The *show ip ospf* command provides the following information:

- The local ***Router ID.***
- ***SPF Scheduling*** information, and various ***SPF timers.***
- The number of ***interfaces*** in specific ***areas***, including the ***type*** of area.
- The link-state ***age*** timer.
- The ***sequence number*** and ***checksum*** for each entry.

Troubleshooting OSPF (continued)

To view OSPF-specific information on an interface:

```
Router#  show ip ospf interface s0
```

```
Serial0 is up, line protocol is up
  Internet Address 192.168.79.2/24, Area 0
  Process ID 10, Router ID 9.9.9.9, Network Type POINT_TO_POINT, Cost: 64
  Transmit Delay is 1 sec, State POINT_TO_POINT,
  Timer intervals configured, Hello 10, Dead 40, Wait 40, Retransmit 5
    Hello due in 00:00:04
  Index 1/1, flood queue length 0
  Next 0x0(0)/0x0(0)
  Last flood scan length is 1, maximum is 1
  Last flood scan time is 0 msec, maximum is 0 msec
  Neighbor Count is 1, Adjacent neighbor count is 1
    Adjacent with neighbor 7.7.7.7
  Suppress hello for 0 neighbor(s)
```

The *show ip ospf interface* command provides the following information:

- The local ***Router ID***.
- The interface ***network type***.
- The OSPF ***cost*** for the interface.
- The interface ***Hello*** and ***Dead*** timers.
- A list of neighbor ***adjacencies***.

To view routing protocol specific information for OSPF:

```
Router#  show ip protocols
```

```
Routing Protocol is “ospf 10"
  Invalid after 0 seconds, hold down 0, flushed after 0
  Outgoing update filter list for all interfaces is
  Incoming update filter list for all interfaces is
  Routing for Networks:
    192.168.79.0  0.0.0.255 area 0
    192.168.109.0 0.0.0.255 area 0
  Routing Information Sources:
    Gateway         Distance      Last Update
    7.7.7.7          110            00:01:05
  Distance: (default is 110)
```

The *show ip protocols* command provides the following information:

- Locally originated ***networks*** that are being advertised.
- Neighboring ***sources*** for routing information
- The ***administrative distance*** of neighboring sources.

Troubleshooting OSPF (continued)

To reset an OSPF process, including neighbor adjacencies:

```
Router#  clear ip ospf process
```

To display information about OSPF virtual-links:

```
Router#  show ip ospf virtual-links
```

To display routes to both ABRs and ASBRs:

```
Router#  show ip ospf border-routers
```

To debug OSPF in realtime:

```
Router#  debug ip ospf adj
Router#  debug ip ospf events
Router#  debug ip ospf hello
```

Section 13
- IS-IS -

IS-IS Fundamentals

IS-IS (**Intermediate System -to- Intermediate System)** is a standardized link-state protocol that was developed to be the definitive routing protocol for the **OSI (Open Systems Interconnect) Model**, which was developed by **ISO (International Standards Organization**). IS-IS shares many similarities to OSPF. Though it was designed as an interior gateway protocol (IGP), IS-IS is predominantly used by ISPs, due to its scalability.

IS-IS adheres to the following Link State characteristics:

- IS-IS allows for a hierarchical network design using **Areas.**
- IS-IS will form **neighbor** relationships with adjacent routers of the same IS-IS type.
- Instead of advertising the distance to connected networks, IS-IS advertises the status of directly connected **"links"** in the form of **Link-State Packets (LSPs**). IS-IS will only send out updates when there is a change to one of its links, and will *only* send the change in the update.
- IS-IS uses the **Dijkstra Shortest Path First** algorithm to determine the shortest path.
- IS-IS is a classless protocol, and thus supports VLSMs.

Other characteristics of IS-IS include:

- IS-IS was originally developed to route the ISO address space, and thus is *not* limited to IP routing.
- IS-IS routes have an administrative distance is **115**.
- IS-IS uses an arbitrary **cost** for its metric. IS-IS additionally has three optional metrics: **delay, expense,** and **error**. Cisco does not support these optional metrics.
- IS-IS has no hop-count limit.

The IS-IS process builds and maintains three separate tables:

- A **neighbor table** – contains a list of all neighboring routers.
- A **topology table** – contains a list of *all* possible routes to all known networks within an area.
- A **routing table** – contains the *best* route for each known network.

IS-IS Protocols and Addressing

IS-IS consists of three sub-protocols that work in tandem to achieve end-to-end routing which ISO defined as **Connectionless Network Service (CLNS)**:

- **CLNP (Connectionless Network Protocol) –** serves as the Layer-3 protocol for IS-IS (and was developed by ISO).
- **ES-IS** (**End System -to- Intermediate System)** – used to route between *hosts* and *routers*.
- **IS**-**IS (Intermediate System -to- Intermediate System)** – used to route between *routers*.

IS-IS was originally developed to route ISO CLNP addresses (outlined in RFC 1142). However, CLNP addressing never became prominently used. Thus, IS-IS was modified to additionally support IP routing, and became **Integrated** (or **Dual) IS-IS** (outlined in RFC 1195).

The IS-IS CLNP address is **hexadecimal** and of **variable length**, and can range from **64** to **160 bits** in length. The CLNP address contains three "sections," including:

- **Area** field – (variable length)
- **ID** field – (from 8 to 64 bits, though usually 48 bits)
- **Selector (SEL)** field - (8 bits)

Thus, the CLNP address identifies the "Area" in which a device is located, the actual host "ID," and the destination application on that host, in the form of the "SEL" field. The CNLP address is logically segmented even further, as demonstrated by the following table:

IDP		**DSP**		
AFI	*IDI*	*HO-DSP*	*System-ID*	*NSEL*
Area Field			**ID Field**	**SEL Field**

Observe the top row of the above figure. The ISO CLNP address provides granular control by separating *internal* and *external* routing information:

- The **IDP (Initial Domain Part)** portion of the address identifies the Autonomous System of the device (and is used to route *to* or *between* Autonomous Systems)
- The **DSP (Domain Specific Part)** portion of the address is used to route *within* the autonomous system.

IS-IS Protocols and Addressing (continued)

IDP		**DSP**		
AFI	*IDI*	*HO-DSP*	*System-ID*	*NSEL*
Area Field			**ID Field**	**SEL Field**

The IDP portion of the address is separated into two "sections," including:

- **AFI (Authority and Format Identifier) –** specifies the organization authorized to assign addresses, and the format and length of the rest of the CLNP address. The AFI is always **8 bits**.
- **IDI (Initial Domain Identifier) –** identifies the "sub-organization" under the parent AFI organization. The length of the IDI is dependent on the chosen AFI.

An AFI of 0x**49** indicates a *private* CLNP address, which cannot be routed globally (the equivalent of an IPv4 private address). An AFI of 0x**47** is commonly used for global IS-IS networks, with the IDI section identifying specific organizations.

The AFI plus the IDI essentially identify the *autonomous system* of the address. However, this is not the equivalent of a BGP AS number, nor is it compatible with BGP as an exterior routing protocol.

The DSP portion of the address is separated into three "sections," including:

- **HO-DSP (High Order DSP) –** identifies the *area* within an autonomous system
- **System ID –** identifies the specific host. Usually **48 bits** (or **6 octets)** in length, to accommodate MAC addresses
- **NSEL –** identifies the destination upper layer protocol of the host (always **8 bits**)

Two "types" of CLNP addresses are defined:

- **NET address –** does not contain upper-layer information (in other words, the SEL field is always set to 0x00)
- **NSAP address –** the "full" CLNP address, with populated Area, ID, and SEL fields.

Please note: A NET address is simply an NSAP address with a zero value in the SEL field.

CLNS Address Example

The following is an example of a full ISO CLNS address:

47.1234.5678.9abc.def0.0001.1111.2222.3333.00

Correlating the above address to the appropriate fields:

IDP		*DSP*		
AFI	*IDI*	*HO-DSP*	*System-ID*	*NSEL*
47.	**1234.5678.9abc.def0.**	**0001.**	**1111.2222.3333.**	**00**
Area Field			*ID Field*	*SEL Field*

The *System-ID* is usually populated by the device's MAC address or IP v4 address.

Recall that CLNS addresses are of variable length. We can specify addresses without an *IDI* field:

47.0001.1111.2222.3333.00

Thus, the above address contains an *AFI* (Autonomous System), *HO-DSP* (Area), *System-ID* (in this example, a MAC Address), and the *NSEL (*SEL). Because the SEL field has a zero value (0x00), the above address is defined as a NET address, and not an NSAP address.

ISO CLNS addresses are not applied on an interface-by-interface basis. Instead, a single CLNS address is applied to the entire *device.*

Even if Integrated IS-IS is being used (thus indicating that IPv4 is being routed instead of CLNS), a CLNS address ***is still required*** on the IS-IS router. This is configured under the IS-IS router process.

Routers within the same area must share identical *AFI, IDI,* and *HO-DSP* values, but each must have a unique System-ID

IS-IS Packet Types

IS-IS defines two categories of network devices:

- **ES (End System)** – identifies an *end* host.
- **IS (Intermediate System)** – identifies a Layer 3 router.

IS-IS additionally defines four categories of packet types:

- **Hello**
- **LSP**
- **CSNP**
- **PSNP**

Hello packets are exchanged for neighbor discovery. Three types of IS-IS Hello packets exist:

- **IIH (IS-IS Hello)** – exchanged between routers (or IS's) to form neighbor adjacencies.
- **ESH (ES Hello)** – sent from an ES to discover a router.
- **ISH (IS Hello)** – sent from an IS to announce its presence to ES's

An **LSP (Link State Packet)** is used to share topology information between routers. There are separate LSPs for Level 1 and Level 2 routing. LSP's are covered in great detail shortly.

A **CSNP (Complete Sequence Number PDU)** is an update containing the *full* link-state database. IS-IS routers will refresh the full database every **15 minutes.**

A **PSNP (Partial Sequence Number PDU)** is used by IS-IS routers to both *request* and *acknowledge* a link-state update.

IS-IS Neighbors

IS-IS routers form neighbor relationships, called **adjacencies,** by exchanging **Hello** packets (often referred to as IS-IS Hellos or **IIH**'s). Hello packets are sent out every **10 seconds,** regardless of media type. Only after an adjacency is formed can routers share routing information.

IS-IS supports three IIH packet formats; one for **point-to-point** links, and two for **broadcast** (or **LAN**) links (Level-1 and Level-2 broadcast Hellos).

Unlike OSPF, IS-IS neighbors *do not* have to share a common IP subnet to form an adjacency. Adjacencies are formed across *CLNP* connections, not *IP* connections, even when using Integrated IS-IS. Thus, IS-IS actually requires *no* IP connectivity between its routers to route IP traffic!

There are two types of adjacencies:

- **Level-1 adjacency** – for routing *within* an area (intra-area routing)
- **Level-2 adjacency –** for routing *between* areas (intra-area routing)

IS-IS routers *must* share a common physical link to become neighbors, and the *System-ID* must be **unique** on each router. Additionally, the following parameters must be identical on each router:

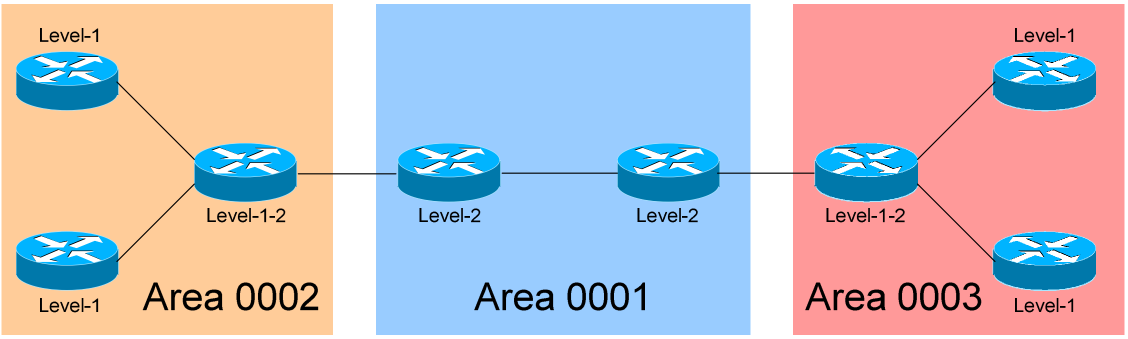
- Hello packet format (point-to-point or broadcast)
- Hello timers
- Router "level" (explained shortly)
- Area (only for Level-1 adjacencies)
- Authentication parameters (Cisco devices currently support only **clear-text authentication** for IS-IS).
- MTU

Neighbors will elect a **DIS (Designated Intermediate System)** on broadcast links. A DIS is the equivalent of an OSPF DR (Designated Router). Unlike OSPF, however, there is no Backup DIS, and thus a new election will occur immediately if the DIS fails. Additionally, the DIS election is *preemptive.*

Whichever IS-IS router has the **highest priority** will be elected the DIS (default priority is **64**). In the event of a tie, whichever IS-IS router has the **highest SNPA** (usually MAC) **address** will become the DIS. The DIS sends out hello packets every **3.3 seconds**, instead of every 10 seconds.

(Reference: http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cgcr/fipr_c/ipcprt2/1cfisis.pdf)

The IS-IS Hierarchy



IS-IS defines three types of IS-IS routers:

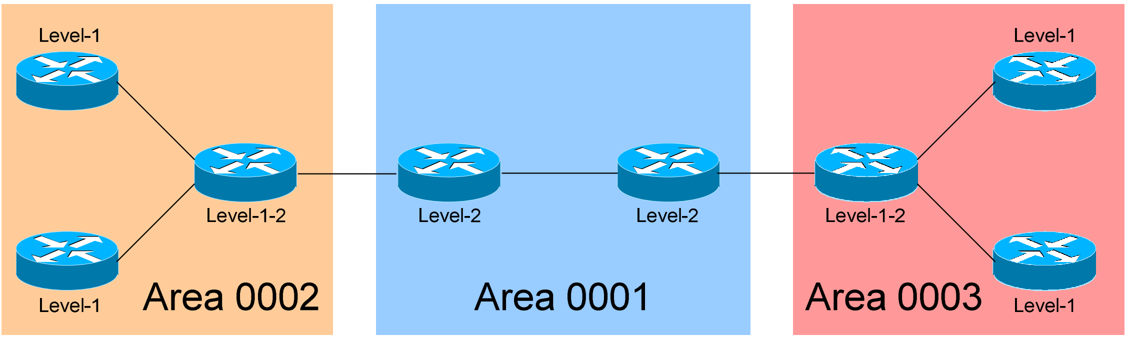
- **Level-1 Router –** contained within a single area, with a topology table limited to only its local area (called the **Level-1 Database**)
- **Level-2 Router -** a *backbone* router that routes *between* areas, and builds a **Level-2 Database.**
- **Level-1-2 Router** – similar to an *area border router*. Interfaces between a local area and the *backbone* area, and builds both a **Level-1** *and* a **Level-2** database.

Each type of IS-IS router will form only specific adjacencies:

- Level-1 routers form **Level-1 adjacencies** with other Level-1 routers *and* Level-1-2 routers.
- Level-2 routers form **Level-2 adjacencies** with other Level-2 routers *and* Level-1-2 routers.
- Level-1-2 routers form *both* **Level-1** and **Level-2 adjacencies** with other Level-1-2 routers.
- Level-1 routers ***will never*** form adjacencies with Level-2 routers.

The IS-IS backbone consists of multiple contiguous Level-2 routers, each of which can exist in a separate area.

The IS-IS Hierarchy (continued)



Neighbors build their topology tables by sharing **LSP's (Link-State Packets**), which are roughly the equivalent of OSPF LSA's. Depending on the type of adjacency, a router will send out either a Level-1 or Level-2 LSP.

Level-1 routers share Level-1 LSP's, and will build a Level-1 topology table consisting of *solely* its own area (thus forming the equivalent of an OSPF Totally Stubby area). If a Level-1 router has a packet destined for the *local* area, it simply routes the packet to the System ID by using the local topology table (Level-1 database).

If a Level-1 router has a packet destined for a *remote* area, it forwards it to the nearest Level-1-2 router. Level-1-2 routers set an **Attach (ATT) bit** in their Level-1 LSP's, informing other Level-1 routers that they are attached to another area.

Level-2 routers share Level-2 LSP's, and will build a Level-2 topology table, which contains a list of reachable areas across the IS-IS domain.

Level-1-2 routers will share *both* Level-1 *and* Level-2 LSP's with its appropriate adjacencies. Level-1-2 routers maintain separate Level-1 and Level-2 topology tables.

Level-1 routes (locally originated) are *always* **preferred** over Level-2 routes (originated from another area).

IS-IS routers will refresh the Link-State topology table every **15 minutes** (as opposed to every 30 minutes for OSPF).

(Reference: http://www.cisco.com/warp/public/cc/pd/iosw/prodlit/insys_wp.htm)

Basic IS-IS Configuration

To configure IS-IS, the IS-IS process must first be established:

```
Router(config)#  router isis
```

The router must then be configured with a CLNP address:

```
Router(config)#  router isis
Router(config-router)#  net 49.0001.1921.6800.5005.00
```

To globally dictate the router-type of all interfaces (default is **level-1-2**):

```
Router(config)#  router isis
Router(config-router)#  is-type level-1
Router(config-router)#  is-type level-1-2
Router(config-router)#  is-type level-2
```

Finally, IS-IS must be explicitly enabled on the interface:

```
Router(config)#  interface fa0/0
Router(config-if)#  ip router isis
```

This not only allows IS-IS to form neighbor relationships out of this interface, it also adds the interface's network to the routing table.

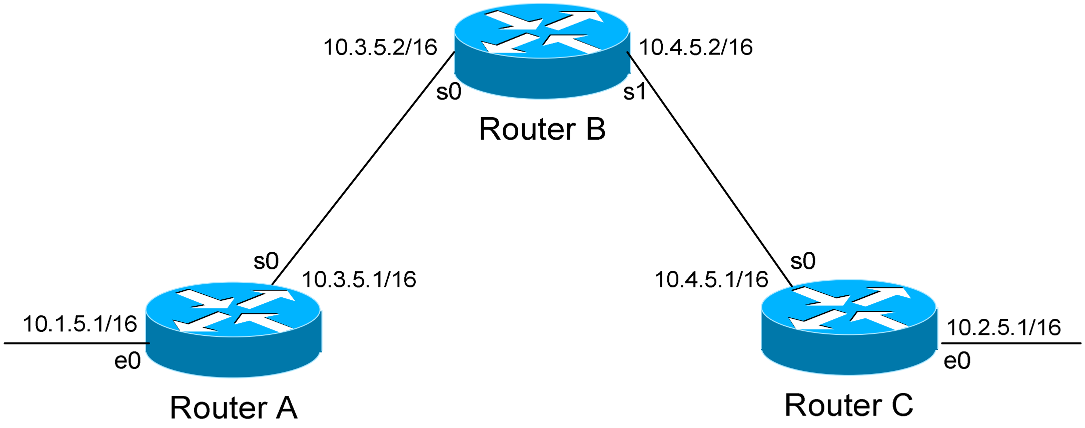
The globally configured router-type can be overridden on each individual interface:

```
Router(config)#  interface fa0/0
Router(config-if)#  isis circuit-type level-1
Router(config-if)#  isis circuit-type level-1-2
Router(config-if)#  isis circuit-type level-2
```

To adjust the priority (default is **64**) of interface, increasing the likelihood that the router will be elected the DIS:

```
Router(config)#  interface e0/0
Router(config-if)#  isis priority 100
```

IS-IS Passive-Interfaces



It is possible to control which router interfaces will participate in the IS-IS process. Just as with EIGRP and OSPF, we can use the *passive-interface* command.

However, please note that the *passive-interface* command works differently with IS-IS than with RIP or IGRP. IS-IS will no longer form neighbor relationships out of a "passive" interface, thus this command prevents updates from being *sent* or *received* out of this interface:

```
RouterC(config)#  router isis
RouterC(config-router)#  passive-interface s0
```

Router C will not form a neighbor adjacency with Router B.

We can configure **all** interfaces to be passive using the *passive-interface default* command, and then individually use the *no passive-interface* command on the interfaces we **do** want neighbors to be formed on:

```
RouterC(config)#  router isis
RouterC(config-router)#  passive-interface default
RouterC(config-router)#  no passive-interface e0
```

Always remember, that the *passive-interface* command will prevent IS-IS (and OSPF) from forming neighbor relationships out of that interface. **No** routing updates are passed in either direction.

However, unlike OSPF, using the *passive-interface* command will still inject that interface's network into the routing table. Thus, the *passive-interface* command can be useful when creating "stub" networks.

The IS-IS Metric

IS-IS utilizes an arbitrary **cost** for its metric (the optional metrics of **delay, expense**, and **error** are not supported by Cisco). By default, interfaces of all types (regardless of speed) are assigned a metric of **10**.

To adjust the metric on an interface:

```
Router(config)#  interface e0/0
Router(config-if)#  isis metric 30
```

IS-IS Authentication

IS-IS authentication can be applied to a **link**, to an **area**, or to a **domain**. Remember, Cisco supports only clear-text authentication for IS-IS.

To configuration authentication on an interface-by-interface basis:

```
Router(config)#  interface fa0/0
Router(config-if)#  isis password MYPASSWORD level-1
Router(config-if)#  isis password MYPASSWORD2 level-2
```

Note that separate authentication passwords can be applied to Level-1 or Level-2 Adjacencies. To configure authentication for an entire IS-IS area:

```
Router(config)#  router isis
Router(config-router)#  area-password MYPASSWORD
```

IS-IS Summarization

IS-IS supports both **inter-area** and **external** summarization, and uses the same command to accomplish both. If we wished to summarize the following networks into one summary route:

- 172.16.0.0/16
- 172.17.0.0/16
- 172.18.0.0/16
- 172.19.0.0/16
- 172.20.0.0/16
- 172.21.0.0/16
- 172.22.0.0/16
- 172.23.0.0/16

The following command would be required:

```
RouterC(config)#  router isis
RouterC(config-router)#  summary-address 172.16.0.0 255.248.0.0
```

IS-IS and WAN Technologies

When configuring IS-IS over Frame-Relay, additional map statements are required:

```
Router(config)#  interface s0/0
Router(config-if)#  frame-relay map clns 105 broadcast
Router(config-if)#  frame-relay map clns 106 broadcast
```

Additionally, we can map CLNP addresses in ISDN:

```
Router(config)#  interface bri0
Router(config-if)#  dialer map clns 49.0001.1921.6800.5005.00 name MYNAME broadcast 3331111
```

IS-IS Troubleshooting

To view any CLNS neighbors, including the type of adjacency:

```
Router#  show clns neighbors
```

To view only IS neighbors:

```
Router#  show clns is-neighbors
```

To view specific IS-IS information about an interface:

```
Router#  show clns interface e0/0
```

To view the IS-IS link-state topology table:

```
Router#  show isis database
```

To view a list of all known IS-IS routers in all areas:

```
Router#  show isis topology
```

(Reference: http://www.cisco.com/en/US/products/sw/iosswrel/ps5187/products_command_reference_chapter09186a008017d02e.html)

IS-IS vs. OSPF

IS-IS is often compared and contrasted to OSPF. Both protocols share several similarities, including:

- Both are *Link-State* routing protocols.
- Both use the *Dijkstra* algorithm to determine the shortest path.
- Both are *classless* and support *VLSMs*.
- Both use a *cost* metric.
- Both use *areas* to minimize the size of topology and routing tables.
- Both elect a *designated router* on broadcast links to contain link-state update traffic.

Despite these similarities, there are a multitude of crucial differences between IS-IS and OSPF, including:

- OSPF supports *only* IP, IS-IS supports *both* IP and CLNS.
- IS-IS *does not* require IP connectivity between routers to share routing information. Updates are sent via CLNS instead of IP.
- In OSPF, *interfaces* belong to areas. In IS-IS, the *entire router* belongs to an area.
- An IS-IS router belongs to only one Level-2 area, which results in less LSP traffic. IS-IS is thus more efficient and scalable than OSPF, and supports more routers per area.
- There is no Area 0 backbone area for IS-IS. The IS-IS backbone is a contiguous group of Level 1-2 and Level 2 routers.
- IS-IS *does not* elect a backup DIS. Additionally, DIS election is preemptive.
- On broadcast networks, even with an elected DIS, IS-IS routers still form adjacencies with *all* other routers. In OSPF, routers will only form adjacencies with the DR and BDR on broadcast links.
- IS-IS uses an arbitrary cost metric. OSPF's cost metric is based on the bandwidth of the link.
- IS-IS provides far more granular control of link-state and SPF timers than OSPF.

(Reference: http://geocities.com/mnvbhatia/draft-bhatia-manral-diff-isis-ospf-00.txt, http://www.ciscopress.com/articles/article.asp?p=31319&rl=1)

Section 14
- Border Gateway Protocol -

Border Gateway Protocol (BGP)

BGP is a standardized *exterior* gateway protocol (EGP), as opposed to RIP, OSPF, and EIGRP which are *interior* gateway protocols (IGP's). BGP Version 4 (**BGPv4)** is the current standard deployment.

BGP is considered a "Path Vector" routing protocol. BGP was not built to route *within* an Autonomous System (AS), but rather to route *between* AS's. BGP maintains a **separate routing table** based on shortest **AS Path** and various other attributes, as opposed to IGP metrics like distance or cost.

BGP is the routing protocol of choice on the Internet. Essentially, the Internet is a collection of interconnected Autonomous Systems.

BGP Autonomous Systems are assigned an Autonomous System Number (ASN), which is a 16-bit number ranging from **1 – 65535**. A specific subset of this range, **64512 – 65535**, has been reserved for private (or internal) use.

BGP utilizes **TCP** for reliable transfer of its packets, on **port 179**.

When to Use BGP

Contrary to popular opinion, BGP is not a necessity when multiple connections to the Internet are required. Fault tolerance or redundancy of outbound traffic can easily be handled by an IGP, such as OSPF or EIGRP.

BGP is also completely unnecessary if there is only one connection to an external AS (such as the Internet). There are over 100,000 routes on the Internet, and interior routers should not be needlessly burdened.

BGP should be used under the following circumstances:

- Multiple connections exist to external AS's (such as the Internet) via different providers.
- Multiple connections exist to external AS's through the same provider, but connect via a separate CO or routing policy.
- The existing routing equipment can handle the additional demands.

BGP's true benefit is in controlling how traffic *enters* the local AS, rather than how traffic *exits* it.

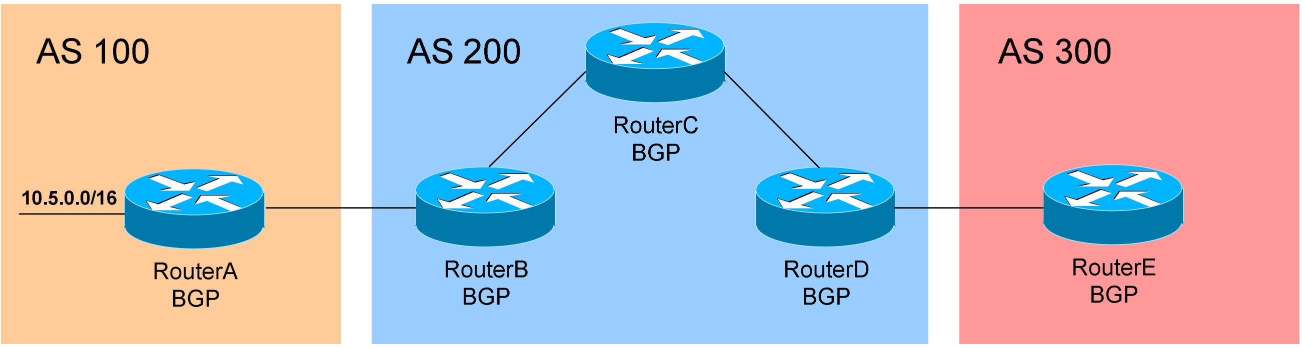
BGP Peers (Neighbors)

For BGP to function, BGP routers (called **speakers)** must form neighbor relationships (called **peers**).

There are two types of BGP neighbor relationships:

- **iBGP Peers** – BGP neighbors within the same autonomous system.
- **eBGP Peers** – BGP neighbors connecting separate autonomous systems.

Note: Do not confuse an *IGP*, such as OSPF, with *iBGP*!



In the above figure, RouterB and RouterC in AS 200 would form an **iBGP** peer relationship. RouterA in AS 100 and RouterB in AS 200 would form an **eBGP** peering.

Once BGP peers form a neighbor relationship, they share their full routing table. Afterwards, only changes to the routing table are forwarded to peers.

By default, BGP assumes that eBGP peers are a maximum of one hop away. This restriction can be bypassed using the *ebgp-multihop* option with the *neighbor* command (demonstrated later in this guide).

iBGP peers do not have a hop restriction, and are dependent on the underlying IGP of the AS to connect peers together. By default, all iBGP peers must be **fully meshed** within the Autonomous System.

A Cisco router running BGP can belong to **only one AS**. The IOS will only allow one BGP process to run on a router.

The Administrative Distance for routes learned outside the Autonomous System (eBGP routes) is **20**, while the AD for iBGP and locally-originated routes is **200**.

BGP Peers Messages

BGP forms its peer relationships through a series of **messages**. First, an **OPEN** message is sent between peers to initiate the session. The **OPEN** message contains several parameters:

- BGP Version – must be the same between BGP peers
- Local AS Number
- BGP Router ID

KEEPALIVE messages are sent periodically (every **60 seconds** by default) to ensure that the remote peer is still available. If a router does not receive a KEEPALIVE from a peer for a Hold-time period (by default, **180 seconds**), the router declares that peer dead.

UPDATE messages are used to exchange routes between peers.

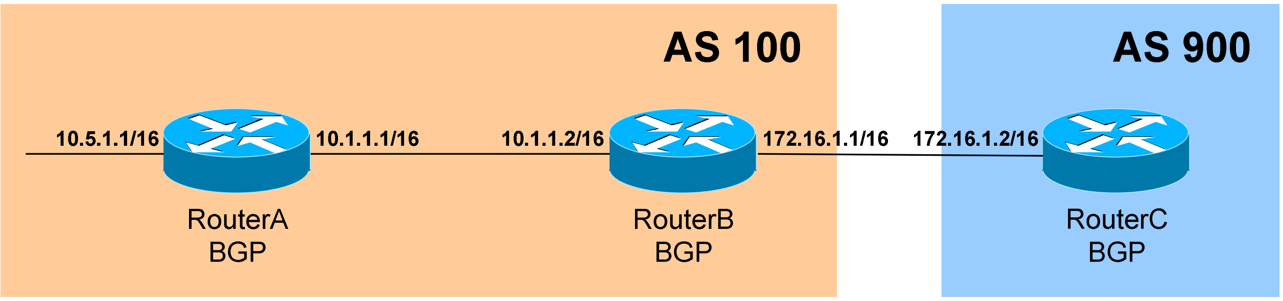
Finally, **NOTIFICATION** messages are sent when there is a fatal error condition. If a NOTIFICATION message is sent, the BGP peer session is torn down and reset.

As a BGP peer session is forming, it will pass through several **states**. This process is known as the BGP **Finite-State Machine (FSM)**:

- **Idle** – the initial BGP state
- **Connect** - BGP waits for a TCP connection with the remote peer. If successful, an OPEN message is sent. If unsuccessful, the session is placed in an Active state.
- **Active** – BGP attempts to initiate a TCP connection with the remote peer. If successful, an OPEN message is sent. If unsuccessful, BGP will wait for a ConnectRetry timer to expire, and place the session back in a Connect State.
- **OpenSent** – BGP has both established the TCP connection *and* sent an OPEN Message, and is awaiting a reply OPEN Message. Once it receives a reply OPEN Message, the BGP peer will send a KEEPALIVE message.
- **OpenConfirm –** BGP listens for a reply KEEPALIVE message.
- **Established –** the BGP peer session is fully established. UPDATE messages containing routing information will now be sent.

If a peer session is stuck in an **Active** state, potential problems can include: no IP connectivity (no route to host), an incorrect *neighbor* statement, or an access-list filtering TCP port 179.

Configuring BGP Neighbors



The first step in configuring BGP is to enable the BGP process, and specify the router's Autonomous System (AS):

```
RouterB(config)#  router bgp 100
```

RouterB is now a member of AS *100*. Next, neighbor relationships must be established. To configure a neighbor relationship with a router in the *same* AS (iBGP Peer):

```
RouterB(config)#  router bgp 100
RouterB(config-router)#  neighbor 10.1.1.1 remote-as 100
```

To configure a neighbor relationship with a router in a *separate* AS (eBGP Peer):

```
RouterB(config)#  router bgp 100
RouterB(config-router)#  neighbor 172.16.1.2 remote-as 900
```

Notice that the syntax is the same, and that the *remote-as* argument is always used, regardless if the peering is iBGP or eBGP.

For stability purposes, the *source* interface used to generate updates to a particular neighbor can be specified:

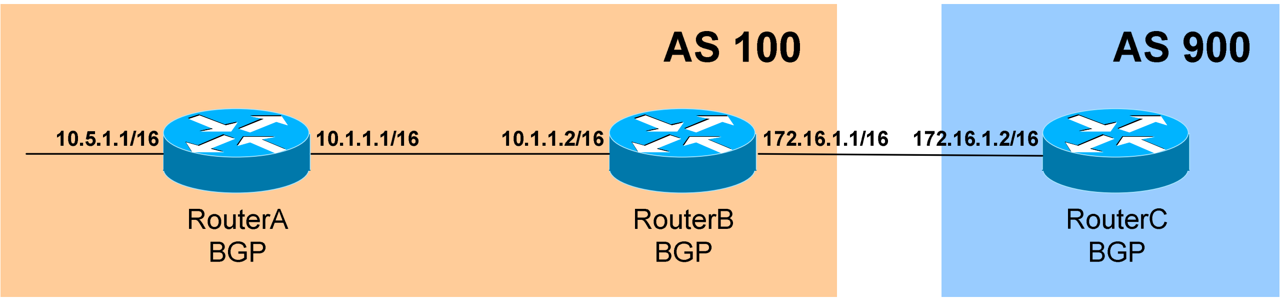
```
RouterB(config)#  router bgp 100
RouterB(config-router)#  neighbor 172.16.1.2 update-source lo0
```

RouterC must then point to RouterB's loopback (assume the address is 1.1.1.1/24) in its neighbor statement:

```
RouterC(config)#  router bgp 900
RouterC(config-router)#  neighbor 1.1.1.1 remote-as 100
```

RouterC *must* have a route to RouterB's loopback in its routing table.

Configuring BGP Neighbors (continued)



Remember though: by default, BGP assumes that external peers are exactly one hop away. Using the loopback as a source interface puts RouterB two hops away from RouterC. Thus, the *ebgp-multihop* feature must be enabled*:*

```
RouterC(config)# router bgp 900
RouterC(config-router)# neighbor 1.1.1.1 ebgp-multihop 2
```

The *2* indicates the number of hops to the eBGP peer. If left blank, the default is 255.

To authenticate updates between two BGP peers:

```
RouterB(config)# router bgp 100
RouterB(config-router)# neighbor 172.16.1.2 password CISCO
```

Configuring BGP Timers

To globally adjust the Keepalive and Hold-time timers for all neighbors:

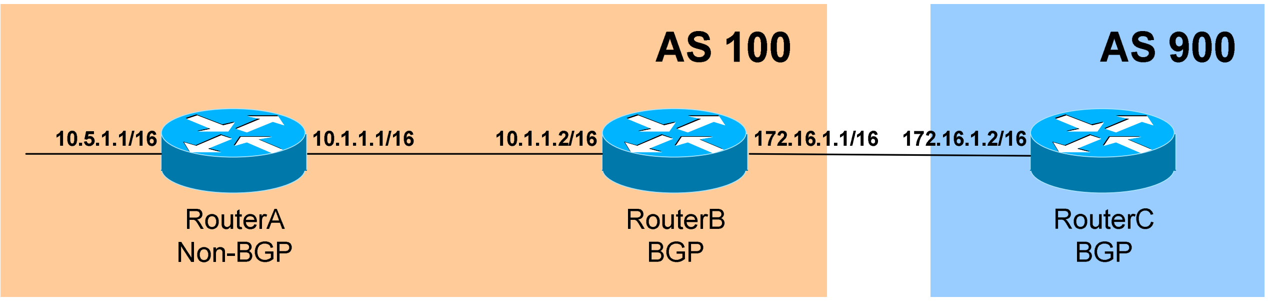
```
RouterB(config)# router bgp 100
RouterB(config-router)# timers bgp 30 90
```

The above command sets the Keepalive timer to *30* seconds, and the Hold-time timer to *90* seconds. If the configured Hold-time timers between two peers are different, the peer session **will still** be established, and the **smallest** timer value will be used.

To adjust the timers for a *specific* neighbor (which overrides the global timer configuration):

```
RouterB(config)# router bgp 100
RouterB(config-router)# neighbor 172.16.1.2 timers 30 90
```

Viewing BGP Neighbors



To view the status of *all* BGP neighbors:

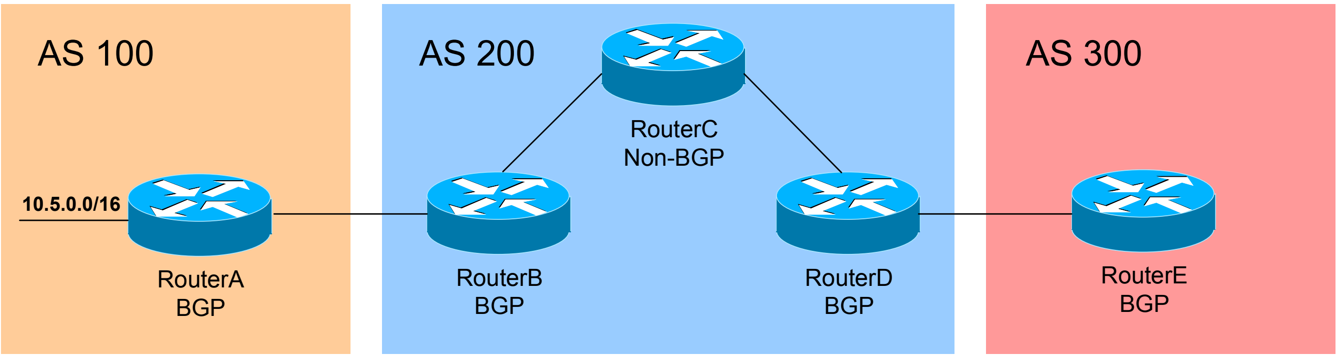
RouterB# *show ip bgp neighbors*

```
BGP neighbor is 172.16.1.2, remote AS 900, external link
 Index 1, Offset 0, Mask 0x2
  Inbound soft reconfiguration allowed
  BGP version 4, remote router ID 172.16.1.2
  BGP state = Established, table version = 27, up for 00:03:45
  Last read 00:00:19, hold time is 180, keepalive interval is 60
seconds
  Minimum time between advertisement runs is 30 seconds
  Received 25 messages, 0 notifications, 0 in queue
  Sent 20 messages, 0 notifications, 0 in queue
  Inbound path policy configured
  Route map for incoming advertisements is testing
  Connections established 2; dropped 1
Connection state is ESTAB, I/O status: 1, unread input bytes: 0
Local host: 172.16.1.1, Local port: 12342
Foreign host: 172.16.1.2, Foreign port: 179

Enqueued packets for retransmit: 0, input: 0, saved: 0

Event Timers (current time is 0x530C294):
Timer          Starts    Wakeups            Next
Retrans            15          0             0x0
TimeWait            0          0             0x0
AckHold            15         13             0x0
SendWnd             0          0             0x0
KeepAlive           0          0             0x0
GiveUp              0          0             0x0
PmtuAger            0          0             0x0

<snip>
```

To view the status of a *specific* BGP neighbor:

RouterB# *show ip bgp neighbors 172.16.1.2*

BGP Synchronization



Consider the above example. AS 200 is serving as a **transit** between AS 100 and AS 300. BGP follows a synchronization rule that states that *all* routers in a transit AS, *including* non-BGP routers, must learn of a route before BGP can advertise it to an external peer.

Confused?

Consider the above example again. If RouterA advertises a BGP route to RouterB (an eBGP peer) for the 10.5.0.0/16 network, that same BGP route will eventually be forwarded to RouterD (an iBGP peer).

However, a **blackhole** would exist if RouterD then advertised that update to RouterE, as RouterC would not have the 10.5.0.0/16 network in its routing table. If RouterE attempts to reach the 10.5.0.0 network, RouterC will drop the packet.

BGP's synchronization rule will force RouterD to wait until RouterC learns the 10.5.0.0/16 route, before forwarding that route to RouterE. How will RouterD know when RouterC learns the route? Simple! When it receives an update from RouterC via an IGP (such as OSPF), containing that route.

BGP synchronization can be disabled under two circumstances:

- The local AS is not a transit between two other AS's
- All routers in the transit AS run iBGP, and are fully meshed.

To disable BGP synchronization:

```
RouterD(config)#  router bgp 200
RouterD(config-router)#  no synchronization
```

As of IOS 12.2(8)T, synchronization is **disabled** by default.

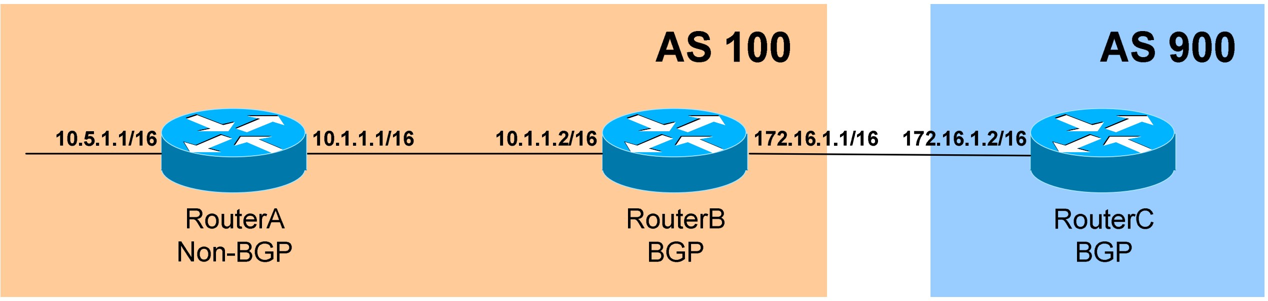
Originating Prefixes in BGP

There are three ways to originate a prefix (in other words, advertise a network) into BGP:

- By using *network* statements
- By using *aggregate-address* statements (explained later in this guide)
- By redistributing an IGP into BGP

Using the *network* statement informs BGP which networks to advertise to eBGP peers, *not* which interfaces to run BGP on. The *network* command can be used to inject *any* network from the local AS into BGP, include dynamic routes learned from an IGP, and not just the routes directly connected to the router.

However, the route ***must* be in the routing table** before BGP will advertise the network to an eBGP peer. **This is a fundamental BGP rule.**



Consider the above example. RouterB may inject the 10.5.0.0/16 network into BGP using the *network* command. However, unless that route is in the local routing table (in this case, via an IGP), RouterB will *not* advertise the route to RouterC.

Furthermore, the *network* statement **must match the route exactly** as it is in the routing table:

```
RouterB(config)#  router bgp 100
RouterB(config-router)#  neighbor 172.16.1.2 remote-as 900
RouterB(config-router)#  network 10.5.0.0 mask 255.255.0.0
```

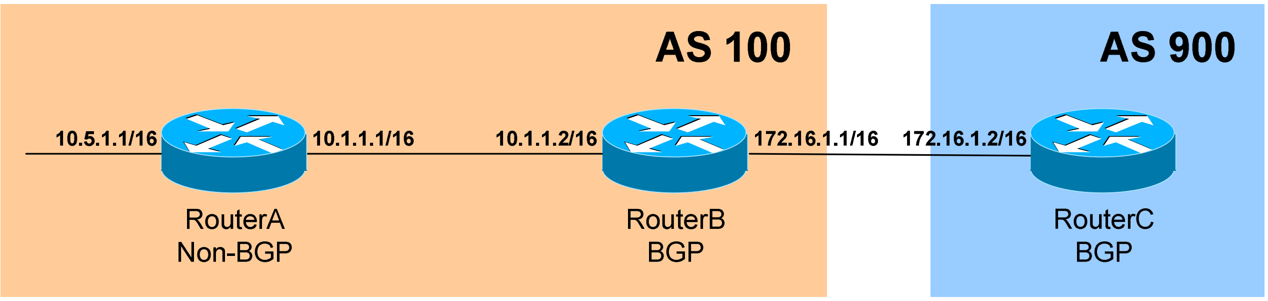
The above configuration would match the route perfectly, while the following configuration would not:

```
RouterB(config-router)#  network 10.5.0.0 mask 255.255.255.0
```

If no mask is specified, a **classful** mask will be assumed.

The BGP Routing Table

Recall that BGP maintains its own **separate routing table**. This table contains a list of routes that can be advertised to BGP peers.



To view the BGP routing table on RouterB:

RouterB# *show ip bgp*

```
        BGP table version is 426532, local router ID is 2.2.2.2
        Status codes: s suppressed, * valid, > best, i - internal
        Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*> 10.5.0.0         0.0.0.0               0      0      32768   i
```

The route has been injected into BGP using the *network* command. The *Next Hop* of *0.0.0.0* indicates that the route was locally originated into BGP. The *Path* is empty, as the route originated in the Autonomous Systems.

Notice the Status Codes of "*>". The * indicates that this route is *valid* (i.e. in the routing table). The > indicates that this is the *best* route to the destination.

BGP will **never** advertise a route to an eBGP peer unless it is both *valid* and the *best* route to that destination. BGP routes that are both *valid* and *best* will also added the **IP routing table** as well.

To view the BGP routing table on RouterC:

RouterC# *show ip bgp*

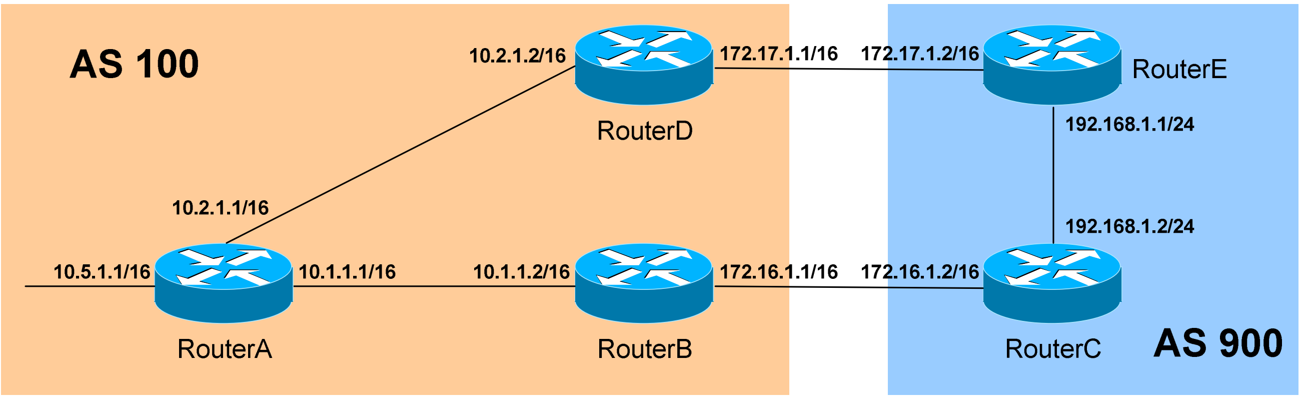
```
   Network          Next Hop            Metric LocPrf Weight Path
*> 10.5.0.0         172.16.1.1            0      100     0     100 i
```

Notice that AS 100 has been added to the path, and that the Next Hop is now RouterB.

BGP Route-Reflectors

Recall that BGP requires all iBGP peers to be fully meshed. **Route-Reflectors** allow us to bypass this restriction. Fewer neighbor connections will result in less bandwidth and CPU usage.

Route-reflector **clients** form neighbor adjacencies with the route-reflector **server**. BGP updates will flow from the server to the clients, without the clients having to interact with each other.



Consider the above example. In AS 100, there are three BGP speakers. Normally, these iBGP peers *must* be fully-meshed. For example, RouterB would need a *neighbor* statement for both RouterA and RouterD.

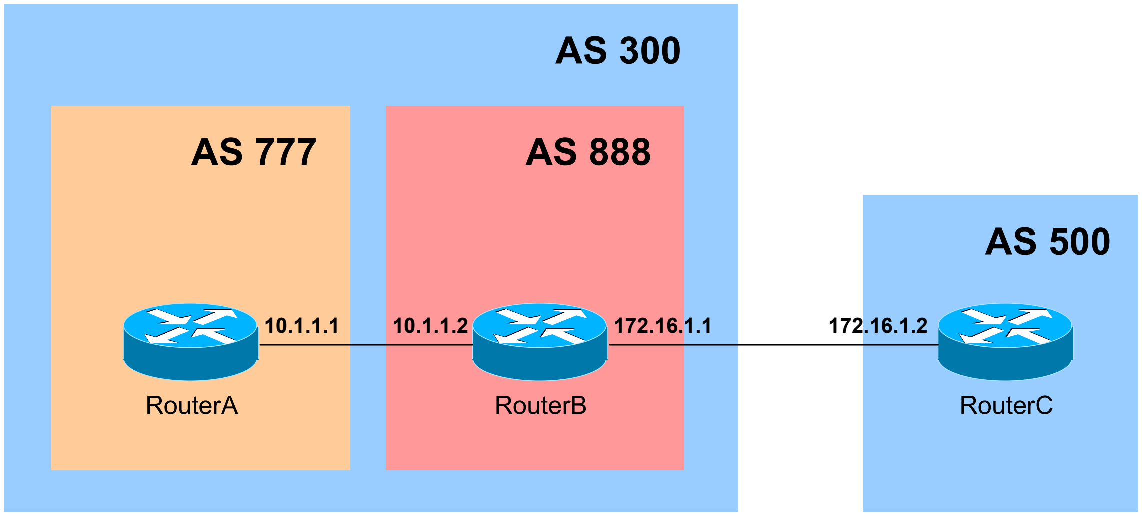
As an alternative, RouterA can be configured as a route-reflector server. Both RouterB and RouterD would *only* need to peer with RouterA.

All route-reflector specific configuration takes place on the route reflector server:

```
RouterA(config)#  router bgp 100
RouterA(config-router)#  neighbor 10.2.1.2 remote-as 100
RouterA(config-router)#  neighbor 10.2.1.2 route-reflector-client
RouterA(config-router)#  neighbor 10.1.1.2 remote-as 100
RouterA(config-router)#  neighbor 10.1.1.2 route-reflector-client
```

Route-reflectors are Cisco's recommended method of alleviating the iBGP full-mesh requirement.

BGP Confederations



Confederations are an alternative method to alleviate the requirement that all iBGP routers be fully meshed. Confederations are essentially AS's within an AS, and are sometimes referred to as **sub-AS's**.

In the above example, RouterA belongs to AS 777 and RouterB belongs to AS 888. Both of those AS's belong to a **parent** AS of 300. RouterA and RouterB will form an **eBGP** peer session.

Configuration is simple:

```
RouterB(config)#  router bgp 888
RouterB(config-router)#  bgp confederation identifier 300
RouterB(config-router)#  bgp confederation peer 777
RouterB(config-router)#  neighbor 10.1.1.1 remote-as 777
RouterB(config-router)#  neighbor 172.16.1.2 remote-as 500
```

Notice that the sub-AS (*777*) is used in the *router bgp* statement. Additionally, the parent AS must be specified using a *bgp confederation identifier* statement. Finally, any *confederation peers* must be identified.

RouterC will be unaware of RouterB's confederation status. Thus, RouterC's neighbor statement will point to AS 300, and not AS 888:

```
RouterC(config)#  router bgp 500
RouterC(config-router)#  neighbor 172.16.1.1 remote-as 300
```

(Reference: http://www.cisco.com/univercd/cc/td/doc/cisintwk/ics/icsbgp4.htm#wp6834)

BGP Peer-Groups

Peer-groups simplify configuration of groups of neighbors, assuming those neighbors share identical settings. Additionally, peer-groups conserve processor/memory resources by sending updates to all peer-group members *simultaneously*, as opposed to sending *individual* updates to each neighbor.

All neighbor parameters are applied to the peer-group itself. Configuration is simple:

```
Router(config)#  router bgp 200
Router(config-router)#  neighbor MYPEERGROUP peer-group
Router(config-router)#  neighbor MYPEERGROUP remote-as 200
Router(config-router)#  neighbor MYPEERGROUP update-source lo0
Router(config-router)#  neighbor MYPEERGROUP route-reflector-client
```

The above configuration creates a peer-group named *MYPEERGROUP*, and applies the desired settings. Next, we must “assign” the appropriate neighbors to the peer-group:

```
Router(config-router)#  neighbor 10.10.1.1 peer-group MYPEERGROUP
Router(config-router)#  neighbor 10.10.2.2 peer-group MYPEERGROUP
Router(config-router)#  neighbor 10.10.3.3 peer-group MYPEERGROUP
```

The above neighbors now inherit the settings of the peer-group named *MYPEERGROUP*.

All “members” of a peer-group must *exclusively* be internal (iBGP) peers *or* external (eBGP) peers. A mix of internal and external peers is not allowed in a peer-group.

Outbound route filtering (via a distribution-list, route-map, etc.) must be identical on all members of a peer-group. Inbound route filtering can still be applied on a per-neighbor basis.

(Reference: http://www.cisco.com/warp/public/459/29.html)

BGP Attributes

BGP utilizes several **attributes** to determine the best path to a destination.

Well-known attributes are supported by all implementations of BGP, while **optional** attributes may *not* be supported by all BGP-speaking routers.

Several **subcategories** of attributes exist:

- **Well-known Mandatory –** Standard attributes supported by all BGP implementations, and *always* included in every BGP update.

- **Well-known Discretionary –** Standard attributes supported by all BGP implementations, and are *optionally* included BGP updates.

- **Optional Transitive –** Optional attribute that may not be supported by all implementations of BGP. *Transitive* indicates that a non-compliant BGP router will forward the unsupported attribute unchanged, when sending updates to peers.

- **Optional Non-Transitive -** Optional attribute that may not be supported by all implementations of BGP. *Non-Transitive* indicates that a non-compliant BGP router will strip out the unsupported attribute, when sending updates to peers.

BGP Attributes (continued)

The following describes several **specific** BGP attributes:

- **AS-Path (well-known mandatory)** – Identifies the list (or path) of traversed AS's to reach a particular destination.

- **Next-Hop (well-known mandatory)** – Identifies the next hop IP address to reach a particular destination.

- **Origin (well-known mandatory) –** Identifies the originator of the route.

- **Local Preference (well-known, discretionary) –** Provides a preference to determine the best path for *outbound* traffic.

- **Atomic Aggregate (well-known discretionary) –** Identifies routes that have been summarized, or *aggregated.*

- **Aggregator (optional transitive) –** Identifies the BGP router that performed an address aggregation.

- **Community (optional transitive) –** Tags routes that share common characteristics into *communities*.

- **Multi-Exit-Discriminator (MED) (optional non-transitive) –** Provides a preference to eBGP peers to a specific *inbound* router.

- **Weight (Cisco Proprietary) –** Similar to Local Preference, provides a *local* weight to determine the best path for outbound traffic.

Each attribute is identified by a **code**:

Origin	Code 1
AS-Path	Code 2
Next Hop	Code 3
MED	Code 4
Local Preference	Code 5
Automatic Aggregate	Code 6
Aggregator	Code 7
Community	Code 8

BGP "Best Path" Determination

If BGP contains multiple routes to the same destination, it compares the routes in **pairs**, starting with the **newest** entries (listed higher in the routing table), and working towards the **oldest** entries (listed lower in the table).

BGP determines the best path by successively comparing the attributes of each "route pair." The attributes are compared in a specific order:

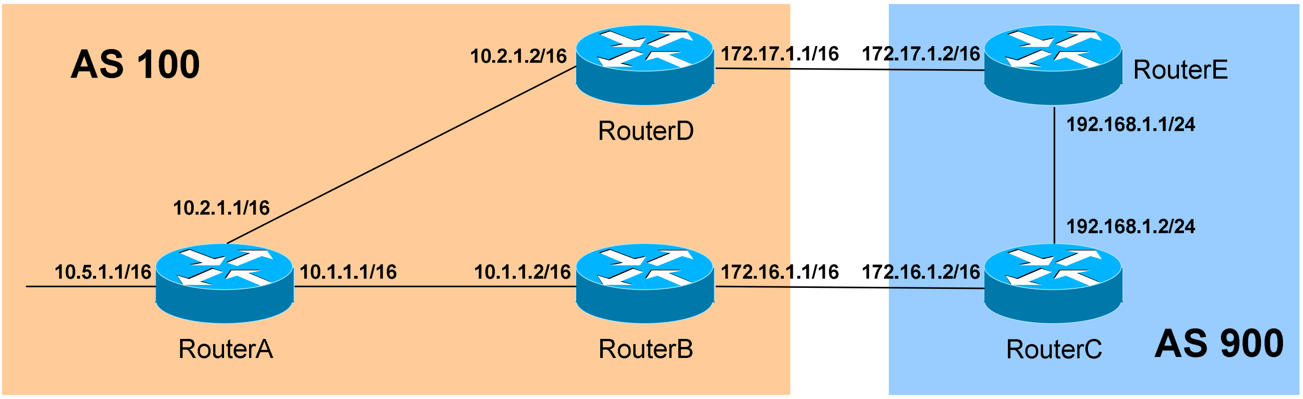
- **Weight –** Which route has the *highest* weight?
- **Local Preference –** Which route has the *highest* local preference?
- **Locally Originated –** Did the local router originate this route? In other words, is the next hop to the destination 0.0.0.0?
- **AS-Path –** Which route has the *shortest* AS-Path?
- **Origin Code –** Where did the route originate? The following origin codes are listed in order of preference:
    - IGP (originated from an interior gateway protocol)
    - EGP (originated from an exterior gateway protocol)
    - ? (Unknown origin)
- **MED –** Which path has the *lowest* MED?
- **BGP Route Type –** Is this an *eBGP* or *iBGP* route? (eBGP routes are preferred)
- **Age –** Which route is the oldest? (oldest is preferred)
- **Router ID –** Which route originated from the router with the lowest BGP router ID?
- **Peer IP Address –** Which route originated from the router with the lowest IP?

When applying attributes, Weight and Local Preference are applied to *inbound* routes, dictating the best *outbound* path.

AS-Path and MED are applied to *outbound* routes, dictating the best *inbound* path.

(Reference: http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a0080094431.shtml)

Weight



The **Weight** attribute is applied to *inbound* routes, dictating the best *outbound* path. It is a Cisco-proprietary attribute, and is only **locally significant** (and thus, is *never* passed on to BGP neighbors).

The weight value can range from 0 – 65535, and the **highest** weight is preferred. By default, a route originated on the local router will be assigned a weight of **32768**. All other routes will be assigned a weight of **0**, by default.

A weight value can be specified for *all routes* advertised from a specific neighbor:

```
RouterA(config)#  router bgp 100
RouterA(config)#  neighbor 10.1.1.2 weight 200
```

Otherwise, a weight value can be specified for *specific routes* from a particular neighbor. First, the prefixes in question must be identified:

```
RouterA(config)#  ip prefix-list MYLIST 192.168.1.0/24
```

Then, a route-map is used to apply the appropriate weight:

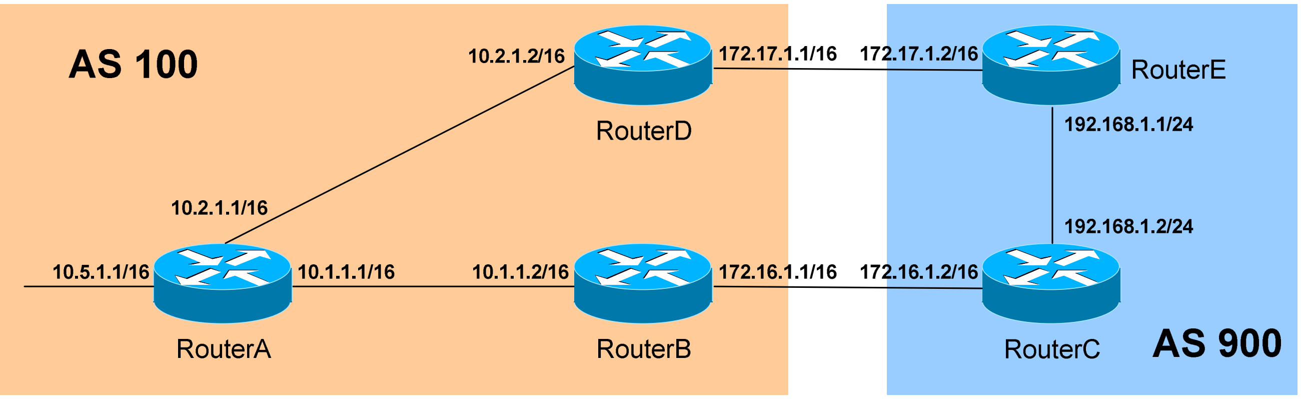
```
RouterA(config)#  route-map WEIGHT permit 10
RouterA(config-route-map)#  match ip address prefix-list MYLIST
RouterA(config-route-map)#  set weight 200
RouterA(config-route-map)#  route-map WEIGHT permit 20
```

Finally, the route-map is applied to the preferred neighbor:

```
RouterA(config)#  router bgp 100
RouterA(config)#  neighbor 10.1.1.2 route-map WEIGHT in
```

(Reference: http://www.cisco.com/warp/public/459/bgp-toc.html#weight)

Local Preference



The **Local Preference** attribute is applied to *inbound* external routes, dictating the best *outbound* path. Unlike the Weight attribute, Local Preference is passed on to **iBGP** peers when sending updates. Local Preference informs iBGP routers how to *exit* the AS, if multiple paths exist.

Local Preference is a 32-bit number, and can range from 0 to 4294967295. The **highest** Local Preference is preferred, and the default preference is **100.**

The Local Preference value can be specified for *all inbound external routes*, on a global basis for BGP:

```
RouterB(config)# router bgp 100
RouterB(config-router)# bgp default local-preference 200

RouterD(config)# router bgp 100
RouterD(config-router)# bgp default local-preference 300
```

Both RouterB and RouterD will include the Local Preference attribute in updates to iBGP neighbors. Thus, RouterA (*and* RouterB) will now prefer the route through RouterD to reach any destination outside the local AS.

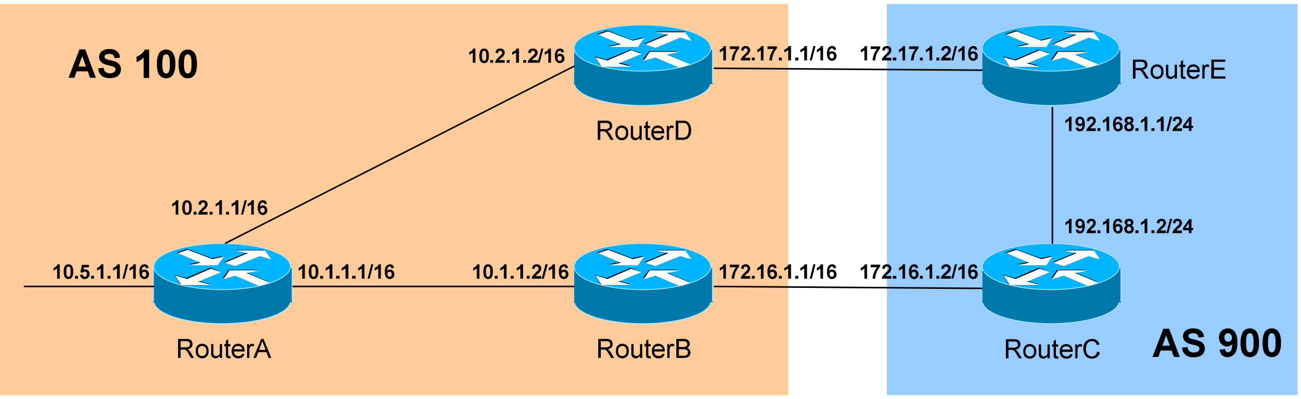
Local Preference can be applied on a per-route basis:

```
RouterD(config)# ip prefix-list MYLIST 192.168.1.0/24

RouterD(config)# route-map PREFERENCE permit 10
RouterD(config-route-map)# match ip address prefix-list MYLIST
RouterD(config-route-map)# set local-preference 300

RouterD(config)# router bgp 10
RouterD(config)# neighbor 172.17.1.2 route-map PREFERENCE in
```

(Reference: http://www.cisco.com/warp/public/459/bgp-toc.html#localpref)

* * *

AS-Path Prepend



The **AS-Path attribute** is applied to *outbound* routes, dictating the best *inbound* path. Two things can be accomplished with the AS-Path attribute, *prepend* or *filter*.

To *prepend* to (or add to) the existing AS-Path results in a *longer* AS-Path, which makes the route *less* desirable for inbound traffic:

RouterB(config)# *access-list 5 permit 10.5.0.0 0.0.255.255*

RouterB(config)# *route-map ASPREPEND permit 10*
RouterB(config-route-map)# *match ip address 5*
RouterB(config-route-map)# *set as-path prepend 200 200*
RouterB(config-route-map)# *route-map ASPREPEND permit 20*

RouterB(config)# *router bgp 100*
RouterB(config-router)# *neighbor 172.16.1.2 route-map ASPREPEND out*

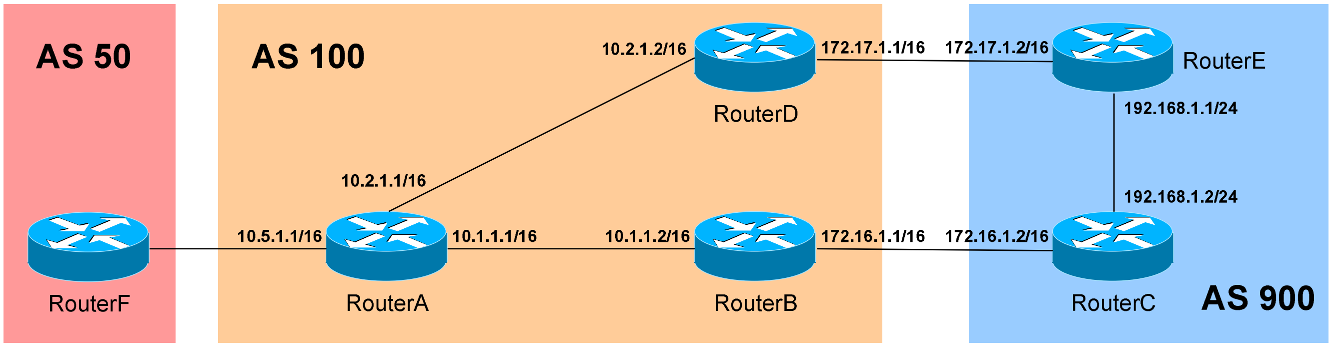
The artificial AS-Path information is not added to a route until it is advertised to an eBGP peer. RouterC's BGP routing table will now look as follows:

RouterC# *show ip bgp*

```
   Network          Next Hop          Metric LocPrf Weight Path
*  10.5.0.0         172.16.1.1          0      100    0      100 200 200 i
*> 10.5.0.0         172.17.1.1          0      100    0      100 i
```

Notice the inflated AS-Path through RouterB. RouterC will prefer the path through RouterD to reach the 10.5.0.0/16 network.

AS-Path Filtering



Additionally, routes can be *filtered* based on AS-Path values, using an *as-path access-list*. This requires the use of **regular expressions:**

- **^** = Start of a string
- **$** = End of a string
- **.** = Any one character
- ***** = Any one or more characters, including none
- **+** = Any one or more characters
- ? = Any one character, including none
- _ = Serves the function of virtually all of the above

The following examples illustrate the use of regular expressions:

- ^100_ = learned from AS 100
- _100$ = originated from AS 100
- ^$ = originated locally
- .* = matches everything
- _100_ = any instance of AS 100

To configure RouterF to only accept routes that *originated* from AS100:

```
RouterF(config)#  ip as-path access-list 15 permit  _100$

RouterF(config)#  route-map ASFILTER permit 10
RouterF(config-route-map)#  match as-path 15

RouterF(config)#  router bgp 50
RouterF(config-router)#  neighbor 10.5.1.1 route-map ASFILTER in
```

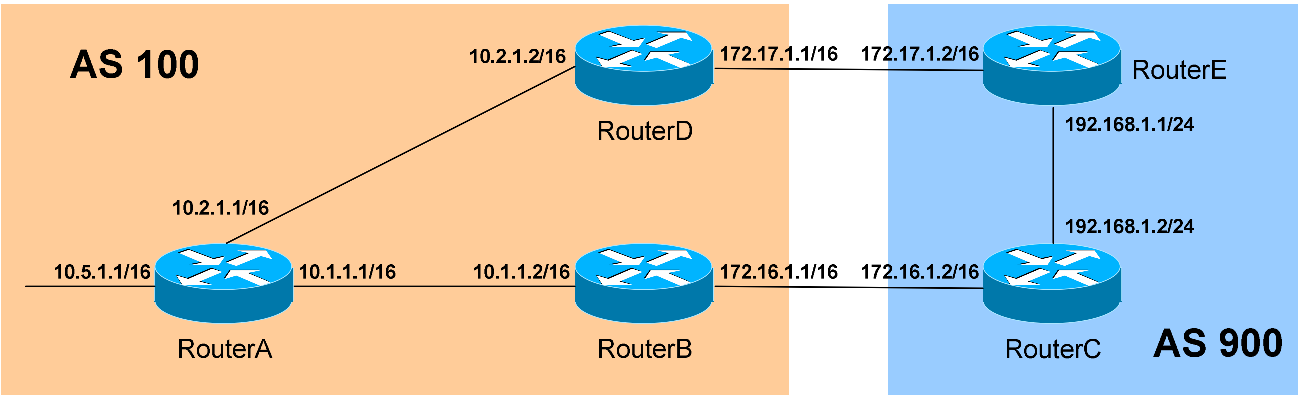
To view what BGP routing entries the AS-Path access-list will match:

```
RouterF#  show ip bgp regexp _100$
```

(Reference: http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a0080094a92.shtml)

* * *

Origin



The **Origin attribute** identifies the originating *source* of the route. The origin codes are as follows (listed in order of preference for route selection):

- **i (IGP) –** Originated from an interior gateway protocol, such as OSPF. This usually indicates the route was injected into BGP via the *network* command under the BGP process. An origin code of “**i**” is most preferred.
- **e (EGP) –** Originated from an external gateway protocol.
- **? (incomplete)** - Unknown origin. This usually indicates the route was redistributed into BGP (from either connected, static, or IGP routes). An origin code of “**?**” is the least preferred.

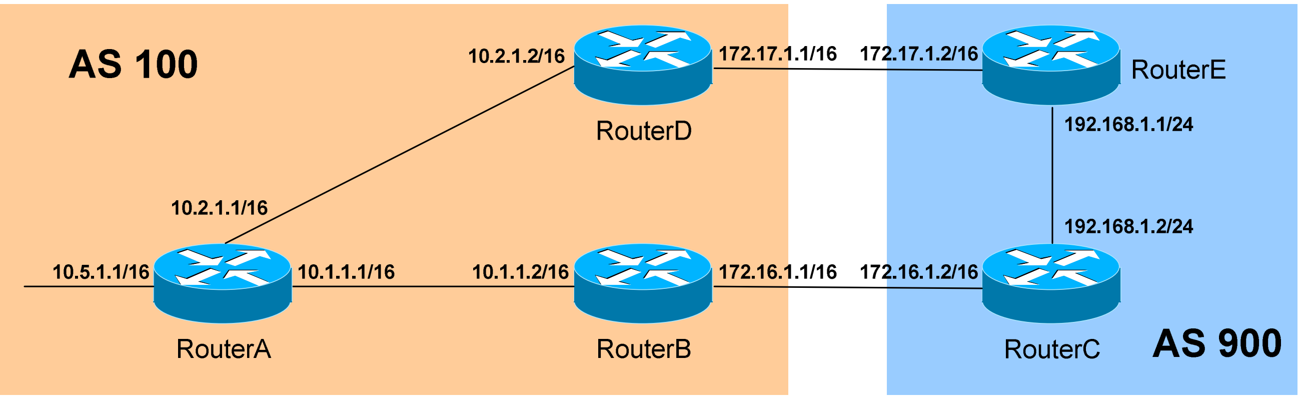
When viewing the BGP routing table, the origin code is listed at the *end* of each line in the table:

RouterB# *show ip bgp*

```
   Network          Next Hop         Metric LocPrf Weight Path
*> 10.5.0.0         10.1.1.1           0      0      0      i
*> 192.168.1.0      172.16.1.2         0      100    0      900 ?
```

The *i* at the end of the first routing entry indicates the *10.5.0.0* network was originated via an IGP, probably with the BGP *network* command. The *192.168.1.0* network was most likely redistributed into BGP in AS 900, as evidenced by the *?* at the end of that routing entry.

MED



The **MED (MultiExit Discriminator) attribute** is applied to *outbound* routes, dictating the best *inbound* path into the AS (assuming multiple paths exist). The MED is identified as the BGP **metric** when viewing the BGP routing table. A **lower** metric is preferred, and the default MED value is **0**.

In the above example, there are two entry points into AS 100. To force AS 900 to prefer that path through RouterD to reach the 10.5.0.0/16 network, the *set metric* command can be used with a route-map:

RouterB(config)# *access-list 5 permit 10.5.0.0 0.0.255.255*

RouterB(config)# *route-map SETMED permit 10*
RouterB(config-route-map)# *match ip address 5*
RouterB(config-route-map)# *set metric 200*

RouterB(config)# *router bgp 100*
RouterB(config-router)# *neighbor 172.16.1.2 route-map SETMED out*

RouterC will now have two entries for the 10.5.0.0/16 route:

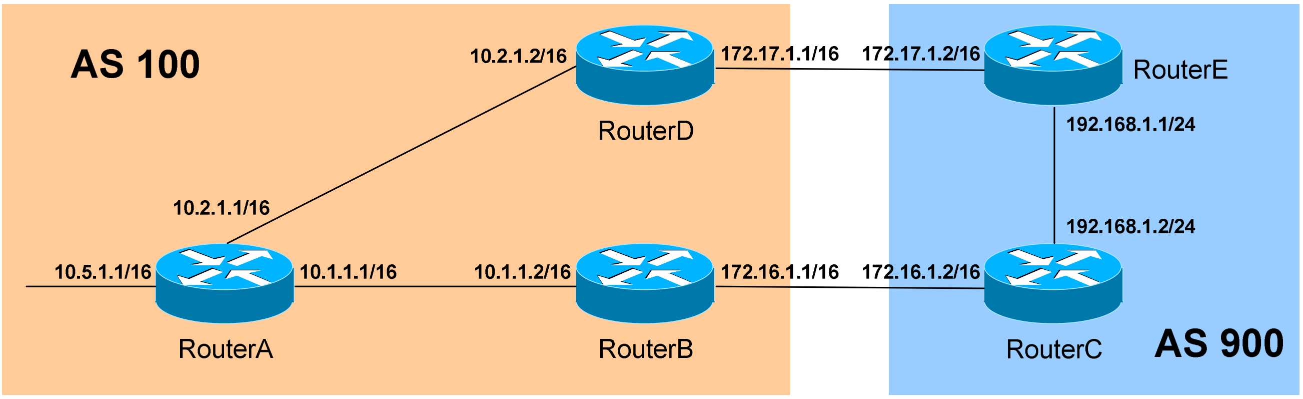
RouterC# *show ip bgp*

```
   Network          Next Hop        Metric LocPrf Weight Path
*  10.5.0.0         172.16.1.1       200     100      0   100 i
*> 10.5.0.0         172.17.1.1       0       100      0   100 i
```

Notice that the route from RouterB has a higher metric, and thus is less preferred. Note specifically the lack of a > on the route with a higher metric.

The MED value is exchanged from one AS to another, but will *never* be advertised further than that. Thus, the MED value is passed from AS 100 to all BGP routers in AS 900, but the metric will be reset to **0** if the route is advertised beyond AS 900.

MED (continued)



A key aspect to consider when using the MED attribute is BGP's method of route selection. Recall that if BGP contains multiple routes to the same destination, it compares the routes in **pairs**, starting with the **newest** entries and working towards the **oldest** entries.

This can lead to sub-optimal routing, depending on the order of routes in the BGP routing table. BGP employs two MED-related commands to alleviate potential sub-optimal routing selections.

The ***bgp deterministic-med*** command forces the MED value to be compared, when multiple routes to the same network are received via *multiple routers from the **same** AS*, regardless of the order of routes in the BGP routing table.

```
RouterE(config)#  router bgp 100
RouterE(config-router)#  bgp deterministic-med
```

The *bgp deterministic-med* command is disabled by default. If used, the command should be enabled on *all* routers within the AS.

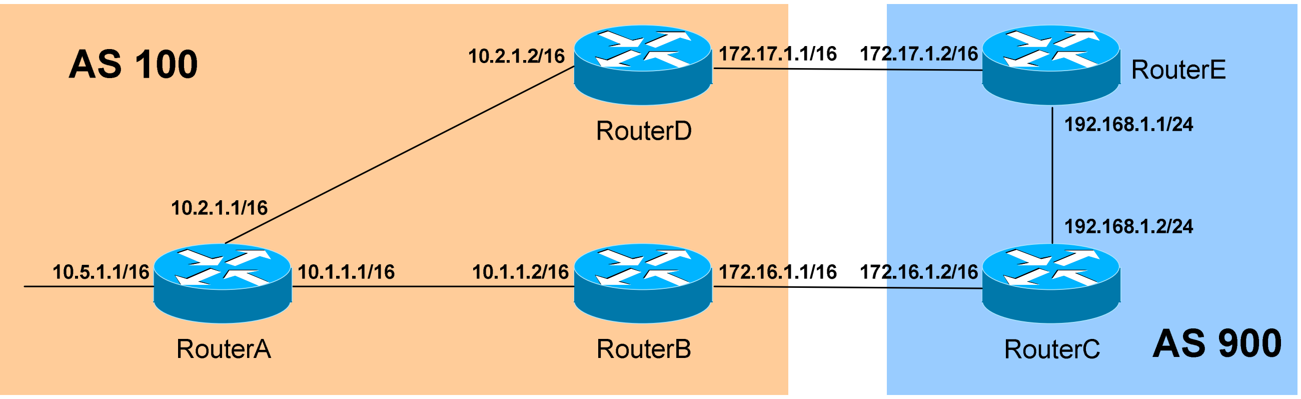
The ***bgp always-compare-med*** command forces the MED value to be compared, when multiple routes to the same network are received via *multiple routers from **different** AS's*, regardless of the order of routes in the BGP routing table.

```
RouterE(config)#  router bgp 100
RouterE(config-router)#  bgp always-compare-med
```

The *bgp always-compare-med* command is disabled by default. Thus, by default, the MED value is *not* compared between paths from different AS's.

(Reference: http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a0080094925.shtml; http://www.cisco.com/warp/public/459/bgp-toc.html#metricattribute)

MED (continued)



The MED metric on routes sent to eBGP neighbors can be dynamically set to the actual metric of an IGP (such as OSPF). This is accomplished using the *set metric-type internal* command with a route-map:

```
RouterB(config)#  access-list 5 permit 10.5.0.0 0.0.255.255

RouterB(config)#  route-map MED_INTERNAL  permit 10
RouterB(config-route-map)#  match ip address 5
RouterB(config-route-map)#  set metric-type internal

RouterB(config)#  router bgp 100
RouterB(config-router)#  neighbor 172.17.1.2 route-map MED_INTERNAL out
```

If the *10.5.0.0/16* network originated in OSPF, the link-state **cost** metric for that route will be applied as the MED metric.

Communities

BGP allows routes to be placed (or *tagged*) into certain **Communities.** BGP routers can make route policy decisions based on a route's community membership.

BGP communities can be assigned using one of three **32-bit** formats:

- **Decimal** (1000000**)**
- **Hexadecimal (**0x1A2B3C)
- **AA:NN** (100:20)

The AA:NN format specifies a 16-bit AS number (the AA), and a 16-bit generic community identifier (NN).

By default, the *decimal* format for communities will be displayed when viewing a route. To force the router to display the AA:NN format:

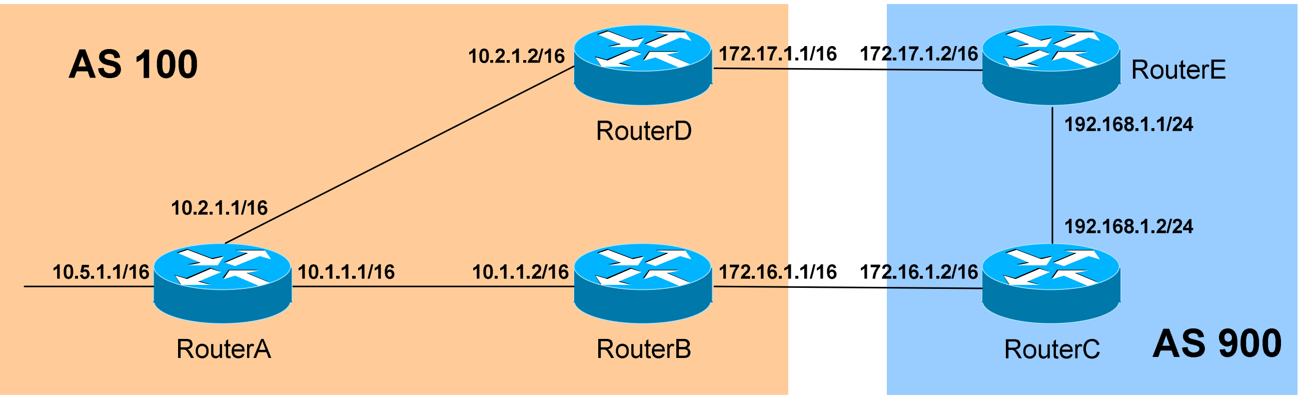
```
RouterA(config)#  ip bgp-community new-format
```

Additionally, there are four **well-known** communities that can be referenced by name:

- **No-export** – prevents the route from being advertised outside the local AS to eBGP peers.
- **No-advertise** – prevents the route from being advertised to either internal or external peers.
- **Internet** – allows the route to be advertised outside the local AS.
- **Local**-**AS –** prevents the route from being advertised outside the local AS to either eBGP *or* confederate peers.

(Reference: http://www.cisco.com/en/US/tech/tk365/technologies_q_and_a_item09186a00800949e8.shtml#four; http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a00800c95bb.shtml#communityattribute)

Communities (continued)



To set the community for a specific route, using a route-map:

```
RouterB(config)# access-list 5 permit 10.5.0.0 0.0.255.255

RouterB(config)# route-map COMMUNITY permit 10
RouterB(config-route-map)# match ip address 5
RouterB(config-route-map)# set community no-export
RouterB(config)# route-map COMMUNITY permit 20

RouterB(config)# router bgp 100
RouterB(config-router)# neighbor 172.16.1.2 send-community
RouterB(config-router)# neighbor 172.16.1.2 route-map COMMUNITY out
```

The community attribute will not be advertised to a neighbor unless the *send-community* parameter is applied to the *neighbor* command, regardless if a community value is applied using a route-map.

The above configuration will place the *10.5.0.0/16* route into the *no-export* community once it is advertised into AS 900. RouterC will advertise this network to all iBGP peers, but the community attribute will prevent RouterC (and all iBGP peers) from advertising the route outside of AS 900.

By default, the *set community* route-map command will **overwrite** any existing community parameters for a route. To instead **append** additional community values, the *additive* parameter must be specified:

```
RouterB(config)# route-map COMMUNITY permit 10
RouterB(config-route-map)# match ip address 5
RouterB(config-route-map)# set community no-export additive
RouterB(config)# route-map COMMUNITY permit 20
```

BGP Summarization

Routes that are **redistributed** into BGP are **automatically summarized**. To disable auto-summary:

```
Router(config)#  router bgp 100
Router(config-router)#  no auto-summary
```

To manually create a summary address for the following group of networks:

- 172.16.0.0/24
- 172.16.1.0/24
- 172.16.2.0/24
- 172.16.3.0/24

The *aggregate-address* command must be used:

```
Router(config)#  router bgp 100
Router(config-router)#  aggregate-address 172.16.0.0 255.255.252.0
```

BGP's default configuration is to send ***both*** the summary (or aggregated) address ***and*** the more specific individual routes. To *only* send the summary route:

```
Router(config)#  router bgp 100
Router(config-router)#  aggregate-address 172.16.0.0 255.255.252.0 summary-only
```

To **suppress** (or *summarize*) only specific routes, instead of all routes, a route-map must be used:

```
Router(config)#  access-list 5 permit 172.16.0.0 0.0.0.255
Router(config)#  access-list 5 permit 172.16.1.0 0.0.0.255

Router(config)#  route-map SUPPRESS permit 10
Router(config-route-map)#  match ip address 5

Router(config)#  router bgp 100
Router(config-router)#  aggregate-address 172.16.0.0 255.255.252.0 summary-only suppress-map SUPPRESS
```

The access-list details the routes that *should* be suppressed. To allow the summarized routes to retain their AS-Path information:

```
Router(config)#  router bgp 100
Router(config-router)#  aggregate-address 172.16.0.0 255.255.252.0 summary-only suppress-map SUPPRESS as-set
```

BGP Route Dampening

Route dampening "suppresses" routes that are flapping, minimizing unnecessary convergence and updates. If a route **flaps** (goes up and down), it is assigned a penalty (default is **1000**). All routes start with a penalty of 0, and the local router maintains a history of routes that have flapped.

Once the penalty reaches a specific threshold, the route is suppressed. When a route is suppressed, it is neither advertised nor used locally on the router.

First, the routes to be "observed" must be identified using an access-list or prefix-list:

```
Router(config)#  ip prefix-list MYLIST seq 10 permit 10.1.0.0/16
Router(config)#  ip prefix-list MYLIST seq 20 permit 10.2.0.0/16
```

Next, dampening values must be configured using a route-map:

```
Router(config)#  route-map MYMAP permit 10
Router(config-route-map)#  match ip address prefix-list MYLIST
Router(config-route-map)#  set dampening 15 750 2000 60
```

The above values for the *set dampening* command represent the defaults.

The *15* (measured in minutes) indicates the half-life timer. If a route is assigned a penalty, half of the penalty will decay after this timer expires.

The *750* (arbitrary penalty measurement) indicates the bottom threshold. Once a penalized route falls below this threshold, it will no longer be suppressed.

The *2000* (arbitrary penalty measurement) indicates the top threshold. If a route flaps to the point that its penalty exceeds this threshold, it is suppressed.

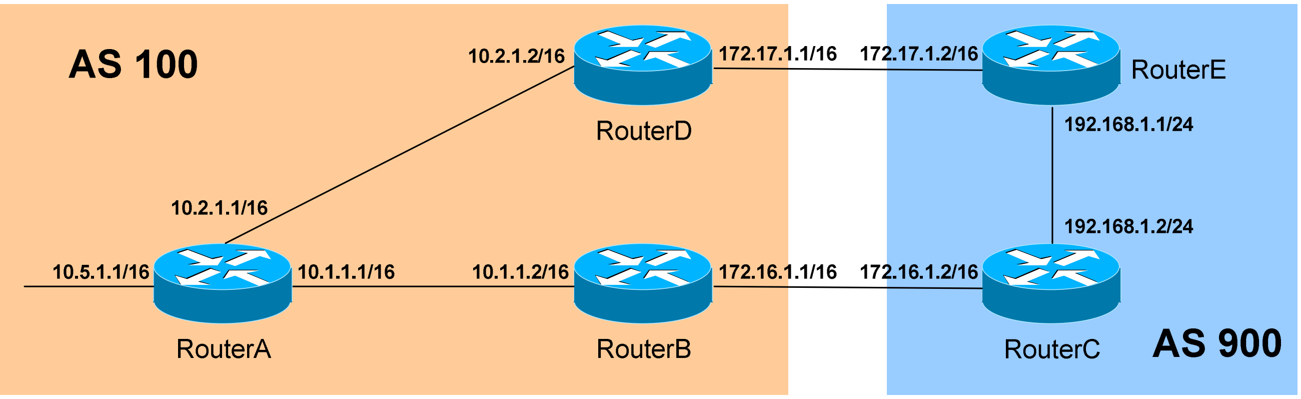
The *60* (measured in minutes) indicates the maximum amount of time a route can be suppressed.

Finally, route-dampening must be enabled under the BGP process:

```
Router(config)#  router bgp 100
Router(config-router)#  bgp dampening route-map MYMAP
```

(Reference: http://www.cisco.com/warp/public/459/bgp-rec-routing.html)

BGP Next-Hop-Self



Consider the above diagram. If RouterC sends the 192.168.1.0/24 route to its eBGP peer RouterB, the Next Hop for that route will be through RouterC:

```
RouterB# show ip bgp
```

```
   Network          Next Hop          Metric LocPrf Weight Path
*> 192.168.1.0      172.16.1.2          0     100      0   900 i
```

A serious problem arises when RouterB sends this route to its iBGP peers (RouterA and RouterD). The Next Hop value is *not* changed:

```
RouterA# show ip bgp
```

```
   Network          Next Hop          Metric LocPrf Weight Path
*  192.168.1.0      172.16.1.2          0     100      0   900 i
```

Notice the lack of >, indicating this is no longer the *best* route to the destination. This is because RouterA has no route to the next hop address.

There are two workarounds. Either the 172.16.0.0/16 network must be added to RouterA's and RouterD's routing tables, or the Next-Hop field must be adjusted to identify RouterB as the next hop.

The configuration is simple, and is completed on RouterB:

```
RouterB(config)# router bgp 200
RouterB(config-router)# neighbor 10.1.1.1 next-hop-self
RouterB(config-router)# neighbor 10.2.1.2 next-hop-self
```

RouterB now advertises itself as the next hop for all eBGP routes it learns:

```
RouterA# show ip bgp
```

```
   Network          Next Hop          Metric LocPrf Weight Path
*> 192.168.1.0      10.1.1.2             0     100     0    900 i
```

* * *

BGP Backdoor

Recall that an external BGP route has an Administrative Distance (AD) of **20**, which is less than the default AD of IGP's, such as OSPF or EIGRP.

Under certain circumstances, this may result in sub-optimal routing. If both an IGP route and eBGP route exist to the same network, and the IGP route *should* be preferred, there are two workarounds:

- Globally change BGP's default Administrative Distance values.
- Use the BGP *network backdoor* command.

Cisco does not recommend changing BGP's default AD values. If necessary, however, the *distance bgp* will adjust the AD for **external, internal**, and **locally-originated** BGP routes, respectively:

```
Router(config)#  router bgp 100
Router(config-router)#  distance bgp 150 210 210
```

The preferred workaround is to use the BGP *network backdoor* command, which adjusts the AD for a specific eBGP route (by default, from **20** to **200**), resulting in the IGP route being preferred:

```
Router(config)#  router bgp 100
Router(config-router)#  network 10.5.0.0 mask 255.255.0.0 backdoor
```

(Reference: http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a00800c95bb.shtml#bgpbackdoor)

Misc. BGP Commands

To restrict the number of routes a BGP router can receive from its neighbor:

```
Router(config)#  router bgp 200
Router(config-router)#  neighbor 10.1.1.1 maximum-prefix 10000
```

To immediately reset an eBGP session if a link connecting two peers goes down, the *bgp fast-external-fallover* feature must be enabled. To enable this feature globally:

```
Router(config)#  router bgp 200
Router(config-router)#  bgp fast-external-fallover
```

To enable this feature on a per-interface basis:

```
Router(config)#  int serial0/0
Router(config-if)#  ip bgp fast-external-fallover permit
```

To reset the BGP session between all neighbors:

```
Router#  clear ip bgp *
```

To force a resend of routing updates, *without* resetting any BGP sessions between neighbors:

```
Router#  clear ip bgp * soft
```

To view a summary of all BGP connections, including the total number of BGP routes and a concise list of neighbors:

```
Router#  show ip bgp summary
```

Part IV

Advanced Routing Functions

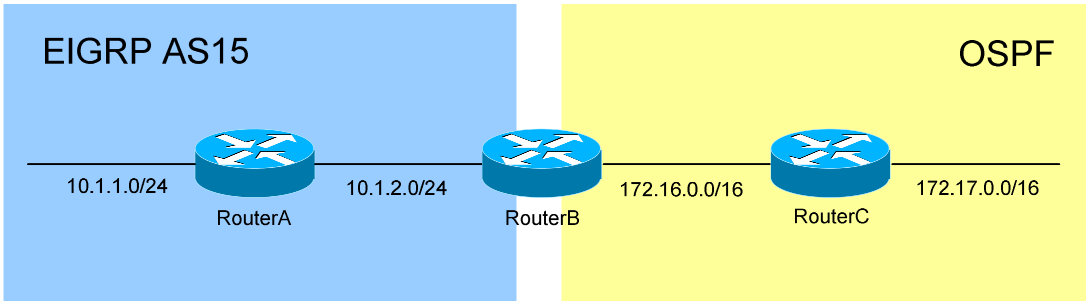
Section 15
- Route Redistribution -

Route Redistribution Basics

It is preferable to employ a single routing protocol in an internetwork environment, for simplicity and ease of management. Unfortunately, this is not always possible, making multi-protocol environments common.

Route Redistribution allows routes from one routing protocol to be advertised into another routing protocol. The routing protocol receiving these redistributed routes usually marks the routes as **external.** External routes are usually *less* preferred than locally-originated routes.

At least one **redistribution point** needs to exist between the two routing domains. This device will actually run *both* routing protocols. Thus, to perform redistribution in the following example, RouterB would require at least one interface in both the EIGRP *and* the OSPF routing domains:



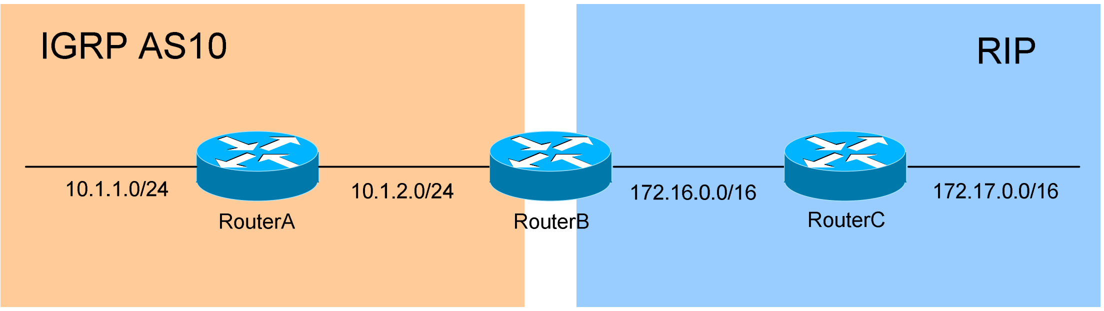
It is possible to redistribute from one routing protocol to the *same* routing protocol, such as between two separate OSPF domains (distinguished by unique *process ID's*). **Static routes** and **connected interfaces** can be redistributed into a routing protocol as well.

Routes will *only* be redistributed if they exist in the routing table. Routes that are simply in a topology database (for example, an EIGRP Feasible Successor), will *never* be redistributed.

Routing **metrics** are a key consideration when performing route redistribution. With the exception of IGRP and EIGRP, each routing protocol utilizes a unique (and thus **incompatible**) metric. Routes redistributed from the *injecting* protocol must be manually (or globally) stamped with a metric that is understood by the *receiving* protocol.

Redistributing into RIP

RIP is a standardized Distance-Vector routing protocol that uses **hop-count** as its distance metric. Consider the following example:



RouterB is our redistribution point between IGRP and RIP. To redistribute all IGRP routes *into* RIP:

```
RouterB(config)#  router rip
RouterB(config-router)#  network 172.16.0.0
RouterB(config-router)#  redistribute igrp 10 metric 2
```

First, the *router rip* process was enabled. Next, RIP was configured to advertise the *network* of *172.16.0.0*/16. Finally, RIP was configured to *redistribute* all *igrp* routes from Autonomous System *10*, and apply a hop-count *metric* of *2* to the redistributed routes. If a metric is *not* specified, RIP will assume a metric of **0**, and **will not advertise** the redistributed routes.

Redistributing into IGRP

IGRP is a Cisco-proprietary Distance-Vector routing protocol that, by default, uses a composite of **bandwidth** and **delay** as its distance metric. IGRP can additionally consider Reliability, Load, and MTU for its metric.

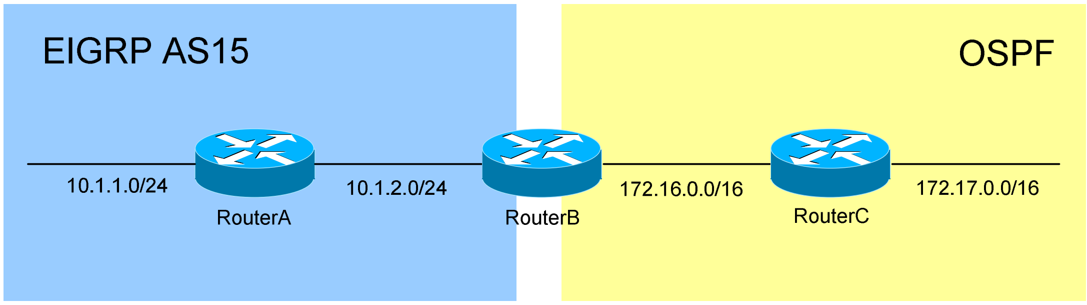
Still using the above example, to redistribute all RIP routes *into* IGRP:

```
RouterB(config)#  router igrp 10
RouterB(config-router)#  network 10.0.0.0
RouterB(config-router)#  redistribute rip metric 10000 1000 255 1 1500
```

First, the *router igrp* process was enabled for Autonomous System *10*. Next, IGRP was configured to advertise the *network* of *10.0.0.0*/8. Finally, IGRP was configured to *redistribute* all *rip* routes, and apply a *metric* of *10000* (bandwidth), *1000* (delay), *255* (reliability), *1* (load), and *1500* (MTU) to the redistributed routes.

Redistributing into EIGRP

EIGRP is a Cisco-proprietary hybrid routing protocol that, by default, uses a composite of **bandwidth** and **delay** as its distance metric. EIGRP can additionally consider Reliability, Load, and MTU for its metric.



To redistribute all OSPF routes *into* EIGRP:

```
RouterB(config)#  router eigrp 15
RouterB(config-router)#  network 10.1.2.0 0.0.0.255
RouterB(config-router)#  redistribute ospf 20 metric 10000 1000 255 1 1500
```

First, the *router eigrp* process was enabled for Autonomous System *15*. Next, EIGRP was configured to advertise the *network* of *10.1.2.0*/24. Finally, EIGRP was configured to *redistribute* all *ospf* routes from process-ID *20*, and apply a *metric* of *10000* (bandwidth), *1000* (delay), *255* (reliability), *1* (load), and *1500* (MTU) to the redistributed routes.

It is possible to specify a *default-metric* for **all** redistributed routes:

```
RouterB(config)#  router eigrp 15
RouterB(config-router)#  redistribute ospf 20
RouterB(config-router)#  default-metric 10000 1000 255 1 1500
```

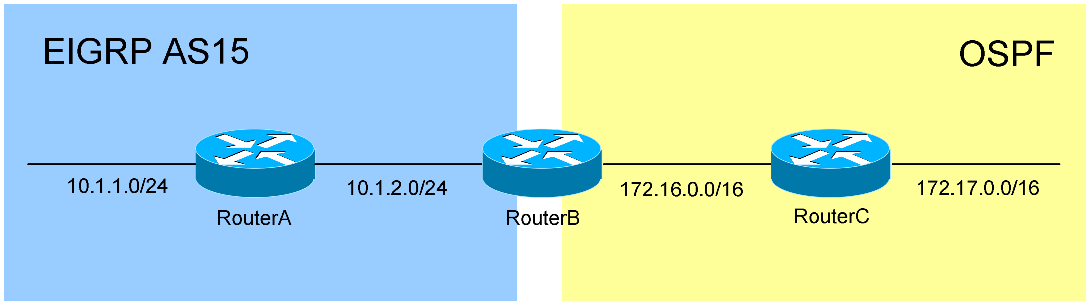
RIP and IGRP also support the *default-metric* command. Though IGRP/EIGRP use only bandwidth and delay by default to compute the metric, it is still necessary to specify *all* five metrics when redistributing. If the *default-metric* or a manual metric is not specified, IGRP/EIGRP will assume a metric of **0**, and **will not advertise** the redistributed routes.

Redistribution will **occur automatically** between IGRP and EIGRP on a router, if both processes are using the same Autonomous System number.

EIGRP, by default, will auto-summarize internal routes unless the *no auto-summary* command is used. However, EIGRP ***will not auto-summarize external routes*** unless a connected or internal EIGRP route exists in the routing table from the same major network of the external routes.

Redistributing into OSPF

OSPF is a standardized Link-State routing protocol that uses **cost** (based on bandwidth) as its link-state metric. An OSPF router performing redistribution automatically becomes an **ASBR**.



To redistribute all EIGRP routes *into* OSPF:

```
RouterB(config)#  router ospf 20
RouterB(config-router)#  network 172.16.0.0 0.0.255.255 area 0
RouterB(config-router)#  redistribute eigrp 15
RouterB(config-router)#  default-metric 30
```

First, the *router ospf* process was enabled with a process-ID of *20*. Next, OSPF was configured to place any interfaces in the *network* of *172.16.0.0*/16 into *area 0*. Then, OSPF will *redistribute* all *eigrp* routes from AS *15*. Finally, a *default-metric* of *30* was applied to all redistributed routes.

If the *default-metric* or a manual metric is not specified for the redistributed routes, a **default metric of 20** will be applied to routes of all routing protocols except for BGP. Redistributed BGP routes will have a **default metric of 1** applied by OSPF.

By default, OSPF will only redistribute **classful routes** into the OSPF domain. To configure OSPF to accept **subnetted networks** during redistribution, the *subnets* parameter must be used:

```
RouterB(config)#  router ospf 20
RouterB(config-router)#  redistribute eigrp 15 subnets
```

Routes redistributed into OSPF are marked **external**. OSPF identifies two types of external routes, **Type-1** (which is *preferred*) and **Type-2** (which is *default*). To change the type of redistributed routes:

```
RouterB(config)#  router ospf 20
RouterB(config-router)#  redistribute eigrp 15 subnets metric-type 1
```

Redistributing Static and Connected Routes

Redistributing **static routes** into a routing protocol is straightforward:

```
RouterB(config)# router eigrp 15
RouterB(config-router)# redistribute static
```

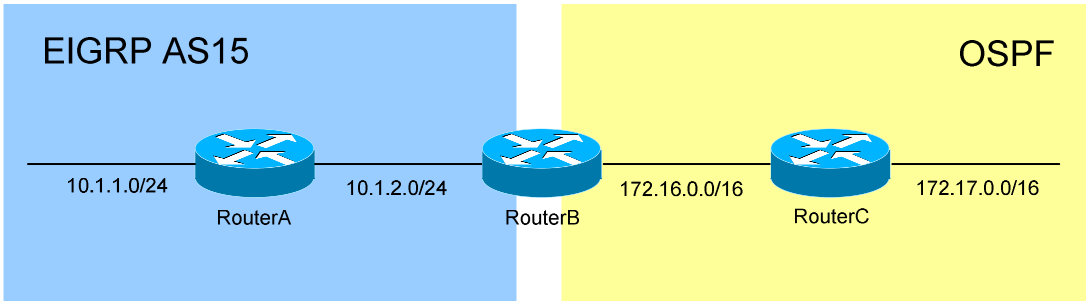
Redistributing networks on **connected** interfaces into a routing protocol is equally straightforward:

```
RouterB(config)# router eigrp 15
RouterB(config-router)# redistribute connected
```

The above commands redistribute *all* connected networks into EIGRP. **Route-maps** can be used to provide more granular control:

```
RouterB(config)# route-map CONNECTED permit 10
RouterB(config-route-map)# match interface fa0/0, fa0/1, s0/0, s0/1

RouterB(config)# router eigrp 15
RouterB(config-router)# redistribute connected route-map CONNECTED
```

Connected networks can be *indirectly* redistributed into a routing protocol. Recall that routes will *only* be redistributed if they exist in the routing table, and consider again the following example:



If RouterB is configured as follows:

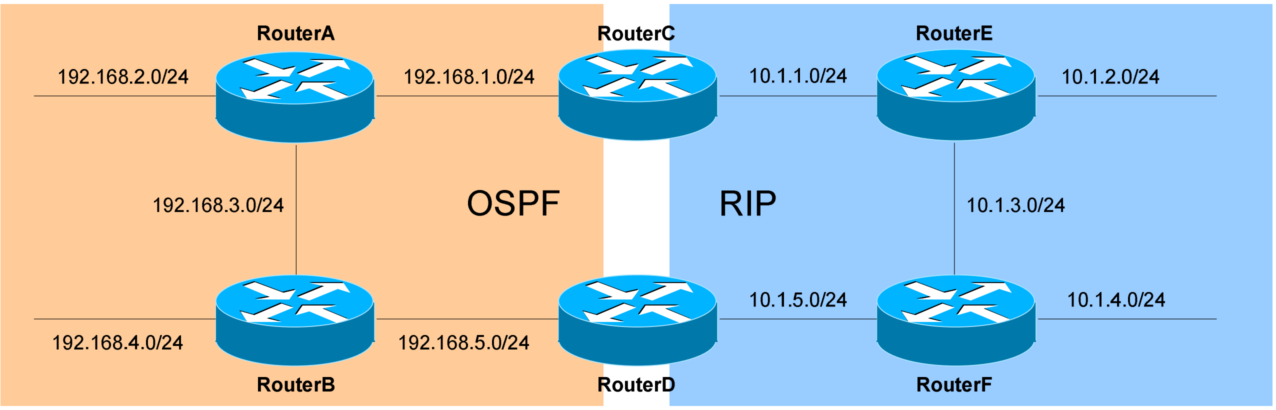
```
RouterB(config)# router eigrp 15
RouterB(config-router)# network 10.1.2.0 0.0.0.255
```

RouterB will *advertise* the 10.1.2.0/24 network to RouterA, but it *will not* have an *EIGRP route* in its routing table for that network, as the network is directly connected.

Despite this, when redistributing EIGRP into OSPF, the 10.1.2.0/24 is *still* injected into OSPF. The *network 10.1.2.0 0.0.0.255* command under the EIGRP process will *indirectly* redistribute this network into OSPF.

Pitfalls of Route Redistribution – Administrative Distance

Route redistribution introduces unique problems when there are multiple points of redistribution. Consider the following diagram:



The first issue is caused by **Administrative Distance (AD)**, which determines which routing protocol is “trusted” the most. By default, OSPF routes have an AD of 110, whereas RIP routes have an AD of 120. Lowest AD is preferred, thus making the OSPF routes the most trusted.

Assume mutual redistribution has been performed on RouterC and RouterD. The following networks will be injected *from* RIP *into* OSPF: 10.1.1.0/24, 10.1.2.0/24, 10.1.3.0/24, 10.1.4.0/24, and 10.1.5.0/24.

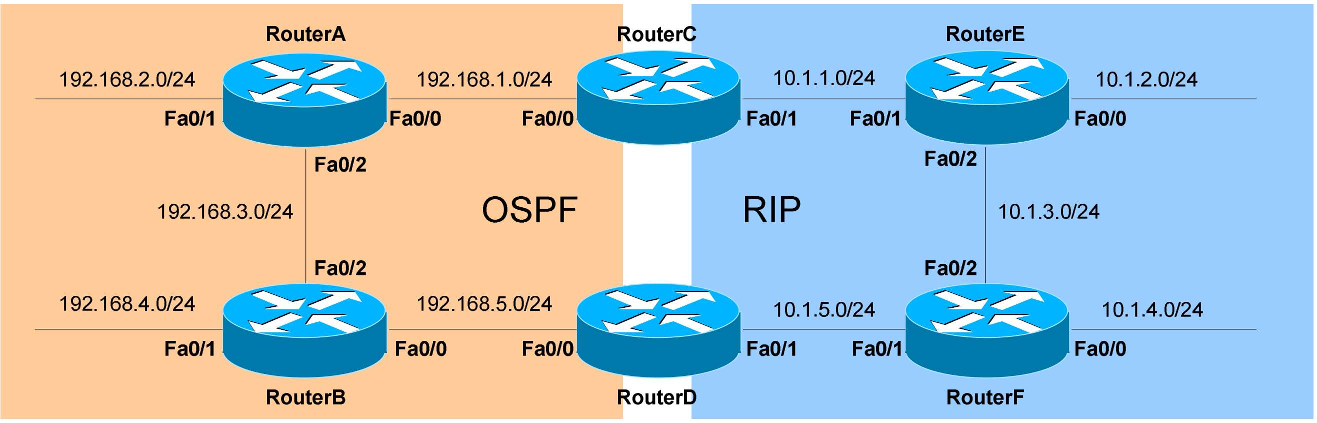
RouterC will eventually receive OSPF routes to the above networks from RouterD, *in addition* to the RIP routes already in its table. Likewise, RouterD will receive OSPF routes to these networks from RouterC.

Because OSPF’s AD is lower than RIP’s, both RouterC and RouterD will prefer the *sub-optimal* path through OSPF to reach the *non-connected* networks. Thus, RouterC will choose the OSPF route for all the 10.x.x.x/24 networks except for 10.1.1.0/24, as it is already directly connected.

This actually creates a **routing loop.** RouterC will prefer the OSPF path through RouterA to reach the 10.x.x.x networks (except for 10.1.1.0/24), and RouterA will likely consider RouterC its shortest path to reach those same networks. Traffic will be continuously looped between these two routers.

Even if RouterC managed to send the traffic through RouterA and RouterB to RouterD, the preferred path to the 10.x.x.x networks for RouterD is *still* through OSPF. Thus, the routing loop is inevitable.

Pitfalls of Route Redistribution – Administrative Distance (continued)



There are two methods to correct this particular routing loop. The first method involves filtering incoming routes using a **distribution-list**, preventing RouterC and RouterD from accepting any routes that originated in RIP from their OSPF neighbors.

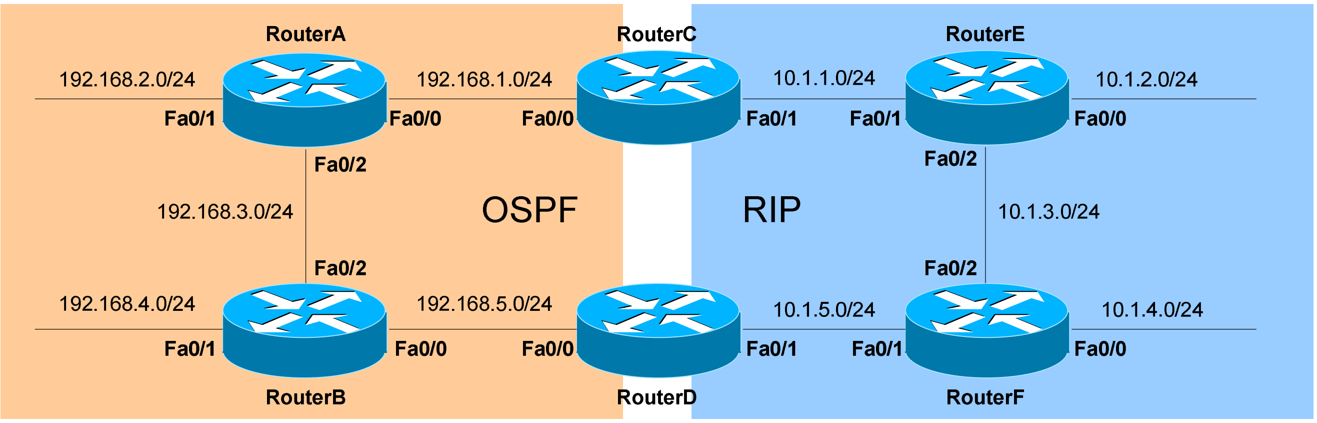
RouterC's configuration would be as follows:

```
RouterC(config)#  access-list 10 deny 10.1.2.0 0.0.0.255
RouterC(config)#  access-list 10 deny 10.1.3.0 0.0.0.255
RouterC(config)#  access-list 10 deny 10.1.4.0 0.0.0.255
RouterC(config)#  access-list 10 deny 10.1.5.0 0.0.0.255
RouterC(config)#  access-list 10 permit any

RouterC(config)#  router ospf 20
RouterC(config-router)#  distribute-list 10 in fastethernet0/0
```

An *access-list* was created that is *deny*ing the RIP networks in question, and permitting all other networks. Under the OSPF process, a *distribute-list* is created for routes coming *in*bound off of the *fastethernet0/0* interface. The *access-list* and *distribute-list* numbers must match. RouterD's configuration would be similar.

This prevents each router from building OSPF routes for the networks that originated in RIP, and thus eliminates the possibility of a loop. However, redundancy is also destroyed – if RouterC's *fa0/1* interface were to fail, it could not choose the alternate path through OSPF.

Pitfalls of Route Redistribution – Administrative Distance (continued)



The second method involves using the ***distance*** command to adjust the AD of specific routes. This can accomplished two ways:

- Lowering the AD of the local RIP-learned routes
- Raising the AD of the external OSPF-learned routes

To force the RIP routes to be preferred, RouterC's configuration would be as follows:

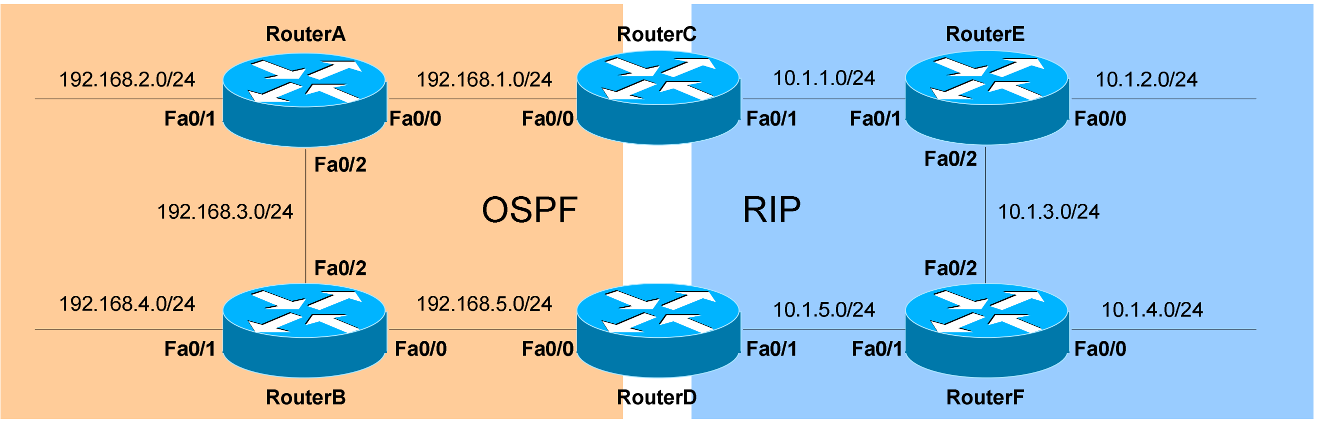
```
RouterC(config)#  access-list 10 permit 10.1.2.0 0.0.0.255
RouterC(config)#  access-list 10 permit 10.1.3.0 0.0.0.255
RouterC(config)#  access-list 10 permit 10.1.4.0 0.0.0.255
RouterC(config)#  access-list 10 permit 10.1.5.0 0.0.0.255
RouterC(config)#  access-list 10 deny any

RouterC(config)#  router rip
RouterC(config-router)#  distance 70 10.1.1.0 0.0.0.255 10
```

An *access-list* was created that is *permit*ting the RIP networks in question, and denying all other networks. Under the RIP process, an administrative *distance* of *70* is applied to updates from routers on the *10.1.1.0* network, for the specific networks matching access-list *10*. RouterD's configuration would be similar.

Thus, the RIP-originated networks will now have a lower AD than the redistributed routes from OSPF. The loop has again been eliminated. Another solution would be to raise the AD of the external OSPF routes. OSPF provides a simple mechanism to accomplish this:

```
RouterC(config)#  router ospf 20
RouterC(config-router)#  distance ospf external 240
```

Pitfalls of Route Redistribution – Route Feedback



A routing loop is only one annoying issue resulting from the above design. **Route feedback** is another problem that must be addressed.

OSPF routes redistributed into RIP on RouterC will eventually reach RouterD, and then be redistributed *again* back into OSPF. This is a basic example of route feedback.

Depending on the metrics used, this could potentially cause RouterB to prefer the route through RouterD (and through the RIP domain), to reach the 192.168.2.0/24 network. This is an obvious example of **suboptimal routing.**

Thus, routes that originated in a routing domain should not to be *re-injected* into that domain. Distribution-lists and the *distance* command can be utilized to accomplish this, but **route tags** may provide a more robust solution.

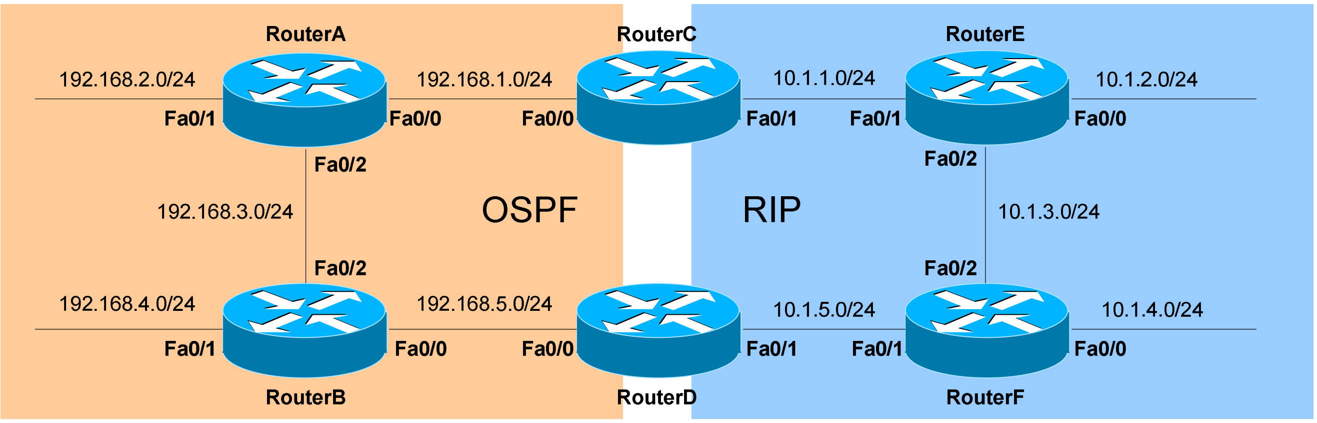
Tagging routes provides a mechanism to both identify and filter those routes further along in the routing domain. A route retains its tag as it passes from router to router. Thus, if a route is tagged when redistributed into RIP on RouterC, that same route can be selectively filtered once it is advertised to RouterD.

Route tags are applied using **route-maps**. Route-maps provide a sequential list of commands, each having a *permit* or *deny* result:

```
RouterC(config)#  route-map OSPF2RIP deny 5
RouterC(config-route-map)#  match tag 33
RouterC(config-route-map)#  route-map OSPF2RIP permit 15
RouterC(config-route-map)#  set tag 44
```

Route-maps are covered in great detail in a separate guide.

Pitfalls of Route Redistribution – Route Feedback (continued)



The full configuration necessary on RouterC would be as follows:

```
RouterC(config)# route-map OSPF2RIP deny 5
RouterC(config-route-map)# match tag 33
RouterC(config-route-map)# route-map OSPF2RIP permit 15
RouterC(config-route-map)# set tag 44

RouterC(config)# router rip
RouterC(config)# redistribute ospf 20 route-map OSPF2RIP


RouterC(config)# route-map RIP2OSPF deny 5
RouterC(config-route-map)# match tag 44
RouterC(config-route-map)# route-map RIP2OSPF permit 15
RouterC(config-route-map)# set tag 33

RouterC(config)# router ospf 20
RouterC(config)# redistribute rip route-map RIP2OSPF
```

Thus, OSPF routes being redistributed into RIP are *set* with a *tag* of *44*. When RIP is redistributed back into OSPF, any route with a *tag* that *match*es *44* is denied.

Similarly, RIP routes being redistributed into OSPF are *set* with a *tag* of *33*. When OSPF is redistributed back into RIP, any route with a *tag* that *match*es 33 is denied.

The net result: routes originating from a routing domain will not redistributed *back* into that domain.

Section 16
- Access Control Lists -

Access Control Lists (ACLs)

Access control lists (ACLs) can be used for two purposes on Cisco devices:

- To **filter** traffic
- To **identify** traffic

Access lists are a set of rules, organized in a rule table. Each rule or line in an access-list provides a condition, either **permit** or **deny**:

- When using an access-list to filter traffic, a *permit* statement is used to “allow” traffic, while a *deny* statement is used to “block” traffic.
- Similarly, when using an access list to identify traffic, a *permit* statement is used to “include” traffic, while a *deny* statement states that the traffic should “not” be included. It is thus interpreted as a **true/false** statement.

Filtering traffic is the primary use of access lists. However, there are several instances when it is necessary to identify traffic using ACLs, including:

- Identifying interesting traffic to bring up an ISDN link or VPN tunnel
- Identifying routes to filter or allow in routing updates
- Identifying traffic for QoS purposes

When filtering traffic, access lists are applied on interfaces. As a packet passes through a router, the top line of the rule list is checked first, and the router continues to go down the list until a match is made. Once a match is made, the packet is either permitted or denied.

There is an implicit ‘deny all’ at the end of all access lists. You don’t create it, and you can’t delete it. Thus, access lists that contain **only deny statements** will **prevent all traffic**.

Access lists are applied either inbound (packets received on an interface, before routing), or outbound (packets leaving an interface, *after* routing). Only one access list **per interface**, **per protocol**, **per direction** is allowed.

More specific and frequently used rules should be at the top of your access list, to optimize CPU usage. New entries to an access list are added to the bottom. You **cannot remove individual lines** from a numbered access list. You must delete and recreate the access to truly make changes. Best practice is to use a text editor to manage your access-lists.

Types of Access Lists

There are two categories of access lists: **numbered** and **named**.

Numbered access lists are broken down into several ranges, each dedicated to a specific protocol:

1–99	IP standard access list
100-199	IP extended access list
200-299	Protocol type-code access list
300-399	DECnet access list
400-499	XNS standard access list
500-599	XNS extended access list
600-699	Appletalk access list
700-799	48-bit MAC address access list
800-899	IPX standard access list
900-999	IPX extended access list
1000-1099	IPX SAP access list
1100-1199	Extended 48-bit MAC address access list
1200-1299	IPX summary address access list
1300-1999	IP standard access list (expanded range)
2000-2699	IP extended access list (expanded range

Remember, individual lines *cannot* be removed from a numbered access list. The entire access list must be deleted and recreated. All new entries to a numbered access list are added to the bottom.

Named access lists provide a bit more flexibility. Descriptive names can be used to identify your access-lists. Additionally, individual lines *can* be removed from a named access-list. However, like numbered lists, all new entries are still added to the bottom of the access list.

There are two common types of named access lists:

- IP standard named access lists
- IP extended named access lists

Configuration of both numbered and named access-lists is covered later in this section.

Wild Card Masks

IP access-lists use **wildcard masks** to determine two things:

1. Which part of an address must match exactly
2. Which part of an address can match any number

This is as opposed to a **subnet mask**, which tells us what part of an address is the network (subnet), and what part of an address is the host. Wildcard masks look like inversed subnet masks.

Consider the following address and wildcard mask:

Address:	172.16.0.0
Wild Card Mask:	0.0.255.255

The above would match any address that begins “172.16.” The last two octets could be anything. How do I know this?

Two Golden Rules of Access Lists:

1. If a bit is set to **0** in a wild-card mask, the corresponding bit in the address must be **matched exactly.**
2. If a bit is set to **1** in a wild-card mask, the corresponding bit in the address can **match any number.** In other words, we “don’t care” what number it matches.

To see this more clearly, we’ll convert both the address and the wildcard mask into binary:

Address:	10101100.00010000.00000000.00000000
Wild Card Mask:	00000000.00000000.11111111.11111111

Any **0** bits in the wildcard mask, indicates that the corresponding bits in the address must be matched exactly. Thus, looking at the above example, we must exactly match the following in the first two octets:

10101100.00010000 = 172.16

Any **1** bits in the wildcard mask indicates that the corresponding bits can be anything. Thus, the last two octets can be any number, and it will still match this access-list entry.

Wild Card Masks (continued)

If wanted to match a **specific address** with a wildcard mask (we'll use an example of 172.16.1.1), how would we do it?

Address:	172.16.1.1
Wild Card Mask:	0.0.0.0

Written out in binary, that looks like:

Address:	10101100.00010000.00000001.00000001
Wild Card Mask:	00000000.00000000.00000000.00000000

Remember what a wildcard mask is doing. A **0** indicates it must match exactly, a **1** indicates it can match anything. The above wildcard mask has all bits set to 0, which means we must match all four octets exactly.

There are actually two ways we can match a host:

- Using a wildcard mask with all bits set to 0 – **172.16.1.1 0.0.0.0**
- Using the keyword "host" – **host 172.16.1.1**

How would we match **all addresses** with a wildcard mask?

Address:	0.0.0.0
Wild Card Mask:	255.255.255.255

Written out in binary, that looks like:

Address:	00000000.00000000.00000000.00000000
Wild Card Mask:	11111111.11111111.11111111.11111111

Notice that the above wildcard mask has all bits set to 1. Thus, each bit can match anything – resulting in the above address and wildcard mask matching all possible addresses.

There are actually two ways we can match all addresses:

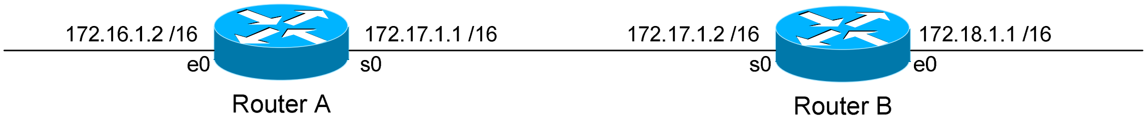
- Using a wildcard mask with all bits set to 1 – **0.0.0.0 255.255.255.255**
- Using the keyword "any" – **any**

Standard IP Access List

access-list *[1-99] [permit | deny] [source address] [wildcard mask] [log]*

Standard IP access-lists are based upon the source host or network IP address, and should be placed closest to the destination network.

Consider the following example:



In order to block network 172.18.0.0 from accessing the 172.16.0.0 network, we would create the following access-list on Router A:

```
Router(config)#  access-list 10 deny 172.18.0.0 0.0.255.255
Router(config)#  access-list 10 permit any
```

Notice the wildcard mask of 0.0.255.255 on the first line. This will match (*deny*) all hosts on the 172.18.x.x network.

The second line uses a keyword of *any*, which will match (*permit)* any other address. Remember that you must have at least one permit statement in your access list.

To apply this access list, we would configure the following on Router A:

```
Router(config)#  int s0
Router(config-if)#  ip access-group 10 in
```

To view all IP access lists configured on the router:

```
Router#  show ip access-list
```

To view what interface an access-list is configured on:

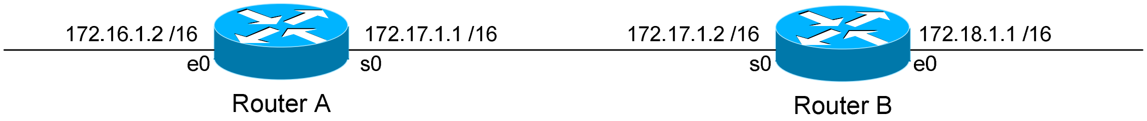
```
Router#  show ip interface
Router#  show running-config
```

Extended IP Access List

access-list *[100-199] [permit | deny] [protocol] [source address] [wildcard mask] [destination address] [wildcard mask] [operator [port]] [log]*

Extended IP access-lists block based upon the source IP address, destination IP address, and TCP or UDP port number. Extended access-lists should be placed closest to the source network.

Consider the following example:



Assume there is a webserver on the 172.16.x.x network with an IP address of 172.16.10.10. In order to block network 172.18.0.0 from accessing anything on the 172.16.0.0 network, EXCEPT for the HTTP port on the web server, we would create the following access-list on Router B:

```
Router(config)#  access-list 101 permit tcp 172.18.0.0 0.0.255.255 host 172.16.10.10 eq 80
Router(config)#  access-list 101 deny ip 172.18.0.0 0.0.255.255 172.16.0.0 0.0.255.255
Router(config)#  access-list 101 permit ip any any
```

The first line allows the 172.18.x.x network access only to port 80 on the web server. The second line blocks 172.18.x.x from accessing anything else on the 172.16.x.x network. The third line allows 172.18.x.x access to anything else.

We could have identified the web server in one of two ways:

```
Router(config)#  access-list 101 permit tcp 172.18.0.0 0.0.255.255 host 172.16.10.10 eq 80
Router(config)#  access-list 101 permit tcp 172.18.0.0 0.0.255.255 172.16.10.10 0.0.0.0  eq 80
```

To apply this access list, we would configure the following on Router B:

```
Router(config)#  int e0
Router(config-if)#  ip access-group 101 in
```

Extended IP Access List Port Operators

In the preceding example, we identified TCP port 80 on a specific host use the following syntax:

```
Router(config)#  access-list 101 permit tcp 172.18.0.0 0.0.255.255 host 172.16.10.10 eq 80
```

We accomplished this using an operator of *eq*, which is short for **equals**. Thus, we are identifying host *172.16.10.10* with a port that *eq*uals 80.

We can use several other operators for port numbers:

eq	Matches a specific port
gt	Matches all ports greater than the port specified
lt	Matches all ports less than the port specified
neq	Matches all ports except for the port specified
range	Match a specific inclusive range of ports

The following will match all ports *greater* than *100*:

```
Router(config)#  access-list 101 permit tcp any host 172.16.10.10 gt 100
```

The following will match all ports *less* than *1024*:

```
Router(config)#  access-list 101 permit tcp any host 172.16.10.10 lt 1024
```

The following will match all ports that do *not equal 443*:

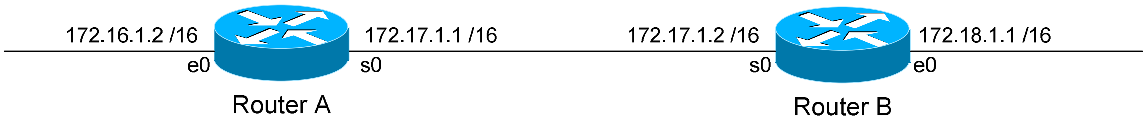
```
Router(config)#  access-list 101 permit tcp any host 172.16.10.10 neq 443
```

The following will match all ports between *80* and *88*:

```
Router(config)#  access-list 101 permit tcp any host 172.16.10.10 range 80 88
```

Access List Logging

Consider again the following example:



Assume there is a webserver on the 172.16.x.x network with an IP address of 172.16.10.10.

We wish to keep track of the number of packets permitted or denied by each line of an access-list. Access-lists have a built-in logging mechanism for such a purpose:

```
Router(config)#  access-list 101 permit tcp 172.18.0.0 0.0.255.255 host 172.16.10.10 eq 80 log
Router(config)#  access-list 101 deny ip 172.18.0.0 0.0.255.255 172.16.0.0 0.0.255.255 log
Router(config)#  access-list 101 permit ip any any log
```

Notice we added an additional keyword *log* to each line of the access-list. When viewing an access-list using the following command:

```
Router#  show access-list 101
```

We will now have a counter on each line of the access-list, indicating the number of packets that were permitted or denied by that line. This information can be sent to a syslog server:

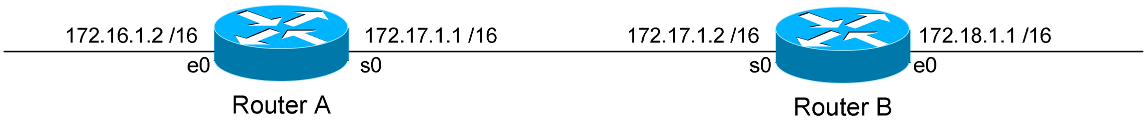
```
Router(config)#  logging on
Router(config)#  logging 172.18.1.50
```

The *logging on* command enables logging. The second *logging* command points to a syslog host at *172.18.1.50.*

We can include more detailed logging information, including the source MAC address of the packet, and what interface that packet was received on. To accomplish this, use the *log-input* argument:

```
Router(config)#  access-list 101 permit ip any any log-input
```

ICMP Access List



Consider this scenario. You've been asked to block anyone from the 172.18.x.x network from "pinging" anyone on the 172.16.x.x network. You want to allow everything else, including all other ICMP packets.

The specific ICMP port that a "ping" uses is **echo**. To block specific ICMP parameters, use an extended IP access list. On Router B, we would configure:

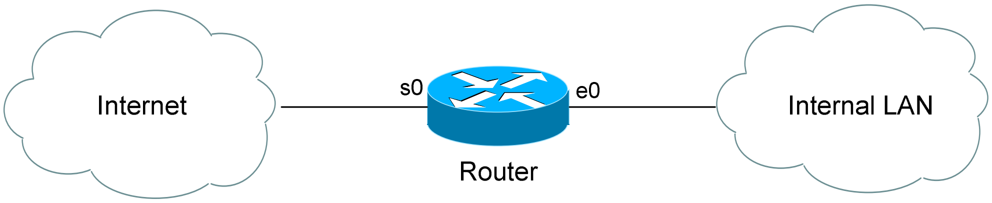
```
Router(config)#  access-list 102 deny icmp 172.18.0.0 0.0.255.255 172.16.0.0 0.0.255.255 echo
Router(config)#  access-list 102 permit icmp 172.18.0.0 0.0.255.255 172.16.0.0 0.0.255.255
Router(config)#  access-list 102 permit ip any any
```

The first line blocks only ICMP echo requests (pings). The second line allows all other ICMP traffic. The third line allows all other IP traffic.

Don't forget to apply it to an interface on Router B:

```
Router(config)#  int e0
Router(config-if)#  ip access-group 102 in
```

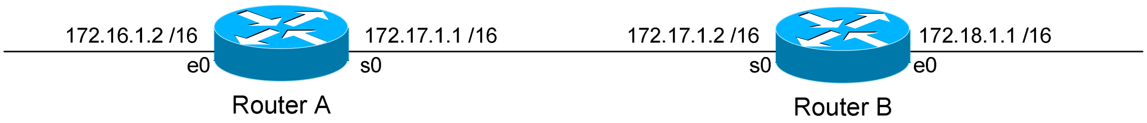
Untrusted networks (such as the Internet) should usually be blocked from pinging an outside router or any internal hosts:



```
Router(config)#  access-list 102 deny icmp any any
Router(config)#  access-list 102 permit ip any any

Router(config)#  interface s0
Router(config-if)#  ip access-group 102 in
```

The above access-list completed disables ICMP on the serial interface. However, this would effectively disable ICMP traffic *in both directions* on the router. Any replies to pings initiated by the Internal LAN would be blocked on the way back in.

Telnet Access List



We can create access lists to restrict telnet access to our router. For this example, we'll create an access list that prevents anyone from the evil 172.18.x.x network from telneting into Router A, but allow all other networks telnet access.

First, we create the access-list on Router A:

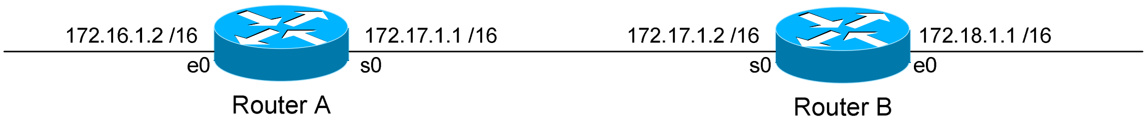
```
Router(config)#  access-list 50 deny 172.18.0.0 0.0.255.255
Router(config)#  access-list 50 permit any
```

The first line blocks the 172.18.x.x network. The second line allows all other networks.

To apply it to Router A's telnet ports:

```
Router(config)#  line vty 0 4
Router(config-line)#  access-class 50 in
```

Named Access Lists



Named access lists provide us with two advantages over numbered access lists. First, we can apply an identifiable name to an access list, for documentation purposes. Second, we can remove individual lines in a named access-list, which is not possible with numbered access lists.

Please note, though we can *remove* individual lines in a named access list, we cannot *insert* individual lines into that named access list. New entries are always placed at the bottom of a named access list.

To create a standard named access list, the syntax would be as follows:

```
Router(config)#  ip access-list standard NAME
Router(config-std-nacl)#  deny 172.18.0.0 0.0.255.255
Router(config-std-nacl)#  permit any
```

To create an extended named access list, the syntax would be as follows:

```
Router(config)#  ip access-list extended NAME
Router(config-ext-nacl)# permit tcp 172.18.0.0 0.0.255.255 host 172.16.10.10 eq 80
Router(config-ext-nacl)#  deny ip 172.18.0.0 0.0.255.255 172.16.0.0 0.0.255.255
Router(config-ext-nacl)#  permit ip any any
```

Notice that the actual configuration of the named access-list is performed in a separate router "mode":

```
Router(config-std-nacl)#

Router(config-ext-nacl)#
```

Time-Based Access-Lists

Beginning with IOS version 12.0, access-lists can be based on the time and the day of the week.

The first step to creating a time-based access-list, is to create a *time-range*:

```
Router(config)#  time-range BLOCKHTTP
```

The above command creates a *time-range* named *BLOCKHTTP*. Next, we must either specify an *absolute* time, or a *periodic* time:

```
Router(config)#  time-range BLOCKHTTP
Router(config-time-range)#  absolute start 08:00 23 May 2006 end 20:00 26 May 2006

Router(config)#  time-range BLOCKHTTP
Router(config-time-range)#  periodic weekdays 18:00 to 23:00
```

Notice the use of military time. The first *time-range* sets an *absolute* time that will *start* from May 23, 2006 at 8:00 a.m., and will *end* on May 26, 2006 at 8:00 p.m.

The second time-range sets a *periodic* time that is always in effect on *weekdays* from 6:00 p.m. to 11:00 p.m.

Only one *absolute* time statement is allowed per time-range, but multiple *periodic* time statements are allowed.

After we establish our time-range, we must reference it in an access-list:

```
Router(config)#  access-list 102 deny any any eq 80 time-range BLOCKHTTP
Router(config)#  access-list 102 permit ip any any
```

Notice the *time-range* argument at the end of the access-list line. This will result in HTTP traffic being blocked, but only during the time specified in the time-range.

Source:
(http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120t/120t1/timerang.htm)

Advanced Wildcard Masks

Earlier in this section, we discussed the basics of wildcard masks. The examples given previously matched one of three things:

- A specific host
- A specific octet(s)
- All possible hosts

It is also possible to match groups or ranges of hosts with wildcard masks. For example, assume we wanted a standard access-list that denied the following hosts:

172.16.1.4
172.16.1.5
172.16.1.6
172.16.1.7

We could create an access-list with four separate lines:

```
Router(config)#  access-list 10 deny 172.16.1.4 0.0.0.0
Router(config)#  access-list 10 deny 172.16.1.5 0.0.0.0
Router(config)#  access-list 10 deny 172.16.1.6 0.0.0.0
Router(config)#  access-list 10 deny 172.16.1.7 0.0.0.0
```

However, it is also possible to match all four addresses in **one** line:

```
Router(config)#  access-list 10 deny 172.16.1.4 0.0.0.3
```

How do I know this is correct? Let's write out the above four addresses, and my wildcard mask in binary:

172.16.1.4:	10101100.00010000.00000001.00000100
172.16.1.5:	10101100.00010000.00000001.00000101
172.16.1.6:	10101100.00010000.00000001.00000110
172.16.1.7:	10101100.00010000.00000001.00000111
Wild Card Mask:	00000000.00000000.00000000.00000011

Notice that the first 30 bits of each of the four addresses are identical. Each begin "*10101100.00010000.00000001.000001*". Since those bits must match exactly, the first 30 bits of our wildcard mask are set to **0**.

Advanced Wildcard Masks (continued)

Notice now that the *only* bits that are different between the four addresses are the last two bits. Not only that, but we use every computation of those last two bits: 00, 01, 10, 11.

Thus, since those last two bits can be anything, the last two bits of our wildcard mask are set to **1**.

The resulting access-list line:

```
Router(config)#  access-list 10 deny 172.16.1.4 0.0.0.3
```

We also could have determined the appropriate address and wildcard mask by using AND/XOR logic.

To determine the address, we perform a logical **AND** operation:

1. If all bits in a column are set to **0**, the corresponding address bit is **0**
2. If all bits in a column are set to **1,** the corresponding address bit is **1**
3. If the bits in a column are a mix of **0**'s and **1's**, the corresponding address bit is a **0.**

Observe:

172.16.1.4:	10101100.00010000.00000001.00000100
172.16.1.5:	10101100.00010000.00000001.00000101
172.16.1.6:	10101100.00010000.00000001.00000110
172.16.1.7:	10101100.00010000.00000001.00000111
Result:	10101100.00010000.00000001.00000100

Our resulting address is **172.16.1.4**. This gets us half of what we need.

Advanced Wildcard Masks (continued)

To determine the wildcard mask, we perform a logical **XOR** (exclusive OR) operation:

1. If all bits in a column are set to **0**, the corresponding wildcard bit is **0**
2. If all bits in a column are set to **1,** the corresponding wildcard bit is **0**
3. If the bits in a column are a mix of **0**'s and **1's**, the corresponding wildcard bit is a 1**.**

Observe:

172.16.1.4:	10101100.00010000.00000001.00000100
172.16.1.5:	10101100.00010000.00000001.00000101
172.16.1.6:	10101100.00010000.00000001.00000110
172.16.1.7:	10101100.00010000.00000001.00000111
Result:	00000000.00000000.00000000.00000011

Our resulting wildcard mask is **0.0.0.3**. Put together, we have:

```
Router(config)#  access-list 10 deny 172.16.1.4 0.0.0.3
```

Please Note: We can determine the number of addresses a wildcard mask will match by using a simple formula:

$$2^n$$

Where "n" is the number of bits set to **1** in the wildcard mask. In the above example, we have two bits set to 1, which matches exactly **four addresses** ($2^2 = 4$).

There *will* be occasions when we cannot match a range of addresses in one line. For example, if we wanted to deny 172.16.1.4-6, instead of 172.16.1.4-7, we would need two lines:

```
Router(config)#  access-list 10 permit 172.16.1.7 0.0.0.0
Router(config)#  access-list 10 deny 172.16.1.4 0.0.0.3
```

If we didn't include the first line, the second line would have denied the 172.16.1.7 address. Always remember to use the above formula (2^n) to ensure your wildcard mask doesn't match more addresses than you intended (often called overlap).

Advanced Wildcard Masks (continued)

Two more examples. How would we deny all **odd** addresses on the 10.1.1.x/24 subnet in one access-list line?

```
Router(config)#  access-list 10 deny 10.1.1.1 0.0.0.254
```

Written in binary:

10.1.1.1:	00001010.00000001.00000001.00000001
Wild Card Mask:	00000000.00000000.00000000.11111110

What would the result of the above wildcard mask be?

1. The first three octets must match exactly.
2. The last bit in the fourth octet must match exactly. Because we set this bit to **1** in our address, every number this matches will be **odd**.
3. All other bits in the fourth octet can match any number.

Simple, right? How would we deny all **even** addresses on the 10.1.1.x/24 subnet in one access-list line?

```
Router(config)#  access-list 10 deny 10.1.1.0 0.0.0.254
```

Written in binary:

10.1.1.0:	00001010.00000001.00000001.00000000
Wild Card Mask:	00000000.00000000.00000000.11111110

What would the result of the above wildcard mask be?

4. The first three octets must match exactly.
5. The last bit in the fourth octet must match exactly. Because we set this bit to **0** in our address, every number this matches will be **even**.
6. All other bits in the fourth octet can match any number.

Section 17
- Route Filtering and Route-Maps -

Prefix-Lists

Prefix-lists are used to match *routes* as opposed to *traffic*. Two things are matched:

- The **prefix** (the *network* itself)
- The **prefix-length** (the length of the *subnet mask*)

Consider the following prefix-list:

```
Router(config)#  ip prefix-list MYLIST 10.1.1.0/24
```

The above *prefix-list* matches the *10.1.1.0/24* network exactly. It will not match 10.1.0.0/16, or 10.1.1.4/30.

A **range** of prefix-lengths can be specified:

```
Router(config)#  ip prefix-list MYLIST 10.1.1.0/24 le 30
Router(config)#  ip prefix-list MYLIST 10.1.1.0/24 ge 26 le 30
```

The first command dictates that the first *24* bits of the prefix must match (meaning, the prefix *must* begin 10.1.1), and the subnet mask must be less than or equal to *30* bits.

The second command dictates again that the first *24* bits of the prefix must match, and the subnet mask must be between 26 to 30 bits (or equal to).

To match all prefixes:

```
Router(config)#  ip prefix-list MYLIST 0.0.0.0/0 le 32
```

To view information about all prefix lists:

```
Router#  show ip prefix-list detail
```

Distribute-Lists

Distribute-lists are used to filter routing updates, either *in*bound or *out*bound. Routes must first be matched using an access-list or prefix-list, and then applied using a distribute-list under the routing process:

To use an *access-list* to identify routes:

```
Router(config)# access-list 10 permit ip 172.16.0.0 0.0.255.255

Router(config)# router rip
Router(config-router)# distribute-list 10 in serial0/0
```

The above *distribute-list* will control routes sent *in*bound on *serial0/0*. Specifically, the referenced *access-list* will only *permit* routes matching *172.16* in the first two octets.

To use a *prefix-list* to identify routes:

```
Router(config)# ip prefix-list MYLIST 10.1.0.0/16

Router(config)# router rip
Router(config-router)# distribute-list prefix MYLIST out fastethernet0/0
```

The above *distribute-list* will control routes sent *out*bound on *fastethernet0/0*. Specifically, the referenced *prefix-list* will only match the **exact** *10.1.0.0/16* route.

Route-Maps

Route-maps are advanced access-lists that serve several functions on IOS devices, including (but not limited to):

- Controlling **redistribution** between routing protocols.
- Adjusting the **attributes** of routes (especially for BGP).
- Implementing **Policy Based Routing (PBR).**

As with access-lists, route-maps are organized as a sequential set of rules or **statements**, each with a **permit** or **deny** condition. However, access-lists can merely permit or deny traffic, while a route-map can additionally modify or perform a specific action on traffic.

Route-maps follow a very simple logic:

- Traffic must be first **matched**, based on specified criteria.
- A particular attribute or action is **set** on the matched traffic.

Each statement in a route-map is assigned a **sequence number**, and contains a series of *match* and *set* statements. The route-map is parsed from the lowest sequence number to the highest, and will stop once a match is found.

The following demonstrates the syntax of a route-map:

```
Router(config)#  access-list 1 permit 10.1.1.0 0.0.0.255

Router(config)#  route-map MYMAP permit 10
Router(config-route-map)#  match ip address 1
Router(config-route-map)#  set ip next-hop 192.168.1.1
```

First, an *access-list* was created that matched traffic from 10.1.1.0/24.

Then, a *route-map* called *MYMAP* was created, and assigned a sequence number of *10* with a *permit* condition. If a route-map contains multiple statements, the sequence number dictates the order of those statements.

The route-map will then *match* any traffic listed in access-list *1*. Notice that the syntax to call an access-list *match ip address*.

Finally, the desired attributed is *set* to this traffic. In this instance, the *ip next hop* attribute has been modified to *192.168.1.1*.

Route-Maps (continued)

A single route-map statement can contain multiple *match* commands:

```
Router(config)#  route-map MYMAP permit 10
Router(config-route-map)#  match ip address 1 2 3
```

The above line would match traffic in access-list *1*, **or** access-list *2*, **or** access-list *3*. Thus, when match criteria is contained within a single line, a logical OR is applied.

However, if match criteria is specified on separate lines:

```
Router(config-route-map)#  match ip address 1
Router(config-route-map)#  match ip address 2
```

Then the traffic must match access-list *1* **and** access-list *2* (a logical AND). **Remember this distinction!**

If **no match criteria** is specified, **all traffic is matched!**

Additionally, a single route-map statement can contain multiple *set* commands:

```
Router(config)#  route-map MYMAP permit 10
Router(config-route-map)#  match ip address 1
Router(config-route-map)#  set weight 50
Router(config-route-map)#  set local-preference 200
```

Any traffic matching access-list *1* will have both *set* attributes applied.

There is an implicit **deny any** statement at the bottom of every route-map. The impact of this deny any statement is dependent on the function of the access-list:

- If using a route-map for *policy-based routing* or adjusting *attributes*, any routes/traffic not specifically matched will **remain unchanged.**
- If using a route-map for *redistribution*, any routes not specifically matched (and permitted) will **not be redistributed**.

(Reference: http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a008047915d.shtml)

Route-Map Criteria

The following are example attributes that can be **match**ed by a route-map:

- *match ip address*
- *match interface*
- *match ip address prefix-list*
- *match ip next-hop*
- *match metric*
- *match route-type*
- *match tag*
- *match community*

```
Router(config)#  route-map MYMAP permit 10
Router(config-route-map)#  match ip address 1
Router(config-route-map)#  match interface serial0/0
Router(config-route-map)#  match ip address prefix-list MYLIST
Router(config-route-map)#  match ip next-hop 192.168.1.2
Router(config-route-map)#  match metric 40
Router(config-route-map)#  match route-type internal
Router(config-route-map)#  match tag 33
Router(config-route-map)#  match community 123
```

The following are example attributes that can be **set** by a route-map:

- *set interface*
- *set ip next-hop*
- *set metric*
- *set tag*
- *set community*
- *set local-preference*
- *set weight*
- *set ip precedence*

```
Router(config)#  route-map MYMAP permit 10
Router(config-route-map)#  set interface fastethernet0/1
Router(config-route-map)#  set ip next-hop 10.1.1.1
Router(config-route-map)#  set metric 200
Router(config-route-map)#  set tag 44
Router(config-route-map)#  set community 321
Router(config-route-map)#  set local-preference 250
Router(config-route-map)#  set weight 300
Router(config-route-map)#  set ip precedence 2
```

Route-Map Examples

The following route-map is applying a BGP attribute to a specific route:

```
Router(config)#  access-list 1 permit 10.1.1.0 0.0.0.255

Router(config)#  route-map MYMAP permit 10
Router(config-route-map)#  match ip address 1
Router(config-route-map)#  set metric 100
Router(config-route-map)#  route-map MYMAP permit 20

Router(config)#  router bgp 100
Router(config-router)#  neighbor 172.16.1.1 route-map MYMAP out
```

The following route-map is controlling routes being redistributed between routing protocols:

```
Router(config)#  access-list 1 deny 192.168.1.0 0.0.255
Router(config)#  access-list 1 deny 192.168.2.0 0.0.255
Router(config)#  access-list 1 permit any

Router(config)#  route-map MYMAP permit 10
Router(config-route-map)#  match ip address 1
Router(config-route-map)#  set tag 150

Router(config)#  router ospf 1
Router(config-router)#  redistribute eigrp 10 metric 3 subnets route-map MYMAP
```

The following route-map is manipulating inbound traffic on a specific interface:

```
Router(config)#  access-list 1 permit 10.1.1.0 0.0.0.255

Router(config)#  route-map MYMAP permit 10
Router(config-route-map)#  match ip address 1
Router(config-route-map)#  set ip next-hop 192.168.1.1

Router(config)#  interface s0/0
Router(config-if)#  ip policy route-map MYMAP
```

Section 18
- Multicast -

Types of "packets"

Three types of packets can exist on an IPv4 network:

Unicast – A packet sent from one host to only one other host. A hub will forward a unicast out all ports. If a switch has a table entry for the unicast's MAC address, it will forward it out only the appropriate port.

Broadcast – A packet sent from one host to *all* hosts on the IP subnet. Both hubs and switches will forward a broadcast out all ports. By definition, a router will not forward a broadcast from one segment to another.

Multicast – A packet sent from one host to a specific group of hosts. Switches, *by default*, will forward a multicast out all ports. A router, *by default,* will not forward a multicast from one segment to another.

Multicast Concepts

Remember, a multicast is a packet sent from one computer to a **group** of hosts. A host must **join** a **multicast group** in order to accept a multicast.

Joining a multicast group can be accomplished statically or dynamically.

Multicast traffic is generally sent *from* a multicast server, *to* multicast clients. Very rarely is a multicast packet sent back from a client to the server.

Multicasts are utilized in a wide range of applications, most notably voice or video systems that have one source "serving" out data to a very specific group of clients.

The key to configuring multicast is to ensure *only* the hosts that require the multicast traffic actually receive it.

Multicast Addressing

IPv4 addresses are separated into several "classes."

Class A: 1.1.1.1 – 127.255.255.255
Class B: 128.0.0.0 – 191.255.255.255
Class C: 192.0.0.0 – 223.255.255.255
Class D: 224.0.0.0 – 239.255.255.255

Class D addresses have been reserved for multicast. Within the Class D address space, several ranges have been reserved for specific purposes:

- **224.0.0.0 – 224.0.0.255 –** Reserved for routing and other network protocols, such as OSPF, RIP, VRRP, etc.
- **224.0.1.0 – 238.255.255.255 –** Reserved for "public" use, can be used publicly on the Internet. Many addresses in this range have been reserved for specific applications
- **239.0.0.0 – 239.255.255.255 –** Reserved for "private" use, and cannot be routed on the Internet.

The following outlines several of the most common multicast addresses reserved for routing protocols:

- 224.0.0.1 – all hosts on this subnet
- 224.0.0.2 – all routers on this subnet
- 224.0.0.5 – all OSPF routers
- 224.0.0.6 – all OSPF Designated routers
- 224.0.0.9 – all RIPv2 routers
- 224.0.0.10 – all IGRP routers
- 224.0.0.12 – DHCP traffic
- 224.0.0.13 – all PIM routers
- 224.0.0.19-21 – ISIS routers
- 224.0.0.22 – IGMP traffic
- 224.0.1.39 – Cisco RP Announce
- 224.0.1.40 – Cisco RP Discovery

Multicast MAC Addresses

Unfortunately, there is no ARP equivalent protocol for multicast addressing. Instead, a reserved range of MAC addresses were created for multicast IPs. All multicast MAC addresses begin with:

0100.5e

Recall that the first six digits of a MAC address identify the vendor code, and the last 6 digits identify the specific host address. To complete the MAC address, the last **23 bits** of the multicast IP address are used.

For example, consider the following multicast IP address and its binary equivalent:

224.65.130.195 = 11100000.01000001.10000010.11000011

Remember that a MAC address is 48 bits long, and that a multicast MAC must begin with *0100.5e*. In binary, that looks like:

00000001.00000000.01011110.0

Add the last 23 bits of the multicast IP address to the MAC, and we get:

00000001.00000000.01011110.01000001.10000010.11000011

That should be exactly 48 bits long. Converting that to Hex format, our full MAC address would be:

0100.5e41.82c3

How did I convert this to Hex? Remember that hexadecimal is Base 16 mathematics. Thus, to represent a single hexadecimal digit in binary, we would need 4 bits ($2^4 = 16$). So, we can break down the above binary MAC address into groups of four bits:

Binary	0000	0001	0000	0000	0101	1110	0100	0001	1000	0010	1100	0011
Decimal	0	1	0	0	5	14	4	1	8	2	12	3
Hex	0	1	0	0	5	e	4	1	8	2	c	3

Hence the MAC address of *0100.5e41.82c3*.

Multicast MAC Addresses (continued)

Ready for some more math, you binary fiends?

Calculate what the multicast MAC address would be for the following IP addresses:

225.2.100.15 = 11100001.00000010.01100100.00001111
231.130.100.15 = 11100111.10000010.01100100.00001111

Remember that all multicast MACs begin with:

0100.5e = 00000001.00000000.01011110.0

So, add the last 23 digits of each of the above IP addresses to the MAC address, and we get:

225.2.100.15 = 00000001.00000000.01011110.00000010.01100100.00001111
231.130.100.15 = 00000001.00000000.01011110.00000010.01100100.00001111

In Hex, that would be:

225.2.100.15 = 0100.5e02.640f
231.130.100.15 = 0100.5e02.640f

Wait a second…. That's the *exact* same multicast MAC address, right? Double-checking our math, we see that it's perfect.

Believe it or not, each multicast MAC address can match **32 multicast IP addresses**, because we're only taking the last 23 bits of our IP address.

We already know that all multicast IP addresses MUST begin 1110. Looking at the 225.2.100.15 address in binary:

1110***0001.0***0000010.01100100.00001111

That leaves 5 bits in between our starting 1110, and the last 23 bits of our IP. Those 5 bits could be anything, and the multicast MAC address would be the same. Because 2^5 = 32, there are 32 multicast IP's per multicast MAC.

According to the powers that be, the likelihood of two multicast systems utilizing the same multicast MAC is rare. The worst outcome would be that hosts joined to either multicast system would receive multicasts from both.

Multicasts and Routing

A router, by default, will drop multicast traffic, unless a **Multicast routing protocol** is utilized. Multicast routing protocols ensure that data sent from a multicast source are received by (and *only* by) its corresponding multicast clients.

Several multicast routing protocols exist, including:

- **Protocol Independent Multicast** (PIM)
- **Multicast OSPF** (MOSPF)
- **Distance Vector Multicast Routing Protocol** (DVMRP)
- **Core-Based Trees** (CBT)

Multicast routing must be enabled globally on a Cisco router or switch, before it can be used:

```
Switch(config)#  ip multicast-routing
```

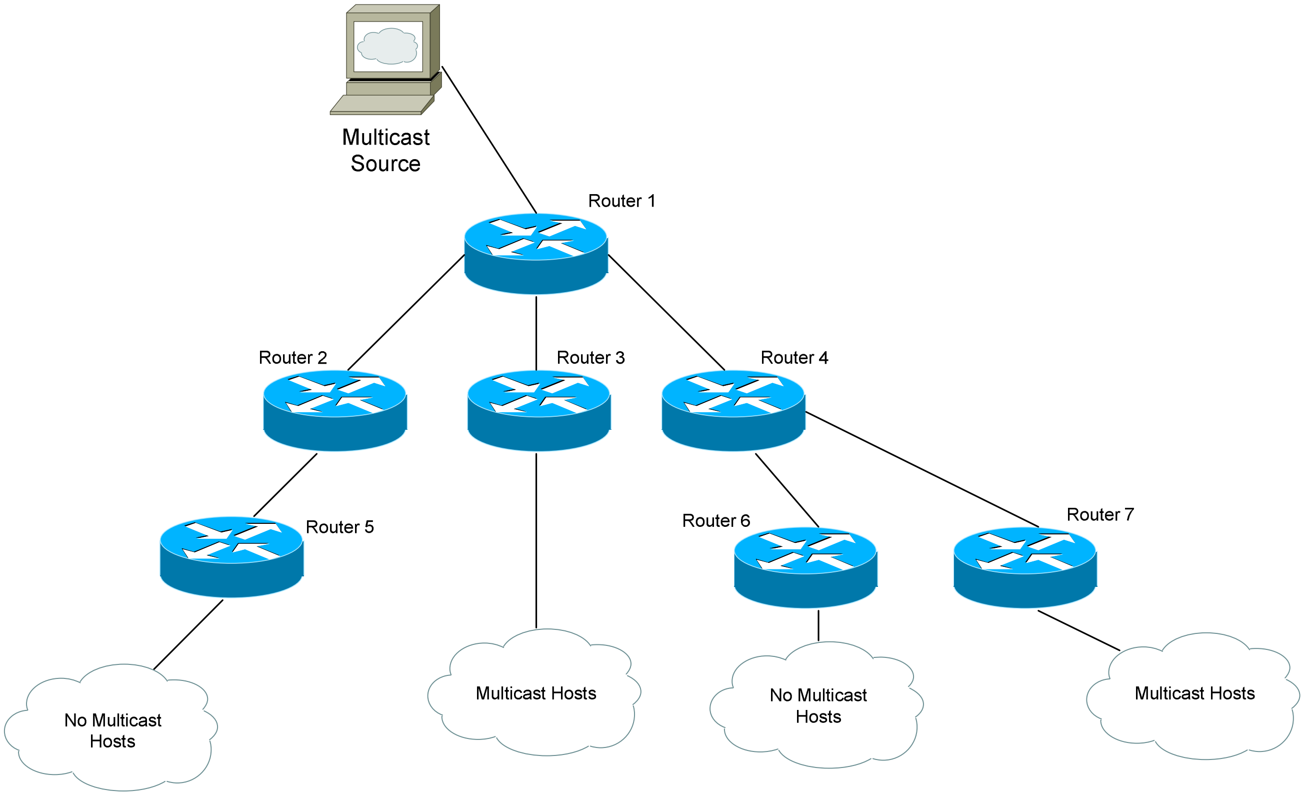
Multicast Path Forwarding

Normally, routers build routing tables that contain **destination** addresses, and route packets *towards* that destination. With multicast, routers are concerned with routing packets *away* from the multicast source. This concept is called **Reverse Path Forwarding (RPF)**.

Multicast routing protocols build tables that contain several elements:

- The multicast **source**, and its associated multicast address (labeled as "**S,G**", or "**Source,Group")**
- **Upstream** interfaces that point *towards* the source
- **Downstream** interfaces that point *away* from the source towards multicast hosts.

Multicast Path Forwarding Example



A router interface will not be designated as a **downstream** interface unless multicast hosts actually exist downstream. In the above example, no multicast hosts exist downstream of Router 5.

In fact, because no multicast hosts exist downstream of Router 1 towards Router 2, no multicast traffic for this multicast group will be forwarded down that path. Thus, Router 1's interface connecting to Router 2 will not become a downstream port.

This pruning allows for efficient use of bandwidth. No unnecessary traffic is sent down a particular link. This "map" of which segments contain multicast hosts is called the **multicast tree**. The multicast tree is dynamically updated as hosts join or leave the multicast group (otherwise known as **pruning** the branches).

By designating upstream and downstream interfaces, the multicast tree remains loop-free. No multicast traffic should ever be sent back upstream *towards* the multicast source.

Internet Group Management Protocol (IGMP)

Remember, multicast works by having a **source** send data to a specific set of **clients** that belong to the same multicast **group.** The multicast group is configured (or assigned) a specific multicast address.

The multicast clients need a mechanism to *join* multicast groups. **Internet Group Management Protocol (IGMP)** allows clients to send "requests" to multicast-enabled routers to join a multicast group.

IGMP only handles group membership. To actually route multicast data to a client, a multicast routing protocol is required, such as PIM or DVMRP.

Three versions of IGMP exist, **IGMPv1, IGMPv2,** and **IGMPv3**.

IGMPv1 routers send out a "query" every **60 seconds** to determine if any hosts need access to a multicast server. This query is sent out to the 224.0.0.1 address (i.e., all hosts on the subnet). Interested hosts must reply with a **Membership Report** stating what multicast group they wish to join.

Unfortunately, IGMPv1 does not allow hosts to dynamically "leave" a group. Instead, if no Membership Reports are received after 3 times the query interval, the router will flush the hosts out of its IGMP table.

IGMPv2 adds additional functionality. Queries can be sent out either as **General Queries** (224.0.0.1) or **Group-Specific Queries** (only sent to specific group members). Additionally, hosts can send a **Leave Group** message to IGMPv2 routers, to immediately be flushed out of the IGMP table. Thus, IGMPv2 allows the multicast tree to by updated more efficiently.

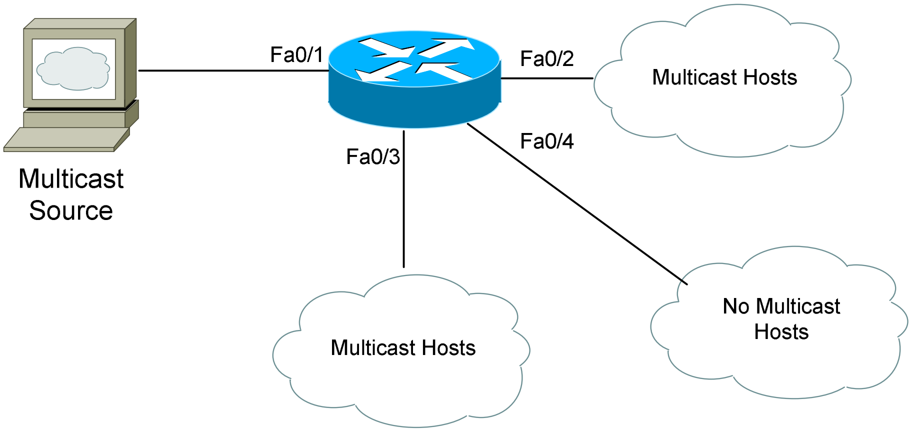
All versions of IGMP elect one router to be the **Designated Querier** for that subnet. The router with the **lowest IP address** becomes Designated.

IGMPv1 is not compatible with IGMPv2. If any IGMPv1 routers exist on the network, *all* routers must operate in IGMPv1 mode.

Cisco IOS version 11.1 and later support IGMPv2 by default.

IGMPv3 enhances v2 by supporting **source-based filtering** of multicast groups. Essentially, when a host responds to an IGMP query with a Membership Report, it can specifically identify which *sources* within a multicast group to join (or even *not* join).

IGMP Example



In the above example, assume the router is using IGMPv2. Interface fa0/1 points towards the multicast source, and thus becomes the **upstream** interface.

Initially, the router will sent out Group Specific Queries out all non-upstream interfaces. Any multicast hosts will respond with a Membership Report stating what multicast group they wish to join.

Interfaces fa0/2 and fa0/3 will become **downstream** interfaces, as they contain multicast hosts. No multicast traffic will be sent out fa0/4.

If all multicast hosts leave the multicast group off of interface fa0/2, it will be removed from the multicast tree. If a multicast host is ever added off of interface fa0/4, it will become a downstream interface.

IGMP Configuration

No configuration is required to enable IGMP, except to enable IP multicast routing (*ip multicast-routing*). We can change the version of IGMP running on a particular interface (by default, it is Version 2):

```
Switch(config-if)#  ip igmp version 1
```

To view which multicast groups the router is aware of:

```
Switch#  show ip igmp groups
```

We can join a router interface to a specific multicast group (forcing the router to respond to ICMP requests to this multicast group):

```
Switch(config-if)#  ip igmp join-group 226.1.5.10
```

WE can also simply force a router interface to *always* forward the traffic of a specific multicast group out an interface:

```
Switch(config-if)#  ip igmp static-group 226.1.5.10
```

We can also restrict which multicast groups a host, off of a particular interface, can join:

```
Switch(config)#  access-list 10 permit 226.1.5.10
Switch(config)#  access-list 10 permit 226.1.5.11

Switch(config-if)#  ip igmp access-group 10
```

Protocol Independent Multicast (PIM)

While IGMP concerns itself with allowing multicast hosts to join multicast groups, **Protocol Independent Multicast (PIM)** is a multicast routing protocol that is concerned about *getting* the multicast data to its destination (or, more accurately, *taking* the data away from the multicast source).

PIM is also responsible for creating the multicast tree, and "pruning" the tree so that no traffic is sent unnecessarily down a link.

PIM can operate in three separate modes:

- **PIM Dense Mode (**PIM-DM**)**
- **PIM Sparse Mode** (PIM-SM)
- **PIM Sparse-Dense Mode** (PIM-SM-DM, Cisco proprietary)

The key difference between PIM Dense and Sparse Mode is how the multicast tree is created. With **PIM Dense Mode**, all networks are flooded with the multicast traffic from the source. Afterwards, networks that don't need the multicast are pruned off of the tree. The network that contains the multicast source becomes the "root" of the multicast network.

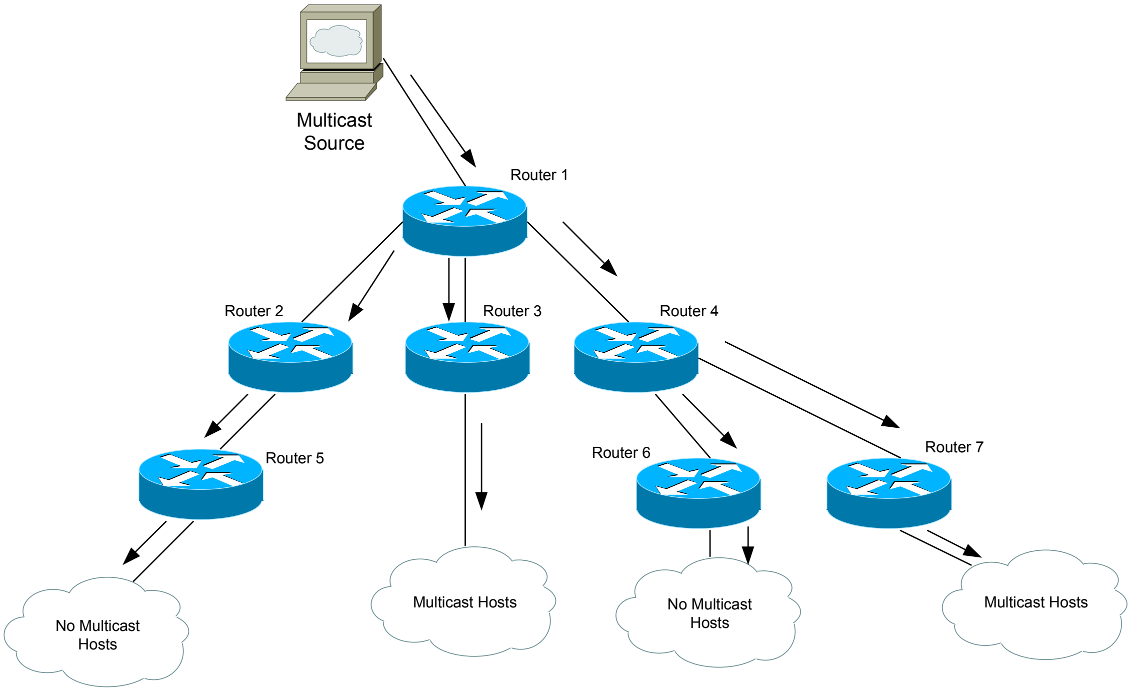
With **PIM Sparse Mode**, no "flooding" occurs. Only networks that contain "requesting" multicast hosts are added to the multicast tree. A centralized PM router, called the **Rendezvous Point (RP),** is elected to be the "root" router of the multicast tree. PIM routers operating in Sparse Mode build their tree towards the RP, instead of towards the multicast source. The RP allows multiple multicast "sources" to utilize the same multicast tree.

PIM Sparse-Dense Mode allows either Sparse or Dense Mode to be used, depending on the multicast group. Any group that points to an RP utilizes Sparse Mode. PIM Sparse-Dense Mode is Cisco proprietary.

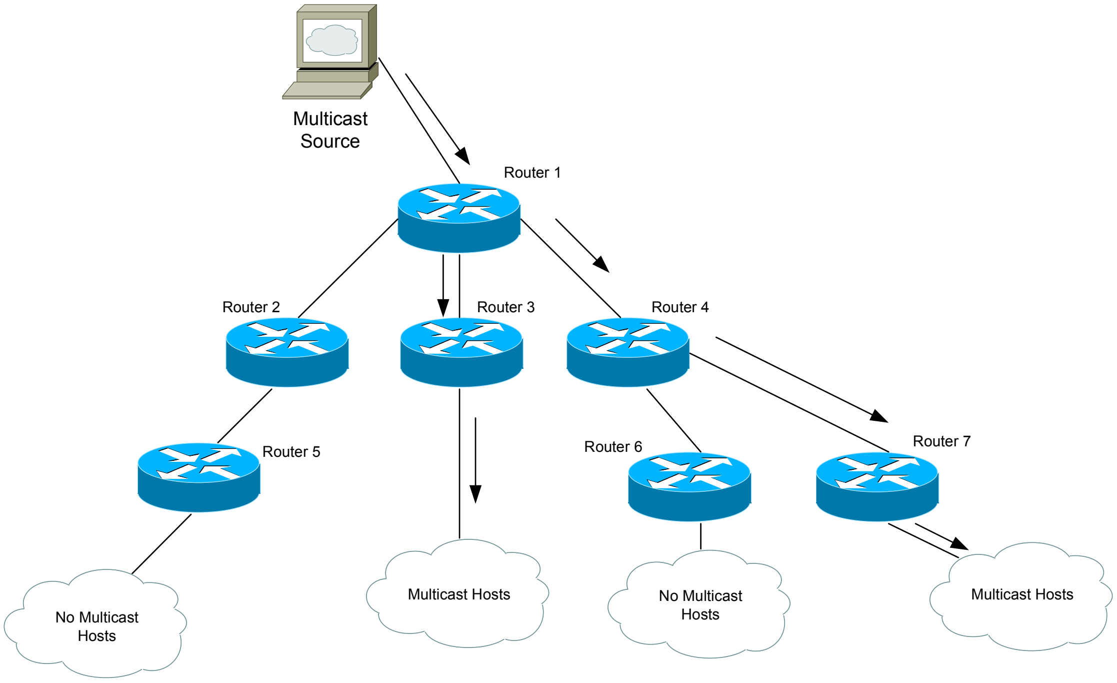
Consider these key points:

- **Dense Mode** should be used when a **large number** of multicast hosts exist across the internetwork. The "flooding" process allows for a quick creation of the multicast tree, at the expense of wasting bandwidth.
- **Sparse Mode** should be used when only a **limited number** of multicast hosts exist. Because hosts must explicitly join before that network segment is added to the multicast tree, bandwidth is utilized more efficiently.

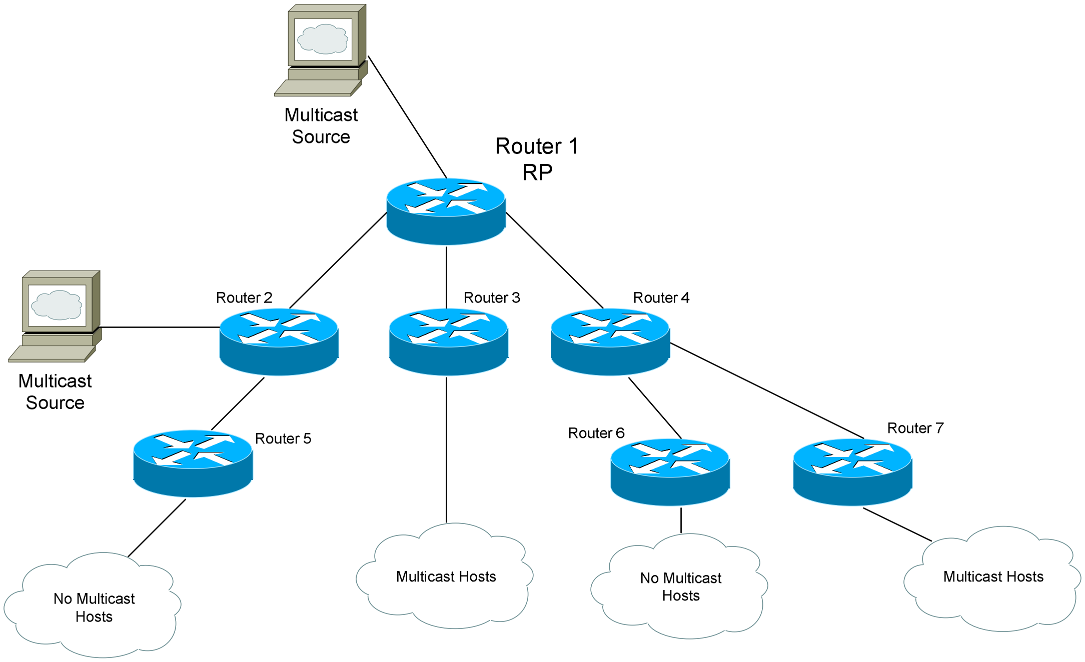
PIM Dense Mode Example



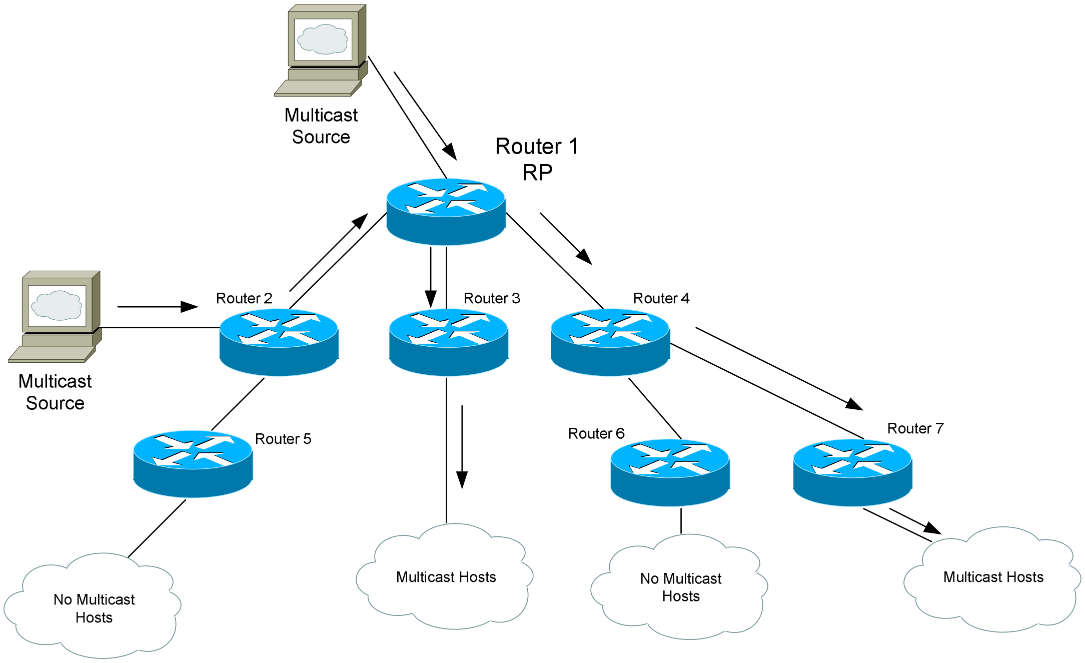
Consider the above example. When PIM routers operate in **Dense Mode**, all segments of the multicast tree are flooded initially. Eventually, "branches" that do not require the multicast traffic are pruned off:



PIM Sparse Mode Example



When PIM routers operate in **Sparse Mode**, multicast traffic is *not* initially flooded throughout the entire multicast tree. Instead, a Rendezvous Point (RP) is elected or designated, and all multicast sources and clients must explicitly register with the RP. This provides a centralized method of directing the multicast traffic of multiple multicast sources:



Configuring Manual PIMv1

Two versions of PIM exist (**PIMv1** and **PIMv2**), though both are very similar. PIM must be enabled on each participating interface in the multicast tree.

To enable PIM and specify its mode on an interface:

```
Switch(config)#  interface fa0/10
Switch(config-if)#  no switchport
Switch(config-if)#  ip pim dense-mode
Switch(config-if)#  ip pim sparse-mode
Switch(config-if)#  ip pim sparse-dense-mode
```

When utilizing PIM-SM, we must configure a Rendezvous Point (RP). RP's can be identified manually, or dynamically chosen using a process called **auto-RP (**Cisco-proprietary).

To manually specify an RP on a router:

```
Switch(config)#  ip pim rp-address 192.168.1.1
```

The above command must be configured on *every* router in the multicast tree, including the RP itself.

To restrict the RP to a specific set of multicast groups:

```
Switch(config)#  access-list 10 permit 226.10.10.1
Switch(config)#  access-list 10 permit 226.10.10.2
Switch(config)#  ip pim rp-address 192.168.1.1 10
```

The first two commands create an *access-list 10* specifying the multicast groups this RP will support. The third command identifies the *RP*, and applies *access-list 10* to the RP.

Configuring Dynamic PIMv1

When using Cisco's auto-RP, one router is designated as a **Mapping Agent**. To configure a router as a mapping agent:

```
Switch(config)#  ip pim send-rp-discovery scope 10
```

The *10* parameter in the above command is a TTL (Time to Live) setting, indicating that this router will serve as a mapping agent for up to 10 hops away.

Mapping agents listen for **candidate** RP's over multicast address 224.0.1.39 (Cisco RP Announce). To configure a router as a candidate RP:

```
Switch(config)#  access-list 10 permit 226.10.10.1
Switch(config)#  access-list 10 permit 226.10.10.2
Switch(config)#  ip pim send-rp-announce fa0/10 scope 4 group-list 10
```

The first two commands create an *access-list 10* specifying the multicast groups this RP will support. The third command identifies this router as a candidate RP for the multicast groups specified in *group-list 10*. This RP's address will be based on the IP address configured on *fa0/10*. The *scope 4* parameter indicates the maximum number of hops this router will advertise itself for.

The above commands essentially create a "mapping" of specific RP's to specific multicast groups. Once a mapping agent learns of these mappings from candidate RPs, it sends the information to all PIM routers over multicast address 224.0.1.40 (Cisco RP Discovery).

Configuring Dynamic PIMv2

Configuring PIMv2 is very similar to PIMv1, except that PIMv2 is a standards-based protocol. Also, there are terminology differences. Instead of mapping agents, PIMv2 uses **Bootstrap Routers (BSR),** which performs the same function.

To configure a router as a BSR:

```
Switch(config)# ip pim bsr-candidate fa0/10
```

To configure candidate RP's in PIMv2:

```
Switch(config)# access-list 10 permit 226.10.10.1
Switch(config)# access-list 10 permit 226.10.10.2
Switch(config)# ip pim rp-candidate fa0/10 4 group-list 10
```

The first two commands create an *access-list 10* specifying the multicast groups this RP will support. The third command identifies this router as a candidate RP for the multicast groups specified in *group-list 10*. This RP's address will be based on the IP address configured on *fa0/10*. The *4* parameter indicates the maximum number of hops this router will advertise itself for.

With PIMv2, we can create **border routers** to prevent PIM advertisements (from the BSR or Candidate RPs) from passing a specific point.

To configure a router as a PIM border router:

```
Switch(config)# ip pim border
```

Multicasts and Layer 2 Switches

Up to this point, we've discussed how multicasts interact with routers or multilayer switches.

By default, a Layer 2 switch will forward a multicast out all ports, excluding the port it received the multicast on. To eliminate the need of "flooding" multicast traffic, two mechanisms have been developed for Layer 2 switches:

- **IGMP snooping**
- **CGMP**

IGMP snooping allows a Layer 2 switch to "learn" the multicast MAC address of multicast groups. It does this by eavesdropping on IGMP Membership Reports sent from multicast hosts to PIM routers. The Layer 2 switch then adds a multicast MAC entry in the CAM for the specific port that needs the multicast traffic.

IGMP snooping is **enabled by default** on the Catalyst 2950 and 3550. If disabled, it can be enabled with the following command:

```
Switch(config)#  ip igmp snooping
```

If a Layer 2 switch does not support IGMP snooping, **Cisco Group Membership Protocol (CGMP)** can be used. Three guesses as to whether this is Cisco-proprietary or not.

Instead of the Layer 2 switch "snooping" the IGMP Membership Reports, CGMP allows the PIM router to actually **inform** the Layer 2 switch of the multicast MAC address, and the MAC of the host joining the group. The Layer 2 switch can then add this information to the CAM.

CGMP must be configured on the **PIM router** (or **multilayer switch**). It is **disabled** by default on all PIM routers. To enable CGMP:

```
Switch(config-if)#  ip cgmp
```

No configuration needs to occur on the Layer 2 switch.

Troubleshooting Multicasting

To view IGMP groups and current members:

Switch# *show ip igmp groups*

To view the IGMP snooping status:

Switch# *show ip igmp snooping*

To view PIM "neighbors":

Switch# *show ip pim neighbor*

To view PIM RPs:

Switch# *show ip pim rp*

To view PIM RP-to-Group mappings:

Switch# *show ip pim rp mapping*

To view the status of PIMv1 Auto-RP:

Switch# *show ip pim autorp*

To view PIMv2 BSRs:

Switch# *show ip pim bsr-router*

We can also debug multicasting protocols:

Switch# *debug ip igmp*
Switch# *debug ip pim*

Viewing the Multicast Table

Just like unicast routing protocols (such as OSPF, RIP), multicast routing protocols build a routing table.

Again, these tables contain several elements:

- The multicast **source**, and its associated multicast address (labeled as "**S,G**", or "**Source,Group")**
- **Upstream** interfaces that point *towards* the source
- **Downstream** interfaces that point *away* from the source towards multicast hosts.

To view the multicast routing table:

Switch# *show ip mroute*

If using PIM in **Dense Mode**, the output would be similar to the following:

```
IP Multicast Routing Table
Flags: D - Dense, S - Sparse, C - Connected, L - Local, P - Pruned
       R - RP-bit set, F - Register flag, T - SPT-bit set
Timers: Uptime/Expires
Interface state: Interface, Next-Hop, State/Mode

(10.1.1.1/24, 239.5.222.1), uptime 1:11:11, expires 0:04:29, flags: C
  Incoming interface: Serial0, RPF neighbor 10.5.11.1
  Outgoing interface list:
    Ethernet0, Forward/Sparse, 2:52:11/0:01:12
```

Remember that a multicast source with its associated multicast address is labeled as (S,G). Thus, in the above example, 10.1.1.1/24 is the multicast source, while 239.5.222.1 is the multicast address/group that the source belongs to.

The Incoming interface indicates the **upstream** interface. The RPF neighbor is the next hop router "upstream" towards the source. The outgoing interface(s) indicate **downstream** interfaces.

Notice that the **S – Sparse** flag is not set. That's because PIM is running in Dense Mode.

Viewing the Multicast Table (continued)

Remember, to view the multicast routing table:

Switch# *show ip mroute*

If using PIM in **Sparse Mode**, the output would be similar to the following:

```
IP Multicast Routing Table
Flags: D - Dense, S - Sparse, C - Connected, L - Local, P - Pruned
       R - RP-bit set, F - Register flag, T - SPT-bit set
Timers: Uptime/Expires
Interface state: Interface, Next-Hop, State/Mode

(*, 224.59.222.10), uptime 2:11:05, RP is 10.1.1.10, flags: SC
  Incoming interface: Serial0, RPF neighbor 10.3.35.1,
  Outgoing interface list:
    Ethernet0, Forward/Sparse, 4:41:22/0:05:21
```

Notice that the (S,G) pairing is labeled as (*, 224.59.222.10). In Sparse Mode, we can have *multiple* sources share the same multicast tree.

The Rendezvous Point (RP) is 10.1.1.10. The flags are set to **SC**, indicating this router is running in Sparse Mode.

Just like with Dense Mode, the Incoming interface indicates the **upstream** interface, and the outgoing interface(s) indicate **downstream** interfaces.

However, the RPF neighbor is the next hop router "upstream" towards the **RP** now, and not the source.