Copperbelt University
Computer Science Department

Internet Technologies

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES I

Subnets and Supernets

The rapid growth of the Internet has led to an enormous increase in the number of requests submitted for registered IP addresses. The result is a drastic reduction in available registered addresses. Virtually all class A and Class B addresses have already been assigned. Although there are still many Class C addresses, their numbers are rapidly dropping. In addition, the relatively small number of available Class C hosts place unrealistic configuration constraints on intranet infrastructures that support heavily populated network segments.

By definition, every registered IP address must identify a unique connection to the Internet. The 32-bit address specifies both a unique network on the Internet and a unique host on that network. With the number of registered IP addresses limited, as it is in the current version of IP (IPv4), this requirement poses two commonly encountered dilemmas.

Suppose a corporation had a registered Class B network address and a corporate intranet that includes ten separate networks. Because the registered Class B network address must uniquely identify a single LAN attached to the Internet, only one of the corporate networks is able to connect to the Internet without additional networks within that single registered IP address needing to be defined. *Subnetting* addresses this problem.

A company that has a single network populated with 400 hosts illustrates another common dilemma. With the number of available Class B addresses in short supply, the company is unable to obtain a registered Class B address but instead is granted two Class C addresses. Without some way of defining its corporate network as a single network within the two registered IP addresses, the company will be forced to divide the network, possibly at considerable expense. This problem is addressed by *supernetting*.

It is therefore, time for us to learn how the technique of subnetting allows us to define multiple networks within a single IP address as well as how to plan and implement subnetting. We shall also learn how supernetting allows us to define contiguous network addresses as a single network segment. Furthermore, we shall learn how to plan and implement a supernetting strategy.

Subnets

When necessary, a network administrator can divide a single IP network into multiple, connected networks, *subnets*. Reasons for creating subnets include the following:

1. *Network extension*. You can extend a network by adding routers and creating subnetworks to support connecting additional hosts when the network needs to grow beyond its physical limits defined by the Physical Layer protocols.

2. *Reducing network congestion*. Communication between hosts on a single network creates intranetwork traffic. Adding hosts to the network can result in network congestion when the amount of traffic supporting interhost communication becomes excessive. By dividing a network into subnetworks and group hosts that share communications on the same subnets, most interhost communications are isolated to the individual subnets and overall network congestion is reduced.
3. *Using multiple network media types*. When a network is divided into subnets, different physical media types can be used on each subnet.
4. *Isolating network problems*. Subnetting can reduce the overall impact of a network communication problem, such as a cable break on a 10Base2 Ethernet network, isolating it to the subnet on which it occurs.
5. *Improving network security*. Subnetting allows you to restrict communications containing sensitive information to a specific subnet. In addition, because only registered IP addresses are visible from the Internet, the structure of a private corporate network that is connected to the Internet through a single registered IP address is made invisible from the outside.

Subnet Masks

Each IP address contains both a complete network address and a complete host address within its 32-bits address. The TCP/IP protocol uses a technique called subnet masking to indicate which of the bits in an IP address represent the network address and which represent the host address. For this reason, each IP network address must be assigned a subnet mask even if the network is not segmented into subnetworks.

The *subnet mask* is a 32-bit number commonly expressed in dotted octet (binary) or dotted decimal format. Binary *ones* indicate the corresponding bit positions in an IP address that are to be read as the network portion of the address. The binary zeros in a subnet mask indicate the corresponding bit positions in an IP address that are to be read as the host portion of the address.

A router identifies the network and host addresses of a host by logically ANDing the subnet mask with an IP address. This helps the router to determine whether the value of a bit position is to be read as part of the network address or host address. The router only reads the value of the bit position as part of the network address when X AND Y is true. This occurs only when both the IP address and the subnet mask have 1 in corresponding bit positions. When ANDing the IP address and the subnet mask, the router reads the network address from bit positions that are true for both the IP address and the subnet mask. The mask hides the values at all other bit positions in the IP address from the router. This means that the router sees the value of only the network portion of the IP address. This is the portion necessary for routing a packet to the proper destination network. Default masks are assigned to Class A, Class B and Class C networks.

The default Class A mask has all 8 bits in the first octet set to 1, with the remaining 24 bits set to 0. The Class B mask has all 16 bits in the first two octets set to 1 with the remaining 16 bits set to 0. The Class C mask has all 24 bits in the first three octets set to 1 with the remaining 8 bits set to 0. Because a bit position value is read as part of the network address only if both the IP address and the subnet mask have a 1 in the same position, Class A network addresses can be defined only in the first octet of an IP address. The process applies for both Class B and Class C network addresses, which must be defined in the first two and first three octets of an IP address, respectively.

Classful and Classless Subnet Hierarchies

IP positions that conform to the default mask assignments for different network classes are recognised as members of a *classful subnet hierarchy*. In a classful subnet hierarchy, Class A networks always use 8 bits for the network address and 24 bits for the host address, Class B networks always use 16 bits for the network address and 16 bits for the host address. Internet routers route packets in a classful subnet hierarchy using the default mask for the network class, as defined in the initial bits of the first octet.

Dividing a network into smaller subnetworks involves borrowing bits from the host portion of the IP address to define the additional subnet addresses. When you borrow bits from the host portion of the IP address to create subnets, you have IP addresses that use non-standard numbers of bits to represent the network and host portions of the address. IP addresses that do not conform to the default mask assignments make up a classless subnet hierarchy. Routers inside a subnetted network are configured with subnet masks that recognise subnet addresses and route to subnets using classless addresses. Figure 45 gives examples of classless and classful addresses.

	Classless	Classful
1.	202.44.7.32 / 3	96.40.4.35
2.	202.44.7.32 / 10	96.40.4.42
3.	202.44.7.32 / 21	96.40.4.53
4.	202.44.7.32 / 29	96.40.4.61

Figure 1: Examples of classless and classful IP addresses

Planning Subnet Numbers

The number of bits borrowed from the host address field determines the maximum number of possible subnets that can be created and the number of host addresses that will be available on each subnet. The number of possible networks made available by assigning a subnet mask is calculated as follows:

Possible number of subnets = $2^n - 2$

Where *n* is the number of bits masked from the host portion of the IP address. The two addresses subtracted from the total are the *"all 1s"* and *"all 0s"* network addresses, because these addresses are not supported by all routers. The number of possible host addresses that can be assigned to each subnet is calculated as follows:

Possible number of hosts: $2^m - 2$

Where *m* is the number of unmasked bits remaining in the host portion of the IP address after some bits have been borrowed for subnetting. Again, the two addresses subtracted from the total are the *"all 1s"* and *"all 0s"* addresses. These methods can be used when masking partial and complete octets.

When calculating the number of available subnets or hosts, always assume that all 1s and all 0s are not supported unless specified otherwise.

Subnetting with Partial Octets

Creating subnets by masking a full octet may be inappropriate or even impossible in some cases. For example, you can mask the entire fourth octet of a Class C address, but doing so leaves no host bits available for assigning host addresses. It is common to mask part of the octet to provide subnet addresses while leaving unmasked bits available for assigning host addresses. This provides multiple networks within the constraints of a single Class C address.

Consider a situation in which a Class C address needs to support three subnets with less than 20 hosts per subnet. Masking the first three bits (high-order) of the fourth octet easily accommodates this. This solution allows one to assign up to 6 subnet addresses, with each subnet supporting up to 30 hosts. This can be summarised as given below:

Network Address:	202.44.7.0
Subnet Mask:	255.255.255.224
Possible Subnet Addresses:	$2^3 – 2 = 6$
Possible Host Addresses per Subnet:	$2^5 – 2 = 30$

A network component

The first three octets. The network component reflects the network class – in this case, a Class C network. The network components for Class A and Class B networks would consist of one and two octets, respectively. This component is common to all hosts on the network. It is the *classful address* recognised by Internet routers as a unique network on the Internet. When IP addresses are assigned, every host on this network will use 202.44.7.0 for the network component.

A subnet component

These are the bits that are masked. In this case it is the first three bits of the fourth octet. Each network segment or subnet is assigned a unique value for the subnet component from the list of available subnet addresses.

A host component

These are the bits that are unmasked. In this case, it is the last five bits of the fourth octet. Within each subnet, every host is assigned a unique host value from 1 to 31. Bits are referred to as high-order bits when they are read starting from the left side of an octet and low-order bits when they are read starting from the right side of an octet.

Note that this solution physically segregates contiguous sets of IP addresses onto network segments based on the first three bits of the fourth octet. For instance, IP addresses that include 001 as the first three bits in the fourth octet are assigned only to hosts on subnet 202.44.7.32 (the masked bits are 001).

The range of host addresses that can be assigned on subnet 202.44.7.32 is 202.44.7.33 to 202.44.7.63 when expressed as classful addresses, which is how we normally view them. Note that the available host addresses are equal to 32 + 1 through 32 + 31, which is the subnet plus host portions of the classless address.

The hosts on the next subnet (202.44.7.64) have 010 as the first three bits in the fourth octet. The address range assigned to subnet 202.44.7.64 is 202.44.7.65 to 202.44.7.95. Again you will see that the host addresses are equal to the subnet plus host portions of the classless address (64 + 1 through 64 + 31). The pattern is fundamental to all subnetted networks and is repeated on each subnet; it includes subnet 0 (mask 000) and subnet 244 (mask 111) on networks that support all 0s and all 1s for subnet addresses.

All hosts on a given subnet have a bit pattern in the host portion of their IP addresses (the subnet addresses) that is unique to the hosts on that subnet. Routers within the network are configured with a subnet mask to recognise the bit patterns, allowing each subnet to be uniquely identified and routed to within the network. The process is the same whether you subnet a Class A address by masking the second and third octets, or you subnet a Class C address by masking the four high-order bits in the fourth octet. Once the process is understood, one will learn common patterns and become familiar with conventions and rules that provide the quickest paths to managing subnets efficiently.

By convention, subnet masks are assigned with contiguous bits, starting with the high-order bit in an octet. The decimal value of the rightmost bit in the mask can be used to determine the start of the range of available subnet addresses in an octet. Each address in the range is a multiple of the address up to, but not including the multiple that has the same decimal value as the subnet mask, as shown in figure 46.

Example 1:	IP Address	= 203.88.6.0
	Subnet Mask	= 255.255.255.240
	Available Subnet Addresses	= $2^4 - 2 = 14$

$$\texttt{11111111.11111111.11111111.111}\underset{\substack{\uparrow\\16}}{\texttt{1}}\texttt{0000}$$

The decimal value of the rightmost bit in the mask is 16. The subnet addresses are obtained by starting with address 16 and adding 16 to each successive address in the sequence. The final address is the highest multiple of 16 that is less than the value of the subnet mask.

The available subnet addresses obtained by using this mask are:

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

Example 2:	IP Address	= 183.60.0.0
	Subnet Mask	= 255.255.192.0
	Available Subnet Addresses	= $2^2 - 2 = 2$

$$\texttt{11111111.11111111.1}\underset{\substack{\uparrow\\64}}{\texttt{1}}\texttt{000000.00000000}$$

The decimal value of the rightmost bit in the mask is 64. The subnet addresses are obtained by starting with address 64 and adding 64 to each successive address in the sequence. The final address is the highest multiple of 64 that is less than the value of the subnet mask.

The available subnet addresses obtained by using this mask are:
64 and 128

Figure 2: Using the rightmost bit in the subnet mask to determine the subnet addresses.

Planning Subnets

Before creating subnets on your network, you need to develop a strategy based on the following suggestions:

(a) Accommodate subnet needs by rounding up the maximum number of hosts currently needed to the nearest power of two, after determining the maximum number of subnets needed. For instance, if you determine that you need nine subnets, 2^3 (or 8) subnets do not provide enough subnet addressing space. You must round up to 2^4 (or 16) subnets to provide for your current needs.

(b) You should also ensure that you are providing subnet addressing space that will accommodate network growth. In this case, even providing for 16 subnets when you currently need 9 may prove inadequate if your network is growing rapidly. It may be better to plan for 2^5 (or 32) maximum subnets to ensure adequate addressing space in the future.

(c) Ensure that your strategy includes enough host bits to accommodate all hosts on your largest subnet. Again, round up the maximum number of hosts currently needed to the nearest power of two (2) to determine how many host bits will be required. For instance, if your largest subnet must support 35 hosts, 5 host bits will provide for only 2^5 (or 32) host addresses. You will need 2^6 (or 64) maximum addresses to accommodate the subnet with 35 hosts. You should consider planning to accommodate growth by providing address space for more hosts per subnet as well as planning for additional subnets.

Assigning Subnet Addresses

You should make subnet and host address assignments in a way that allows you to create additional subnets or support additional hosts without having to reassign existing addresses. You can do this by assigning subnet addresses starting with the leftmost of the subnet address bits and assigning host addresses starting with the rightmost of the host address bits and progressing in numeric order.

In figure 47, the subnet addresses have been assigned starting from the left of the subnet address field. Although a maximum of 16 subnets are available, only five (5) have been assigned. This means that only the first three bits are used to make all subnet address assignments. If the number of available hosts needs to be increased, the existing subnet and host addresses remain valid when the subnet mask is changed to 224. The mask is altered by changing the fourth bit in the subnet address field from 1 to 0, as shown in figure 48.

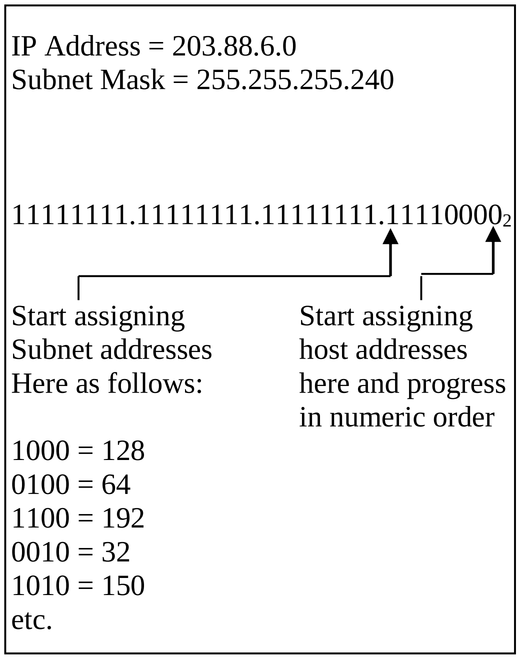
If you start with a subnet mask of 224, you may eventually find that more subnets are needed and the number of hosts per subnet will probably not exceed 16. If the greatest number of hosts supported on any subnet is currently 10 and host addresses were assigned by starting with the rightmost bit in the host field, progressing in numeric order, only the four rightmost bits were used to make the host address assignments. This means that the unused host bit (fifth from the right) is available to increase the maximum number of subnets possible without necessitating any changes beyond changing the subnet mask to 240 by changing the fourth high-order bit from 0 to 1. If the addresses were assigned by starting from the left of the host address field, the host addresses using the fifth bit would need to be reassigned in order to make the bit available for assigning subnet addresses.

Following are the steps to take when creating subnets:

1. Determine the number of subnets needed and the number of hosts that must be supported on each subnet, taking into account current and future needs. Calculate the maximum number that must be available to support your needs, using these general formulas:

   (a) Number of Subnets = $2^n - 2$, where *n* is the number of masked bits
   (b) Number of Hosts = $2^m - 2$, where *m* is the number of unmasked bits

2. Determine the subnet mask and subnet address assignments based on the maximum number of subnets needed. Once the mask is determined, the available subnet addresses will be multiples of the decimal value of the rightmost bit in the mask, up to, but not including the decimal value of the mask. Assign subnet addresses starting with the leftmost bits in the subnet address field.


IP Address = 203.88.6.0
Subnet Mask = 255.255.255.240
11111111.11111111.11111111.11110000₂
Start assigning Subnet addresses Here as follows:
Start assigning host addresses here and progress in numeric order
1000 = 128
0100 = 64
1100 = 192
0010 = 32
1010 = 150
etc.

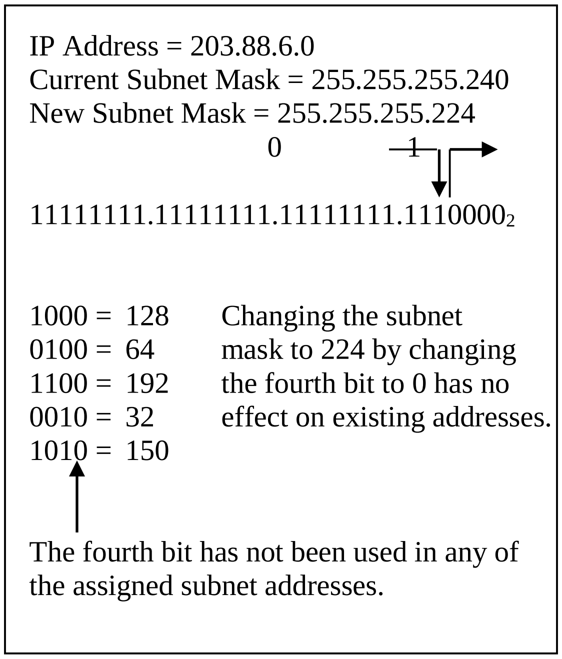

IP Address = 203.88.6.0
Current Subnet Mask = 255.255.255.240
New Subnet Mask = 255.255.255.224
0
1
11111111.11111111.11111111.1110000₂
1000 = 128
0100 = 64
1100 = 192
0010 = 32
1010 = 150
Changing the subnet mask to 224 by changing the fourth bit to 0 has no effect on existing addresses.
The fourth bit has not been used in any of the assigned subnet addresses.

Figure 3: Assigning subnet and host addresses

Figure 4: Changing the mask in a subnet address

3. Assign IP addresses to subnet hosts, taking the following into account:

   (a) All hosts on the network must have the same network address.
   (b) All hosts on a subnet must have the same subnet address.
   (c) Each host on a subnet must have a unique host address.
   (d) Host address fields cannot be all zeroes or all ones.
   (e) Host addresses on each subnet must be assigned starting with the rightmost bit in the host address field and progressing in numeric order.

Supernets

Supernets combine multiple Class C addresses into a block of addresses that can be assigned to hosts on a single network segment. Even though multiple Class C addresses are used to define the supernet, it is identified as a single network by routers.

Supernets were invented to address the following issues concerning the current IP addressing structure:

(a) Virtually all Class B addresses have been assigned, whereas a relatively large number of Class C network addresses are still available.
(b) A single Class C address supports a maximum of only 254 hosts.
(c) Continued growth of the routing tables in the Internet routers will render them virtually unmanageable due to their enormous size.
(d) The existing IP address space must accommodate continued growth.

The process of creating a supernet mask to combine multiple Class C addresses is the reverse of the process involved in creating a subnet mask to subdivide a network. Instead of borrowing bits from the host portion of the IP address, bits are borrowed from the default network address and used to increase the number of host address bits.

Please, recall that the default mask for a Class C network is 255.255.255.0 or 11111111.1111 1111.11111111.00000000. The first 24 bits are used to identify the network portion of the IP address. Using the formula

$2^m - 2$ = number of hosts

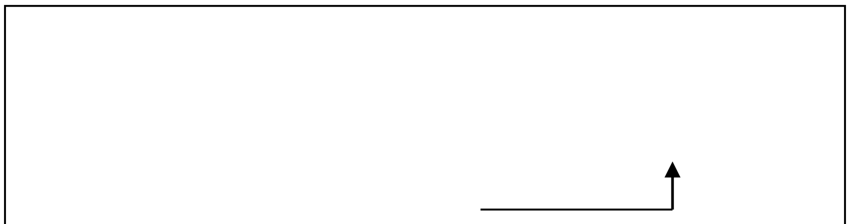
gives a potential $2^8 - 2$ (or 254) hosts within a Class C IP network.

By using a subnet mask that defines 23 bits for the network portion of the address (255.255.254.0), the number of unmasked bits defining the host portion of the address becomes 9. Substituting 9 for *m* in the formula provides a potential $2^9 - 2$ (510) hosts on a single network. Combining Class C networks to create supernets is also referred to as *address aggregation*.

IP Address Requirements for Creating Supernets

For one to create supernets from Class C IP addresses, the network addresses must be consecutive and the third octet in the first address must be evenly divisible by two. This ensures that a single bit (the rightmost bit in the third octet) defines the difference between the two addresses. Using these criteria, for example, you can create a supernet with the Class C network addresses 211.34.16.0 and 211.34.17.0, but not with network addresses 211.34.19.0 and 211.34.20.0. The comparison of binary addresses shown in figure 49 makes the difference between the two pairs more obvious.

Figure 49 shows that the addresses in the valid pair are contiguous at the binary level. This means that the blocks of host addresses that they contain are also contiguous. The pair that cannot be used to create a supernet is not contiguous at the binary level (a fact that is not readily apparent when viewing the dotted decimal format) and the blocks of host addresses they contain are not contiguous. To create a supernet from network addresses 211.34.16.0 and 211.34.17.0, assign a mask using 23 bits. The 24th bit becomes part of the host address space, allowing 510 hosts on the combined network instead of 254.



Valid for Supernet

211.34.16.0	11010011.00100010.0001000**0**.00000000
211.34.17.0	11010011.00100010.0001000**1**.00000000

The network addresses are contiguous

Not Valid for Supernet

211.34.19.0	11010011.00100010.00010**011**.00000000
211.34.20.0	11010011.00100010.00010**100**.00000000

The network addresses are not contiguous

Figure 5: Comparison of valid and nonvalid pairs of network addresses for creating a supernet.

The router on the new supernet advertises only the first Class C address (211.34.16.0) and the subnet mask /23. Receiving routers read the IP address and mask and recognise that the mask is made up of 23 bits, indicating that one bit of the Class C address has been used to make a supernet. This information is used by the routers to identify 211.34.16.0 and 211.34.17.0 as two contiguous Class C networks that have been combined. The first network in the pair is identified as supernet 0 and the second network is identified as supernet 1.

Combining More than Two Class C Addresses

You can combine more than two Class C addresses by borrowing more bits from the network portion of the address to extend the host portion of the address. Figure 50 shows the effect of borrowing additional bits to support more hosts on a single network. You can use the information shown in figure 50 to determine how many networks to combine or how many bits to borrow to support a given number of hosts. Figure 50 shows, for example, how many bits should be borrowed in order to support 4,098 hosts. In this case 5 bits need to be borrowed.

The lovely student is humbly requested to explore and seriously reflect on the issue of combining more than two Class C addresses. Please, try out several examples.

Supernets and Classless InterDomain Routing (CIDR)

CIDR was invented to provide a process for enabling routers to identify non-default or classless IP addresses. Because it allows routers to use subnet masks other than the defaults for Class A, B or C networks, the routers can identify these addresses. Earlier routing schemes used the default mask to identify the network portion of an IP address. With CIDR, network addresses are identified by the number of bits in the network portion of the address. The IP address 211.34.16.0/23, for example, is read by routers as having a subnet mask of 11111111.11111111.11111110.0000 0000. The /23 in the address tells the router that the first 23 bits from the left make up the subnet mask.

Being able to combine networks and advertise all the address ranges combined together as one block greatly decreases the volume of information that a router is required to store and route on the Internet. *Please, be advised not to create supernets without ensuring beforehand that your network routers support CIDR.*

Number of Bits Borrowed	7	6	5	4	3	2	1	0
Subnet Mask	128	192	224	240	248	252	254	255
Number of Contiguous Addresses to Combine	128	64	32	16	8	4	2	0
Number of Hosts Supported		>8,192 <16,384	>4,096 <8,192	>2,048 <4,096	>1,024 <2,048	<1,024 >512	>256 <512	< 256

Figure 6: Combining more than two Class C networks.

Copperbelt University
Computer Science Department

Internet Technologies

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

IP Routing

Routing is the glue that binds the Internet together. Without it, TCP/IP is limited to a single physical network. Routing allows traffic from your local network to reach its destination somewhere else in the world – perhaps after passing through many intermediate networks.

The important role of routing and the complex interconnection of Internet networks make the design of routing protocols a major challenge to network software developers.

Common Routing Configurations

First we must make a distinction between routing and routing protocols. All systems route data, but not all systems run routing protocols. Routing is the act of forwarding datagrams based on the information contained in the routing table. Routing protocols are programs that exchange the information used to build routing tables.

A network's routing configuration does not always require a routing protocol. In situations where the routing information does not change – for example, when there is only one possible route, the systems administrator usually builds the routing table manually. Some networks have no access to any other TCP/IP networks, and therefore do not require that the systems administrator builds the routing table at all. The three most common routing configurations are:

Minimal routing: A network completely isolated from all other TCP/IP networks requires only minimal routing. A minimal routing table is usually built by **ifconfig** when the network interface is configured. If your network doesn't have direct access to other TCP/IP networks, and if you are not using subnetting, this may be the only routing table you may require.

Static routing: A network with a limited number of gateways to other TCP/IP networks can be configured with static routing. When a network has only one gateway, a static route is the best choice. A static routing table is constructed manually by the systems administrator. Static routing tables do not adjust to network changes, so they work best where routes do not change.

Dynamic routing: A network with more than one possible route to the same destination should use dynamic routing. A dynamic routing table is built from the information exchanged by routing protocols. The protocols are designed to distribute information that dynamically adjusts routes to reflect changing network conditions. Routing protocols handle complex routing situations more quickly and accurately than the systems administrator can. Routing protocols are designed not only to switch to a backup route when the primary route becomes inoperable; they are also designed to decide which route is the "best" route to a destination. On any network where there are multiple paths to the same destination, a routing protocol should be used.

Routes are built automatically by **ifconfig**, manually by the systems administrator or dynamically by routing protocols. But no matter how routes are entered, they all end up in the routing table.

Interior Routing Protocols

Routing protocols are divided into two general groups: *interior* and *exterior* protocols. An interior protocol is a routing protocol used inside – interior to – an independent network system. In TCP/IP terminology, these independent network systems are called autonomous systems (AS). Within an autonomous system (AS), routing information is exchanged using an interior protocol chosen by the autonomous system's administration.
All interior routing protocols perform the same basic functions. They determine the "best" route to each destination and they distribute routing information among the systems on a network. How they perform these functions, in particular, how they decide which routes are best, is what makes routing protocols different from each other. There are several interior routing protocols:

- The *Routing Information Protocol (RIP)* is the interior routing protocol most commonly used in UNIX systems. RIP is included as part of the UNIX software delivered with most systems. It is adequate for local area networks and is simple to configure.

- *Hello* is a protocol that uses delay as the deciding factor when choosing the best route. Delay is the length of time it takes a datagram to make the round trip between its source and destination. A *Hello* packet contains a time stamp indicating when it was sent. When the packet arrives at its destination, the receiving system subtracts the time stamp from the current time, to estimate how long it took the packet to arrive. *Hello* is not widely used. It was the interior protocol of the original 56 kbps NSFNET backbone and has had very little use otherwise.

- *Intermediate System to Intermediate System (IS-IS)* is an interior routing protocol from the OSI protocol suite. It is a *Shortest Path First (SPF) link-state* protocol. It was the interior routing protocol used on the T1 NSFNET backbone and is still used by some large service providers.

- *Open Shortest Path First (OSPF)* is another link state protocol developed for TCP/IP. It is suitable for very large networks and provides several advantages over RIP.

Routing Information Protocol (RIP)

The Routing Information Protocol (RIP) is run by the routing daemon **routed** (pronounced "route" "d"). When **routed** starts, it issues a request for routing updates and then listens for responses to its request. When a system configured to supply RIP information "hears" the request, it responds with an update packet based on the information in its routing table. The update packet contains the destination addresses from the routing table and the routing metric associated with each destination. Update packets are issued in response to requests, as well as periodically to keep routing information accurate.

To build a routing table, **routed** uses the information in the update packets. If the routing update contains a route to the destination that does not exist in the local routing table, the new route is added. If the update describes a route whose destination is already in the local routing table, the new route is used only if it has a lower cost. The cost of a route is determined by adding the cost of reaching the gateway that sent the update to the metric contained in the RIP update packet. If the total metric is less than the metric of the current route, the new route is used.
RIP also deletes routes from the routing table. It accomplishes this in two ways.

(a) If the gateway to a destination says that the cost of the route is greater than 15, then the route is deleted.
(b) RIP assumes that a gateway that does not send updates is dead. All routes through a gateway are deleted if no updates are received from that gateway for a specified time period

In general, RIP issues routing updates every 30 seconds. In many implementations, if a gateway does not issue routing updates for 180 seconds, all routes through that gateway are deleted from the routing table.

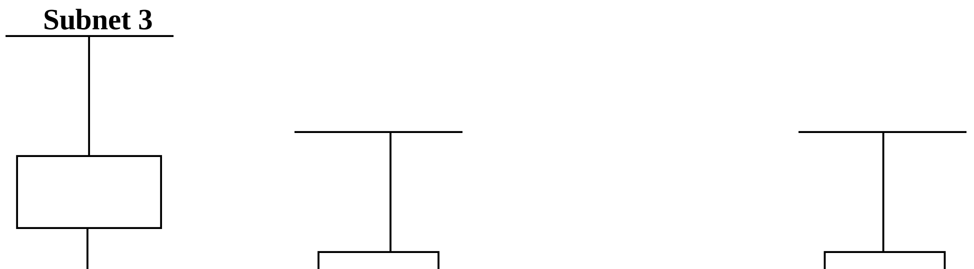
RIP is easy to implement and simple to configure. However, RIP has three serious disadvantages.

(a) *Limited network diameter*. The longest RIP route is 15 hops. A RIP router cannot maintain a complete routing table for a network that has destinations more than 15 hops away. The hop count cannot be increased because of the second shortcoming.
(b) *Slow convergence*. Deleting a bad route sometimes requires the exchange of multiple routing update packets until the route's cost reaches 16 or more. This is called *counting to infinity*, because RIP keeps incrementing the route's cost until it becomes greater than the largest valid RIP metric. *In this case, 16 is infinity*. Additionally, RIP may wait 180 seconds before deleting the invalid routes. We can say that these conditions delay the "convergence of routing", i.e., it takes a long time for the routing table to reflect the current state of the network.
(c) *Classful routing*. RIP interprets all addresses using the IP class rules for addressing. For RIP all addresses are class A, B or C, which makes RIP incompatible with CIDR supernets and incapable of supporting variable-length subnets.

Nothing can be done to change the limited network diameter. A small metric is essential to reduce the impact of counting to infinity. However, limited network size is the least important of RIP's shortcomings. The real work of improving RIP concentrates on the other two problems, slow convergence and classful routing.

Features have been added to RIP to address slow convergence. Figure 51 illustrates a network where a counting-to-infinity problem might happen.

Figure 51 shows that *mubilo* reaches subnet 3 through *mumbole* and then through *muchinga*. Subnet 3 is 2 hops away from *mubilo* and 1 hop away from *mumbole*. Therefore, *mumbole* advertises a cost of 1 for subnet 3 and *mubilo* advertises a cost of 2 and traffic continues to be routed through *mumbole*. That is, until something goes wrong. If *muchinga* crushes, *mumbole* waits for an update from *muchinga* for 180 seconds. While waiting, *mumbole* continues to send updates to *mubilo* that keep the route to subnet 3 in *mubilo's* routing table. When *mumbole's* timer finally expires, it removes all routes through *muchinga* from its routing table, including the route to subnet 3. It then receives an update from *mubilo* advertising that *mubilo* is 2 hops away from subnet 3. *mumbole* installs this route and announces that it is 3 hops away from subnet 3. *mubilo* receives this update, installs the route and announces that it is 4 hops away from subnet 3. Things continue on in this manner until the cost of the route to subnet 3 reaches 16 in both routing tables. If the update interval is 30 seconds, this could take a long time.



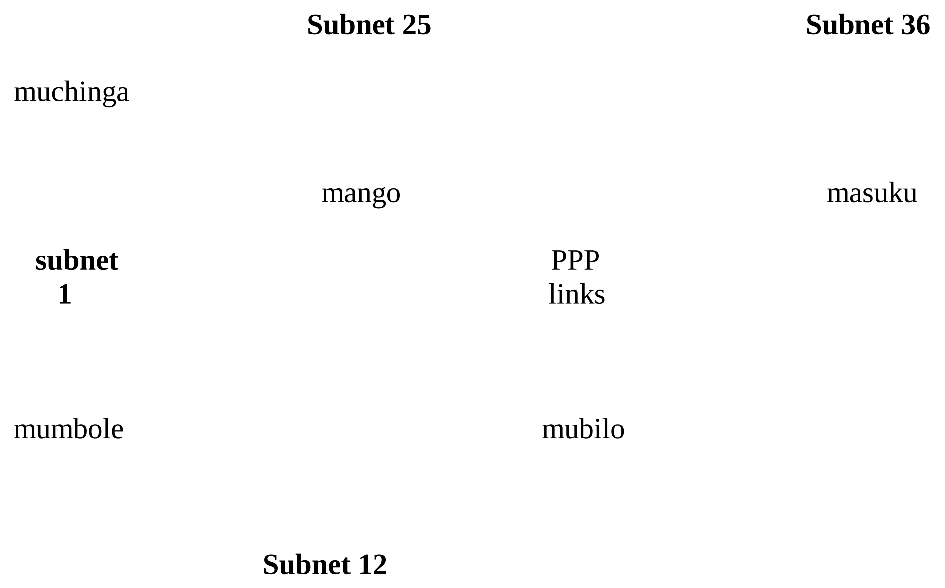


Figure 7: A sample network

Split horizon and poison reverse are the two features that attempt to avoid counting to infinity.

Split Horizon. With this feature, a router does not advertise routes on the link from which those routes were obtained. This would solve the count-to-infinity problem described above. Using the spilt horizon rule, mubilo would not announce the route to subnet 3 on subnet 12 because it learned that route from the updates it received from mumbole on subnet 12. While this feature works for the example described above, it does not work for all count-to-infinity configurations.

Poison Reverse.

This feature is an enhancement of split horizon. It uses the same idea: "Do not advertise routes on the link from which those routes were obtained." But it adds a positive action to that essentially negative rule. Poison reverse says that a router should advertise an infinite distance for routes on this link. With poison reverse, mubilo would advertise subnet 3 with a cost of 16 to all systems on subnet 12. The cost of 16 means that subnet 3 cannot be reached through mubilo.

Split horizon and poison reverse solve the problem described above. But what happens if mubilo crashes? With split horizon, mango and masuku do not advertise to mubilo the route to subnet 12 because they learned the route from mubilo. They do, however, advertise the route to subnet 12 to each other. When mubilo goes down, mango and masuku perform their own count to infinity before they remove the route to 12. Triggered updates address this problem.

Triggered updates are a big help. Instead of waiting for the normal 30 seconds update interval, a triggered update is sent immediately. Therefore, when an upstream router crashes or a local link goes down, immediately after the router updates its local routing table, it sends the changes to its neighbours. Without triggered updates, counting to infinity can take almost 8 minutes. With triggered updates, neighbours are informed in a few seconds. Triggered updates also use network bandwidth efficiently. They do not include the full routing table; they include only the routes that have changed.
Triggered updates take positive action to eliminate bad routes. Using triggered updates, a router advertises the routes deleted from its routing table with an infinite cost to force downstream routers to also remove them. Figure 51 can still help us to understand this scenario. If mubilo crashes, masuku and mango wait 180 seconds and remove the routes to subnet 1, 3 and 12 from their routing tables. They then send each other triggered updates with a metric of 16 for subnets 1, 3 and 12. Thus they tell each other that they cannot reach these networks and no count to infinity occurs. Split horizon, poison reverse and triggered updates go a long way to eliminating counting to infinity.

It is the final shortcoming – the fact that RIP is incompatible with CIDR supernets and variable-length subnets – that necessitated the migration to a higher version of RIP, which could address the shortcomings. RIP is not compatible with current and future plans for the TCP/IP protocol stack. A new version had to be created to address this final problem.

RIP Version 2

RIP Version 2 (RIP-2), is a new version of RIP. It is not a completely new protocol. It simply defines extensions to the RIP packet format. RIP-2 adds a network mask and a next hop address to the destination address and metric found in the original RIP packet.

The network mask frees the RIP-2 router from the limitation of interpreting addresses based on strict address class rules. The mask is applied to the destination address to determine how the address should be interpreted. Using the mask, RIP-2 routers support variable-length subnet and CIDR supernets.

The next hop address is the IP address of the gateway that handles the route. If the address is 0.0.0.0, the source of the update packet is the gateway for the route. The next hop route permits a RIP-2 supplier to provide routing information about gateways that do not speak the RIP-2.

RIP-2 adds other new features to RIP. It transmits updates via the multicast address 224.0.0.9 to reduce the load on systems that are not capable of processing a RIP-2 packet. RIP-2 also introduces a packet authentication scheme to reduce the possibility of accepting erroneous updates from the misconfigured system.

Despite these changes, RIP-2 is compatible with RIP. The original RIP specification allowed for future versions of RIP. RIP has a version number in the packet header and it had several empty fields for extending the packet. The new values used by RIP-2 did not require any changes to the structure of the packet. The new values are simply placed in the empty fields that the original protocol reserved for future use. Properly implemented RIP routers can receive RIP-2 packets and extract the data that they need from the packet without becoming confused by the new data.

Split horizon, poison reverse, triggered updates and RIP-2 eliminate most of the problems with the original protocol. But RIP-2 is still a distance vector protocol. There are other newer routing technologies that are considered superior for large networks. In particular, link-state routing protocols are favoured because they provide rapid routing convergence and reduce the possibility of routing loops.

Open Shortest Path First (OSPF)

Open Shortest Path First (OSPF) is a *link-state* protocol. As such it is very different from RIP. A router running RIP shares information about the entire network with its neighbours. Conversely, a router running OSPF shares information about its neighbours with the entire network. The "entire network" means, at most, a single autonomous system. RIP does not try to learn about the entire Internet, and OSPF does not try to advertise to the entire Internet. That is not their job. These are interior routing protocols and so their job is to construct the routing inside of an autonomous system. OSPF further refines this task by defining a hierarchy of routing areas within an autonomous system:

Areas. An *area* is an arbitrary collection of interconnected networks, hosts and routers. Areas exchange routing information with other areas within the autonomous system through *area border routers*.

Backbone. A *backbone* is a special area that interconnects all of the other areas within an autonomous system. Every area must connect to the backbone, because the backbone is responsible for distributing routing information between the areas.

Stub area. A *stub area* has only one area border router, which means that there is only one route out of the area. In this case, the area border router does not need to advertise external routes to the other routers within the stub area. It can simply advertise itself as the default route.

Only a large autonomous system needs to be subdivided into areas. The sample network in figure 51 is small and would not need to be divided. However, using figure 51 as an example, we could divide this autonomous system into any areas we wish. Let us assume we divide it into areas: area 1 contains subnet 3; area 2 contains subnet 1 and subnet 12; and area 3 contains subnet 25, subnet 36 and the PPP links. Furthermore, we could define area 1 as a stub area because muchinga is that area's only area border router. We could also define area 2 as the backbone area because it interconnects the other two areas and all routing information between area 1 and 3 must be distributed by area 2. Area 2 contains two border routers, mubilo and muchinga and one interior router, mumbole. Area 3 contains three routers: mubilo, masuku and mango.

OSPF provides lots of flexibility for subdividing an autonomous system. Why is this necessary? One problem for a link-state protocol is the large quantity of data that can be collected in the *link-state database* and the amount of time it can take to calculate the routes from that data. Every OSPF router builds a *directed graph* of the entire network using Dijkstra's Shortest Path First (SPF) algorithm. A directed graph is a map of the network from the perspective of the router, that is, the root of the graph is the router. The graph is built from the link-state database, which includes information about every router on the network and all the neighbours of every router.

The information in the link-state database is gathered and distributed in a simple and efficient manner. An OSPF router discovers its neighbours through the use of *Hello packets*. It sends Hello packets and listens for Hello packets from adjacent routers. The Hello packet identifies the local router and lists the adjacent routers from which it has received packets. When a router receives a Hello packet that lists it as an adjacent router, it knows that it has found a neighbour. It knows this because it can hear packets from that neighbour and because the neighbour lists it as an adjacent router, the neighbour must be able to hear packets from it. The newly discovered neighbour is added to the local system's neighbour list.

The OSPF router then advertises all of its neighbours. It does this by flooding a Link-State Advertisement (LSA) to the entire network. The LSA contains the address of every neighbour and the cost of reaching that neighbour from the local system. Flooding means that the router sends the LSA out of very interface and that every router that receives the LSA sends it out of every interface except the one from which it was received. To avoid flooding duplicate LSAs, the routers store a copy of the LSAs they receive and discard duplicates. In this way, every router in the entire network receives every other router's link-state advertisement.

OSPF routers track the state of their neighbours by listening for Hello packets. Hello packets are issued by all routers on a periodic basis. When a router stops issuing Hello packets, it or the link it is attached to is assumed to be down. Its neighbours update their LSA and flood them through the network. The new LSAs are included into the link-state database on every router on the network and every router recalculates their network map based on this new information. Clearly, limiting the number of routers by limiting the size of the network reduces the burden of recalculating the map. For many networks the entire autonomous system is small enough. For others, dividing the autonomous system into areas improves efficiency.

Another feature of OSPF that improves efficiency is the designated router. The designated router is one router on the network that treats all other routers on the network as its neighbours, while all other routers treat only the designated router as their neighbour. This helps reduce the size of the link-state database and thus improves the speed of the shortest-path-first calculation. Assume a broadcast network with 5 routers. Five routers each with 4 neighbours produce a link-state database with 20 entries. But if one of those routers is the designated router then that router has 4 neighbours and all other routers have only one neighbour for a total of 8 link-state database entries. While there is no need for a designated router on such a small network, the larger the network, the more dramatic the gains. For example, a broadcast network with 25 routers has a link-state database of 48 entries when a designated router is used, versus a database of 600 entries without one.

OSPF provides the router with an end-to-end view of the route between two systems instead of limited next-hop view provided by RIP. Flooding quickly disseminates routing information throughout the network. Limiting the size of the link-state database through areas and designated routers speeds the SPF calculation. Taken altogether, OSPF is an efficient link-state routing protocol.

OSPF also offers additional features. It provides password authentication to ensure that the update comes from a valid router. OSPF uses Message Digest 5 (MD5) crypto-checksum for stronger authentication.

OSPF also supports equal-cost multi-path routing. This means that OSPF routers can maintain more than one path to a single destination. Given the proper conditions, this feature can be used for load balancing across multiple network links. With all these features, OSPF is the preferred TCP/IP interior routing protocol for dedicated routers.

Exterior Routing Protocols

Exterior routing protocols are used to exchange routing information between autonomous systems. The routing information passed between autonomous systems is called reachability information. Rechability information is simply information about which networks can be reached through specific autonomous system. Moving routing information into and out of an autonomous system is the function of routing protocols. Exterior routing protocols are also called exterior gateway protocols. We are, however, requested not to confuse an exterior gateway protocol with the Exterior Gateway protocol (EGP). EGP is not a generic term; it is a particular exterior routing protocol, and an old one for that matter.

Exterior Gateway Protocol (EGP)

A gateway running EGP announces that it can reach networks that are part of its autonomous system. It does not announce that it can reach networks outside its autonomous system.

Before sending routing information, the systems first exchange EGP Hello and I-Heard-You (I-H-U) messages. These messages establish a dialogue between two EGP gateways. Computers communicating via EGP are called EGP neighbours and the exchange of Hello and I-H-U messages is called acquiring a neighbour.

Once a neighbour is acquired, routing information is requested via a poll. The neighbour responds by sending a packet of reachability information called an update. The local system includes the routes from the update into its local routing table. If the neighbour fails to respond to three consecutive polls, the system assumes that the neighbour is down and removes the neighbour's routes from its table. If the system receives a poll from its EGP neighbour, it responds with its own update packet.

Unlike the interior protocols discussed earlier, EGP does not attempt to choose the "best" route. EGP updates contain distance-vector information, but EGP does not evaluate this information. The routing metrics from different autonomous systems are not directly comparable. Each AS may use different criteria for developing theses values. Therefore, EGP leaves the choice of a "best" route to someone else.

When EGP was designed, the network relied upon a group of trusted core gateways to process and distribute the routing information received from all the autonomous systems. These core gateways were expected to have the information necessary to choose gateways, where the information was combined and passed back out to the autonomous systems.

A routing structure that depends on a centrally controlled group of gateways does not scale well and it is therefore inadequate for the rapidly growing Internet. As the number of autonomous systems and networks connected to the Internet grew, it became difficult for the core gateways to keep up with the expanding workload. This is one reason why the Internet moved to a more distributed architecture that places a share of the burden of processing routes on each autonomous system. Another reason is that no central authority controls the commercialised Internet. The Internet is composed of many equal networks. In a distributed architecture, the autonomous systems require routing protocols, both interior and exterior, that can make intelligent routing choices. Because of this, EGP is no longer popular.

Border gateway Protocol (BGP)

Border Gateway Protocol (BGP) is the leading exterior routing protocol of the Internet. It is based on the OSI InterDomain Routing Protocol (IDRP). BGP supports policy-based routing, which uses non-technical reasons (for example, political, organisational or security considerations) to make routing decisions. Thus BGP enhances an autonomous system's ability to choose between routes and to implement routing policies without relying on a central routing authority. This feature is important in the absence of core gateways to perform these tasks.

Routing policies are not part of the BGP protocol. Policies are provided externally as configuration information. The network administrator enforces the routing policy through configuring the router.

BGP is implemented on top of TCP, which provides BGP with a reliable delivery service. BGP uses well-known TCP port 179. It acquires its neighbours through the standard TCP three-way handshake. BGP neighbours are called peers. Once connected, BGP peers exchange OPEN messages to negotiate session parameters, such as the version of BGP that is to be used.

The UPDATE message lists the destinations that can be reached through a specific path and the attributes of the path. BGP is a path vector protocol. It is called a path vector protocol because it provides the entire end-to-end path of a route in the form of a sequence of autonomous system numbers. Having the complete AS path eliminates the possibility of routing loops and count-to-infinity problems. A BGP UPDATE contains a single path vector and all the destinations reachable through that path. Multiple UPDATE packets may be sent to build a routing table.

BGP peers send each other complete routing table updates when the connection is first established. After that, only changes are sent. If there are no changes, a small (19-byte) KEEP-ALIVE message is sent to indicate that the peer and the link are still operational. BGP is very efficient in its use of network bandwidth and system resources.

By far the most important issue to remember about exterior protocols is that most systems do not run them. Exterior protocols are only required when an AS must exchange routing information with another AS. Most routers within an AS run an interior protocol such as OSPF. Only those gateways that connect the AS to another AS need to run an exterior gateway protocol. Your network is probably an independent part of an AS run by someone else. Internet Service Providers are good examples of autonomous systems made up of many independent networks. Unless you provide a similar level of service, you probably don't need to run an exterior routing protocol.

Choosing a Routing Protocol

Although there are many routing protocols, choosing one is usually easy. Most of the interior routing protocols mentioned above were developed to handle the special routing problems of very large networks. Some of the protocols have only been used by large national and regional networks. For local area networks, RIP is still the most common choice. For larger networks, OSPF is the choice.
If you must run an exterior routing protocol, the protocol that you use is often not a matter of choice. For two autonomous systems to exchange routing information, they must use the same exterior protocol. If the other AS is already in operation, its administrators have probably decided which protocol to use and you will be expected to conform to their choice. Most often this choice is BGP.

Copperbelt University
Computer Science Department

Internet Technologies

By Dr Derrick Ntalasha

An Error Reporting Mechanism (ICMP)

Best effort semantics and error detection

IP defines a best-effort communication service in which datagrams can be lost, duplicated, delayed or delivered out of order. It may seem that a best-effort service does not need any error detection. However, it is important to realise that a best-effort service is not careless - actually IP attempts to avoid errors and to report problems when they occur. The checksum that is used to detect transmission errors is already an example of error detection in IP. Whenever a datagram is received, the checksum is verified to ensure that the header arrived intact. To verify the checksum, the receiver re-computes the checksum including the value in the checksum field. If a bit in the IP header is damaged during transmission across a physical network, the receiver will find that the checksum does not result in zero. After changing fields in the header (e.g., after decrementing the TIME TO LIVE field), a router must re-compute the checksum before forwarding the datagram to its next hop.
If a checksum error is detected then the datagram must be discarded without any further processing. The receiver cannot trust any fields in the datagram header because the receiver can not know which bits were altered. In particular, the receiver can not send an error message back to the computer that sent the datagram because the receiver can not trust the source address in the header. Further, the receiver can not forward the damaged datagram because the receiver can not trust the destination address in the header. Thus, the receiver has no option but to discard the damaged datagram.

Internet Control Message Protocol (ICMP)

Problems that are less severe than transmission errors result in error conditions that can be reported. The TCP/IP suite includes a protocol that IP uses to send error messages when errors occur. This protocol is known as the Internet Control Message Protocol (ICMP). ICMP is required for a standard implementation of IP. The two protocols are co-dependant. IP uses ICMP when it sends an error message, and ICMP uses IP to transport messages.

Examples of ICMP error messages include

- *Source Quench*. A router sends a source quench whenever it has received so many datagrams that it has no more buffer space available. A router that has temporarily run out of buffer space must discard incoming datagrams. When it discards a datagram, the router sends a source quench message to the host that created the datagram. When it receives a source quench, a host is required to reduce the rate at which it is transmitting.

- *Time Exceeded*. A time exceeded message is sent in two cases. Whenever a router reduces the TIME TO LIVE field in a datagram to zero, the router discards the datagram and sends a time exceeded message. In addition, a time exceeded message is sent by a host if the re-assembly timer expires before all fragments from a given datagram arrive.

- *Destination Unreachable*. Whenever a router determines that a datagram can not be delivered to its final destination, the router sends a destination unreachable message to the host that created the datagram. The message specifies whether the specific destination host is unreachable or the network to which the host attaches is unreachable. In other words, the error message distinguishes between a situation where an entire network is temporarily disconnected from an internet (e.g., because a router has failed) and the case where a particular host is temporarily off-line (e.g., because the host is powered down).

- *Redirect*. When a host creates a datagram destined for a remote network, the host sends the datagram to a router, which forwards the datagram to its destination. If a router determines that a host has incorrectly sent a datagram that should be sent to a different router, the router uses a redirect message to cause the host to change its route. A redirect message can specify either a change for a specific host or a change for a network; the latter is more common.

- *Parameter Problem*. One of the parameters specified in a datagram is incorrect.

In addition to error messages, ICMP defines informational messages that include:

- *Echo Request / Reply*. An echo request message can be sent to the ICMP software on any computer. In response to an incoming echo request message, ICMP software is required to send an ICMP echo reply message. The reply carries the same data as the request.

- *Address Mask Request / Reply.* A host broadcasts an address mask request when it boots, and routers than receive the request send an address mask reply that contains the correct 32-bit subnet mask being used on the network.

ICMP message transport

ICMP uses IP to transport each message. When a router has an ICMP message to send, it creates an IP datagram and encapsulates the ICMP message in the datagram. That is, the ICMP message is placed in the data area of the IP datagram. The datagram is then forwarded as usual, with the complete datagram being encapsulated in a frame for transmission.

ICMP messages are always created in response to a datagram. Either the datagram has encountered a problem (e.g., a router finds that the destination specified in the datagram is unreachable), or the datagram carries an ICMP message back to the source of the datagram. Sending a message back to the source is straightforward because each datagram carries the IP address of its source in the header. A router extracts the source address from the header of the incoming datagram and places the address in the DESTINATION field of the header of the datagram that carries the ICMP message.

Datagrams carrying ICMP messages do not have special priority - they are forwarded like any other datagram, with one minor exception. If a datagram carrying an ICMP error message causes an error, no error message is sent. In this case, the designers of IP wanted to avoid an internet becoming congested carrying error messages about error messages.

Using ICMP messages to test reachability

The *ping* program is one of the most commonly used utilities for network management on a network. The ping program is used to test and see if a given destination can be reached. Ping uses ICMP echo request and echo reply messages. When invoked, ping sends an IP datagram that contains an ICMP echo request message to the specified destination. After sending the request, it waits a short time for the reply. If no reply arrives, ping retransmits the request. If no reply arrives for the retransmissions (or if an ICMP destination unreachable message arrives), ping declares that the remote machine is not reachable.

ICMP software on a remote machine replies to the echo request message. According to the protocol, whenever an echo request arrives, the ICMP software must send an echo reply.

Using ICMP to trace a route

The TIME TO LIVE field in a datagram is used to recover from routing errors. To prevent a datagram from following a cycle of routes forever, each router that handles a datagram decrements the TIME TO LIVE counter in the header. If the counter reaches zero, the router discards the datagram and sends an ICMP time exceeded error message back to the source.

ICMP messages are used by the traceroute tool when it constructs a list of all routers along a path to a given destination. Traceroute is a network management tool which sends a series of datagrams and waits for a response to each. Traceroute sets the TIME TO LIVE in the first datagram to 1 before sending the datagram. The first router that receives the datagram decrements the TIME TO LIVE, discards the datagram and sends back an ICMP time exceeded message. Because the ICMP message travels in an IP datagram, traceroute can extract the IP source address and announce the address of the first router along the path to the destination.

After it discovers the address of the first router, traceroute sends a datagram with TIME TO LIVE set to 2. The first router decrements the counter and forwards the datagram. The second router discards the datagram and sends an error message. Similarly, once it has received an error message from the router that is distance 2, traceroute sends a datagram with TIME TO LIVE set to 3, and then 4, and so on.

Several details remain that traceroute must handle. Because IP uses best-effort delivery, datagrams can be lost, duplicated, or delivered out of order. Thus traceroute must be prepared to handle duplicate responses and to retransmit datagrams that are lost. Choosing a retransmission time can be difficult because traceroute can not know how long to wait for a response - traceroute allows the user to decide how long to wait.

Traceroute faces another problem: routes can change dynamically. If routes change between two probes, the second probe may take a longer or shorter path than the first. More important, the sequence of routers that traceroute finds may not correspond to a valid path through the internet. Thus, traceroute is most useful in an internet where routes are relatively stable.

Traceroute also needs to handle the situation when the TIME TO LIVE is large enough to reach the destination host. To determine when a datagram succeeds in reaching the destination, traceroute sends a datagram to which the destination host will reply. Although it could send an ICMP echo request, traceroute does not. Instead, traceroute either uses ICMP message type 30 (traceroute), or it uses the User Datagram Protocol (UDP), a protocol that allows application programs to send and receive individual messages. When using UDP, traceroute sends the UDP datagram to a non-existent program, on the destination machine. When the UDP message arrives for a non-existent program, ICMP sends a destination unreachable message. Thus, each time it transmits a datagram, traceroute either receives an ICMP time exceeded message from a router along the path or an ICMP destination unreachable message from the ultimate destination computer.

Using ICMP for path MTU discovery

In a router, IP software fragments any datagram that is larger than the MTU of the network over which the datagram is being transmitted. Although fragmentation solves the problem of heterogeneous networks, fragmentation often impacts performance. A router uses memory and CPU time to construct fragments. Similarly, a destination host uses memory and CPU time to collect incoming fragments and reassemble them into a complete datagram. In some applications, fragmentation can be avoided if the original sender chooses a smaller datagram size. For example, a file transfer application can send an arbitrary amount of data in each datagram. If the application chooses a datagram size less than or equal to the smallest network MTU along the path to the destination, no router will need to fragment the datagram.

Technically, the smallest MTU along a path from a source to a destination is known as the *path MTU*. If routes change (i.e., the path changes), the path MTU may change as well. However, in many parts of the Internet, routes tend to remain stable for days or weeks. In such cases, it makes sense for a computer to find the path MTU, and create datagrams that are small enough.

What mechanism can a host use to determine the path MTU? An ICMP error message and a probe that will cause the error message to be sent will have to be used. The error message consists of an ICMP message that reports fragmentation was required but not permitted, and the technique for requesting it is a bit in the FLAGS field of the IP header that specifies the datagram should not be fragmented. When a router determines that a datagram must be fragmented, the router examines the bit in the header to verify that fragmentation is allowed. If the bit is set, the router does not perform fragmentation. Instead, the router sends an ICMP message back to the source, and discards the datagram.

To determine the path MTU, IP software on a host sends a sequence of probes, where each probe consists of a datagram that has the header bit set to prevent fragmentation. If a datagram is larger than the MTU of a network along the path, the router connected to that network will discard the datagram and send the appropriate ICMP message to the host. The host can then send a smaller probe until one succeeds. As with traceroute, a host must be prepared to retransmit probes for which no response is received.

Copperbelt University
Computer Science Department

<u>Internet Technologies</u>

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES I

Internet Group Management Protocol (IGMP)

The Internet Group Management Protocol (IGMP) is used by hosts and routers that support multicasting. It lets all the systems on a physical network know which hosts currently belong to which multicast groups. This information is required by the multicast routers, so that they know which multicast datagrams to forward onto which interfaces. IGMP is defined in RFC 1112.

Like ICMP, IGMP is considered to be part of the IP layer. Also like ICMP, IGMP messages are transmitted in IP datagrams. Unlike other protocols that we have seen, IGMP has a fixed-size message, with no optional data. Figure 52 shows the encapsulation of an IGMP message within an IP datagram.

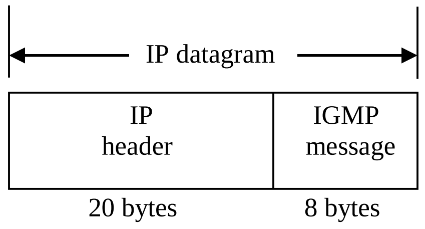


Figure 8: Encapsulation of an IGMP message within an IP datagram.

IGMP messages are specified in the IP datagram with a protocol value of 2.

IGMP Message

Figure 53 shows the format of the 8-byte IGMP message. The IGMP *version* that we are considering is 1. An IGMP *type* of 1 is a query sent by a multicast router and 2 is a response sent by a host. The checksum takes 16 bits. The *group address* is a class D IP address. In a query the group address is set to 0 and in a report it contains the group address being reported.

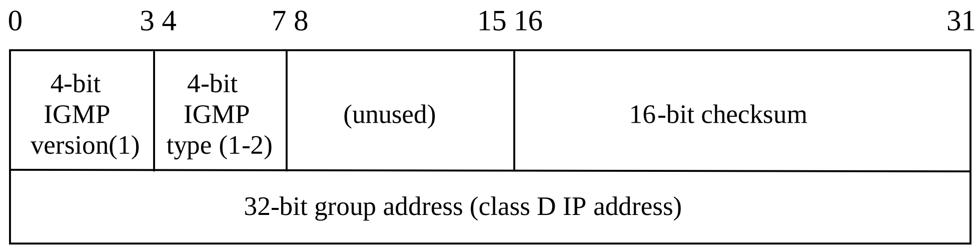


Figure 9: Format of fields in IGMP message

The IGMP Protocol

The IGMP protocol is described in details in the following text.

Joining a multicast Group

Fundamental to multicasting is the process of joining a multicast group on a given interface on a host. Here the term process is used to mean a program being executed by the operating system. Membership in a multicast group on a given interface is dynamic – it changes over time as processes join and leave the group. This means that a process must have a way of joining a multicast group on a given interface. A process can also leave a multicast group that it previously joined. The term *interface* is used here because membership in a group is associated with an interface. A process can join the same group on multiple interfaces.

What is implied here is that a host identifies a group by the group address *and* the interface. A host must therefore, keep a table of all the groups that at least one process belongs to and a reference count of the number of processes belonging to the group.

IGMP Reports and Queries

IGMP messages are used by multicast routers to keep track of group membership on each of the router's physically attached networks. The following rules apply:

1. A host sends an IGMP report when the first process joins a group. If multiple processes on a given host join the same group, only one report is sent, the first time a process joins that group. This report is sent out through the same interface on which the process joined the group.
2. A host does not send a report when processes leave a group, even when the last process leaves a group. The host knows that there are no members in a given group, so when it receives the next query, it will not report the group.
3. A multicast router sends an IGMP query at regular intervals to see if any hosts still have processes belonging to any groups. The router must send one query out through each interface. The group address in the query is 0 since the router expects one response from a host for every group that contains one or more members on the host.
4. A host responds to an IGMP query by sending one IGMP report for each group that still contains at least one process.

Using these queries and reports, a multicast router keeps a table of which of its interfaces have one or more hosts in a multicast group. When the router receives a multicast datagram to forward, it forwards the datagram (using the corresponding multicast link-layer address) only out through the interfaces that still have hosts with processes belonging to that group. Figure 54 shows these two IGMP messages, reports sent by hosts and queries sent by routers. The router is asking each host to identify each group on that interface.

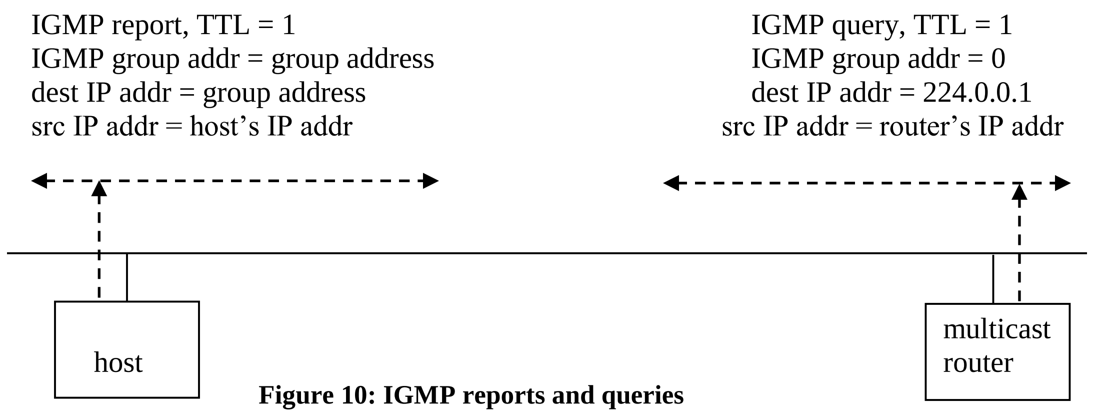

IGMP report, TTL = 1
IGMP group addr = group address
dest IP addr = group address
src IP addr = host's IP addr
IGMP query, TTL = 1
IGMP group addr = 0
dest IP addr = 224.0.0.1
src IP addr = router's IP addr
host
multicast router

Figure 10: IGMP reports and queries

Implementation Details

There are implementation details in this protocol that improve its efficiency. First, when a host sends an initial IGMP report (when the first process joins a group), there is no guarantee that the report is delivered (since IP is used as the delivery service). Another report is sent at a later time. This later time is chosen by the host to be a random value between 0 and 10 seconds.

Next, when a host receives a query from a router it does not respond immediately, but schedules the responses for later times. (The plural word responses is used because the host must send one for each group that contains one or more members.) Since multiple hosts can be sending a report for the same group, each schedules its response using random delays. It is also important to realise that all the hosts on a physical network receive all the reports from other hosts in the same group, because the destination address of the report is the group's address. This means that if a host is scheduled to send a report, but receives a copy of the same report from another host, the response can be cancelled. This is because a multicast router does not care how many hosts belong to the group – only whether at least one host belongs to the group. In deed, a multicast router does not even care which host belongs to the group. It only needs to know that at least one host belongs to a group on a given interface.

On a single physical network without any multicast routers, the only IGMP traffic is the reports issued by the hosts that support IP multicasting, when the host joins a new group.

Time-to-Live Field

In figure 54 we noted that the TTL field of the reports and queries is set to 1. This refers to the normal TTL field in the IP header. A multicast datagram with an initial TTL of 0 is restricted to the same host. By default, multicast datagrams are sent with a TTL of 1. This restricts the datagram to the same subnet. Higher TTLs can be forwarded by multicast routers.

As per standard, an ICMP error message is never generated in response to a datagram destined to a multicast address. Multicast routers do not generate ICMP "time exceeded" errors when the TTL reaches 0.

By increasing the TTL an application can perform an expanding ring search for a particular server. The first multicast datagram is sent with a TTL of 1. If no response is received, a TTL of 2 is tried, then 3 and so on. In this way the application locates the closest server, it terms of hops.

The special range of addresses 224.0.0.0 through 224.0.0.255 is intended for applications that never need to multicast further than one hop. A multicast router should never forward a datagram with one of these addresses as the destination, regardless of the TTL.

All Hosts Group

Figure 54 also shows that the router's IGMP query is sent to the destination IP address of 224.0.0.1. This is called the all-hosts group address. It refers to all the multicast-capable hosts and routers on a physical network. Each host automatically joins this multicast group on all multicast-capable interfaces, when the interface is initialised. Membership in this group is never reported.

To summarise, we can say multicasting is a way to send a message to multiple recipients. In many applications it is better than broadcasting, since multicasting imposes less overhead on hosts that are not participating in the communication. The simple host membership reporting protocol (IGMP) is the basic building block for multicasting.

Multicasting on a LAN or across closely connected LANs uses the techniques that have just been described. Since broadcasting is often restricted to a single LAN, multicasting could be used instead of broadcasting for many applications that use broadcasting today.

A problem that has not been completely solved, however, is multicasting across wide area networks.

Copperbelt University
Computer Science Department

Internet Technologies

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES I

User Datagram Protocol (UDP)

UDP is a simple, datagram-oriented, transport layer protocol; each output operation by a process produces exactly one UDP datagram, which causes one IP datagram to be sent. This is different from a stream-oriented protocol such as TCP where the amount of data written by an application may have little relationship to what actually gets in a single IP datagram. Figure 55 shows the encapsulation of a UDP datagram as an IP datagram.

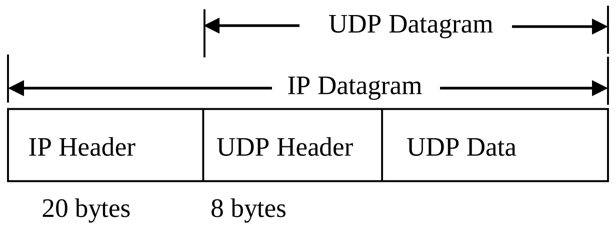


Figure 11: UDP encapsulation

UDP provides no reliability: it sends the datagram that the application writes to the IP layer, but there is no guarantee that they ever reach their destination. Given this lack of reliability, we are tempted to think that we should avoid UDP and always use a reliable protocol such as TCP.

The application needs to worry about the size of the resulting IP datagram. If the datagram exceeds the network's Maximum Transmission Unit (MTU), the IP datagram is fragmented. This applies to each network that the datagram traverses from the source to the destination, not just the first network connected to the sending host.

UDP Header

Figure 56 shows the fields in the UDP header.

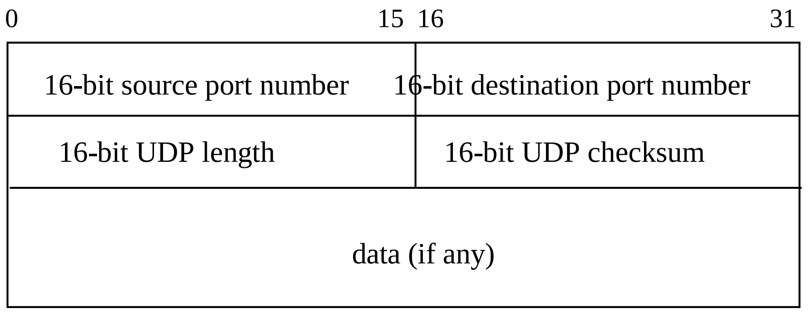


Figure 12: UDP Datagram format

The port numbers identify the sending process and the receiving process. TCP and UDP use the destination port number to demultiplex incoming data from IP. Since IP has already demultiplexed the incoming IP datagram to either TCP or UDP (based on the protocol value in the IP header), it means that TCP port numbers are looked at by TCP and the UDP port numbers by UDP. The TCP port numbers are independent of the UDP port numbers.

Despite this independence, if a well-known service is provided by both TCP and UDP then the port number is normally chosen to be same for both transport layer protocols. This is purely for convenience and is not required by the protocols.

The UDP length field is the length of the UDP header and the UDP data in bytes. The minimum value of this field is 8 bytes. (Sending a UDP datagram with 0 bytes of data is OK.) This UDP length is redundant. The IP datagram contains its total length in bytes, so the length of the UDP datagram is this total length minus the length of the IP header, which is specified by the header length field in the IP datagram format.

UDP Checksum

The UDP checksum covers the UDP header and the UDP data. Let us recall that the checksum in the IP header only covers the IP header – it does not cover any data in the IP datagram. Both TCP and UDP have checksums in their headers to cover their header and their data. With UDP the checksum is optional, while with TCP it is mandatory.

The length of a UDP datagram can be an odd number of bytes, while the checksum algorithm adds 16-bit words. The solution is to append a pad byte of 0 to the end, if necessary, just for the checksum computation. That is, this possible pad byte is not transmitted.

Both UDP and TCP include a 12-byte pseudo-header with the UDP datagram (or TCP segment) just for the checksum computation. This pseudo header includes certain fields from the IP header. The purpose is to let UDP double-check that the data has arrived at the correct destination (i.e., that IP has not accepted a datagram that is not addressed to this host and that IP has not given UDP a datagram that is for another upper layer protocol). Figure 57 shows the pseudo-header along with a UDP datagram.

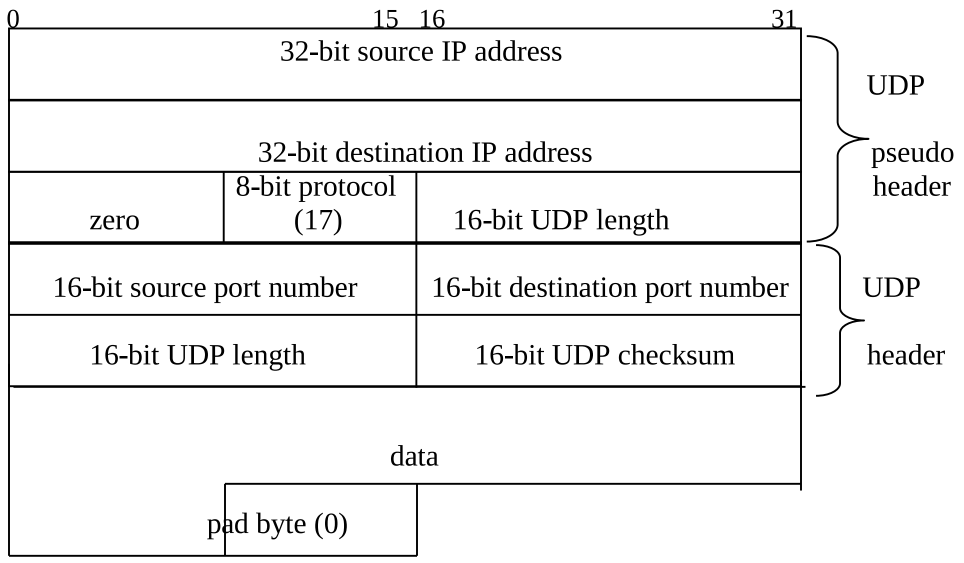


Figure 13: Fields used for computation of UDP checksum

Figure 57 shows explicitly a diagram with an odd length, requiring a pad byte for the checksum computation. Notice that the length of the UDP datagram appears twice in the checksum computation.

If the calculated checksum is 0, it is stored as all one bits (65535), which is equivalent in ones-complement arithmetic. If the transmitted checksum is 0, it indicates that the sender did not compute the checksum.

If the sender did compute a checksum and the receiver detects a checksum error then the UDP datagram is silently discarded. No error message is generated. This is what happens if an IP header checksum error is detected by IP.

The UDP checksum is an end-to-end checksum. It is calculated by the sender and then verified by the receiver. It is designed to catch any modification of the UDP header or data anywhere between the sender and the receiver.

Copperbelt University
Computer Science Department

Internet Technologies

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES I

Transmission Control Protocol (TCP): Reliable Transport Service

TCP achieves a seemingly impossible task: it uses the unreliable datagram service offered by IP when sending data to another computer, but provides a reliable data delivery service to application programs. TCP must compensate for the loss or delay in an internet to provide efficient data transfer, and it must do so without overloading the underlying networks and routers.

The service that the Transmission Control Protocol (TCP) provides to applications

Reliability is the responsibility of a transport protocol. Applications interact with a transport service to send and receive data. In the TCP/IP suite, the *Transmission Control Protocol (TCP)* provides reliable transport service.

From an application program's point of view, the service offered by TCP has seven major features:

1. *Connection Orientation*. TCP provides connection-oriented service in which an application must first request a connection to a destination, and then use the connection to transfer data.

2. *Point-To-Point Communication*. Each TCP connection has exactly two endpoints.

3. *Complete Reliability*. TCP guarantees that the data sent across a connection will be delivered exactly as sent, with no data missing or out of order.

4. *Full Duplex Communication*. A TCP connection allows data to flow in either direction, and allows either application program to send data at any time. TCP can buffer outgoing and incoming data in both directions, making it possible for an application to send data and then continue computation while the data is being transferred.

5. *Stream Interface*. We say that TCP provides a stream interface in which an application sends a continuous sequence of octets across a connection. That is, TCP does not provide a notion of records, and does not guarantee that data will be delivered to the receiving application in the same size pieces that it was transferred by the sending application.

6. *Reliable Connection startup.* TCP requires that when two applications create a connection, both must agree to the new connection; duplicate packets used in previous connections will not appear to be valid responses or otherwise interfere with the new connection.

7. *Graceful Connection Shutdown.* An application program can open a connection, send arbitrary amounts of data, and then request that the connection be shut down. TCP guarantees to deliver all the data reliably before closing the connection.

End-To-End Service and Datagrams

TCP is called an *end-to-end* protocol because it provides a connection directly from an application on one computer to an application on a remote computer. The applications can request that TCP form a connection, send and receive data, and close the connection.

The connections provided by TCP are called virtual connections because they are achieved in software. The underlying internet system does not provide hardware or software support for connections. Instead, the TCP software modules on two machines exchange messages to achieve the illusion of a connection.

TCP uses IP to carry messages. Each TCP message is encapsulated in an IP datagram and sent across the internet. When the datagram arrives on the destination host, IP passes the contents to TCP. Although TCP uses IP to carry messages, IP does not read or interpret the messages. Thus TCP treats IP as a packet communication system that connects hosts at two endpoints of a connection, and IP treats each TCP message as data to be transferred.

Achieving Reliability

A transport protocol like TCP must be carefully designed to achieve reliability. The major problems are: unreliable delivery by the underlying communication system and computer reboot. Consider a situation in which two application programs form a TCP connection, communicate, close the connection, and then form a new connection. Because any message can be lost, duplicated, delayed or delivered out of order, messages from the first connection can be duplicated and a copy delayed long enough for the second connection to be established. Messages must be unambiguous, or the protocol will accept duplicate messages from the old connection and allow them to interfere with the new connection.

Computer system reboot poses another serious challenge to reliable transport protocol designers. Consider a situation where two application programs establish a connection and then one of the computers is rebooted. Although protocol software on the computer that reboots has no knowledge of the connection, protocol software on the computer that did not reboot considers the connection valid. Duplicate packets that are delayed pose an especially difficult challenge because a protocol must be able to reject packets from a previous reboot (i.e., the packets must be rejected even if the protocol software has no record of the previous connection).

Packet loss and retransmission

TCP uses a variety of techniques to achieve reliability. One of the most important techniques is *retransmission*. When TCP sends data, the sender compensates for packet loss by implementing a retransmission scheme. Both sides of a communication participate in this scheme. When TCP receives data, it sends an *acknowledgement* back to the sender. Whenever it sends data, TCP starts a timer. If the timer expires before an acknowledgement arrives, the sender retransmits the data.

TCP's retransmission scheme is the key to its success because it handles communication across an arbitrary internet and allows multiple application programs to communicate concurrently. For example, one application can send data across a satellite channel to a computer in another country, while another application sends data across a local area network to a computer in the next room. TCP must be ready to retransmit any message that is lost on either connection. Acknowledgements from a computer on a local area network are expected to arrive within a few milliseconds. Waiting too long for such an acknowledgement leaves the network idle and does not maximise throughput. Thus, on a local area network, TCP should not wait a long time retransmitting. However, retransmitting after a few milliseconds does not work well on a long-distance satellite connection because the unnecessary traffic consumes network bandwidth and lowers throughput.

TCP faces a more difficult challenge than distinguishing between local and remote destinations: bursts of datagrams can cause congestion, which causes transmission delays along a given path to change rapidly. In fact, the total time required to send a message and receive an acknowledgement can increase or decrease by an order of magnitude in a few milliseconds. In other words, the delay required for data to reach a destination and an acknowledgement to return depends on traffic in the internet as well as the distance to the destination. Because TCP allows multiple application programs to communicate with multiple destinations concurrently and because traffic conditions affect delay, TCP must handle a variety of delays that can change rapidly.

Adaptive retransmission

Before TCP was invented, transport protocols used a fixed value for retransmission delay - the protocol designer chose a value that was large enough for the expected delay. TCP designers, however, realised that a fixed timeout would not work well for an internet. Thus, they chose to make TCP's retransmission adaptive. That is TCP monitors current delay on each connection, and adapts (i.e., changes) the retransmission timer to accommodate changing conditions.

TCP can not know the exact delays for all parts of an internet at all times. Instead, TCP estimates round-trip delay for each active connection by measuring the time needed to receive a response. Whenever it sends a message to which it expects a response, TCP records the time at which the message was sent. When a response arrives, TCP subtracts the time the message was sent from the current time to produce a new estimate of the round-trip delay for that connection. As it sends data packets and receives acknowledgements, TCP generates a sequence of round-trip estimates and uses a statistical function to produce a weighted average. In addition to a weighted average, TCP keeps an estimate of the variance, and uses a linear combination of the estimated mean and variance as a value for retransmission.

Using the variance helps TCP react quickly when delay increases following a burst of packets. Using a weighted average helps TCP reset the retransmission timer if the delay returns to a lower value after a temporary burst. When the delay remains constant, TCP adjusts the retransmission timeout to a value that is slightly longer than the mean round-trip delay. When delays start to vary, TCP adjusts the retransmission timeout to a value greater than the mean to accommodate peaks. In other words, if the delay is large, TCP uses a large retransmission timeout, if the delay is small, TCP uses a small timeout. The goal is to wait long enough to determine that a packet was lost, without waiting longer than necessary.

Buffer, flow control and windows

TCP uses a *window* mechanism to control the flow of data. When a connection is established, each end of the connection allocates a buffer to hold incoming data, and sends the size of the buffer to the other end. As data arrives, the receiver sends acknowledgements, which also specify the remaining buffer size. The amount of buffer space available at any time is called the *window*, and a notification that specifies the size is called a *window advertisement*. A receiver sends a window advertisement with each acknowledgement.

If the receiving application can read data as quickly as it arrives, a receiver will send a positive window advertisement along with each acknowledgement. However, if the sending side operates faster than the receiving side (e.g., because the CPU is faster), incoming data will eventually fill the receiver's buffer, causing the receiver to advertise a *zero window*. A sender that receives a zero window advertisement must stop sending until the receiver again advertises a positive window.

Three-way handshake

To guarantee that connections are established or terminated reliably, TCP uses *a 3-way handshake* in which three messages are exchanged. During the 3-way handshake, each end of the connection includes an initial window advertisement in the message that carries the SYN. A 3-way handshake is necessary and sufficient to ensure unambiguous agreement despite packet loss, duplication and delay.

TCP uses the term *synchronisation segment (SYN segment)* to describe messages in a 3-way handshake used to create a connection, and the term *FIN segment* (short for *finish*) to describe messages in a 3-way handshake used to close a connection. TCP retransmits lost SYN or FIN segments. Furthermore, the handshake guarantees that TCP will not open or close a connection until both ends have interacted. Part of the 3-way handshake used to create a connection requires each end to generate a random 32-bit sequence number. If an application attempts to establish a new TCP connection after a computer reboots, TCP chooses a new random number. Because each new connection receives a new random sequence number, a pair of application programs can use TCP to communicate, close the connection, and then establish a new connection without interference from duplicate or delayed packets.

Congestion control

In most modern internets, packet loss (or extremely long delay) is more likely to be caused by congestion than a hardware failure. Transport protocols that retransmit can worsen the problem of congestion by injecting additional copies of a message. If congestion triggers excessive retransmission, the entire system can reach a state of *congestion collapse*, analogous to a traffic jam on a highway. To avoid the problem, TCP always uses packet loss as a measure of congestion, and responds by reducing the rate at which it retransmits data.

Whenever a message is lost, TCP begins congestion control. Instead of retransmitting enough data to fill the receiver's buffer (i.e., the receiver's window size), TCP begins by sending a single message containing data. If the acknowledgement arrives without additional loss, TCP doubles the amount of data being sent, and sends two additional messages. If acknowledgements arrive for those two, TCP sends four more, and so on. The exponential increase continues until TCP is sending half of the receiver's advertised window, at which time TCP slows down the rate of increase. TCP's congestion control scheme responds well to increased traffic in an internet. By backing of quickly, TCP is able to alleviate congestion. More important, because it avoids adding retransmissions to a congested internet, TCP's congestion control scheme helps prevent congestion collapse.

TCP segment format

TCP uses a single format for all messages, including messages that carry data, acknowledgements and messages that are part of the 3-way handshake used to create or terminate a connection. TCP uses the term segment to refer to a message. Figure 56 illustrates the TCP segment. A TCP connection contains two streams of data, one flowing in each direction. If the applications at each end are sending data simultaneously, TCP can send a single segment that carries the acknowledgement for incoming data, a window advertisement that specifies the amount of additional buffer space that is available for incoming data, and outgoing data. Thus, some of the fields in the segment refer to the data stream travelling in the forward direction, while other fields refer to the data stream that is travelling in the reverse direction.

When a computer sends a segment, the *ACKNOWLEDGEMENT NUMBER* and *WINDOW* fields refer to incoming data: the *ACKNOWLEDGEMENT NUMBER* specifies the sequence number of the data that has been received, and the window specifies how much additional buffer space is available for more data. The *SEQUENCE NUMBER* field refers to outgoing data. It gives the sequence number for the data being carried in the segment. The receiver uses the sequence number to reorder segments that arrive out of order and to compute an acknowledgement number. Field *DESTINATION PORT* identifies which application program on the receiving computer should receive the data, while field *SOURCE PORT* identifies the application program that sent the data. Finally, the *CHECKSUM* filed contains a checksum that covers the TCP segment header and the data. This is a mandatory field that must be calculated and stored by the sender, and then verified by the receiver.

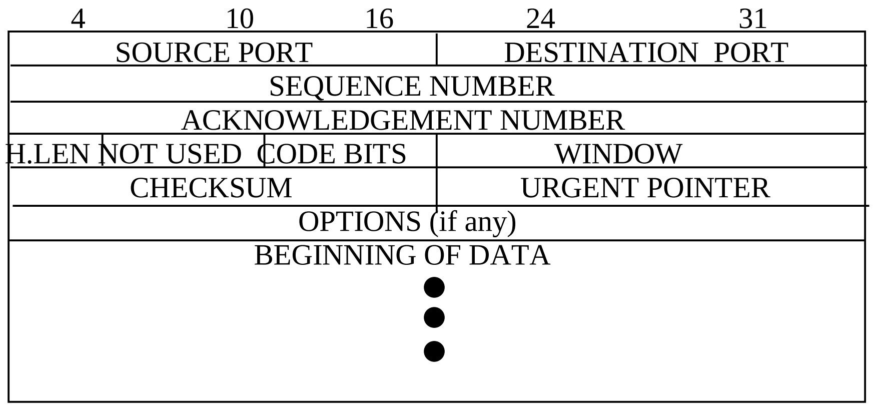


Figure 14: The TCP segment format

Just like with the IP_{v4} header, the field H.len gives the length of the TCP header in 32-bit words. This field includes the options field when options are present. Without options, the size of the TCP header is 20 bytes.

The *CODE BITS* field in the TCP header consists of six flag bits. The code bits are the *SYN*, *FIN*, *ACK*, *URG*, *PSH* and *RST*. One or more of the code bits (flag bits) can be turned on at the same time. These bits are as follows:

URG: The urgent pointer is valid. TCP provides what it calls urgent mode, allowing one end to tell the other end that "urgent data" of some form has been placed into the normal stream of data. The other end is notified that this urgent data has been placed into the data stream, and it is up to the receiving end to decide what to do. The notification from one end to the other that urgent data exists in the data stream is done by setting two fields in the TCP header. The URG bit is turned on and the 16-bit urgent pointer is set to a positive offset that must be added to the sequence number field in the TCP header to obtain the sequence number of the last byte of urgent data.

TCP must inform the receiving process when an urgent pointer is received and one was not already pending on the connection, or if the urgent pointer advances in the data stream. The receiving application can then read the data stream and must be able to tell when the urgent pointer is encountered.

What is urgent mode for? The two most commonly used applications are Telnet and Rlogin, when the interactive user types the interrupt key, for instance. Another application is FTP, when the interactive user aborts a file transfer, for example. It is important that the contents of the data stream that are written by the sender when it enters urgent mode are specially marked by the sender itself. TCP's urgent mode is a way for the sender to transmit emergency data to the other end.

ACK: The acknowledgement number is valid.

PSH: The receiver should pass this data to the application as soon as possible. There should be no need for prolonged buffering.

RST: Reset the connection. A reset is sent by TCP whenever a segment arrives that doesn't appear correct for the referenced connection. The referenced connection means the connection specified by the destination IP address and port number, and the source IP address and the port number (socket).

A common case for generating a reset is when a connection request arrives and no process is listening on the destination port. In the case of the User Datagram Protocol (UDP), an ICMP port unreachable error message is generated when a datagram arrives for a destination port that is not in use. TCP uses a reset instead. Unlike the FIN bit which leads to an *orderly release* of the connection, the RST bit leads to an *abortive release*. Aborting a connection provides two features to the application: (1) any queued data is thrown n away. The receiver of the RST can tell that the other end did an abort instead of a normal close of the connection. Please, note that the RST segment elicits no response from the other end; hence it is not acknowledged at all. (2) The receiver TCP of the RST aborts the connection and advises the application that the connection was reset.

SYN: Synchronise sequence numbers to initiate a connection.

FIN: The sender has finished sending data.

The most common *OPTION* field is the maximum segment size option, called the MSS. Each end of a connection normally specifies this option on the first segment exchanged (the one with the SYN flag set to establish a connection). It specifies the maximum sized segment that the sender wants to receive.

The *DATA* portion of a TCP segment is optional. For instance, when a connection is established and when a connection is terminated, segments are exchanged that contain only the TCP header with possible options. A header without any data is also used to acknowledge received data, if there is no data to be transmitted in that direction. There are also some cases dealing with timeouts when a segment can be sent without any data.

Miscellaneous

A TCP connection is said to be *half-open* if one end has closed or aborted the connection without the knowledge of the other end. This can happen any time one of the two hosts crashes. As long as there is no attempt to transfer data across a half-open connection, the end that is still up will not detect that the other end has crashed. It is possible, although improbable, for two applications to both perform an active open to each other at the same time. Each end must transmit a SYN, and the SYNs must pass each other on the network. It also requires each end to have a local port number that is well known to the other end. This is called *simultaneous open*.

Although often, but not always, the client in a TCP connection performs the active close, causing the first FIN to be sent. It is also possible for both sides to perform an active close and the TCP protocol allows for this *simultaneous close*. When establishing a TCP connection, the side that sends the first SYN is said to perform an *active open*. The other side, which receives this SYN and sends the next SYN performs a passive open. On the other hand, when terminating a connection, the side that send the first FIN is said to perform the active close and the other side that receives this FIN performs the *passive close*.

Unit 1 Client Server Interaction

The Internet is a vast interconnected collection of computer networks of many different types. Programs running on the computers connected to it interact by passing messages, employing a common means of communication. The

design and construction of the Internet communication mechanisms (the internet protocols) is a major technical achievement, enabling a program running anywhere to address messages to programs anywhere else.
Internets provide a general communication infrastructure without specifying which services will be offered, which computers will run those services, how the availability of services will become known, or how services will be used - such issues are left to application software and users. Protocol software does not know when to initiate contact with, or when to accept incoming communication from, a remote computer. Instead, communication across an internet requires a pair of application programs to co-operate. An application on one computer attempts to communicate with an application on another (the analogue of placing a telephone call), and an application on the other computer answers the incoming request (the analogue of answering a telephone call). In other words, one application initiates communication and the other accepts it.

Initializing Communication
Protocol software does not have a mechanism analogous to a telephone bell - there is no way for protocol software to inform an application that communication has arrived, and no way for an application to agree to accept arbitrary incoming messages. Instead of waiting for an arbitrary message to arrive, an application that expects communication must interact with protocol software before an external source attempts to communicate. The application informs local protocol software that a specific type of message is expected, and then the application waits. When an incoming message matches exactly what the application has specified, protocol software passes the message to the application. Two applications involved in a communication can not both wait for a message to arrive - one application must actively initiate interaction, while the other application waits passively.

Copperbelt University
Computer Science Department

<u>Internet Technologies</u>

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

The client-server paradigm

The paradigm of arranging for one application program to wait for another application to initiate communication pervades so much of distributed computing that it has been given the name: *client-server paradigm of interaction*. The terms *client* and *server* refer to the two applications involved in a communication. The application that actively initiates contact is called a client, while the application that passively waits for contact is called a server.

Characteristics of Clients and Servers

In general, client software has the following characteristics;

- It is an application program that becomes a client temporarily when remote access is needed, but performs other computation locally.
- It is invoked by a user and executes for one session.
- It runs locally on the user's computer.
- It actively initiates contact with a server (*CONNECT* primitive).
- It can access multiple services as needed.

In general, server software has the following characteristics;

- It is a special-purpose program dedicated to providing one service.
- It is invoked automatically when a system boots, and continues to execute through many sessions.
- It runs on a shared computer.
- It waits passively for contact from arbitrary remote clients (*LISTEN* primitive).
- It accepts contact from arbitrary clients, but offers a single service.

Note that the word *server* is (strictly) referring to a piece of software. However, a computer running one or more servers is often (incorrectly) called a *server*. Like most application programs, a client and a server use a transport protocol to communicate. The figure below illustrates a client and a server using the TCP/IP protocol stack.

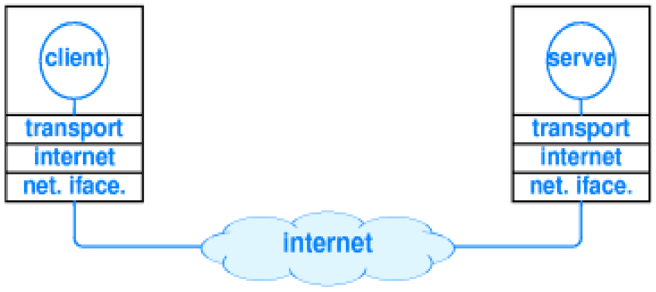


Figure showing a client and a server

Requests, responses and direction of data flow
Information can pass in either or both directions between a client and a server. Typically, a client sends a request to a server and the server returns a response to the client. In some cases, a client sends a series of requests and the server issues a series of responses (e.g., a database client might allow a user to look up more than one item at a time). In other cases, the server provides continuous output without any request - as soon as the client contacts the server, the server begins sending data (e.g., a local weather server might send continuous weather reports with updated temperature and barometric pressure).

It is important to understand that servers can accept incoming information as well as deliver outgoing information. For example, most file servers are configured to export a set of files to clients. That is, a client sends a request that contains the name of a file, and the server responds by sending a copy of the file. However, a file server can also be configured to import files (i.e., to allow a client to send a copy of a file which the server accepts and stores on disk). In other words, information can flow in either or both directions between a client and server.

Transport protocols and client-server interaction
Like most application programs, a client and server use a transport protocol to communicate. A client or server application interacts directly with a transport-layer protocol to establish communication and send or receive information. The transport protocol then uses lower layer protocols to send and receive individual messages. Thus a computer needs a complete stack of protocols to run either a client or a server.

Multiple services on one computer
The client and server each interact with a protocol in the transport layer of the stack. A sufficiently powerful computer can run multiple servers and clients at the same time. Such a computer must have the necessary hardware resources (e.g. a fast CPU and sufficient memory) and have an operating system capable of running multiple applications concurrently (e.g. UNIX or Windows). The figure below illustrates such a setup.

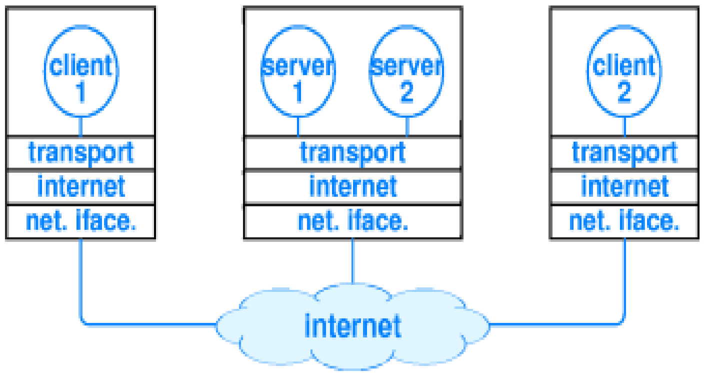


Figure showing two servers

The computer in the middle might be running an FTP server and a WWW server. Modern computers are often capable of running many servers at the same time.

On such systems, one server program runs for each service being offered. For example, a single computer might run a file server as well as a World Wide Web server. Although a computer can operate multiple servers, the computer needs only a single physical connection to the internet.

Allowing a computer to operate multiple servers is useful because the hardware can be shared by multiple services. Consolidating servers on to a large, server-class computer also helps reduce the overhead of system administration because it results in fewer computer systems to administer. Experience has shown that the demand for a server is often sporadic - a given server can remain idle for long periods of time. An idle server does not use the computer's CPU while waiting for a request to arrive. Thus, if demand for services is low, consolidating servers on a single computer can dramatically reduce cost without significantly reducing performance. It is worthy noting, however, that the highest performance is achieved by running a single server or client on each single computer.

Identifying a particular service
Transport protocols provide a mechanism that allows a client to specify unambiguously which service is desired. The mechanism assigns each service a unique identifier, and requires both the server and client to use the identifier. When a server begins execution, it registers with the local protocol software by specifying the identifier for the service it offers. When a client contacts a remote server, the client specifies the identifier for the desired service. Transport protocol software on the client's machine sends the identifier to the server's machine when making a request. Transport protocol software on the server's machine uses the identifier to determine which server program should handle the request.

As an example of service identification, consider the Transmission Control Protocol (TCP). TCP uses 16-bit integer values known as *protocol port numbers* to identify services, and assigns a unique protocol port number to each service. A server specifies the protocol port number for the service it offers, and then waits passively for communication. A client specifies the protocol port number of the desired service when sending a request. Conceptually, TCP software on the server's computer uses the protocol port number in an incoming message to determine which server should receive the request.

Multiple copies of a server for a single service
Technically, a computer system that permits multiple application programs to execute at the same time is said to support *concurrency*, and a program that has more than one thread of control (also called *process* or *task*) is called a

concurrent program. Concurrency is fundamental to the client-server model of interaction because a concurrent server offers service to multiple clients at the same time, without requiring each client to wait for previous clients to finish.

To understand why simultaneous service is important, consider what happens if a service requires significant time to satisfy each request. For example, a file transfer service allows a client to obtain a copy of a remote file: the client sends the name of the file in a request, and the server returns a copy of the file. If a client requests a small file, the server can send the entire file in a few milliseconds. However, a server may require several minutes to transfer a file that contains a series of high-resolution digital images. If a file server handles one request at a time, all clients must wait while the server transfers a file to one of them. In contrast, a concurrent file server can handle multiple clients simultaneously. When a request arrives, the server assigns the request to a thread of control that can execute concurrently with existing threads. In essence, a separate copy of the server handles each request. Thus, short requests can be satisfied quickly, without waiting for long requests to complete.

Dynamic server creation
Most concurrent servers operate dynamically. The server creates a new thread for each request that arrives. The server program, actually, is constructed in two parts: one that accepts requests and creates a new thread for the request, and another that consists of the code to handle an individual request. When a concurrent server starts executing only the first part runs. That is, the main server thread waits for a request to arrive. When a request arrives, the main thread creates a new service thread to handle the request. The service thread handles one request and then terminates. Meanwhile, the main thread keeps the server alive - after creating a thread to handle a request, the main thread waits for another request to arrive. If N clients are using a given service on a single computer, there are N + 1 threads providing the service: the main thread is waiting for additional requests, and N service threads are each interacting with a single client.

Transport protocol and unambiguous communication
If multiple copies of a server exist, how can a client interact with the correct copy? The answer lies in the method that transport protocols use to identify a server. Each service is assigned a unique identifier and each request from a client includes the service identifier, making it possible for transport protocol software on the server's computer to associate the incoming request with the correct server. As an example, consider the identifier used on a TCP connection. TCP requires each client to choose a local protocol port number that is not assigned to any service. When it sends a TCP segment, a client must place its local protocol number in the *SOURCE PORT* filed and the protocol number of the server in the *DESTINATION PORT* field. On the server's computer, TCP uses the combination of source and destination protocol port numbers (as well as client and server IP addresses) to identify a particular communication. Thus, messages can arrive from two or more clients for the same service without causing a problem. TCP passes each incoming segment to the copy of the server that has agreed to handle the client.

Connection oriented and connectionless transport
Transport protocols support two basic forms of communication: connection-oriented or connectionless. To use a connection-oriented transport protocol, two applications must establish a connection, and then send data across the connection. For example, TCP provides a connection-oriented interface to applications. When it uses TCP, an application must first request TCP to open a connection to another application. Once the connection is in place, two applications can exchange data. When the applications finish communicating, the connection must be closed.

The alternative to connection-oriented communication is a connectionless interface that permits an application to send a message to any destination at a time. When using a connectionless transport protocol, the sending application must specify a destination with each message it sends. For example, in the TCP/IP protocol suite, *the User Datagram Protocol (UDP)* provides connectionless transport. An application using UDP can send a sequence of messages, where each message is sent to a different destination.

Clients and servers can use either connection-oriented or connectionless transport protocols to communicate. When using a connection-oriented transport, a client first forms a connection to a specific server. The connection then stays in place while the client sends requests and receives responses. When it finishes using the service, the client closes the connection.

Clients and servers that use connectionless protocols exchange individual messages. For example, many services that use connectionless transport require a client to send each request in a single message, and the server to return each response in a single message.

A service reachable through multiple protocols
Services need not choose between connectionless and connection-oriented transport; it is possible to offer both. That is, the same service can be made available over two or more transport protocols, with the choice of transport left to the client. Allowing the use of multiple transport protocols increases flexibility by making the service available to clients that may not have access to a particular transport protocol.

There are two possible implementations of a multi-protocol server. In the first implementation, two servers exist for the same service. One server uses connectionless transport, while the other uses connection-oriented transport. In the second implementation, a single server program can interact with two or more transport protocols. The server accepts requests that arrive from either protocol, and uses the protocol in which the request arrived when sending a response.

Complex client-server interactions
Some of the most interesting and useful functionality of client-server computing arises from arbitrary interactions among clients and servers. In particular, it should be noted that:

- A client application is not restricted to accessing a single service. A single application can first become a client of one service, and later become a client of another. The client contacts a different server (perhaps on a different computer) for each service.

- A client application is not restricted to accessing a single server for a given service. In some services, each server provides different information than servers running on other computers. For example, a date server might provide the current time and date for the computer on which it runs. A server on a computer in a different time zone gives a different answer. In other services, all servers provide the same information. In such cases, a client might send a request to multiple servers to improve performance - the client uses the information sent by the server that responds first.

- A server is not restricted from performing further client-server interactions - a server for one service can become a client for another. For example, a file server that needs to record the time that a file was accessed might become a client of a time server. That is, while it is handling a request for a file, the file server sends a request to a time server, waits for the response, and then continues handling the file request.

Interactions and circular dependencies
Servers must be planned carefully to avoid circular dependencies. Consider a file server that uses a time server to obtain the current time whenever a file is accessed. A circular dependency can result if the time server also uses the file server. For example, suppose a programmer is asked to modify the time server so that it keeps a record of each request. If the programmer chooses to have the time server become a client of the file server, a cycle can result. The file server becomes a client of the time server which becomes a client of the file server, and so on. The result is a disaster analogous to an infinite loop in a program.

Although dependencies among a pair of servers are easy to spot and avoid, a larger set of dependencies may not be obvious. Imagine a dependency cycle that includes a dozen servers, each operating on a separate computer. If each server is maintained by a separate programmer, the dependencies among them may be difficult to identify.

Copperbelt University
Computer Science Department

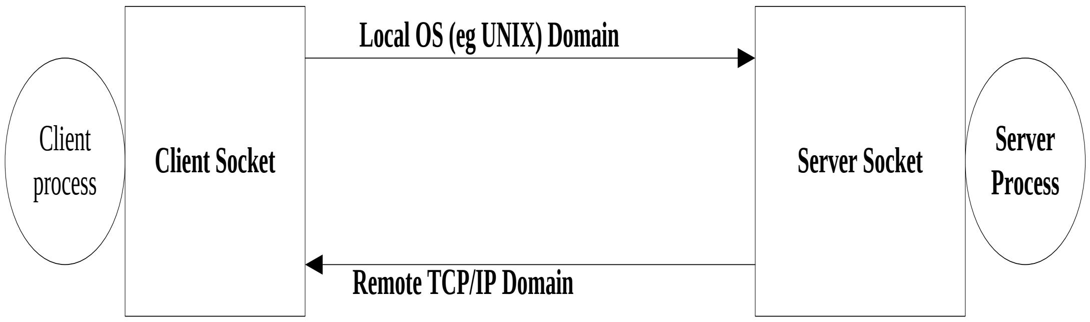
<u>Internet Technologies</u>

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

The Socket Interface

The socket interface provides additional details about client-server interaction by explaining the interface between an application and protocol software. A socket can be thought of as reference point to which messages can be sent and from which messages can be received. Any process can create a socket to communicate with another process but, both processes must create its own socket, since the two sockets are used as a pair. The figure below shows a typical representation of sockets.



Sockets can also be defined as software abstractions of ports used within running processes. Messages are sent between a pair of sockets, each of which is attached or bound to a port. This binding may be automatic or may need to be done explicitly. To send a message to a socket the send must know the IP address of the host and the port number to which the socket is bound. Each port is associated with a particular protocol i.e. UDP (Datagrams) or TCP (Streams).

Application program interface (API)

When it interacts with protocol software, an application must specify details such as whether it is a server or client (i.e., whether it will wait passively or actively initiate communication). In addition, applications that communicate must specify further details (e.g., the sender must specify the data to be sent, and the receiver must specify where incoming data should be placed). The interface an application uses when it interacts with transport protocol software is known as an *Application Program Interface (API)*. An API defines a set of operations that an application can perform when it interacts with protocol software. Thus, the API determines the functionality that is available to an application as well as the difficulty of creating a program to use that functionality. Most programming systems define an API by giving a set of procedures that the application can call and the arguments that each procedure expects. Usually, an API contains a separate procedure for each basic operation. For example, an API might contain one procedure that is used to establish communication and another procedure that is used to send data. The following examples show a client or server can use a Java API for UDP Datagram.

UDP Example Client using Java API for UDP datagram

```
import java.net.*;
import java.io.*;
public class UDPClient { // sends a message to the server and gets a reply
public static void main(String args[]) {
    // args give message contents and server hostname
    try {
            Datagram aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
            aSocket.close();
    }
    catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
    catch (IOException e){System.out.println("IO: " + e.getMessage());}
    }
}
```

UDP Example — Server using Java UDP datagram

```
import java.net.*;
import java.io.*;
public class UDPServer { // receives a request and sends it back to the client
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        }
        catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        finally{ if(aSocket != null) aSocket.close();}
        }
}
```

The socket API

Communication protocol standards do not usually specify an API that applications use to interact with the protocols. Instead, the protocols specify the general operations that should be provided and allow each operating system to define the specific API an application uses to perform the operations. Thus, a protocol standard might suggest an operation is needed to allow an application to send data, and the API specifies the exact name of the function and the type of each argument. Although protocol standards allow operating system designers to choose an API, many have adopted the *socket API* (sometimes just shortened to *sockets*). The socket API is available for many operating systems, including systems used on personal computers (e.g., Windows 98, NT, 2000) as well as various UNIX systems (e.g., Sun Microsystems Solaris). The socket API originated as part of the BSD UNIX operating system and has become the *de facto* standard in the industry. To sum up, the interface between an application program and the communication protocols in an operating system is known as the Application Program Interface (API).

Socket communication and UNIX I/O

Because they were originally developed as part of the UNIX operating system, sockets employ many concepts found in other parts of UNIX. In particular, sockets are integrated with I/O - an application communicates through a socket similar to the way the application transfers data to or from a file. Thus, understanding sockets requires one to understand UNIX I/O facilities.
UNIX uses an *open-read-write-close* paradigm for all I/O. The name is derived from the basic I/O operations that apply to both devices and files. For example, an application must first call *open* to prepare a file for access. The application then calls read or write to retrieve data from the file or store data into the file. Finally, the application calls *close* to specify that it has finished using the file. When an application opens a file or device, the call to *open* returns a *descriptor*, a small integer that identifies the file; the application must specify the descriptor when requesting data transfer (i.e., the descriptor is an argument to the *read* or *write* procedure). For example, if an application calls *open*

to access a file named *cbufile,* the open procedure might return descriptor 4. A subsequent call to *write* that specifies descriptor 4 will cause data to be written to the file *cbufile*; the file name does not appear in the call to *write*.

Sockets, descriptors and network I/O

Socket communication also uses the descriptor approach. Before an application can use protocols to communicate, the application must request the operating system to create a *socket* that will be used for communication. The system returns a small integer descriptor that identifies the socket. The application then passes the descriptor as an argument when it calls procedures to transfer data across the network; the application does not need to specify details about the remote destination each time it transfers data.

Parameters and socket API

Socket programming differs from conventional I/O because an application must specify many details to use a socket. For example, an application must choose a particular transport protocol, provide the protocol address of a remote machine, and specify whether the application is a client or a server. To accommodate all the details, each socket has many parameters and options - an application can supply values for each. To avoid having a single socket function with separate parameters for each option, designers of the socket API chose to define many functions. In essence, an application creates a socket and then invokes functions to specify in detail how the socket will be used. The advantage of the socket approach is that most functions have three or fewer parameters; the disadvantage is that a programmer must remember to call multiple functions when using sockets.

Procedures that implement the socket API

Several procedures have been defined to implement the socket API.

1.The socket procedure

The socket procedure creates a socket and returns an integer descriptor:
descriptor = socket(protofamily, type, protocol)

The argument *protofamily* specifies the protocol family to be used with the socket. For example, the value PF_INET is used to specify the TCP/IP protocol suite, and PF_DECnet is used to specify Digital Equipment Corporation protocols. The argument *type* specifies the type of communication that the socket will use. The two most common types are a connection-oriented stream transfer (specified with the value SOCK_STREAM) and a connectionless message-oriented transfer (specified with the value SOCK_DGRAM). The argument *protocol* specifies a particular transport protocol used with the socket. Having a protocol argument in addition to *type* argument, permits a single protocol suite to include two or more protocols that provide the same service. The values that can be used with the *protocol* argument depend on the protocol family. For example, although the TCP/IP protocol suite includes the protocol TCP, the AppleTalk suite does not.

2.The close procedure

The close procedure tells the system to terminate the use of a socket. It has the form:
close(socket)

where *socket* is the descriptor for the socket being closed. If the socket is using a connection-oriented transport protocol, *close* terminates the connection before closing the socket. Closing a socket immediately terminates use - the descriptor is released, preventing the application from sending more data, and the transport protocol stops accepting incoming messages directed to the socket, preventing the application from receiving more data.

3.The bind procedure

When created, a socket has neither a local address nor a remote address. A server uses the *bind* procedure to supply a protocol port number at which the server will wait for contact. *Bind* takes three arguments:

bind(socket, localaddr, addrlen)

The argument *socket* is the descriptor of a socket that has been created but not previously bound; the call is a request that the socket be assigned a particular protocol port number. The argument *localaddr* is a structure that specifies the local address to be assigned to the socket, and argument *addrlen* is an integer that specifies the length of the address. Because sockets can be used with arbitrary protocols, the format of an address depends on the protocol being used. The socket API defines a generic form used to represent addresses, and then requires each protocol family to specify how their protocol addresses use the generic form. The generic format for representing an address is defined to be a *sockaddr* structure. The structure has three fields as given below:

```
struct sockaddr {
        u_char   sa_len;              /* total length of the address */
        u-char   sa_family;           /* family of the address       */
        char     sa_data[14];         /*  the address itself         */
};
```

The field *sa_len* consists of a single octet that specifies the length of the address. The field *sa_family* specifies the family to which an address belongs (the symbolic constant AF_INET is used for TCP/IP addresses). Finally, the field *sa_data* contains the address.

Each protocol family defines the exact format of addresses used with the *sa_data* field of a *sockaddr* structure. For example, TCP/IP protocols use the structure *sockaddr_in* to define an address:

```
struct sockaddr_in {
        u_char   sin_len;             /* total length of the address */
        u_char   sin_family;          /* family of the address         */
        u_short  sin_port;      /* protocol port number        */
        struct   in_addr  sin-addr;   /* IP address of computer    */
        char     sin_zero[8];         /* not used (set to zero)      */
};
```

The first two fields of the structure *sockaddr_in* correspond exactly to the first two fields of the generic *sockaddr* structure. The last three fields define the exact form of address that TCP/IP protocols expect. There are two points to notice. First, each address identifies both a computer and a particular application on that computer. The field *sin_addr* contains the IP address of the computer and the field *sin_port* contains the protocol port number of an application. Second, although TCP/IP needs only six octets to store a complete address, the generic *sockaddr* structure reserves fourteen octets. Thus, the final field in structure *sockaddr_in* defines an 8-octet field of zeroes, which pad the structure to the same size as *sockaddr*.

A server calls *bind* to specify a protocol port number at which the server will accept contact. However, in addition to a protocol port number, structure *sockaddr_in* contains a field for an IP address. Although a server can choose to fill in the IP address when specifying an address, doing so causes problems when a host is multi-homed because it means the server only accepts requests sent to one specific address. To allow a server to operate on a multi-homed host, the socket API includes a special symbolic constant, INADDR_ANY, that allows a server to use a specific port at any of the computer's IP addresses.

4.The listen procedure

After specifying a protocol port, a server must instruct the operating system to place a socket in passive mode so it can be used to wait for contact from clients. To do so, a server calls the *listen* procedure, which takes two arguments: *listen(socket, queuesize)*

The argument *socket* is the descriptor of a socket that has been created and bound to a local address, and the argument *queuesize* specifies a length for the socket's request queue. The operating system builds a separate request queue for each socket. Initially, the queue is empty. As requests arrive from clients, each is placed in the queue; when the server asks to retrieve an incoming request from the socket, the system returns the next request from the queue. If the queue is full when a request arrives, the system rejects the request. Having a queue of requests allows the system to hold new requests that arrive while the server is busy handling a previous request. The parameter allows each server to choose a maximum queue size that is appropriate for the expected service.

5. The accept procedure

All servers begin by calling *socket* to create a socket and *bind* to specify a protocol port number. After executing the two calls, a server that uses a connectionless transport protocol is ready to accept messages. However, a server that uses a connection-oriented transport protocol requires additional steps before it can receive messages: the server must call *listen* to place the socket in passive mode, and must then accept a connection request. Once a connection has been accepted, the server can use the connection to communicate with a client. After it finishes communication, the server closes the connection.

A server that uses connection-oriented transport must call the procedure *accept* to accept the next connection request. If a request is present in the queue, *accept* returns immediately; if no requests have arrived, the system blocks the server until a client forms a connection. The *accept* call has the form:
newsock = accept(socket, caddress, caddresslen)

The argument *socket* is the descriptor of a socket the server has created and bound to a specific protocol port. The argument *caddress* is the address of a structure of type *sockaddr* and *caddrsslen* is a pointer to an integer. *Accept* fills in fields of argument *caddress* with the address of the client that formed the connection and sets *caddresslen* to the length of the address. Finally, *accept* creates a new socket for the connection, and returns the descriptor of the new socket to the caller. The server uses the new socket to communicate with the client, and then closes the socket when finished. Meanwhile, the server's original socket remains unchanged - after it finishes communicating with a client, the server uses the original socket to accept the next connection from a client.

6. The connect procedure

Clients use the procedure *connect* to establish a connection with a specific server. The form is:
connect(socket, saddress, saddresslen)

The argument *socket* is the descriptor of a socket on the client's computer to use for the connection. The argument *saddress* is a *sockaddr* structure that specifies the server's address and protocol port number. The argument *saddresslen* specifies the length of the server's address measured in octets.

When used with a connection-oriented transport protocol such as TCP, *connect* initiates a transport-level connection to the specified server. In essence, *connect* is the procedure a client uses to contact a server that has called accept. A client that uses a connectionless transport can also call *connect*. However, doing so does not initiate a connection or cause a packet to cross the internet. Instead, *connect* merely marks the socket *connected*, and records the address of the server.

In other words, the connect procedure, which is called by clients, has two uses. When used with connection-oriented transport, connect establishes a transport connection to a specified server. When used with connectionless transport, connect records the server's address in the socket, allowing the client to sent many messages to the same server without requiring the client to specify the destination address with each message.

The combination of an IP address and a protocol port number is sometimes called an *endpoint address.*

7. The send, sendto, and sendmsg procedures

Both clients and servers need to send information. Usually, a client sends a request and a server sends a response. If the socket is connected, procedure *send* can be used to transfer data. *Send* has four arguments:
send(socket, data, length, flags)

The argument *socket* is the descriptor of a socket to use, the argument *data* is the address in memory of the data to send, the argument *length* is an integer that specifies the number of octets of data and the argument *flags* contains bits that request special options. However, many options are intended for system debugging, and are not available to conventional client and server programs.

The procedures sendto and sendmsg allow a client or server to send a message using an unconnected socket; both require the caller to specify a destination. Sendto, takes the destination address as an argument. It has the form:
sendto(socket, data, length, flags, destaddress, addresslen)

The first four arguments correspond to the four arguments of the *send* procedure. The final two arguments specify the address of a destination and the length of that address. The form of the address in argument *destaddress* is the *sockaddr* structure (specifically, structure *sockaddr_in* when used with TCP/IP).

The *sendmsg* procedure performs the same operation as *sendto*, but abbreviates the arguments by defining a structure. The shorter argument list can make programs that use *sendmsg* easier to read:
sendmsg(socket, msgstruct, flags)

The argument *msgstruct* is a structure that contains information about the destination address, the length of the address, the message to be sent, and the length of the message:

```
struct msgstruct{                           /* structure used by sendmsg        */
        struct   sockaddr      *m_saddr;       /* ptr to destination address       */
        struct   datavec       *m_dvec;        /* ptr to message (vector)  */
        int      m_dvlength;                   /* num. of items in vector          */
        struct   access        *m_rights;      /* ptr to access rights list        */
        int      m_alength;                    /* num. of items in list            */
};
```

The structure should be viewed as a way to combine many arguments into a single structure. Most applications use only the first three fields, which specify a destination protocol address and a list of data items that comprise the message.

8. The recv, recvfrom and recvmesg procedures

A client and a server each need to receive data sent by the other. The socket API provides several procedures that can be used. For example, an application can all *recv* to receive data from a connected socket. The procedure has the form:
recv(socket, buffer, length, flags)

The argument *socket* is the descriptor of a socket from which data is to be received. The argument *buffer* specifies the address in memory in which the incoming message should be placed, and the argument length specifies the size of the buffer. The argument *flags* allows the caller to control details (e.g., to allow an application to extract a copy of an incoming message without removing the message from the socket).

If a socket is not connected, it can be used to receive messages from an arbitrary set of clients. In such cases, the system returns the address of the sender along with each incoming message. Applications use the procedure *recvfrom* to receive both a message and the address of the sender:
recvfrom(socket, buffer, length, flags, sndaddr, saddrlen)

The first four arguments correspond to the arguments of *recv*. The two additional arguments, *sndaddr* and *saddrlen*, are used to record the sender's IP address. Argument *sndaddr* is a pointer to a *sockaddr* structure into which the system writes the sender's address, and argument *saddrlen* is a pointer to an integer that the system uses to record the length of the address. *Recvfrom* records the sender's address in exactly the same form that *sendto* expects. Thus, if an application uses *recvfrom* to receive an incoming message, sending a reply is easy - the application simply uses the recorded address as a destination for the reply.

The socket API includes an input procedure analogous to the *sendmsg* output procedure. Procedure *recvmsg* operates like *recvfrom*, but requires fewer arguments. It has the form:
recvmsg(socket, msgstruct, flags)

where argument *msgstruct* gives the address of a structure that holds the address for an incoming message as well as locations for the sender's IP address. The *msgstruct* recorded by *recvmsg* uses exactly the same format as the structure required by *sendmsg*. Thus, the two procedures work well for receiving a message and sending a reply.

Read and write with sockets

The socket API was originally designed to be part of UNIX, which uses *read* and *write* for I/O. Consequently, sockets also allow applications to use *read* and *write* to transfer data. Like *send* and *recv*, *read* and *write* do not have arguments that permit the caller to specify a destination. Instead, read and write each have three arguments: a socket descriptor, the location of a buffer in memory used to store the data, and the length of the memory buffer. Thus, *read* and *write* must be used with connected sockets.

The chief advantage of using *read* and *write* is generality - an application program can be created that transfers data to or from a descriptor without knowing whether the descriptor corresponds to a file or a socket. Thus, a programmer can use a file on a local disk to test a client or server before attempting to communicate across a network. The chief disadvantage of using *read* and *write* is that a socket library implementation may introduce additional overhead in the file I/O of any application that also uses sockets.

Other socket procedures

The socket API contains other useful procedures. For example, after a server calls procedure *accept* to accept an incoming connection request, the server can call procedure *getpeername* to obtain the complete address of the remote client that initiated the connection. A client or server can also call *gethostname* to obtain information about the computer on which it is running.

A socket has many parameters and options. Two general-purpose procedures are used to set socket options or obtain a list of current values. An application calls procedure *setsockopt* to store values in socket options, and procedure *getsockopt* to obtain current option values. Options are used mainly to handle special cases (e.g., to increase performance by changing the internal buffer size the protocol software uses).

Two procedures are used to translate between IP addresses and computer names. Procedure *gethostbyname* returns the IP address for a computer given the computer's name. Clients often use *gethostbyname* to translate a name entered by a user into a corresponding IP address needed by the protocol software. Procedure *gethostbyaddr* provides an inverse mapping - given an IP address for a computer, it returns the computer's name. Clients and servers can use *gethostbyaddr* when displaying information for a person to read.

Example of a client and a sever

Some of the socket API procedures can be demonstrated on an example.

Connection-oriented communication

A client and server must select a transport protocol that supports connectionless service or one that supports connection-oriented service. Connectionless service allows an application to send a message to an arbitrary destination at any time. The destination does not need to agree that it will accept the message before transmission occurs. In contrast, a connection-oriented service requires two applications to establish a transport connection before data can be sent. To establish a connection, the applications each interact with transport protocol software on their local computer, and the two transport protocol modules exchange messages across the network. After both sides agree that a connection has been established, the applications can send data.

This example will show how software for a connection-oriented service uses sockets. In the example, the server keeps a count of the number of clients that have accessed the service and then reports the count whenever a client contacts the server. In the example, a client forms a connection to the server and waits for output. Whenever, a connection request arrives, the server creates a message in printable ASCII form, sends the message over the connection, and then closes the connection. The client displays the data it receives and then exits. For example, the eighth time a client connects to the server, the client receives and prints the following message: *This server has been contacted 8 times.*

The server takes one command-line argument, a protocol port number at which to accept requests. The argument is optional, that is, if no port number is specified, the code uses a port number that the programmer chooses, e.g., port 7653. Care must be taken that the port used should not conflict with existing ports on the computer used.

The example client has two command-line arguments: the name of a host on which to contact a server and a protocol port number to use. Both arguments are optional. If no protocol port number is specified, the client uses 7653. If neither argument is specified, the client uses the default port and the host name *localhost*, which is usually an alias that maps to the computer on which the client is running.

Sequence of socket procedure calls

Figure 1 illustrates the sequence of socket procedures that the example client and server call. As the figure shows, the server calls seven socket procedures and the client calls six. The client begins by calling library procedures *gethostbyname* to convert the name of a computer to an IP address and *getprotobyname* to convert the name of a protocol to the internal binary form used by the socket procedure. The client then calls *socket* to create a socket and *connect* to connect the socket to a server. Once the connection is in place, the client repeatedly calls *recv* to receive the data that the sender sends. Finally, after all data has been received, the client calls *close* to close the socket.

The server also calls *getprotobyname* to generate the internal binary identifier for the protocol before calling *socket* to create a socket. Once a socket has been created, the server calls *bind* to specify a local protocol port for the socket and *listen* to place the socket in passive mode. The server then enters an infinite loop in which it calls *accept* to accept the next incoming connection request, *send* to send a message to the client, and *close* to close the new connection. After closing a connection, the server calls *accept* to extract the next incoming connection.

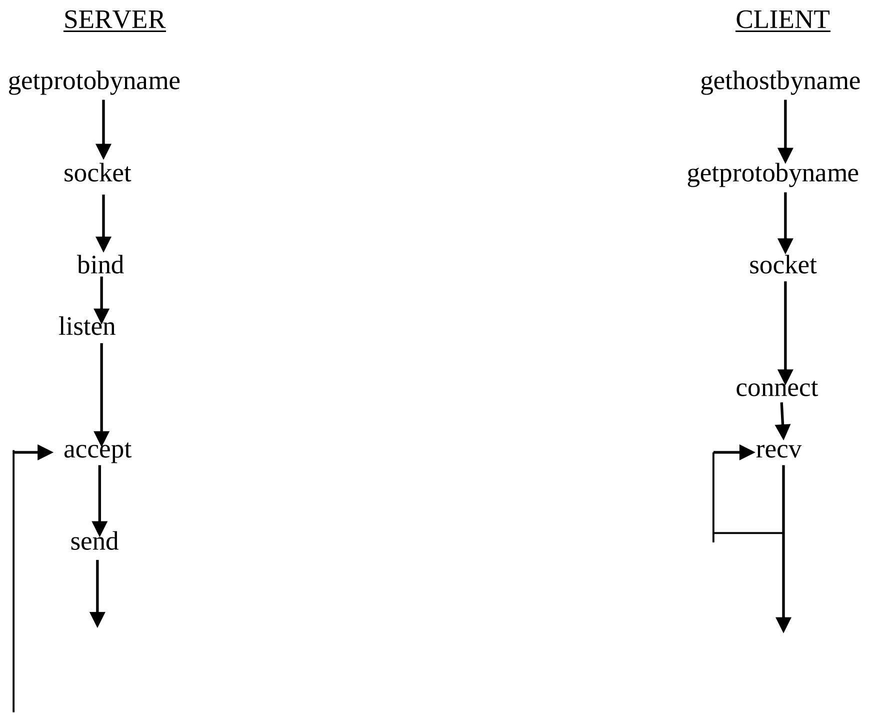


Figure 15: The sequence of socket procedure calls in the example client and server. The server must call *listen* before a client calls *connect*.

Stream service and multiple recv calls

Although the server makes only one call to *send* to transmit data, the client code iterates to receive data. During each iteration, the client calls *recv* to obtain data. The iteration stops when the client obtains an end-of-file condition (i.e., a count of zero). In most cases, TCP on the server's computer will place the entire message in a single TCP segment, and then transmit the segment across a TCP/IP internet in an IP datagram. However, TCP does not guarantee that the data will be sent in a single segment, neither does it guarantee that each call to *recv* will return exactly the same amount of data that the server transferred in a call to *send*. Instead, TCP merely asserts that data will be delivered in order, with each call of *recv* returning one or more octets of data. Consequently, a program that calls *recv* must be prepared to make repeated calls until all data has been extracted.

Socket procedures and blocking

Most procedures in the socket API are synchronous or blocking in the same way as I/O calls. That is, when a program calls a socket procedure, the program is suspended until the procedure completes. There is no time limit for suspension - the operation may take arbitrarily long. For example, after creating a socket, binding a protocol port, and placing the socket in passive mode, the server calls *accept*. If a client has already requested a connection before the server calls *accept*, the call returns immediately. If the server reaches the accept call before any client requests a connection, the server will be suspended until a request arrives. In fact, the server spends most of its time suspended at the *accept* call.

Calls to socket procedures in the client code can also block. For example, some implementations of the library procedure *gethostbyname* send a message across a network to a server and wait for a reply. In such cases, the client remains suspended until the reply is received. Similarly, the call to *connect* blocks until TCP can perform the 3-way handshake to establish a connection.

The most important suspension occurs during data transmission. After the connection has been established, the client calls *recv*. If no data has been received on the connection, the call blocks. Thus, if the server has a queue of connection requests, a client will remain blocked until the server sends data.

Copperbelt University
Computer Science Department

<u>Internet Technologies</u>

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

Naming with the Domain Name System (DNS)

Although IP addresses are fundamental in TCP/IP, users do not need to remember or enter IP addresses. Instead, computers are also assigned symbolic names. Application software allows a user to enter one of the symbolic names when identifying a specific computer. For example, when specifying a destination for an electronic mail message, a user enters a string that identifies the recipient to whom the message should be delivered and the name of the recipient's computer. Similarly, a computer name is embedded in a string that a user enters to specify a site on the World Wide Web. Although symbolic names are convenient for humans, they are inconvenient for computers. Because it is more compact than a symbolic name, the binary form of an IP address requires less computation to manipulate. Furthermore, an address occupies less memory and requires less time to transmit across a network than a name. Thus, although application software permits users to enter symbolic names, the underlying network protocols require addresses - an application must translate each name into an equivalent IP address before using it for communication.

A name service stores a collection of one or more naming contexts — sets of bindings between textual names and attributes for objects such as users, computers, services and remote objects. The major operation that a name service supports is to resolve a name that is, to look up attributes from a given name. Operations are also required for creating new bindings, deleting bindings and listing bound names and adding and deleting contexts.

Name management is separated from other services largely because of the openness of the Internet, which brings the following motivations:

Unification: It is often convenient for resources managed by different services to use the same naming scheme. URLs are a good example of this.
Integration: It is not always possible to predict the scope of sharing on the Internet. It may become necessary to share and therefore name resources that were created in different administrative domains. Without a common name service, the administrative domains may use entirely different naming conventions.

General name service requirements

Name services were originally quite simple, since they were designed only to meet the need to bind names to addresses in a single management domain, corresponding to a single LAN or WAN. The interconnection of networks and the increased scale of the Internet have produced a much large name-mapping problem.
The Global Name Service, developed at the Digital Equipment Corporation Systems Research Center [Lampson 1986] has the following objectives:

- To handle an essentially arbitrary number of names and to serve an arbitrary number of administrative organizations: For example, the system should be capable, among other things, of handling the electronic mail addresses of all of the computer users the world.
- A long lifetime: Many changes will occur in the organization of the set of names a in the components that implement the service during its lifetime.
- High availability: Most other systems depend upon the name service; they can’t work when it is broken.
- Fault isolation: So that local failures do not cause the entire service to fail.
- Tolerance of mistrust: A large open system cannot have any component that is trusted by all of the clients in the system.

The software that translates computer names into equivalent Internet addresses provides an example of a client-server interaction. The database of names is not kept on a single computer. Instead, the naming information is distributed among a potentially large set of servers located at sites across the Internet. Whenever, an application program needs to translate a name, the application becomes a client of the naming system. The client sends a request message to a name server, which finds the corresponding address and sends a reply message. If it cannot answer a request, a name server temporarily becomes the client of another name server, until a server is found that can answer the request. This interaction is illustrated further by figure 1 below.

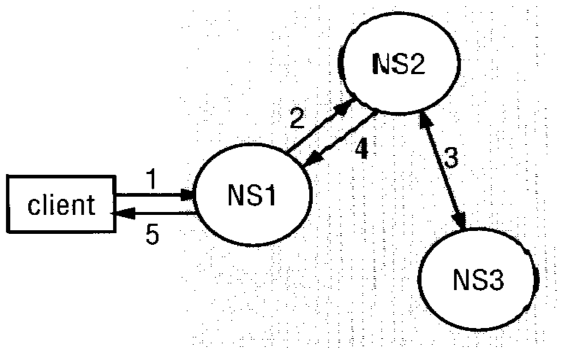


Figure 1 showing naming servers

Name Spaces

A name space is the collection of all valid names recognized by a particular service. For a name to be valid means that the service will attempt to look it up, even though that name may prove not to correspond to any object — to be unbound. Name spaces require a syntactic definition. Names may have an internal structure that represents their position in hierarchic name space, as in the UNIX file system, or in an organizational hierarchy, as is the case for Internet domain names; or they may be chosen from a flat set of numeric or symbolic identifiers. The most important advantage of hierarchic name spaces is that each part of a name is resolved relative to a separate context, and the same name may be used with different meanings in different contexts. In the case of file systems, each represents a context. Hierarchic name spaces are potentially infinite, so they enable a system to grow indefinitely. Flat name spaces are usually finite; their size is determined by fixing a maximum permissible length on names. If no limit is set on the length of the names in a flat name space, then it also is potentially infinite. Another potential advantage of the hierarchic name space is that different contexts can be managed by different people.

The Domain Name System

The Domain Name System is a name service design whose principal naming database is used across the Internet. It was devised principally by Mockapetris [1987] to replace the original Internet naming scheme, in which all host names and addresses were held in a single central master file and downloaded by FTP to all computers that required them [Harrenstien et al. 1985]. This original scheme was soon seen to suffer from three major shortcomings:

- It did not scale to large numbers of computers.
- Local organizations wished to administer their own naming systems.
- A general name service was needed — not one that serves only for look computer addresses.

The objects named by the DNS are primarily computers — for which mainly IP addresses are stored as attributes. In principle, however, any type of object can be named, and its architecture gives scope for a variety of implementations. Organizations and departments within them can manage their own naming data. Millions of names are bound by the Internet DNS, and lookups are made against it from around the world. Any name can be resolved by any client. This is achieved by hierarchical partitioning name database, by replication of the naming data, and by caching.

Structure of Computer Names

The naming scheme used in the Internet is called the Domain Name System (DNS). Syntactically, each computer name consists of a sequence of alpha-numeric segments separated by periods. For example, a computer in the Computer Science Department at the Copperbelt University could have the name: mycomputer.*cs.cbu.edu*

Naming Domains

A naming domain is a name space for which there exists a single overall administrative authority for assigning names within it. This authority is in overall control of which names may be bound within the domain, but it is free to delegate this task. Domains in DNS are collections of domain names; syntactically, a domain's name is the common suffix of the domain names within it, but otherwise it cannot be distinguished from, for example, a computer name. For example,.*cs.cbu.edu* is a domain that contains mycomputer.*cs.cbu.edu*. Note that the term 'domain name' is potentially confusing since only some domain names identify domains. A computer can even have the same name as a domain: for example, yahoo.com is the name of a web server in the domain called yahoo.com.

Responsibility for a naming domain normally goes hand in hand with responsibility for managing and keeping up to date the corresponding part of the database stored in an authoritative name server and used by the name service. Naming data belonging to different naming domains are in general stored by distinct name servers managed by the corresponding authorities.

Domain names are hierarchical, with the most significant part of the name on the right. The left-most segment of a name (*mycomputer* in the example above) is the name of an individual computer. Other segments in a domain name identify the group that owns the name. For example, the segment *cbu* gives the name of the university. Beyond specifying how the most significant segments are chosen, the domain name system does not specify an exact number of segments in each name nor does it specify what those segments represent. Instead, each organization can choose how many segments to use for computers inside the organization and what those segments represent. The domain name system does specify values for the most significant segment, which is called the top-level of the DNS.

The table in figure 2 lists the possible top-level domains.

Domain Name	Assigned To
com	Commercial organisation
edu	Educational institution
gov	Government institution
mil	Military group
net	Major network support centre
org	Organisation other than those above
arpa	Temporary ARPA domain (still in use)
int	International organisation
country code	A country (e.g. Zambia; zm)

Figure 16: Values for the most significant segment of a domain name.

DNS does not distinguish between names in upper or lower case. Some additional top-level domains have been proposed to further divide the namespace and eliminate the overcrowding, which has occurred in the commercial domain. The proposed names include *firm, store, web, arts, rec, info,* and *nom*. *Arts* and *rec* were proposed to accommodate organisations such as art museums and recreational web sites. *Nom* was proposed to permit individuals to register their names. When an organisation wants to participate in the domain name system, the organisation must apply for a name under one of the existing top-level domains. For example, a corporation named *skyways* might request to be assigned domain *skyways* under the top-level domain *com*. If it approves the request, the Internet authority responsible for domain names will assign *skyways* Corporation the domain:
Skyways com

A domain name that ends with a period is called an *absolute domain name* or a *fully qualified domain name (FQDN) e.g., kariba.cbu.ac.zm*. If the domain name does not end with a period, it is assumed that the name needs to be completed.

Geographic structures

In addition to the familiar organisational structure, the DNS allows organisations to use a geographic registration. For example, the Kitwe Teachers Training College in the town of Kitwe, on the Copperbelt Province of Zambia could register the domain:

kttc.kitwe.copperbelt.zm

Thus, names of computers at the college would end with *.zm* instead of *.edu*. Some countries have adopted a combination of geographic and organisational domain names. For example, universities in Zambia register under the domain: *ac.zm*. Where *ac* is an abbreviation for academic and *zm* is the official country code for Zambia.

The DNS server hierarchy

The number of segments in a domain name corresponds to the naming hierarchy. There is no universal standard because each organisation can choose how to structure names in its hierarchy. Furthermore, names within an organisation do not need to follow a uniform pattern because individual groups within the organisation can choose a hierarchical structure that is appropriate for the group.

One of the main features of the Domain Name System is autonomy - the system is designed to allow each organisation to assign names to computers or to change those names without informing a central authority. The naming hierarchy helps achieve autonomy by allowing an organisation to control all names with a particular suffix. Thus, the Copperbelt University is free to create or change any name that ends with *cbu.ac.zm,* while IBM Corporation is free to create or change names that end with *ibm.com*.

In addition to hierarchical names, the DNS uses client-server interaction to aid autonomy. In essence, the entire naming system operates as a large, distributed database. Most organisations that have an Internet connection run a domain name server. Each server contains information that links the server to other domain name servers. The resulting set of servers functions as a large, coordinated database of names. Whenever an application needs to translate a name to an IP address, the application becomes a client of the naming system. The client places the name to be translated in a DNS request message, and sends the result to a DNS server. The server extracts the name from the request, translates the name to an equivalent IP address, and returns the resulting address to the application in a reply message.

DNS servers are arranged in a hierarchy that matches the naming hierarchy, with each being the *authority* for part of the naming hierarchy. A *root server* occupies the top of the hierarchy, and is an authority for the top-level domains (e.g. *.com*). Although it does not contain all possible domain names, a root server contains information about how to reach other servers. For example, although it does not know the names of computers at NIPA, root server knows how to reach a server that handles requests for *nipa.ac.zm.* A corporation can choose to place all its domain names in a single server, or can choose to run several servers.

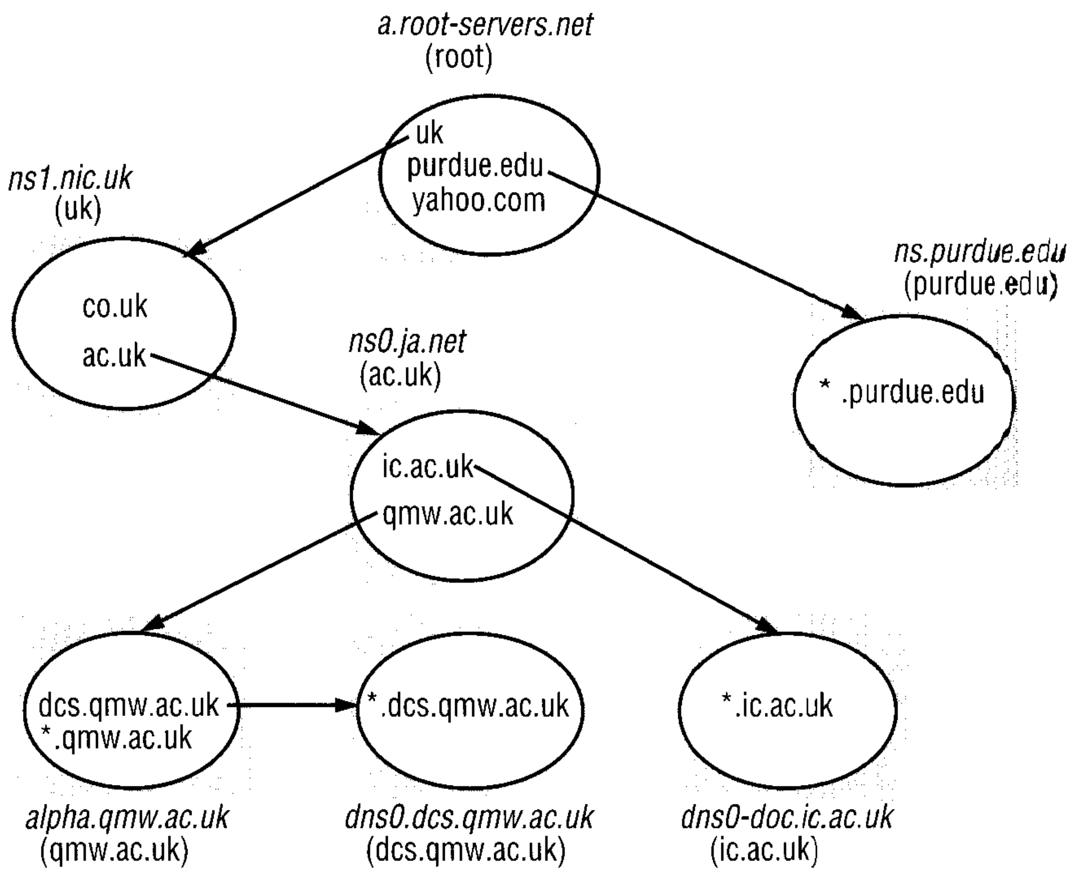


Figure 3 showing domain hierarchies

Figure 3 above shows the arrangement of some of the DNS database. Note that, in practice, root servers such as root-servers, net hold entries for several levels of domain, as well as entries for first-level domain names. This is to reduce the number navigation steps when domain names are resolved. Root name servers hold authoritative entries for the name servers for the top-level domains. They are also authoritative name servers for the generic top-level domains such as com and edu. However, the root name servers are not name servers for the country domains. For example, the uk domain currently has seven name servers, one of which is called nsl.nic.net. These name servers know the name servers for the second level domains in the United Kingdom such as ac.uk and co.uk. The servers for the domain ac.uk know the name servers for the entire university domain in the country, such as manchester.ac.uk.

Server Architectures

An architecture in which an organisation uses a single server is the simplest - a small organisation can minimize cost by placing all its domain information in a single server. Larger organisations usually find that a single, centralized server does not suffice for two reasons:

1. A single server and the computer on which it runs cannot handle arbitrary requests at high speed.
2. Large organisations often find it difficult to administer a central database. The problem is especially severe because most DNS software does not provide automated update - a human must enter changes and additions in the server's database manually. Thus, the group of people who are responsible for administering an organization’s centralized server must co-ordinate to ensure that only one manager attempts to make changes at a given time. If the organisation runs multiple servers, each group can manage a server that is an authority for the group's computers. More important, each group can make changes to its server database without centralized co-ordination. (However, the IETF is working on a standard for automated update of a DNS server database.)

A zone is a subtree of the DNS tree that is administered separately. A common zone is a second-level domain, ac.zm, for example. Many second-level domains then divide their zone into smaller zones. For example, a university might divide itself into zones based on departments and a company might divide itself into zones based on branch offices or internal divisions.

Once the authority of a zone is delegated, it is up to the person responsible for the zone to provide multiple name servers for that zone. Whenever a new system is installed, in a zone, the DNS administrator for the zone allocates a name and IP address for the new system and enters these into the name server's database. This is where the need for delegation becomes obvious. At a small university, for example, one person could do this each time a new system was added, but in a large university the responsibility would have to be delegated (probably by department), since one person couldn't keep up with the work.

A name server is said to have authority for one zone or multiple zones. The person responsible for a zone must provide a *primary name server* for that zone and one or more *secondary name servers*. The primary and secondary name servers must be independent and redundant servers so that a single point of failure does not affect availability of the name service for the zone.

The main difference between a primary and a secondary name server is that the primary loads all the information for the zone from disk files, while the secondary name servers obtain all the information from the primary name server. When a secondary name server obtains information from its primary name server, we call this a *zone transfer*.

When a new host is added to a zone, the administrator adds the appropriate information (name and IP address minimally) to a disk file on the system running the primary name server. The primary name server is then notified to reread its configuration files. The secondary name servers query the primary name server on a regular basis (normally every 3 hours) and if the primary name server contains newer data, the secondary obtains new data using a zone transfer.

Locality of reference, multiple servers and links among servers

The locality of reference principle applies to the domain name system, and helps to explain why multiple servers work well. The domain name system follows the locality of reference principle in two ways.

1. A user tends to look up the names of local computers more often than the names of remote computers.
2. A user tends to look up the same set of domain names repeatedly.

Having multiple servers within an organisation works well because a server can be placed within each group. The local server is an authority for names of computers in the group. Because the DNS obeys the locality principle, the local server can handle most requests. Thus, in addition to being easier to administer, multiple servers help balance the load, and thereby reduce the problems of contention a centralized server may cause.

Although DNS allows the freedom to use multiple servers, a domain hierarchy cannot be split into servers arbitrarily. The rule is: a single server must be responsible for all computers that have a given suffix. Servers in the domain name system are linked together, making it possible for a client to find the correct server by following links. In particular, each server is configured to know the locations of servers of subparts of the hierarchy.

Resolving a Name

In general, resolution is an iterative process whereby a name is repeatedly presented to naming contexts. A naming context either maps a given name onto a set of primitive attributes (such as those of a user) directly, or it maps it onto a further naming context and a derived name to be presented to that context. To resolve a name, it is first presented to some initial naming context; resolution iterates as long as further contexts and derived names are output.
The translation of a domain name into an equivalent IP address is called *name resolution*, and the name is said to be *resolved* to an IP address. Software to perform the translation is known as *name resolver* (or simply *resolver*) software.

Many operating systems provide name resolver software as a library routine that an application can call. For example, on UNIX systems, an application can call library routine *gethostbyname* to resolve a name. *Geshostbyname* takes a single argument and returns a structure. The argument is a character string that contains the domain name to be looked up. If it succeeds, *gethostbyname* returns a structure that contains a list of one or more IP addresses that correspond to the specified name. If it fails to resolve the name, *gethostbyname* returns a *NULL* pointer instead of a structure.
Each resolver is configured with the address of a local domain name server. To become a client of the DNS server, the resolver places the specified name in a DNS request message and sends the message to the local server. The resolver then waits for the server to send a DNS reply message that contains the answer. Although a client can choose to use either UDP or TCP when communicating with a DNS server, most resolvers are configured to use UDP because it requires less overhead for a single request. DNS uses the well-known port 53.

When an incoming request specifies a name for which a server is an authority, the server answers the request directly. That is, the server looks up the name in its local database, and sends a reply to the resolver. However, when a request arrives for a name outside the set for which the server is an authority, further client-server interaction results. The server temporarily becomes a client of another name server. When the second server returns an answer, the original server returns a copy of the answer back to the resolver from which the request arrived.

Each server knows the address of a root server. Knowing the location of a root server is sufficient because the name can be resolved from there. For example, suppose that a resolver at a remote site of an organisation sends a request to its local server (server A) and server A is not an authority for the name, so server A proceeds to act as a client of other servers. In the first place, server A sends a request to the root server. The root server is also not an authority for the name. The response from the root server gives the location of a server (server B) for the required name.

When it receives the response from the root server, server A contacts server B. Although it is not the authority for names in the particular domain, the main server at B knows the location of the server (server C) for the required domain. Thus, it returns a response to inform server A about the details of the server (server C) that is the authority for the required domain. Finally, server A contacts server C that is the authority for the required names. Server C then returns an *authoritative* answer to server A, either the IP address for the name or an indication that no such name exists.

Stepping through the hierarchy of servers to find the server that is an authority for a name is called *iterative query resolution*, and is used only when a server needs to resolve a name. The resolvers that applications call always request *recursive query resolution*. That is, they request complete resolution - the reply to a recursive request is either the IP address being sought or an authoritative statement that no such name exists. In other words:

> *The resolver software in a host always requests recursive resolution in which a name is resolved to an equivalent address. When it becomes a client of another server, a server can request iterative resolution to step through the server hierarchy one level at a time.*

The question really is: What does a name server do when it does not contain the information requested? It must contact another name server. This is the distributed nature of the DNS. Not every name server, however, knows how to contact every other name server. Instead, every name server must know how to contact the *root name server*. All primary name servers must know the IP address of the machines running each root server. These IP addresses are contained in the primary name server's configuration files. The primary servers must know the IP addresses of the machines running the root servers, not their DNS names. The root servers then know the name and location (i.e., the IP address) of each authoritative name server for all the second-level domains. This implies an iterative process: The requesting name server must contact a root server. The root server tells the requesting server to contact another server, and so on.

Optimization of DNS performance

Without optimization, traffic at the root server would be intolerable because the root server would receive a request each time someone mentioned the name of a computer. Furthermore, the principle of locality suggests that a given computer will emit the same request repeatedly - if a user enters the name of a remote computer, the user is likely to specify the same name again. There are two primary optimizations used in the DNS: replication and caching.

Each root server is replicated. Many copies of the server exist around the world. When a new site joins the Internet, the site configures its local DNS server with a list of root servers. The site's server uses whichever root server is most responsive at a given time. In practice, the geographically nearest server usually responds best. Thus, a site in Africa would use a root server in Africa, while a site in Europe would use a root server in Europe.

DNS caching is more important than replication because caching affects most of the system. Each server maintains a cache of names. Whenever it looks up a new name, the server places a copy of the binding in its cache. Before contacting another server to request a binding, the server checks its cache. If the cache contains the answer, the server uses the cached answer to generate a reply. Caching works well because name resolution shows a strong tendency toward temporal locality of reference. That is, on a given day, a user is likely to look up the same name repeatedly.

A fundamental property of the DNS, therefore, is caching. That is, when a name server receives information about a mapping (say, the IP address of a host name) it caches that information so that a later query for the same mapping can use the cached result and not result in additional queries to other name servers.

Types of DNS entries

Each entry in a DNS database consists of three items: a domain name, a record type and a value. The record type specifies how the value is to be interpreted. A query sent to a DNS server specifies both a domain name and a type. The server only returns a binding that matches the type of the query. DNS classifies bindings between a domain name and an equivalent IP address as type A (the A stands for *address type*). Type bindings are common because they are used by most applications. For example, when a user supplies a computer name to an application program such as *FTP*, *ping* or a World Wide Web browser, the application requests a binding that matches type A. In addition to type A, DNS supports several other types. One popular type is MX (for Mail eXchange), which is used to map the computer name found in an e-mail address to an IP address. E-mail software specifies type MX when it sends a request to a DNS server. The answer that the server returns matches the requested type. Thus, an e-mail system will receive an answer that matches type MX. To recap:

> *The domain name system stores a type with each entry. When a resolver looks up a name, the resolver must specify the type that is desired. A DNS server returns only entries that match the specified type.*

Aliases using the CNAME type

CNAME entries are analogous to a symbolic link in a file system - the entry provides an alias for another DNS entry. For example, if Myconnections Corporation has two computers named mugi.*myconnections.com* and *shanks.myconnections.com* and suppose that Mugi decides to run a Web server, and wants to follow the convention of using the name *www* for the computer that runs the organisation's Web server. Although the organisation could choose to rename one of their computers (e.g., stabs), a much easier solution exists. The organisation can create a CNAME entry for *www.myonnections.com* that points to stabs. Whenever a resolver sends a request for *www.myconnections.com*, the server returns the address of the computer stabs.

The use of aliases is especially convenient because it permits an organisation to change the computer used for a particular service without changing the names or addresses of the computers. For example, Myconnections Corporation can move its Web service from computer *mugi* to computer *shanks* by moving the server and changing the CNAME record in the DNS server - the two computers retain their original names and IP addresses.

An important consequence of multiple types

The type system in the DNS is convenient because it permits a manager to use a single name for multiple purposes (e.g., to direct Web traffic to one computer, while sending e-mail to a different computer). However, users are sometimes surprised at the consequence of having specific types in DNS requests - a name that works with one application may not work with another. For example, it may be possible to send e-mail to a computer, while an attempt to communicate with the computer using a program like *ping* or *traceroute* results in a message that no such computer exists. The apparent inconsistency exists because the DNS type requested by e-mail differs from the type requested by other application programs. If the domain database contains a type MX record for the name, a request from the e-mail system will succeed. However, if the database does not also contain a type A record, a request from programs like *ping* will result in a negative reply.

The type system that the DNS uses can produce unexpected results because some applications are configured to use multiple types. For example, the resolvers used in some e-mail systems try two types when resolving a name. The resolver begins by sending the server a type MX request. If the server responds negatively, the resolver then tries a type A request. The scheme can help in situations where an organisation specifies a type A record in their domain database for a given name, but fails to specify a type MX entry for the name as well.

DNS Message Format

There is one DNS message format defined for both queries and responses. Figure 3 shows the overall format of the message.

0 15 16 31

Identification	flags
number of questions	number of answer RRs
number of authority RRs	number of additional RRs
questions	
answers (variable number of resource records)	
authority (variable number of resource records)	
additional information (variable number of resource records)	

Figure 17: General format of DNS queries and responses

The message has a fixed 12-byte header followed by four variable-length fields. The *identification* is set by the client and returned by the server. It lets the client match responses to requests. The 16-bits flags field is divided into numerous pieces as shown in
figure 4.

QR	opcode AA	TC	RD	RA	(zero)	rcode	
1	4 1	1	1	1	3	4	

Figure 18: Flags field in the DNS header

Each of the fields in figure 4 are described below:

- *QR* is a 1-bit field. 0 means the message is a query, 1 means it is a response.
- *Opcode* is a 4-bit field. The normal value is 0 (a standard query). Other values are 1 (an inverse query) and 2 (server status request).
- *AA* is a 1-bit flag that means “authoritative answer.” The name server is authoritative for the domain in the question section.

- *TC* is a 1-bit field that means “truncated.” With UDP this means that the total size of the reply exceeded 512 bytes, and only the first 512 bytes of the reply was returned.
- *RD* is a 1-bit field that means “recursion desired.” This bit can be set in a query and is then returned in the response. This flag tells the name server to handle the query itself, called a *recursive query*. If the bit is not set and the requested name server does not have an authoritative answer then the requested name server returns a list of other name servers to contact for the answer. This is called an *iterative query*.
- *RA* is a 1-bit field that means “recursion available.” This bit is set to 1 in the response if the server supports recursion.
- This is a 3-bit field that must be 0.
- *rcode* is a 4-bit field with the return code. The common values are 0 (no error) and 3 (name error). A name error is returned only from an authoritative name server and means that the domain name specified in the query does not exist.

The next four 16-bit fields specify the number of entries in the four variable-length fields that complete the record. For a query, the *number of questions* is normally 1 and the other three counts are 0. Similarly, for a reply the *number of answers* is at least 1 and the remaining two counts can be 0 or nonzero.

Question Portion of DNS Query Message

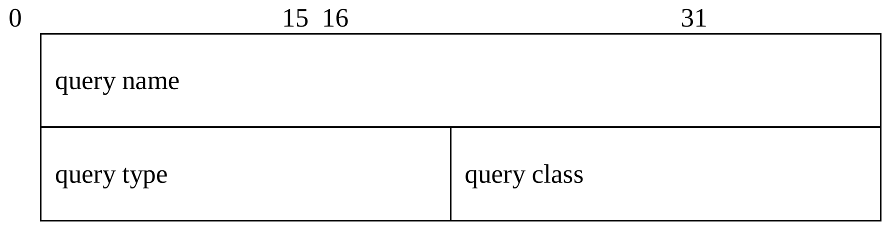


Figure 19: Format of the question portion of DNS query message

The format of each question in the question section is shown in figure 5. There is normally just one question. The query name is the name being looked up. It is a sequence of one or more labels. Each label begins with a 1-byte count that specifies the number of bytes that follow. The name is terminated with a byte of 0, which is a label with a length of 0, which is the label of the root. Each count byte must be in the range of 0 to 63, since labels are limited to 63 bytes. Unlike many other message formats that we have encountered, this field is allowed to end on a boundary other than a 32-bit boundary. No padding is used. Figure 6 shows an example of how the domain name kariba.cbu.ac.zm is stored.

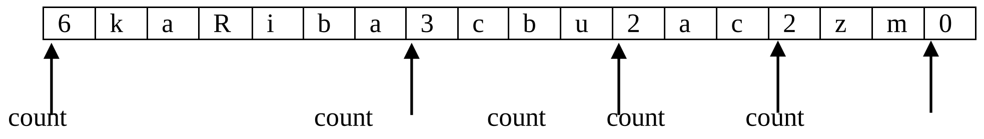


Figure 20: Representation of the domain name kariba.cbu.ac.zm.

Each question has a query type and each response (called a resource record) has a type. There are several different values. Figure 7 shows some of these values. The query type is a superset of the type: two of the values shown in the figure can be used only in questions.

Name	**Numeric value**	**Description**	***type?***	***query type?***
A	1	IP address	●	●
NS	2	Name server	●	●
CNAME	5	Canonical name	●	●
PTR	12	Pointer record	●	●
HINFO	13	Host info	●	●
MX	15	Mail exchange record	●	●
AXFR	252	Request for zone transfer		●
* or ANY	255	Request for all records		●

Figure 21: Type and query type values for DNS questions and responses

The most common query type is an A type, which means an IP address is desired for the query name. A PTR query requests the names corresponding to an IP address. This is a pointer query. The query class is normally 1, meaning Internet address. Some other non-IP values are also supported at some locations.

Resource Record Portion of DNS Response Message

The final three fields in the DNS message, the answers, authority and additional information fields, share a common format called a resource record or RR. Figure 8 shows the format of a resource record.

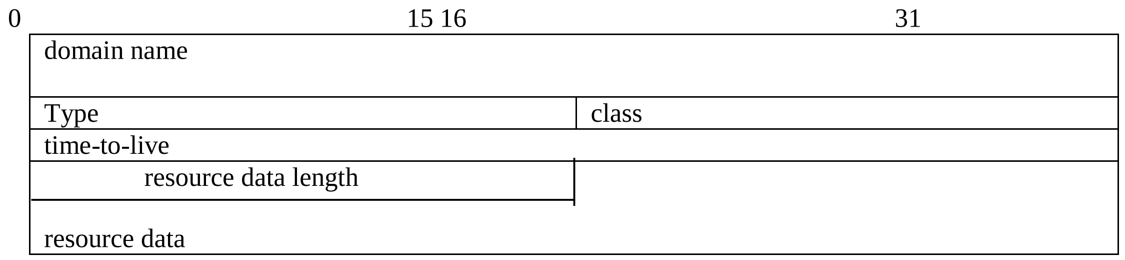


Figure 22: Format of DNS resource record

The *domain name* is the name to which the following *resource data* corresponds. It is the same format as described earlier for the query name field (see figure 5). The type specifies one of the RR type codes. These are the same as the query type values described earlier (see figure 7). The class is normally 1 for the Internet data. The time-to-live field is the number of seconds that the RR can be cached by the client. RRs often have a TTL of 2 days. The resource data length specifies the amount of resource data. The format of this data depends on the type. For a type of 1 (an A record) the resource data is a 4-byte IP address.

Some of the resource records are described in the table in figure 9.

Resource Record Text Name	**Record Type**	**Function**
Start of Authority	SOA	Marks the beginning of a zone's data and defines parameters that affect the entire zone.
Nameserver	NS	Identifies a domain's nameserver
Address	A	Converts a hostname to an address
Pointer	PTR	Converts an address to a hostname
Mail Exchange	MX	Identifies where to deliver mail for a given domain name
Canonical Name	CNAME	Defines an alias hostname
Host Information	HINFO	Describes a host's hardware and operating system
Well-known service	WKS	Advertises network services
Text	TXT	Stores arbitrary text strings

Figure 23: Standard resource records

An Example of a Domain Database

This file converts hostnames to IP addresses, so A records predominate, but it also contains MX, CNAME and other records. Such a file is only created on a primary server. All other servers get information from the primary server.

```
;
;               Addresses and other host information
;
@               IN     SOA    mubilo.ndongo.com      jan.mubilo.ndongo.com

;               Define the name servers and the mail servers
                IN     NS            mubilo.ndongo.com.
                IN     NS            muchinga.ndongo.com.
                IN     NS            orange.ndongo.com.
                IN     MX            10 mubilo.ndongo.com.
                IN     MX            20 mumbole.ndongo.com.
;
;               Define local host
;
localhost       IN     A             127.0.0.1
;
;               Define the hosts in this zone
;
mubilo          IN     A             172.16.12.1
                IN     MX            5 mubilo.ndongo.com.
loghost         IN     CNAME         mubilo.ndongo.com.
lemon           IN     A             172.16.12.2
                IN     MX            5 mubilo.ndongo.com.
masuku          IN     CNAME         lemon.ndongo.com.
mumbole         IN     A             172.16.12.3
mango           IN     A             172.16.12.4
muchinga        IN     A             172.16.1.2
;               host table has BOTH host and gateway entries for 10.104.0.19
our-gateway  IN A             10.104.0.19
;
;               Records for servers within this domain
;
pack.plant      IN     A             172.16.18.15
dress.sales     IN     A             172.16.6.1
;
;               Define sub-domains
;
plant           IN     NS            pack.plant.ndongo.com.
                IN     NS            mumbole.ndongo.com.
sales           IN     NS            dress.sales.ndongo.com.
                IN     NS            pack.plant.ndongo.com.
```

The example file above begins with an SOA record and a few NS records that define the domain and its servers. We shall discuss the records in the order in which they occur in the sample file.

The first MX record identifies a mail server for the entire domain. This record says that *mubilo* is the mail server for ndongo.com with a preference of 10. Mail addressed to user@ndongo.com is redirected to *mubilo* for delivery. For *mubilo* to successfully deliver the mail, it must be properly configured as a mail server.

The second MX record identifies *mumbole* as a mail server for *ndongo.com* with a preference of 20. Preference numbers let you define alternate mail servers. The lower the preference number, the more desirable the server. Therefore, our two sample MX records say "send mail for the *ndongo.com* domain to *mubilo* first; if *mubilo* is unavailable, try sending the mail to *mumbole*." Rather than relying on a single mail server, preference numbers allow you to create backup servers. If the main mail server is unreachable, the domain's mail is sent to one of the backups instead.

The sample MX records redirect mail addressed to *ndongo.com*, but mail addressed to user@mango.ndongo.com will still be sent directly to *mango.ndongo.com* – not *mubilo* or *mumbole*. This configuration allows simplified mail addressing in the form user@ndongo.com for those who want to take advantage of it, but it continues to allow direct mail delivery to individual hosts for those who wish to take advantage of that.

The first A record in this example defines the address for local host. It allows users within the *ndongo.com* domain to enter the name *localhost* and have it resolved to the address 127.0.0.1 by the local nameserver.

The next A record defines the IP address for *mubilo*. Note that the records that relate to a single host are grouped together, which is the most common structure used in the zone files. The A record is followed by an MX record and a CNAME record that both relate to *mubilo*. The *mubilo* MX record points back to the host itself and the CNAME record defines an alias for the host name.

lemon's A record is also followed by an MX record and a CNAME record. However, lemon's MX record serves a different purpose. It directs all mail addressed to user@lemon.ndongo.com to *mubilo*. This MX record is required anyway because the MX records at the beginning of the zone file redirect mail only if it is addressed to user@ndongo.com. If you also want to redirect mail addressed to *lemon*, you need a "lemon-specific" MX record.

The name field of the CNAME record contains an alias for the official hostname. The official name, called the canonical name, is provided in the data field of the record. Because of these records, *mubilo* can be referred to by the name *loghost* and *lemon* can be referred to as *masuku*. Hostname aliases should not be used in other resource records. For example, don't use an alias as the name of a mail server in an MX record. Use only the "canonical" (official) name that is defined in an A record.

The Domain Name System's nslookup Command

nslookup is a debugging tool. It allows anyone to directly query a nameserver and retrieve any of the information known to the DNS system. It is helpful for determining if the server if the server is running correctly and is properly configured for querying for information provided by the remote servers.

The **nslookup** program is used to resolve queries either interactively or directly from the command line. Below is a command-line example of using **nslookup** to query for the IP address of a host:

```
C:\> nslookup mubilo.ndongo.com

Server:     lemon.ndongo.com
Address:    172.16.12.2

Name:       mubilo.ndongo.com
Address:    172.16.12.1
```

Here, a user asks **nslookup** to provide the address of *mubilo.ndongo.com*. **nslookup** displays the name and address of the server used to resolve the query and then it displays the answer to the query. This is useful, but **nslookup** is more often used interactively.

The real power of **nslookup** is seen in interactive mode. To enter interactive mode, type **nslookup** at the command line without any arguments. Terminate an interactive session by entering CTRL – D (^D) or the **exit** command at the **nslookup** prompt. Redone in an interactive session, the previous query shown is:

```
C:\> nslookup

Default Server:         lemon.ndongo.com
Address:        172.16.12.2

>    mubilo.ndongo.com

Server:         lemon.ndongo.com
Address:        172.16.12.2

Name:           mubilo.ndongo.com
Address:        172.16.12.1

> ^D
```

By default **nslookup** queries for A records, but you can use the set type command to change the query to another resource record type or to the special query type "ANY". ANY is used to retrieve all available resource records for the specified host.

The following example checks MX records for *mubilo* and *lemon*. Note that once the query type is set to MX, it stays MX. It does not revert to the default A-type query. Another set type command is required to reset the query type.

```
C:\> nslookup

Default Server:         lemon.ndongo.com
Address:        172.16.12.2


> set type=MX

> mubilo.ndongo.com

Server:         lemon.ndongo.com
Address:        172.16.12.2

mubilo.ndongo.com    preference = 5, mail exchanger = lemon.ndongo.com
mubilo.ndongo.com    inet address = 172.16.12.1

> lemon.ndongo.com

Server:         lemon.ndongo.com
Address:        172.16.12.2

lemon.ndongo.com     preference = 5, mail exchanger = lemon.ndongo.com
lemon.ndongo.com     inet address = 172.16.12.2
```

```
> exit
```

Abbreviations and the DNS

Because users tend to enter names for local computers more often than they enter names for remote computers, abbreviations for local names are convenient. For example, Mazusa Corporation might choose to allow users to omit the suffix *mazusa.com* when entering a domain name. With such an abbreviation in effect, a user could enter the name *mumbole.na.training* to refer to computer *mumbole* in the *na* subdivision of the *training* division. The full domain name for *mumbole* is *mumbole.na.training.mazusa.com.*

Domain name servers do not understand abbreviations - a server only responds to a full name. To handle abbreviations, resolvers are programmed to try a set of suffixes. For example, each resolver at Mazusa Corporation might be programmed to look up a name twice: once with no change and once with the suffix *mazusa.com* appended. The suffix allows both local and remote names to be handled the same way. Given the valid name of a remote computer, a DNS server will return a valid answer, which the resolver uses. However, given an abbreviated local name, a DNS server will return an error message (because no such name exists). The resolver can then try appending each of the suffixes.

Usually, each computer contains resolver software that all applications on the computer use. Because abbreviations are handled by a resolver, the set of abbreviations allowed on each computer can differ. For example, a resolver on a computer in the *training* division might try appending the suffix *na.training* to a name before appending *mazusa.com*. As a result, a user in the *training* division could reference any computer in the division without entering the suffix. Similarly, resolvers on computers in the *purchasing* division might append the suffix *na.purchasing* before appending *mazusa.com*. If each division follows the pattern, a user can abbreviate the name of any computer in his or her division.

Copperbelt University
Computer Science Department

Internet Technologies

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

Electronic Mail Representation and Transfer

In the transmission of messages over communication networks and Internet, E-mail is in the centre of most organizations today. Each organization has a fully computerized network that enables users to send and receive messages. Most of the E-mail systems are fast, flexible and reliable. Messages that are sent from other people are kept in the mailboxes of the receiving systems before they are read by the recipients. In order to retrieve these messages from the mailbox from a specific mail server e.g. yahoo, the recipient uses the keyboard to enter the URL for that email server, username, and password in order to access the mail box. The user can then read, reply to the messaged, search for specific mails using a specified key, delete, compose or forward to other users.

Electronic mail is one of the most widely used network applications. As such, it is important to know the interactions that occur between clients and servers when electronic mail is transferred across an internet.

The electronic mail paradigm

Originally, *electronic mail (e-mail)* was designed as a straightforward extension of the traditional office memo. That is, the original e-mail systems were built to allow a person to communicate with other people. An individual created a message and specified other individuals as recipients. The e-mail software transmitted a copy of the message to each recipient. The electronic mail systems have evolved from the original design, and are automated to permit more complex interactions. In particular, because a computer program can answer an e-mail message and send a reply, e-mail can be used in a variety of ways. For example, a company can establish a computer program that responds automatically to requests for information that arrives in mail messages. A user sends an e-mail request to the program, and receives the desired information in a reply.

Store and Forward Messaging

E-mail uses the store and forward communication technique of client/server messaging. This means that computers or LANs can transmit messages from a terminal or workstations on one system to a terminal or workstation on another LAN through the use of intervening systems. Store and forward refers to the manner in which a message is sent to its destination. The message can be temporarily stored at intermediate sites before it is forwarded to the next node on its way to its final destination.
Another important feature of the communication techniques used in e-mail is the recipients do not have to be at their machines and ready to accept a message when it sent. The message can be stored until they are ready to read it.

Electronic mailboxes and addresses

Before an e-mail can be sent to an individual, the person must be assigned an *electronic mailbox*. The mailbox consists of a passive storage area (e.g., a file on disk). Like a conventional mailbox, an e-mail mailbox is private - the permissions are set to allow the mail software to add an incoming message to an arbitrary mailbox, but to deny anyone except the owner the right to examine or remove messages. In most cases, an electronic mailbox is associated with a computer account. Thus, a person who has multiple computer accounts can have multiple mailboxes. Each electronic mail box is assigned a *unique electronic mail address (e-mail address)*. When someone sends a memo, they use an electronic mail address to specify a recipient. A full e-mail address contains two parts. The second part specifies a computer and the first part specifies a mailbox on that computer. In the most widely used format, *an "at sign"* separates the two components, e.g., *mailbox@computer*, where *mailbox* is a string that denotes a user's mailbox, and *computer* is a string that denotes the computer on which the mailbox is located (i.e., a domain name).

The division of an e-mail address into two parts achieves two goals.

1. The division allows each computer system to assign mailbox identifiers independently. Thus, two computers can use different mailbox identification schemes or they can both choose to use the same mailbox names.
2. The division permits users on arbitrary computer systems to exchange e-mail messages. E-mail software on the sender's computer uses the second part to determine which computer to contact, and e-mail software on the recipient's computer uses the first part of the address to select a particular mailbox into which the message should be placed.

The format used for the mailbox portion of an e-mail address depends on the e-mail software available on a computer as well as on the operating system being used. Some software systems allow the systems administrator to choose mailbox names, while other systems require a user's mailbox identifier to be the same as the user's login identifier.

Electronic mail messages

An electronic mail message consists of ASCII text that is separated into two parts by a blank line. Called a *header*, the first part contains information about the message: the sender, intended recipients, date the message was sent, and the format of the contents. The second part is known as the *body* and it contains the text of the message. Although the body of a message can contain arbitrary text, the header follows a standard form that e-mail software uses when it sends or receives a message. Each header line begins with a *keyword* followed by a colon and additional information. The keyword tells e-mail software how to interpret the remainder of the line. Some keywords are required in each e-mail header, while others are optional. For example, each header must contain a line that begins with the keyword *TO* and specifies a list of recipients. The remainder of the header line following *TO:* contains a list of one or ore e-mail addresses, where each address corresponds to one recipient. E-mail software places a line that begins with the keyword *From* followed by the e-mail address of the sender in the header of each message. Two additional header lines that contain the date the message was sent and the subject of the message. Both are optional and the sender's e-mail software chooses whether or not to include them.

If e-mail software does not understand a header line, the software passes it through unchanged. Thus, application programs that use e-mail messages to communicate can add additional lines to the message header to control processing. As a result, a vendor can build e-mail software that uses header lines to add functionality, and so, if an incoming message contains a special header line, the software knows that the message was created by the company's product (e.g., the brand name of the software that was used to create the message). The table in figure 10 lists some of the key words commonly found in Internet mail, and describes the purpose of each.

Key word	**Meaning**
From	Sender's address
To	Recipients' addresses
Cc	Addresses of carbon copies
Bcc	Addresses of blind carbon copies
Date	Date on which message was sent
Subject	Topic of the message
Reply-To	Address to which reply should go
X-Charset	Character set used (usually ASCII)
X-Mailer	Mail software used to send the message
X-Sender	Duplicate of sender's address
X-Face	Encoded image of the sender's face

Figure 24: Examples of keywords found in Internet mail.

Multipurpose Internet Mail Extensions (MIME)

The original Internet e-mail was designed to handle only text. The body of an e-mail message was restricted to printable characters and could not contain arbitrary bytes. In particular, one could not transfer a binary file directly as the body of a mail message.

The inventers of e-mail devised schemes to allow e-mail to be used to transfer arbitrary data (e.g., a binary program or a graphics image). In general, all the schemes *encode* the data in a textual form, which can then be sent in a mail message. Once it arrives, the body of the message must be extracted and converted back to binary form. For example, one method uses a hexadecimal representation. The binary data is divided into four-bit units, with each unit encoded as one of the sixteen hexadecimal characters. The sequence of hexadecimal characters is then sent in an e-mail message. The receiver must translate the characters back to binary.

To help co-ordinate and unify the various schemes that have been invented for encoding binary data, the IETF invented MIME. MIME does not dictate a single standard for encoding binary data. Instead, MIME permits a sender and receiver to choose an encoding method that is convenient. When using MIME, the sender includes additional lines in the header to specify the message follows MIME format as well as additional lines in the body to specify the type of the data and the encoding. MIME also allows a sender to divide a message into several parts and to specify the encoding for each part independently. Thus, with MIME, a user can send a plain text message and attach a graphics image. When the recipient views the message, the e-mail system displays the text message, and then asks the user how to handle the attached image (e.g., save a copy on the disk or display a copy on the screen). When the user decides how to handle the attachment, the MIME software decodes the attached data automatically. MIME adds two lines to an e-mail header. One to declare that MIME was used to create the message, another to specify how MIME information is included in the body. For example, the header lines:

MIME-Version: 1.0
Content-Type: Multipart/Mixed; Boundary=Mime_separator

specify that the message was composed using version 1.0 of MIME; and that a line containing MIME_separator will appear in the body before each part of the message. When MIME is used to send a standard text message, the second line becomes: *Content-type: text/plain*

The chief advantage of MIME is its flexibility. The standard does not specify a single encoding scheme that all senders and receivers must use. Instead, MIME allows new encodings to be invented at any time. A sender and receiver can use a conventional e-mail system to communicate, provided they agree on the encoding scheme and a unique name for it. In addition, MIME does not specify a value to be used to separate parts of the message or a way to name the encoding scheme used. The sender can choose any separator that does not appear in the body. The receiver uses information in the header to determine how to decode the message. MIME is compatible with older e-mail systems. In particular, an e-mail system that transfers the message does not need to understand the encoding used for the body or the MIME header line. The message can be treated exactly like any other e-mail message. The mail system transfers header lines without interpreting them, and treats the body as a single block of text.

E-mail and application programs

It is possible to configure an e-mail address to correspond to a program instead of a mailbox on disk. When e-mail arrives destined for such an address, the mail system sends a copy to the specified program instead of placing a copy on disk. Allowing a computer program to send or receive mail makes it possible to invent new ways to interact. For example, consider an application that permits a user to obtain information from a database. The user places a request (e.g., a database query) in an e-mail message, and sends the message to the program. The program extracts the request from the incoming message, looks up the answer in the database, and then uses e-mail to send a reply to whoever sent the request.

Mail transfer

After a user composes an e-ail message and specifies recipients, e-mail software transfers a copy of the message to each recipient. In most systems, two separate pieces of software are required. A user interacts with an *e-mail interface* program when composing or reading messages. The underlying e-mail system contains a mail transfer program that handles the details of sending a copy of a message to a remote computer. When a user finishes composing an outgoing message, the e-mail interface places the message in a queue that the mail transfer program handles. The mail transfer program waits for a message to be placed on its queue, and then transfers a copy of the message to each recipient. Sending a copy of a message to a recipient on the local computer is trivial because the transfer program can append the message to the user's mailbox. Sending a copy to a remote user is more complex. The sender's mail transfer program becomes a client that contacts a server on the remote machine. The client sends the message to the server, which places a copy of the message in the recipient's mailbox.

The Simple Mail Transfer Protocol (SMTP)

When a mail transfer program contacts a server on a remote machine, it forms a TCP connection over which it communicates. Once the connection is in place, the two programs follow the Simple Mail Transfer Protocol (SMTP) that allows the sender to identify itself, specify a recipient, and transfer an e-mail message. SMTP handles many details of sending and receiving of e-mails. For example, SMTP requires reliable delivery - the sender must keep a copy of the message until the receiver has stored a copy in non-volatile memory (e.g., on disk). Furthermore, SMTP allows the sender to ask whether a given mailbox exists on the remote computer.

Optimising for multiple recipients on a computer

Most mail transfer programs are optimised to handle all recipients on a given remote computer at the same time. For example, suppose a user at *nonexist.com* sends a message to three users at *mazusa.com*. The mail transfer program on *nonexist.com* does not need to establish three separate connections to the server on *mazusa.com*. Instead, the transfer program forms a single connection to the server, specifies all three recipients, and transfers a single copy of the message. The server accepts the message and then delivers a copy to each of the recipients. Optimising for multiple recipients is important for two reasons. First, it dramatically reduces the network bandwidth required to transfer e-mail. Second, optimisation reduces the delay required for all users to receive a copy of a message. The recipients who have their mailbox on a given computer will receive a copy of the message at approximately the same time. If the internet between the sender and the receiver fails, either all recipients receive a copy of the message or none does.

Mail exploders, list and forwarders

Because they can process e-mail messages, computer programs can be used to manipulate and forward messages. For example, many mail systems include a *mail exploder or mail forwarder*, a program that can forward copies of a message. The exploder uses a database to determine how to handle a message. Commonly called a *mailing list*, each entry in the database is a set of e-mail addresses. Furthermore, each entry in the database is assigned its own e-mail address. When an e-mail message arrives, the mail exploder examines the destination address. If the destination address corresponds to a list in its database, the exploder forwards a copy of the message to each address on the list. The table in figure 11 shows an example mail exploder database.

List	Contents
friends	joe@cbu.ac.zm, size@yahoo.com, simms@hotmail.com
customers	aha@xyz.com, NZ_marketing@yota.ah.com

Figure 25: An example database used by a mail exploder. Each entry is assigned a name and contains a list of e-mail addresses.

The example database specifies only the local part of a list's address. To complete the address, one must append the name of the computer on which the exploder operates to the name that is given. For example, if the exploder runs on computer *mutopo.com*, the first mailing list has the full address: *friends@mutopo.com*. If someone sends a message to *friends@mutopo.com,* the exploder will forward a copy to each of the three recipients.

Mail exploders make it possible for a large group to communicate via e-mail without requiring senders to specify all recipients explicitly. To send mail to the group, a message is sent to the list address. The exploder receives the message and forwards a copy to each member of the list. To receive mail sent to the group, an individual must request their e-mail address to be added to the mailing list.

Mail gateways

Although a mail exploder can operate on any computer, forwarding an e-mail message to a large mailing list can require significant processing time. Thus, many organisations do not permit exploders or large mailing lists on conventional computers. Instead, the organisation selects a small set of computers to run exploders and forward e-mail. A computer dedicated to processing electronic mail is often called a *mail gateway, e-mail gateway* or *e-mail relay*. The mailing lists maintained on most gateways are *public*. That is, anyone is allowed to join the list, and anyone can send a message to the list. Thus, a message often arrives at the gateway from a remote computer. Such a message passes across the Internet at least twice. Initially, a single copy passes from the sender's computer to the mail gateway. After it consults the database of mailing lists, the exploder generates a request to send copies of the message. A conventional mail transfer program on the gateway computer sends each copy of the message across the Internet to the respective recipient's computer, where a server stores it in the recipient's mailbox.

Automated mailing lists

Because computer programs can be created to send and receive e-mail messages, it is possible to build programs that handle routine chores without human intervention. One especially useful form of automated program is used in conjunction with an e-mail exploder. The special program, called a list manager, automatically maintains the exploder's database of mailing lists. A user who wants to create a new mailing list, can add their e-mail address to a list or remove their e-mail address from a list, sends an e-mail message to the list manager program. For example, one list manager expects incoming e-mail to contain commands such as*:*

add mailbox to list

where *mailbox* is an e-mail address and *list* is the name of a mailing list. Other list managers use alternative forms (e.g., *subscribe* instead of *add*).

From the participant's point of view, automated management improves service because it allows a participant to join or leave a list without contacting a human being or waiting for a person to enter a change. From the list owner's point of view, automated management reduces the cost of maintaining the list.

Mail relays and e-mail addresses

Because a user's e-mail address includes the name of the user's computer, an organisation that has many computers can have a variety of e-mail addresses. If each employee's e-mail address includes the name of their computer, the e-mail addresses of the employees will differ. Knowing one employee's e-mail address would not help someone guess another employee's e-mail address. To avoid confusion and make e-mail addresses for all employees uniform, an organisation might choose to run a mail gateway, and assign all e-mail addresses relative to the gateway. For example, if Mazusa Corporation names their gateway computer as *mazusa.com* then the corporation can assign each employee an e-mail address of the form: employee@mazusa.com, where the string employee is chosen to designate a single employee (e.g., the employee's username).

Because each e-mail address includes the name of the gateway, a message sent to an employee at Mazusa Corporation will arrive at the gateway computer. The database on the gateway must contain an entry for each employee that specifies the employee's mailbox on a specific machine in the corporation. The database on the e-mail gateway allows the internal and external addresses to differ - external e-mail addresses can be independent of the mailbox identifiers used by particular computer systems. For example, if the computer that employee James Cheche uses has the 7-digit number 8456311 assigned as a mailbox identifier, the entry in the corporate mail gateway might specify:

List	Contents
James_Cheche	8456311@sales.mazusa.com

In essence, the database would contain a mailing list for John's external e-mail identifier with a single recipient. The exploder at the gateway would forward mail sent to James_Cheche@mazusa.com to James's internal mailbox. In addition to making e-mail addresses uniform across an entire organisation, the gateway scheme permits flexibility. Because no one outside the corporation knows the specific computer an employee uses to receive mail or the employee's internal mailbox identifier, the organisation can move an employee or rename a computer without changing the employee's e-mail address.

Mailbox access

Given a choice, most users prefer to have their mailbox located on the computer they use most. For example, someone who has a workstation on their desktop might choose to place their electronic mailbox on the hard disk of the workstation. Unfortunately, mailboxes can not be placed on all computer systems. A mailbox is merely a storage location on disk. Remote programs do not access the mailbox directly. Instead, each computer system that has mailboxes must run a mail server program that accepts incoming e-mail and stores it in the correct mailbox. On powerful computer systems, mail server programs operate in background, allowing the user to run other applications at the same time. To permit multiple clients to send e-mail simultaneously, most mail servers arrange to run multiple copies of the server program at the same time.

A mailbox cannot be placed on a computer unless the computer runs a mail server. For example, a computer that does not have enough memory, does not have an operating system that allows programs to run in the background, or does not have sufficient CPU capacity, can not run a server. More importantly, servers are expected to run continuously. A computer that remains powered off or disconnected from the Internet for extended periods of time will not suffice as an e-mail receiver. Thus, a personal computer is not usually chosen to run an e-mail server.

Using a separate computer for e-mail can be inconvenient. For example, someone who uses a personal computer on their desktop would find it annoying to move to another computer merely to read e-mail. To avoid the problem, the TCP/IP protocols include a protocol that provides remote access to an electronic mailbox. The protocol allows a user's mailbox to reside on a computer that runs a mail server, and allows the user to access items in the mailbox from another computer. Known as the *Post Office Protocol (POP)*, the protocol requires an additional server to run on the computer with the mailbox. The additional server uses the POP protocol. A user runs e-mail software that becomes a client of the POP server to access the contents of the mailbox. Figure 49 illustrates one way to use POP. As the figure shows, a computer that has a mailbox must run two servers. A conventional mail server accepts incoming e-mail and stores it in the appropriate mailbox. The mail can arrive either directly from the original sender or from a mail gateway. A POP server allows a user on a remote machine to access the mailbox.

Although both the e-mail server and POP server communicate across the Internet, there are several differences. First, the mail server uses the SMTP, while the POP server uses the POP. Second, the mail server accepts a message from arbitrary sender, while the POP server only allows a user to access the mailbox after the user enters authentication information (e.g., a password). Third, the mail server can transfer only e-mail messages, while a POP server can provide information about the mailbox contents.

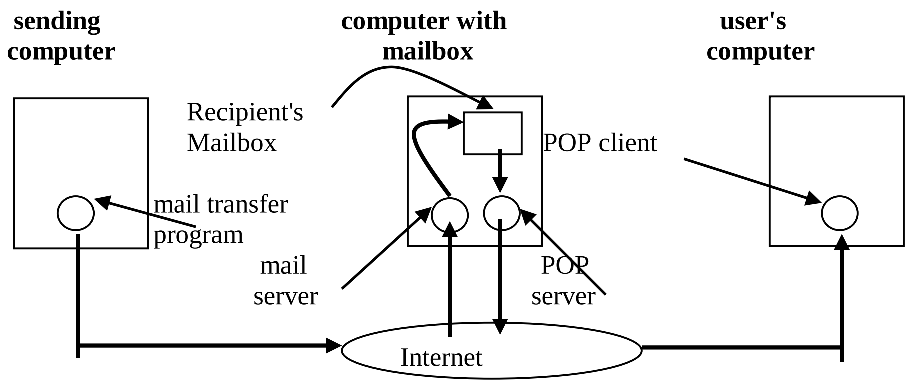


Figure 26: The path of e-mail when POP is used to access a mailbox.

Dialup connections and POP

Although figure 12 shows a client accessing the POP server across the Internet, POP is especially popular among users who rely on dialup telephone connections. In such cases, a computer with the user's mailbox remains attached to the Internet as shown in the figure. However, the user's computer does not need a permanent Internet connection. Instead, the computer can attach to a modem and use a telephone connection. To receive e-mail, the user forms a dialup connection either to the mailbox computer or to some other computer on the Internet. Once the user connects to a computer on the Internet, the user can run a POP client to contact the server and access the e-mail.

Copperbelt University
Computer Science Department

<u>Internet Technologies</u>

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

File Transfer and Remote File Access

Data transfer and distributed computation

Before networks came into being, transferring data from one computer to another required the use of magnetic media such as tapes or disks. Data was written onto the magnetic medium by an application on one computer and the medium was physically moved to another computer. Computer networks reduced the delay substantially, and made possible a new form of computation in which programs on two or more computers co-operate to achieve a solution. Output from one program becomes input of another program. The chief disadvantage of direct communication among programs is that it requires co-ordination of applications on many computers, and such co-ordination can be difficult. One has to ensure that computers are running and that the applications are ready at the same time. Furthermore, to achieve high throughput, one must prevent other programs on the computers from using significant amounts of CPU, memory and network bandwidth. A second disadvantage of direct communication arises from its inability to recover from a failure. If any computer or application program crashes, the entire computation may have to be restarted from the beginning. Because no intermediate results may have been saved, such failures can be especially costly if they occur late in a long computation (e.g., after many hours of processing). Instead of sending data across a network as it is generated, each application stores intermediate results in files on disk. Later, the data is transferred from an output file on one computer to an input file on another. This can help overcome some of the disadvantages mentioned above.

Generalized file transfer

To be useful, file transfer software must be general and flexible. It must allow transfer of an arbitrary file, and must accommodate multiple file types. Because an internet can connect heterogeneous computer systems, file transfer software must accommodate differences among the ways computer systems store files. For example, each computer system has rules about file names. A name that is valid on one computer system may be invalid on another. Furthermore, because most computer systems use login accounts to define file ownership, the owner on one computer system may not have a corresponding login account on another computer. Finally, file transfer software must accommodate other minor differences in file representation, and file protection mechanisms.

Interactive and batch transfer paradigms

A file transfer service can offer the advantages of both interactive and batch approaches. To do so, the service must provide an interface that permits either a human user or a program to invoke the service. To operate interactively, a human invokes the service, enters a request, and waits for a response. To operate in batch mode, a transfer program manages a queue of requests. When handling a request, the transfer program passes a request to the service and waits for the transfer to complete.

The File Transfer Protocol (FTP)

The most widely deployed Internet file transfer service uses the File Transfer Protocol (FTP). FTP permits transfer of an arbitrary file, and include a mechanism that allows files to have ownership and access restrictions. Because it hides the details of individual computer systems, FTP accommodates heterogeneity. It can be used to transfer a copy of a file between an arbitrary pair of computers. FTP is among the oldest application protocols still used in the Internet.

Originally defined as part of the ARPANET protocols, FTP predates both TCP and IP. As TCP/IP was created, a new version of FTP was developed that worked with the new Internet protocols. FTP is among the most heavily used applications.

FTP general model and user interface

FTP is designed to permit interactive or batch use. Most users invoke FTP interactively. They run an FTP client that establishes communication with a specified server to transfer files. However, some software systems invoke FTP automatically without requiring a user to interact with an FTP client. For example, a MIME interface to e-mail can extract and follow an FTP reference. When it invokes FTP, a program handles all details. The program interacts with FTP and then informs the user whether the operation succeeded or failed. The program completely hides the FTP interface from the user. When a user invokes FTP interactively, the user communicates with a command-driven interface. FTP issues a *prompt* to which the user responds by entering a command. FTP executes the command and then issues another prompt. FTP has commands that allow a user to specify a remote computer, provide authorization, find out which remote files are available and request file transfer of one or more files. Some FTP commands require little or no time to execute, while others can take a significant time. For example, it may take seconds to transfer a copy of a large file.

Although the FTP protocol standard specifies exactly how FTP software on one computer interacts with FTP software on another, the standard does not specify a user interface. Consequently, the interface available to a user can vary from one implementation of FTP to another. To help maintain similarity among products, many vendors have chosen to adopt the interface that first appeared in an early version of FTP software written for the BSD UNIX system. The BSD interface for FTP supports over 50 individual commands.

Connections, authorization and file permissions

Most users need only a handful of FTP commands to transfer a file. After starting an FTP program, a user must enter the *open* command before any files can be transferred. *Open* requires the user to give the domain name of a remote computer, and then forms a TCP connection to the computer. Known as a control connection, the TCP connection to a remote machine is used to send commands. For example, once a connection has been opened, FTP requests the user to supply authorization to the remote computer. To do so, the user must enter a login name and a password. The login name which must correspond to a valid account on the remote computer, determines which files can be accessed. If an FTP user supplies login name *shangombo,* the user will have the same file access permissions as someone who logs in as *shangombo* on the remote machine. After a user opens a remote connection and obtains authorization, the user can transfer files. The control connection remains in place as long as it is needed. When a user finishes accessing a particular computer, the user enters the *close* command to terminate the control connection. Closing a control connection does not terminate use of the FTP program - the user can choose to open a new control connection to another computer.

Anonymous file access

Although the use of a login name and password can help keep files secure from unauthorised access, such authorization can also be inconvenient. In particular, requiring each user to have a valid login name and password makes it difficult to allow arbitrary access. To permit arbitrary users to access a file, many sites follow the convention of establishing a special computer account used for FTP. The account, which has the login name *anonymous*, permits an arbitrary user minimal access to files. Early systems used the password *guest* for *anonymous* access. More recent versions of FTP often request that a user sends his e-mail address as a password, making it possible for the remote FTP program to send the user e-mail if problems arise. In either case, the term anonymous FTP is used to describe the process of obtaining access with the *anonymous* login name.

File transfer in either direction

FTP allows file transfer in either direction. After a user establishes a connection to a remote computer, the user can obtain a copy of a remote file or transfer a copy of a local file to the remote machine. Such transfers are subject to access permissions. The remote computer can be configured to prohibit creation of new files or changes to existing files, and the local computer enforces conventional access restrictions on each user. A user enters the *get* or *mget* command to receive a copy of a remote file. The *get* command, which is used most often, handles a single file transfer at a time. *Get* requires a user to specify the name of the remote file to copy. The user can enter a second name if the local file into which the copy should be placed has a different name than the remote file. If the user does not supply a remote file name on the input line along with the command, FTP prompts the user to request a name. Once it knows the name of a file, FTP performs a transfer and informs the user when it completes. The command *mget* permits a user to request multiple files with a single request. The user specifies a list of remote files, and FTP transfers each file to the user's computer.

To transfer a copy of a file from the local computer to a remote computer, a user enters a *put*, *send*, or *mput* command. *Put* and *send* are two names for the command that transfers a single file. As with *get*, a user must enter the name of the file on the local computer, and can also enter a different file name to use on the remote computer. If no file name is present on the command line, FTP prompts the user. The *mput* command is analogous to *mget* - it permits a user to request multiple file transfers with a single command. The user specifies a list of files, and FTP transfers each.

To make it easy for users to specify a set of file names, FTP allows a remote computer system to perform traditional file name expansion. Many computer systems use the asterisk (*) as a wildcard.

File name translation

Because FTP can be used between heterogeneous computer systems, the software must accommodate differences in file name syntax. For example, some computer systems restrict file names to uppercase letters, while others permit a mixture of lowercase and uppercase. Similarly, some computer systems permit a file name to contain as many as 128 characters while others restrict names to 8 or fewer characters. To handle incompatibilities among computer systems, the BSD interface to FTP permits a user to define rules that specify how to translate a file name when moving to a new computer system. Thus, a user can specify that FTP should translate each lowercase letter to its uppercase equivalent.

Changing directories and listing contents

Many computers have a hierarchical system that places each file in a *directory (folder)*. The hierarchy arises because a directory can contain other directories as well as files. FTP supports a hierarchical file system by including the concept of a current directory. At any time, the local and remote sides of a control connection are each in a specific directory. All file names are interpreted in the current directory, and all file transfers affect the current directory. The command *pwd* can be used to find the name of the remote directory. The command *cd* and *cdup* permit a user to control the directory that FTP is using on the remote computer. The command *cd* changes to a specified directory, a valid directory name must be specified on the command line. The command *cdup* changes to the parent directory (i.e., moves up one level in the hierarchy). To determine the set of files available in the current directory on the remote computer, a user can enter the *ls* command. The *ls* command produces a list of file names, but does not tell about the type of contents of each file. Thus, a user can not determine whether a given name refers to a text file, graphics image or another directory.

File types and transfer modes

FTP defines two basic types of transfer that accommodate most files: textual and binary. The user must select a transfer type, and the mode stays in effect for the entire file transfer. Textual transfer is used for basic text files. A text file contains a sequence of characters separated into lines. Most computer systems use either the ASCII or EBCDIC character set to represent characters in a text file. A user, who knows the character set used on a remote computer, can use the *ascii* or *ebcdic* command to specify textual transfer and request FTP to translate between the local and remote character sets when copying a file. The only alternative to text transfer in FTP is binary transfer, which must be used for all non-text files. For example, an audio clip, a graphics image, or a matrix of floating-point numbers must be transferred in binary mode. A user enters the *binary* command to place FTP in binary mode.

FTP does not interpret the contents of a file transferred in binary mode, and does not translate from one representation to another. Instead, binary transfer merely produces a copy - the bits of a file are reproduced without change. Unfortunately, a binary transfer may not produce the expected result. For example, consider a file of 32-bit floating point numbers. In binary mode, FTP will copy the bits of the file from one computer to another without change.

However, if the floating point representations used by the computers differ, the computers will interpret the values in the file differently.

Verbose output

Output from FTP begins with a three-digit number that identifies the message. For example, FTP places 226 at the beginning of a message that informs the user that a transfer has completed. Similarly, the remote FTP places 221 at the beginning of a message that acknowledges a request to close the control connection. A user can choose whether FTP should issue informational messages *(verbose mode)* or omit such messages and report only the results *(quiet mode).* For example, in verbose mode, FTP calculates and prints the total number of bytes in each transfer, the time required for the transfer, and the number of bytes transferred per second. To control the mode, a user enters the *verbose* command. Verbose is a toggle that reverses the mode whenever entered. Thus, entering the verbose command once turns the verbose mode off, and entering the command, a second time turns verbose mode on again.

Client-server interaction in FTP

FTP uses the client-server paradigm. A user runs a local FTP application, which interprets commands that the user enters. When a user enters an open command and specifies a remote computer, the local application becomes an FTP client that uses TCP to establish a control connection to an FTP server on the specified computer. The client and server use the FTP protocol when they communicate across the control connection. That is, the client does not pass the user's keystrokes directly to the server. Instead, when a user enters a command, the client interprets the command. If the command requires interaction with the server, the client forms a request using the FTP protocol and sends the request to the server. The server uses the FTP protocol when it sends a reply.

Control and data connections

FTP uses a control connection only to send commands and to receive responses. When it transfers a file, FTP does not send the data across the control connection. Instead, the client and server establish a separate data connection for each file transfer, use it to send one file and then close the connection. If the user requests another transfer, the client and server establish a new connection. To avoid conflict between the control and data connections, FTP uses a different protocol port number for each (port 20 for data and port 21 for control). Although data connections appear and disappear frequently, the control connection persists for the entire session. Thus, while a transfer is in progress, the client and server have two connections open: a control connection and a data connection for the transfer. Once the transfer completes, the client and server close the data connection and continue to use the control connection. The control connection is always initiated by the client, while the data connection is initiated by the server. In this case, the roles of the client and server are reversed, with the server acting as a client, and the client acting as a server.

Using separate connections for transfer and control has several advantages. First, the scheme keeps the protocols simpler and makes implementation easier. Data from a file is never confused with FTP commands. Second, because the control connection remains in place, it can be used during the transfer (e.g., the client can send a request to abort the transfer).

The Trivial File Transfer Protocol (TFTP)

The Internet protocols include a second file transfer service known as the *Trivial File Transfer Protocol (TFTP).* TFTP differs from FTP in several ways. First, communication between a TFTP client and server uses UDP instead of TCP. Second, TFTP only supports file transfer. That is, TFTP does not support interaction and does not have a large set of commands. TFTP does not permit a user to list the contents of a directory or interrogate the server to determine the names of files that are available. Third, TFTP does not have authorisation. A client does not send a login name and password. A file can be transferred only if the file permissions allow global access.

Although TFTP is less powerful than FTP, TFTP has two advantages. First, TFTP can be used in environments where UDP is available, but TCP is not. Second, the code for TFTP requires less memory than the code for FTP. Although these advantages are not important in a general-purpose computer, they can be important in a small computer or a special-purpose computer hardware device.

TFTP is especially important for bootstrapping a hardware device that does not have a disk on which to store system software. All that the device needs is a network connection and a small amount of *Read-Only Memory (ROM)* into which TFTP, UDP and IP are hardwired. When it receives power, the device executes the code in ROM, which broadcasts a TFTP request across the network. A TFTP server on the network is configured to answer the request by sending a file that contains the binary program to be run. The device receives the file, loads it into memory, and begins to execute the program.

Bootstrapping over the network adds flexibility and reduces costs. Because a separate server exists for each network, a server can be configured to supply a version of the software that is configured for the network. Cost is reduced because software can be changed without changing the hardware. For example, the manufacturer can release a new version of software for the device without changing the hardware or installing a new ROM.

Network File System (NFS)

Although it is useful, file transfer is not optimal for all data transfers. As an example, consider an application running on computer A that needs to append a one-line message to a file located on computer B. Before the message can be appended, a file transfer service requires that the entire file be transferred from computer B to computer A. Then the updated file must be transferred from A back to B. Transferring a large file back and forth introduces long delays and consumes network bandwidth. Also, the transfer is unnecessary because the contents of the file are never used on computer A. To accommodate applications that only need to read or write part of a file, TCP/IP includes a *file access* service. Unlike a file transfer service, a file access service allows a remote client to copy or change small pieces without copying an entire file.

The file access mechanism used with TCP/IP is known as the Network File System (NFS). NFS allows an application to open a remote file, move to a specified position in the file, and read or write data starting at that position. The NFS client software sends the data to the server where the file is stored along with a request to write the data to the file. The server updates the file and returns an acknowledgement. Only the data being read or written travels across the network. A small amount of data can be appended to a large file without copying the entire file. In addition to reducing the bandwidth requirements, the file access scheme used by NFS allows shared access. A file that resides at an NFS server can be accessed by multiple clients. To prevent others from interfering with file updates, NFS allows a client to lock a file. When the client finishes making changes, the client unlocks the file, allowing others access.

Example of using FTP

```
$ ftp
ftp> open (to) ftp.cs.sot.cbu.ac.zm
Connected to nakanzi.cs.sot.cbu.ac.zm
Name: anonymous
Password:
ftp> ls
ftp> get  (remote file)  InternetTechnologiesNotes   (local file)   cbufile
ftp> close
221 Goodbye.
ftp> quit
```

Copperbelt University
Computer Science Department

Internet Technologies

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

World Wide Web (WWW) Pages and Browsing

Web Terminologies

HTTP (Hypertext Transfer Protocol)
The Hypertext Transfer Protocol is used for request-reply interaction implemented over TCP/IP. A client requests and the server accepts a connection at the default server port or at a port specified in the URL. Then the client sends a request message to the server and the server replies the client with a message. The interaction is finished the connection is closed. Some versions of HTTP use persistent connections which are open connections over a series of request-reply exchanges between a client and a server. The Request-Reply interaction is further explained with the following actions. Before a message for a connection is sent to the server there is a process of authentication using a password style. When the connection is established the Request-Reply messages are marshalled into ASCII text strings. The client and sever also negotiation for the contents of the message which they can accept. These content types can be MIME (Multipurpose Internet Mail Exchanges) types as prefix of a content data. HTTP is also called a 'stateless protocol' in that after an HTTP request is serviced by a web server the TCP/IP transaction is completed. No channel of communication is maintained. Subsequent requests are not affected in any way by previous requests. (Unlike FTP or Telnet, which authenticate and open a communication channel that stays open)

Hypertext
The Web is a graphical, platform-independent, distributed, decentralized, multi-formatted, interactive, dynamic, nonlinear, and immediate, two way communication medium. The basic mechanism that enables all of it is actually quite simple – the hypertext link, a kind of "jump point" that allows a visitor to jump from a place in a Web page to any other Web page, document, or binary data-object (a script, graphic, video, etc) on the Web. A link can connect anything anywhere that has an address or URL on the Net.
A hypertext link also referred to as an anchor, actually works much like a cross reference in a book, except that you can immediately go to it simply by clicking on the link, whether it's a link within the same document or to a page or document halfway around the world.

Hypermedia
Given that hypertext linking occurs within and between documents, it makes sense that hypermedia refers to connecting with and between other non-text (binary) media such as graphics, audio, animation, video, or software programs. Many pages on the Web are now generated on the fly from scripts, programs, or database queries. Java applets are being used to generate dynamically interactive Web sites.

Web Page
A Web page is a hypertext (HTML) document contained in a single file. It is simply a plain text document. All the codes entered into the document as ordinary text, with none of the binary level formatting a word processor would embed in it.

Web Site
Often Servers are called Web sites but any grouping of related and linked Web pages sharing a common theme or subject matter is called a Web site. In simple terms a Web site is a collection of related Web pages.

HTML
HTML is a subset of SGML (Standard Generalized Mark-up Language). SGML was developed to standardize the mark-up, or preparation for type-setting, of computer-generated documents. HTML, on the other hand was specifically developed to mark up, or encode, hypertext documents for display on the World Wide Web. An HTML document is a plain text file with codes (called tags) inserted in the text to define elements in the documents. HTML tags generally have two parts, an on-code, and an off-code, which contain the text to be defined. The most important thing to keep mind about HTML is that its purpose isn't necessarily to specify the exact formatting or layout of a Web page but define and specify the specific elements that make up a page- the body of the text, headings, paragraphs, line breaks, text elements, and so on. You use HTML primarily to define the composition of a Web page, and much less to determine its appearance. The particular Web browser you use to view the page controls the display of the Web page.

URLs (Uniform Resource Locators) are addresses that identify a server, a directory, or a specific file. HTTP URLs or Web addresses are the only one type of addresses on the Web. FTP, Gopher, and WAIS are other types of addresses you will find fairly often on the Web as well.

Browser interface
The World Wide Web, also called the WWW, W3, or simply the Web, dates back to 1989, when Tim Berners-Lee, often called "the inventor of the World Wide Web," proposed it. Many others have been critically involved, but Berners-Lee gets the credit for originally proposing and evangelizing the idea as a way to facilitate collaboration between scientists and other academics over the Internet.
On the original web page for the World Wide Web Project, posted on the CERN (the European Laboratory for Particle Physics, birthplace of the WWW) server in 1992, Berners-Lee described the World Wide Web as "a wide area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents. Today you can just define it as "the universal space of all network-accessible information."

The *World Wide Web (WWW)* is a large-scale, on-line repository of information that users can search using an interactive application program called a *browser*. Most browsers have *a point and click* interface. The browser displays information on the computer's screen and permits a user to navigate using the mouse. The information displayed includes both text and graphics. Furthermore, some of the information on the display is highlighted to indicate that an item is *selectable*. When the user places the cursor over a selectable item and clicks a mouse button, the browser displays new information that corresponds to the selected item.

Hypertext and hypermedia

The Web is a distributed *hypermedia* system that supports interactive access. A hypermedia system provides a straightforward extension of a traditional *hypertext* system. In either system, information is stored as a set of documents. Besides the basic information, a document can contain pointers to other documents in the set. Each pointer is associated with a selectable item that allows the user to select the item and follow the pointer to a related document. The difference between hypertext and hypermedia arises from document content: hypertext documents contain only textual information, while hypermedia documents can contain additional representations of information, including digitized photographic images or graphics. The difference between a distributed and a non-distributed hypermedia system is significant. In a non-distributed system, information resides within a single computer, usually on a single disk. Because the entire set of documents is available locally, links among them can be checked for consistency. That is, a non-distributed hypermedia system can guarantee that all links are valid and consistent. In contrast, the Web distributes documents across a large set of computers. Furthermore, a system administrator can choose to add, remove, change or rename a document on a computer without notifying other sites. Consequently, links among Web documents are not always consistent. In other words, because the computers used to store Web documents are administered independently, links among documents can become invalid.

Document representation

A hypermedia document available on the Web is called a *page*. The main page for an organisation or an individual is known as a *homepage*. Because a page can contain many items, the format must be defined carefully so that a browser can interpret the contents. In particular, a browser must be able to distinguish among arbitrary text, graphics and links to other pages. The author of a page should be able to describe the general document layout (e.g., the order in which items are presented). Each Web page that contains a hypermedia document uses a standard representation. Known as the *HyperText Markup Language (HTML)*, the standard allows an author to give general guidelines for display and to specify the contents of the page. HTML is a *markup language* because it does not include detailed formatting instructions. For example, although HTML contains extensions that allow an author to specify the size of text, the font to be used, or the width of a line, most authors choose instead to specify only a level of importance as a number from 1 through 6. The browser chooses a font and display size appropriate for each level. Similarly, HTML does not specify exactly how a browser marks an item as selectable - some browsers underline selectable items, others display selectable items in a different colour, and some do both. Consequently, two browsers may display the same document differently.

HTML PROGRAMMING

When a web browser displays a page it reads from a plain text file, and looks for special codes or "tags" that are marked by the < and > signs. The general format for a HTML tag is:

<tag_name>string of text</tag_name>

As an example, the title for this section uses a **header** tag:

```
 <h3>What are HTML tags?</h3>
```

This tag tells a web browser to display the text **What are HTML tags?** in the style of header level 3. HTML tags may tell a web browser to bold the text, italicize it, make it into a header, or make it be a hypertext link to another web page. It is important to note that the ending tag,

```
 </tag_name>
```

contains the "/" slash character. This "/" slash tells a web browser to stop tagging the text. Many HTML tags are paired this way. If you forget the slash, a web browser will continue the tag for the rest of the text in your document, producing undesirable results.

NOTE: A web browser does not care if you use upper or lower case. For example, <h3>...</h3> is no different from <H3>...</H3>

Unlike computer programming, if you make a typographical error in HTML you will not get a "bomb" or "crash" the system; your web page will simply look, well... wrong. It is quick and easy to go inside the HTML and make the changes.

Your browser has a small but open vocabulary! An interesting aspect of HTML is that if the browser does not know what to do with a given tag, it will ignore it! For example, in this document you are viewing, the header tag for this section *really* looks like this:

```
 <wiggle><h3>What are HTML tags?</h3></wiggle>
```

but since your browser probably does not support a **<wiggle>** tag. it proceeds with what it knows how to do.

Creating Your HTML Document

An HTML document contains two distinct parts, the head and the body. The head contains information about the document that is not displayed on the screen. The body then contains everything else that is displayed as part of the web page.

The basic structure then of any HTML page is:

```
  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
  <html>
  <head>
  <!-- header information used to contain extra information about this document,
not displayed on the page -->
  </head>

  <body>

  <!-- all the HTML for display -->
          :      :
          :      :
          :      :
  </body>
  </html>
```

The very first line:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
is not technically required, but is a code that tells the browser what version of HTML the current page is written for. Enclose all HTML content within `<html>...</html>` tags. Inside is first your

`<head>...</head>` and then the `<body>...</body>` sections.
Also note the comment tags enclosed by `<!-- blah blah blah -->`. The text between the tags is NOT displayed in the web page but is for information that might be of use to you or anyone else who might look at the HTML code behind the web page. When your web pages get complicated. The comments will be very helpful when you need to update a page you may have created long ago.

Here are the steps for creating your first HTML file. Go to a text editor e.g. NOTEPAD
Enter the following text:

```
     <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
     <html>
     <head>
     <title>Volcano Web</title>
     </head>
     <!-- written for the Writing HTML programs by Ntalasha 3rd April, 2006 -->
     <body>
       In this lesson you will use the Internet to research information on
volcanoes and then
       write a report on  your results.
     </body>
     </html>
```

Typical example of an HTML document

A Simple Web Page (source)

```
<html>
<head>
  <title>A Simple Webpage</title>
</head>
<body>
This is a simple webpage.
</body>
</html>
```

NOTE: Look where the `<title>...</title>` tag is located. It is in the `<head>...</head>` portion and thus will not be visible on the screen. The `<title>` tag is used to uniquely identify each document and is also displayed in the title bar of the browser window.

Also note that we have inserted a comment tag that lists the name of the author and the date the document was created. You could write anything in between the comment tags but it is only visible when you look at the source HTML for a web page. Save the document as a file called "`volc.html`"

You can create files with names like `VOLC.HTML` if you use Windows95 or a later Windows operating system. By using this file name extension, a web browser will know to read these text files as HTML and properly display the web page.

Displaying Your Document in a Web Browser

1. Go to the web browser window select **New Window** or **New Browser** from the **File** window.)

2. Select **Open File...** from the **File** menu. (Note: For users of Internet Explorer, click the **Browse** button to select your file)
3. Use the dialog box to find and open the file you created, **"volc.html"**
4. You should now see in the title bar of the window the text "Volcano Web" and in the web page below, the one sentence of **<body>** text you wrote, "In this lesson..."

A Brief Introduction to Tags

The web page `simple.html` uses the following tags: `<html>`, `<head>`, `<title>`, and `<body>`.

Tags do not appear directly when you view a web page. Instead, they serve as **instructions to your browser**. Tags can change the font, create a link, insert an image, and more.

The tags in `simple.html` that begin with a slash (`</html>`, `</head>`, `</title>`, and `</body>`) are called **closing tags**. Closing tags stop the effect of the tag that they correspond to. For example, the `<body>` section ends at the closing tag, `</body>`.

Short List of Tag Properties

- Tags are delineated by angle brackets (`<  >`)
- Tags are "invisible"; they don't directly appear in the web page
- Closing tags begin with a slash and end the effect of a previous tag

We'll discuss the general properties of tags in some detail in the next section. For now, let's focus on the particular tags in the "Simple Web Page" example.

Tags in Example 1 (Structural Tags)

- `<html>`: "Here begins an HTML document."

  All well-formed HTML documents have one `<html>` and one `</html>` tag. These tags, along with the `.html` file extension, identify a file as an HTML document.

- `<head>`: "Here begins the header."

  The header contains tags that apply to the overall document. For example, it might contain tags that are designed for search engines to pick up: keywords, a short description, and so on. Information in the header is *not* meant to be directly displayed as normal webpage content.

  The reasons you would be concerned about the header are a bit abstract at this stage.

- `<title>`: "Here begins the document title." (Must be in the header)

  Along the *very top* of your browser window, you will see the title. These words appear because of the `<title>` tag.

When a search engine indexes your page, it will probably use the `<title>` as the page title it displays to users. If you omit the `<title>`, the search engine will make up one for you. This is Not a Good Thing.

- `<body>`: "Here begins the body."

  All well-formed HTML documents have one `<body>` and one `</body>` tag. This section is where you put all the text and tags that should be displayed in the main browser window.

Adding Text

Adding Text (**source**)
```
<html>
<head>
  <title>A Simple Webpage</title>
</head>
<body>
This is a simple webpage.
And I mean really simple.
</body>
</html>
```

To view this alteration: Add the text in bold to your `simple.html` file. Save the file as `adding.html`. Open the `adding.html` file in your browser.
Adding Text (Results).

This is a simple webpage. And I mean really simple.

HTML and Whitespace

What happened here? Why are the two sentences on the same line? HTML ignores whitespace (spaces, tabs, or carriage returns) in the source. It replaces all continuous chunks of whitespace with a single space: " ".

The **advantage** is that you can use whitespace to make your code more readable, while the browser can freely adjust line breaks for each user's window and font size.
The **disadvantage** is that it might take you a while to get used to this behavior. Of course, there are tags that adjust the whitespace.

Making Text Bold

Change the text of `adding.html` to the following:
Bold Text (**source**)
This is a simple webpage.
And I mean <b>really</b> simple.
Save your changes and view the file in your browser.
Bold Text (Results)

This is a simple webpage. And I mean really simple.
As you can see, the text becomes bold, starting at the `<b>` and ending at the `</b>`. This is our first example of a *physical style tag* -- a tag that directly controls the physical appearance of the text.

Changing the Background Color

Let's do something a little more dramatic:
Background Color (**source**)

```
<html>
<head>
  <title>A Simple Webpage</title>
</head>
<body style="background-color: yellow">
This is a simple webpage.
And I mean <b>really</b> simple.
</body>
</html>
```

Embedding graphics in a web page

Non-textual information such as a graphics image or a digitised photo is not inserted directly in an HTML document. Instead, the data resides in a separate location, and the document contains a reference to the data. When a browser encounters such a reference, the browser goes to the specified location, obtains a copy of the image and inserts the image in the displayed document. HTML uses the *IMG* tag to encode a reference to an external image. For example, the tag: *<IMG SRC="sipalo_photo.gif">* specifies that file *sipalo_photo.gif* contains an image that the browser should insert in the document.

Files listed in an *IMG* tag differ from files used to store Web pages. Image files are not stored as text files and do not follow HTML format. Instead, each image file contains binary data that corresponds to one image, and the file is stored in *graphics interchange format (gif)*. Because the image file does not include any formatting information, the IMG tag includes additional parameters that can be used to suggest positioning. In particular, when an image appears with other items (i.e., text or other images), the keyword ALIGN can be used to specify whether the top, middle, or bottom of the image should be aligned with other items on the line, e.g., *<IMG SRC= "sipalo_photo.gif" ALIGN=middle>*

Identifying a page

When a user invokes a browser, a user must specify an initial page to view. Identifying a page is complicated for several reasons. First, the Web includes many computers, and a page can reside on any of them. Second, a given computer can contain many pages; each must be given a unique name. Third, the Web supports multiple document representations, so a browser must know which representation a page uses (e.g., whether the page refers to an HTML document or to an image stored in binary form). Fourth, because the Web is integrated with other applications, a browser must know which application protocol must be used to access a page. A syntactic format was invented that incorporates all the information needed to specify a remote item. This syntactic form encodes the information in a character string known as a *Uniform Resource Locator (URL*). The general form of a URL is:

protocol://computer_name:port/document_name
where *protocol* is the name of the protocol used to access the document, *computer_name* is the domain name of the computer on which the document resides, *:port* is an optional protocol port number, and *document_name* is the name of the document on the specified computer. A URL contains the information a browser needs to retrieve a page. The browser uses the separator character colon and slash to divide the URL into three components: a protocol, a computer name and a document name. The browser then uses the information to access the specified document.

Hypertext links from one document to another

Although HTML includes many features used to describe the contents and format of a document, the feature that distinguishes HTML from conventional document formatting languages is its ability to include hypertext references. Each hypertext reference is a passive pointer to another document. Unlike an IMG reference, which causes a browser to retrieve an external information immediately, a hypertext reference does not cause an immediate action. Instead, a browser turns a hypertext reference into a selectable item when displaying the document. If the user selects the item, the browser follows the reference, retrieves the document to which it refers and replaces the current display with the new document. HTML allows any item to be designated as a hypertext reference. Thus, a single word, phrase, an entire paragraph or an image can refer to another document. If an entire image is designated as a hypertext reference, the user can select the reference by placing the cursor at any position in the image and clicking a mouse button. Similarly, if an entire paragraph is designated as a hypertext reference, clicking on any character in the paragraph causes the browser to follow the reference. The HTML mechanism for specifying a hypertext reference is known as an *anchor*. To permit arbitrary text and graphics to be included in a single reference, HTML uses tags <A> and </A> as a bracket for the reference. All items in between the two are part of the anchor. Tag <A> includes information that specifies a URL. If the user selects the reference, the browser uses the URL to obtain the document. For example, the following input contains an anchor that references the URL http://www.mazusa.com.

```
These notes were compiled at
<A HREF="http://www.mazusa.com">
Mazusa Computer Services, </A> one of Zambia's upcoming companies.
```

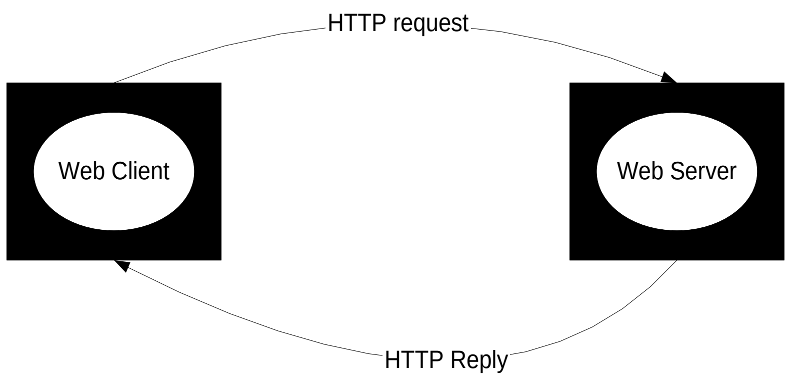
When displayed on screen, the input produces:

These notes were compiled at Mazusa Computer Services
one of Zambia's upcoming companies.

The example shows a convention that many browsers use to indicate that text is selectable. The anchored text is displayed with an underline. Only the words Mazusa Computer Services are underlined because the others are not anchored. Everything in the input between the tags that start and end an anchor form part of the selectable item. The input inside an anchor can include text, tags that specify formatting, or graphics images. Thus, an HTML document can contain a picture or an icon that corresponds to a hypertext link as easily as a sequence of words.

Client-server interaction

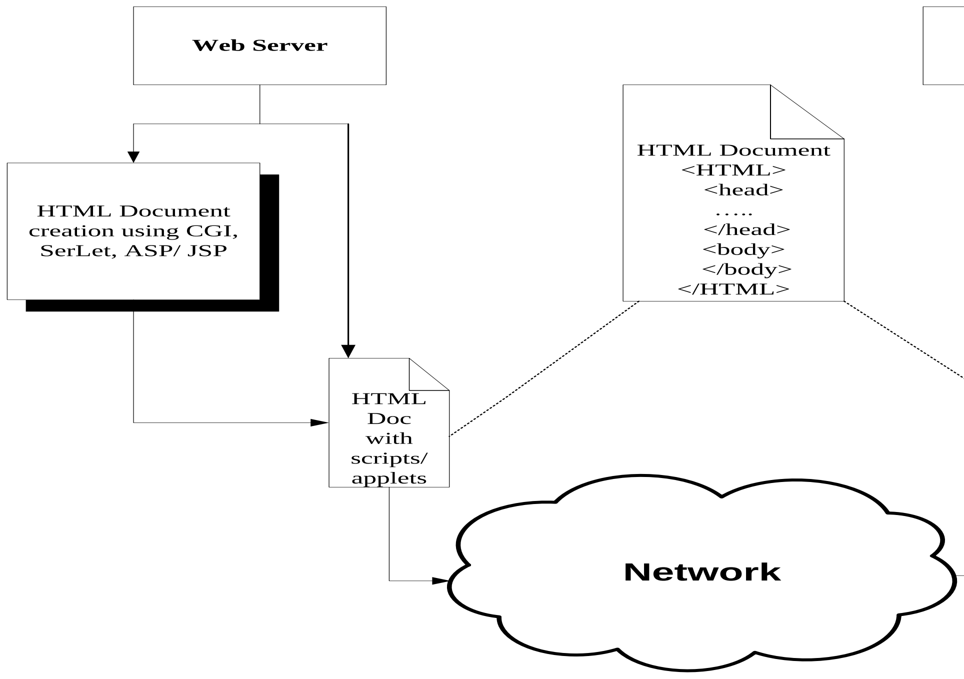
Web browsing uses the client-server paradigm, when given the URL of a document, a browser becomes a client that contacts a server on the computer specified in the URL to request the document. The browser then displays the document for the user. The connection between a Web browser and a server has a short duration. The browser establishes a connection, sends a request, and receives the requested item or a message that no such item exists. As soon as the document or image has been transferred, the connection is closed. Terminating connections quickly works well in most instances because browsing does not exhibit high locality. A user might access a Web page on one computer and then immediately follow a link to a Web page on another computer. However, terminating connections too quickly can introduce overhead in cases where a browser must return to the same server for many documents. For example, consider a page that contains references to several images, all of which reside on the same computer as the page. When a user selects the page, the user's browser opens a connection, obtains the page, and closes the connection. When it needs to display an image (i.e., when it encounters an <IMG> tag in the page), the browser must open a new connection to the same server to obtain a copy of the image. The diagram below illustrates client server interaction in web sites.



The Web Browser (GUI to WWW) requests eg links/ URL/Forms in form of HTML as a HTTP request. The HTTP reply can also be a GUI display or another web link. The Web Server processes or relays HTTP request (e.g. GET or PUT) and then generates HTTP reply (e.g. including a Web page).

Web document transport and HTTP

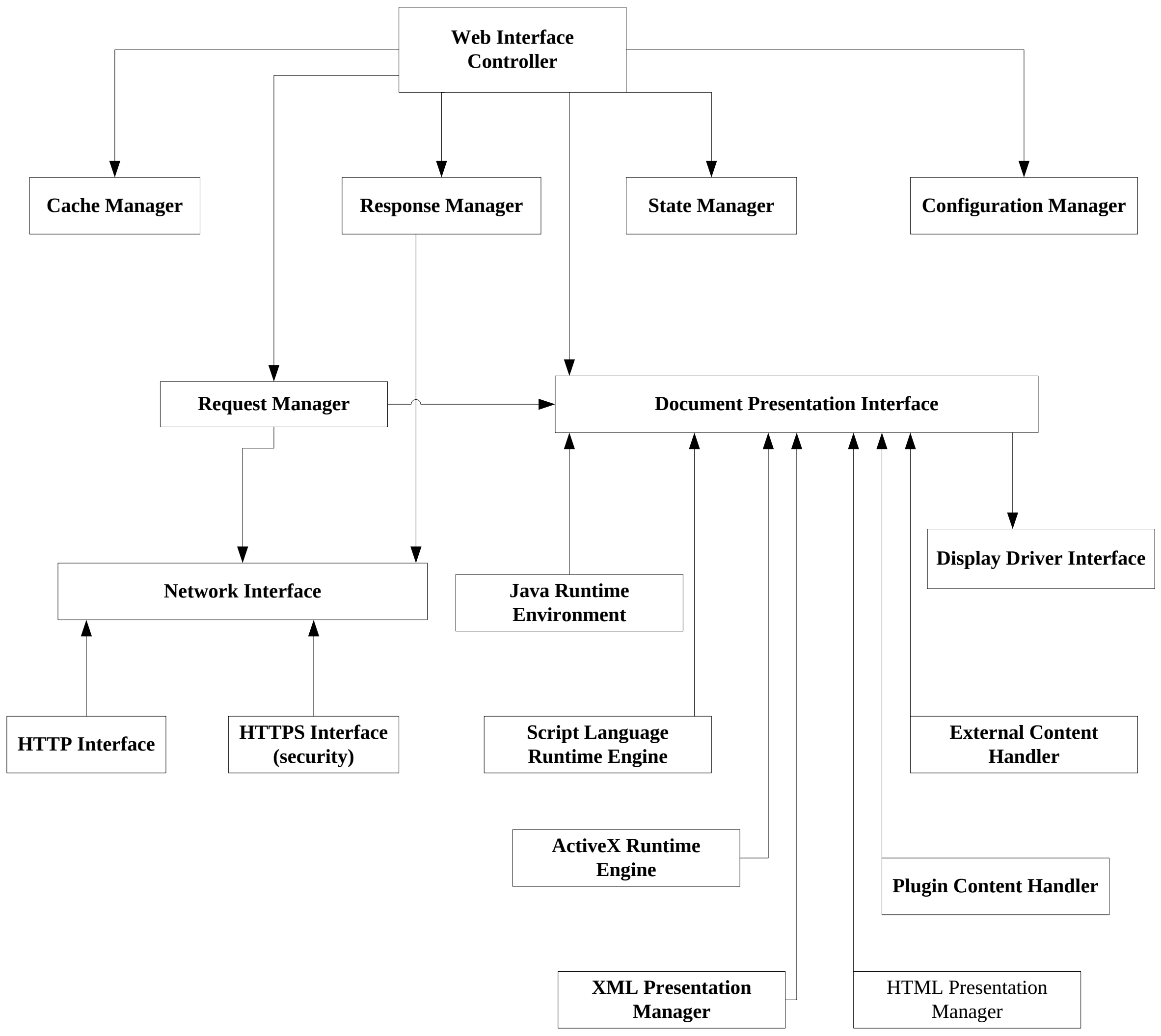
When a browser interacts with a Web server, the two programs follow the *HyperText Transport (Transfer) Protocol (HTTP).* HTTP allows a browser to request a specific item, which the server then returns. To ensure that browsers and servers can interoperate unambiguously, HTTP defines the exact format of requests sent from a browser as well as the format of replies that the server returns.


Web Server
Web Broswer
HTML Document creation using CGI, SerLet, ASP/ JSP
HTML Document <HTML> <head> ….. </head> <body> </body> </HTML>
HTML Doc with scripts/ applets
HTML Doc with scripts/ applets
Network

Browser architecture

Web browsers have a more complex structure than Web servers. A server performs a straightforward task repeatedly, that is, the server waits for a browser to open a connection and request a specific page. The server then sends a copy of the requested item, closes the connection, and waits for the next connection. A browser handles most of the details of document access and display. Consequently, a browser contains several large software components that work together to provide the illusion of a seamless service. Conceptually, a browser consists of a set of clients, a set of interpreters and a controller that manages them. The controller forms the central piece of the browser. It interprets both mouse clicks and keyboard input, and calls other components to perform operations specified by the user. For example, when a user enters a URL or clicks on a hypertext reference, the controller calls a client to fetch the requested document from the remote server on which it resides, and an interpreter to display the document for the user.

Each browser must contain an HTML interpreter to display documents. Other interpreters are optional. Input to an HTML interpreter consists of a document that conforms to the HTML syntax. Output consists of a formatted version of the document on the user's display. The interpreter handles layout details by translating HTML specifications into commands that are appropriate for the user's display hardware. For example, if it encounters a heading tag in the document, the interpreter changes the size of the text used to display the heading. Similarly, if it encounters a break tag, the interpreter begins a new line of output. One of the most important functions in an HTML interpreter involves selectable items. The interpreter must store information about the relationship between positions on the display and anchored items in the HTML documents. When the user selects an item with the mouse, the browser uses the current cursor position and the stored position information to determine which item the user has selected. The figure below illustrates the architectural view of a typical browser.



Optional clients

Besides an HTTP client and an HTML interpreter, a browser can contain components that enable the browser to perform additional tasks. For example, many browsers include an FTP client that is used to access the file transfer service. Some browsers also contain an e-mail client software that enables the browser to send and receive e-mail messages.

When using a browser, a user does not invoke such services as file transfer explicitly nor does he interact with conventional software. Instead, the browser invokes the service automatically, and uses it to perform a task as needed. If the browser is well-designed, it hides details from a user, who may be unaware that an optional service has been invoked. For example, file transfer can be associated with a selectable item on the screen. When the user selects the item, the browser uses its FTP client to obtain a copy of the file. The first field in a URL specifies a protocol. The controller uses the field to determine which client to call. For example, if the URL specifies *http*, the controller calls the HTTP client. Similarly, the URL:

ftp://ftp.cs.cbu.ac/pup/nakanzi/notes/itnotes.doc specifies that the browser should use anonymous FTP to retrieve the file *pub/nakanzi/notes/itnotes.doc* from the computer *ftp.cs.cbu.ac*.

Caching in Web browsers

The pattern of client-server contact in Web browsing is unlike the pattern in many of the other applications. First, because users tend to view Web pages outside of their organisation, Web browsers tend to reference remote pages more frequently than local pages. Second, because users do not search for the same information repeatedly, they do not tend to repeat accesses day after day.

Browsers use a cache to improve document access. The browser places a copy of each item it receives in a cache on the local disk. When a user selects an item, the browser checks the disk cache before retrieving a fresh copy. If the cache contains the item, the browser obtains the copy from the cache without using the network.

Retaining items in the cache for extended periods is not always desirable. First, a cache can take vast amounts of disk space. For example, suppose a user visits ten pages, each of which contains five large images. The browser will store the page documents plus all fifty images in the cache on the local disk. Second, improvements in performance are only helpful if a user decides to view an item again. Unfortunately, users often browse because they are looking for information. When the information has been found, the user stops browsing. For example, a user who views ten pages may decide that nine of the ten pages contain nothing of interest. Thus, storing a copy of such pages in a cache will not improve performance because the user will never return to the pages. In such cases, caching actually lowers performance because the browser takes time to write items to disk unnecessarily.

To help users control how a browser handles the cache, most browsers allow the user to adjust the cache policy. The user can set a time limit for caching, and the browser removes items from the cache after the time limit expires. Browsers usually maintain a cache during a particular session. A user who does not want items to remain in the cache between sessions can request a cache time of zero. In such cases, a browser removes the cache whenever the user terminates the session.

Copperbelt University
Computer Science Department

<u>Internet Technologies</u>

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

CGI Technology for Dynamic Web Documents

Three basic types of Web documents

Although we previously described a Web page as a simple textual document stored in a file, alternative representations exist. In general, all documents can be grouped into three broad categories according to the time at which the contents of the document are determined.

1. *Static*. A Web document resides in a file that is associated with a Web server. The author of a static document determines the contents at the time the document is written. Because the contents do not change, each request for a static document results in exactly the same response.

2. *Dynamic*. A dynamic Web documents does not exist in a predefined form. Instead, a dynamic document is created by a Web server whenever a browser requests the document. When a request arrives, the Web server runs an application program that creates the dynamic document. The server returns the output of the program as a response to the browser that requested the document. Because a fresh document is created for each request, the contents of a dynamic document can vary from one request to another.

3. *Active.* An active document is not fully specified by the server. Instead, an active document consists of a computer program that understands how to compute and display values. When a browser requests an active document, the server returns a copy of the program that the browser must run locally. When it runs, the active document program can interact with the user and change the display continuously. Thus, the contents of an active document are never fixed - they can continue to change as long as the user allows the program to run.

Advantages and disadvantages of each document type

The chief advantages of a static document are simplicity, reliability and performance. Because it contains straightforward formatting specification, a static document can be created by someone who does not know how to write a computer program. After it has been created and tested thoroughly, a static document remains valid indefinitely. Finally, a browser can display a static document rapidly, and place a copy in a cache on a local disk to speed future requests for the document. The chief disadvantage of static a document is inflexibility - the document must be revised whenever information changes. Furthermore, changes are time-consuming because they require a human to edit a file. Thus, a static document is not useful for porting information that changes frequently.

The chief advantage of a dynamic document lies in its ability to report current information. For example, a dynamic document can be used to report information such as current stock prices, current weather conditions, or the current availability of tickets for a concert. Whenever a browser requests the information, the server runs an application that accesses the needed information and creates a document. The server then sends the document to the browser. Dynamic documents place the responsibility on the server; a browser uses the same interaction to obtain a dynamic page as to retrieve a static page. From the browser's point of view, dynamic documents are indistinguishable from static pages. Because both static and dynamic pages use HTML, a browser does not know whether the server extracted the pages from a disk file or obtained the page dynamically from a computer program. The chief disadvantages of a dynamic document approach are increased cost and the inability to display changing information. Like a static document, a dynamic document does not change after a browser retrieves a copy. Thus information in a dynamic document begins to age as soon as it has been sent to the browser. For example, consider a dynamic document that reports current stock prices. Because stock prices change quickly, the document can become obsolete while a user is browsing through its contents.

A dynamic document is more expensive to create and access than a static document. Personnel costs for creating dynamic documents are higher because the creator of a dynamic Web page must know how to write a computer program. Moreover, the program must be designed carefully and tested extensively to ensure that the output is valid. Verifying the correctness of such a program can be difficult because the input can include multiple data values obtained from various sources. In addition to the higher cost of creating dynamic documents, hardware costs increase for dynamic documents because a more powerful computer system is needed to operate the server. A dynamic document takes slightly longer to retrieve than a static document because the server requires additional time to run the application program that creates the document.

The chief advantage of an active document over a dynamic document lies in its ability to update information continuously. For example, only an active document can change the display quickly enough to show an animated image. An active document can access sources of information directly and update the display continuously. For example, an active document that displays stock prices can continue to retrieve stock information and change the display without requiring any action from the user. The chief disadvantage arises from the additional costs of creating and running such documents, and from a lack of security. First, active document display requires more sophisticated browser software and a powerful computer system to run the browser. Second, writing active documents that are correct requires more programming skill than other forms, and the resulting documents are more difficult to test. In particular, because an active document must run on an arbitrary computer instead of a server, the program must be written to avoid depending on features available on one computer system that are not available on other computer systems. Finally, an active document is a potential security risk because the document can export as well as import information.

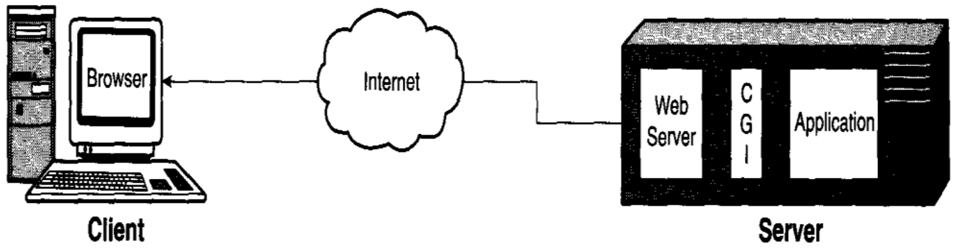
Implementation of dynamic documents

Because the responsibility for creating a dynamic document rests with the Web server that manages the document, changes required to support dynamic documents involve only servers. Three additions are required before a conventional Web server can handle a dynamic document. First, the server program must be extended so it is capable of executing a separate application program that creates a document each time a request arrives. The server must be programmed to capture the output from the application program and return the document to the browser. Second, a separate application program must be written for each dynamic document. Third, the server must be configured so it knows which URLs correspond to dynamic documents and which correspond to static documents. For each dynamic document, the configuration must specify the application program that generates the document. A server uses configuration information and the URL in each incoming request to determine how to proceed. If the configuration information specifies that the URL corresponds to a static document, the server fetches the document from a file as usual. If the URL corresponds to a dynamic document, the server determines the application program that generates the dynamic document, executes the program and uses the output from the program as the document to be returned to the browser.

The Common Gateway Interface

HTML is a markup language and, as such, cannot by itself describe computations, allow interaction with the user, or provide access to a database. Yet, these are things that are commonly needed in HTML documents. Computations are required to provide support for all kinds of Web commerce, for example, to compute the cost of a list of items ordered on a form. The Web also is now a common way to access databases, for example, for making reservations for transportation services. A wide variety of Web sites require interaction with the user, including games and verification of user names and passwords for private sites. One early response to these needs was to develop a technique for the browser to access the software resources of the server machine. Using this approach, the browser can run programs indirectly on the server, including database access systems and the databases that reside on the server. These server-based programs communicate back to the client through HTTP with HTML documents. The interface between a browser and software on the server is called the Common Gateway Interface.

An HTTP request to run a CGI program is like any other HTTP request, except that the requested file can be identified by the server as being a CGI program. Servers can identify CGI programs by their addresses on the server or by their filename extensions. When a server receives a request for a CGI program, it does not return the file - it executes the program in the file and returns that program's output.
A CGI program can produce results in a number of different forms. Most often, an HTML document is returned, although the program may produce only the URL of an existing document. When a new document is generated, it consists of two parts: an HTTP header and a body. The header must be complete if the response is going directly to the client. It can be partial if the response is going to the client through the server, which completes the header. The form of the body, which is specified in the header, can be HTML, plain text, or even an image or audio file.
One common way for a user to interact through a browser with a Web server is through forms. A form is presented to the user, who is invited to fill in the boxes and click the buttons of the form. The user submits the form to the server by clicking its Submit button. The contents of the form are encoded and transmitted to the server. The server must use a program to decode the transmitted form contents, perform whatever computation is necessary on the form data, and produce its output. The figure below shows the communications configuration for CGI.



Both CGI programs and client-side scripts, such as JavaScript, support dynamic documents. However, client-side scripts cannot access files and databases on the server.

The CGI standard

A widely used technology for building dynamic Web documents is known as the *Common Gateway Interface (CGI).* The CGI standard specifies how a server interacts with an application program that implements a dynamic document. The application is called a *CGI program*. CGI provides general guidelines, and allows a programmer to choose most details. For example, CGI does not specify a particular programming language. Instead the standard permits a programmer to choose a language and to use different languages for different dynamic documents. Thus, a programmer can choose an appropriate language for each document. For example, a programmer can use a conventional programming language like FORTRAN, C or C++ for documents that require intense arithmetic computation, and use a scripting language like Perl, Python the UNIX shell for documents that require only minor text formatting.

CGI Linkage

This section describes how the connection between an HTML document being displayed by a browser and a CGI program on the server is established. CGI programs often reside in a specific subdirectory on the server, cgi-bin, although the Web server administrator can choose a different place for these programs. All that is required is for both the Web server and the CGI programmer to know where CGI programs are stored. A server can be configured to recognize the requested file as being a program by the extension on the file's name. For example, if the extension on the requested file's name is . p1, that could indicate that it is a CGI program. Because CGI programs are invoked by the server, their access protection code must be set to allow this. If a CGI program has been compiled and is in machine code, the server can invoke it directly. However, the compiled versions of the programs are not saved and are not in machine code anyway.

Output from a CGI program

In practice output from a CGI program is not restricted to HTML - the standard permits CGI applications to generate arbitrary document types. To distinguish among various document types, the standard allows a CGI program to place

a header on its output. The header consists of text that describes the document type. Headers generated by CGI programs use the same general format as headers that a Web server uses when sending a document to a browser. The header consists of one or more lines of text and a blank line. Each line of the header specifies information about the document and the data representation. After it runs a CGI program, a server examines the header before returning the document to the browser that issued the request. Thus, a CGI program can use the header to communicate with the server when necessary. For example, a header that consists of the line: *Content-type: text/html*
Followed by a blank line specifies that the output is an HTML document. The header in the output from a CGI program can also be used to specify that the document is in a new location. The technique is known as *redirection*. For example, suppose a CGI program associated with the URL: *http://someserver/cgi-bin/wena* needs to refer incoming requests to the document associated with the URL: *http://someserver/new/yebo.txt*, the CGI program can generate two lines of output:

```
Location: /new/yebo.txt
<blank line>
```

When it detects the *Location:* directive in the header, the server will retrieve the document as if the browser had requested */new/yebo.txt* (instead of */cgi-bin/wena*) with the URL:
http://someserver/new/yebo.txt

An example CGI program

Below is the code for a simple CGI program written in the UNIX shell language. When executed, the program creates a plain text document that contains the current date and time.

```
#!/bin/sh
#
# CGI script that prints the date and time at which it was run
#
# Output the document header followed by a blank line
echo Content-type: text/plain
echo
# Output the date
echo This document was created on: `date`
```

In essence, the shell is a command interpreter - a shell script contains commands in the same format a user follows when entering commands on a keyboard. The shell ignores blank lines or any line that begins with the hash symbol (#). The hash symbol denotes the beginning of a comment. The first word of a line is the name of the command being used and subsequent words form the arguments to the command. The only command used in the example script is *echo.* Each invocation of *echo* generates one line of output. When invoked with no arguments, *echo* creates a blank line of output. Otherwise, *echo* writes an exact copy of its arguments. The shell interprets the accent grave quotes in *`date`* as a request to execute the date program and substitute its output. Thus, before it invokes *echo* for the third time, the shell replaces string *`date`* with the current date and time. For example, if the script is run on October 24, 2002 at 20 seconds past 12 hours, the script would generate the following output:

```
Content-type: text/plain

This document was created on Thu Oct 24 12:00:20  2002
```

A server that is capable of running CGI programs must be configured before it can invoke the example script. The configuration specifies a URL that the server uses to locate the script. When a browser contacts the server and requests the specified URL, the server runs the program. When the browser receives the document, the browser displays a single line for the user:

```
This document was created on Thu Oct 24 12:00:20  2002
```

Unlike a static document, the contents of a dynamic document change each time the user instructs the browser to reload the document. Because the browser does not cache a dynamic document, a request to reload the document causes the browser to contact the server. The server invokes the CGI program, which creates a new document with the current time and date. Thus, the user sees the contents of the document change after each request to reload.

Parameters and environment variables

The CGI standard permits a CGI program to be parameterised. That is, a server can pass arguments to a CGI program whenever the program is invoked. Parameterisation is important because it allows a single CGI program to handle a set of dynamic documents that differ only in minor details. Values for the parameters can be supplied by the browser. To do so, the browser adds additional information to a URL. When a request arrives, the server divides the URL in the request into two parts: a prefix that specifies a particular document and a suffix that contains additional information. If the prefix of a URL corresponds to a CGI program, the server invokes the program and passes the suffix to the URL as an argument. Syntactically, a question mark (?) in the URL separates the prefix from the suffix. Everything before the question mark forms a prefix, which must specify a document. The server uses its configuration information to determine how to map the prefix to a particular CGI program. When it invokes the CGI program, a server passes an argument to the program that consists of all characters in the URL following the question mark.

Because it was originally designed for Web servers that run under operating systems like UNIX and Windows, the CGI standard follows an unusual convention for passing arguments to CGI programs. Instead of using the command-line, a server places argument information in UNIX *environment variables* and then invokes the CGI program. The CGI program inherits a copy of the environment variables, from which it extracts the values. For example, the server assigns the suffix of the URL to the environment variable QUERY_STRING. A server also assigns values to other environment variables that the CGI program can use. The table in figure 14 lists examples of CGI environment variables along with their meanings.

Name of variable	**Meaning**
SERVER_NAME	The domain name of the computer running the server.
GATEWAY_INTERFACE	The version of the CGI software that the server is using
SCRIPT_NAME	The path in the URL after the server name
QUERY_STRING	Information following "?" in the URL
REMOTE_ADDR	The IP address of the computer running the browser that sent the request.

Figure 27: Examples of environment variables passed to a CGI program.

State information

A server invokes a CGI program each time a request arrives for the associated URL. Furthermore, because a server does not maintain any history of requests, the server cannot tell the CGI program about previous requests from the same user. Nevertheless, a history is useful because it allows a CGI program to participate in a dialogue. For example, a history of previous interactions makes it possible to avoid having a user answer questions repeatedly when specifying additional requests. To provide a non-repetitive dialogue, a CGI program must save information between requests. Information that a program saves between invocations is called *state information*. In practice, dynamic document programs use two methods to store state information; the choice is determined by the length of time the information must be kept. The first method is used for information that must be kept across invocations of a browser. A dynamic document program uses long-term storage on the server computer (e.g., a file on disk) to store such information. The second method is used for information that must be kept while a browser is running. A dynamic document program encodes such information in URLs that appear in the document.

A CGI script with long-term state information

The script below illustrates how a server uses environment variables to pass arguments to a CGI program. Further, the script shows how a dynamic document program can store long-term state information in a file. The example script keeps a record of IP addresses of computers from which it has been contacted. When a request arrives, the script examines the browser's IP address and generates a document. The document tells whether the request is the first to arrive from the computer.

```
#!/bin/sh
FILE=ipaddrs

echo Content-type: text/plain
echo
# See if IP address of browser's computer appears in our file

if grep -s $REMOTE_ADDR $FILE > /dev/null 2>&1
then
        echo Computer $REMOTE_ADDR has requested this URL previously.
else
        # Append browser's address to the file
        echo $REMOTE_ADDR >>  $FILE
        echo This is the first contact from computer $REMOTE_ADDR
fi
```

The script above maintains a list of IP addresses in a local file named *ipaddrs*. When invoked, the script searches the file for the browser's IP address. If a match is found, the script reports that it has been contacted from the browser's computer previously. Otherwise, the script appends the browser's IP address to the file and reports that the contact was the first from the browser's computer. The script uses environment variable *REMOTE_ADDR* to obtain the IP address of the computer running the browser that issued the request. In the UNIX shell language, variables are referenced by placing a dollar sign in front of the variable name. The main body of the script consists of a single *if-then-else* statement. The statement uses the *grep* command to determine whether the browser's IP address appears in the file *ipaddrs*. The result of the search determines whether the script follows the *then* or *else* part of the *if* statement. If the address is found, the script emits a message that reports that the request is not the first. If the address is not present, the script appends the address to the file and then emits a message that acknowledges the first contact. When it is run the script generates three lines of output: two lines of header followed by a single line of text. The line of text contains the IP address of the browser's computer and tells whether a request has been received from the same computer previously. To sum it all up, we can say:

> *A dynamic document program uses files that reside on the server' computer to store long-term state information. Because such files persist between requests, information written in a file while handling a request will remain available when the program handles subsequent requests.*

A CGI script with short-term state information

The text beyond a question mark in a URL is passed to a dynamic document program as an argument. Because a dynamic document program creates a copy of a document on demand, each invocation of the program can produce a document that contains a new set of URLs. In particular, any text that the program places beyond a question mark in a URL will become an argument if the user selects the URL. The script below illustrates the concept.

```
#!/bin/sh

echo Content-type: text/html
echo

N=$QUERY_STRING
echo "<HTML>"

case "x$N" in

x)              N=1
                 echo "This is the initial page. <BR><BR>"
                 ;;
x[0-9]*) N=`expr $N + 1`
                 echo "You have displayed this page $N times. <BR><BR>"
```

```
                    ;;
*)                  echo "The URL you used is invalid. </HTML>"
                    exit 0
                    ;;
esac
echo "<A HREF=\http://$SERVER_NAME$SCRIPT_NAME?$N\">"
echo "Click here to refresh the page.</A></HTML>"
```

First, the script above emits an HTML document instead of plain text. Second, the script uses environment variable *QUERY_STRING* to determine whether the server passed an argument to the script. Third, the script concatenates the contents of variables *SERVER_NAME* and *SCRIPT_NAME* to obtain a URL for the script, and append a question mark and the count from variable N. Fourth, the script uses a *case* statement to choose among three possibilities: the argument string is empty, the argument script contains an integer or the argument string contains something else. In two cases, the string emits appropriate output. In the last case, the script reports that the URL is invalid.

Let us assume that the server *www.noexist.com* has been configured so the script corresponds to the path /cgi/ex4. When a user invokes URL: *http://www.nonexist/cgi/ex4*, the script generates six lines of output:

```
Content-type: text/html

<HTML>
This is the initial page.<BR><BR>
<A HREF=http://www.nonexist.com/cgi/ex4?1>
Click here to refresh the page.</A></HTML>
```

Because the header specifies that the document is HTML, a browser will interpret the HTML commands and display three lines of output:

This is the initial page.

Click here to refresh the page.

Although it does not appear on the user's screen, state information is embedded in a URL in the document. The state information consists of the last two characters, *?1*, and they encode information about the number of times the page has been refreshed. If the user clicks on the second sentence, the browser will request URL: *http://www.nonexist.com/cgi/ex4?1* When invoked, the script will find that QUERY_STRING contains 1. Thus the script will generate six lines of output:

```
Content-type: text/html

<HTML>
This is the initial page.<BR><BR>
<A HREF=http://www.nonexist.com/cgi/ex4?2>
Click here to refresh the page.</A></HTML>
```

When a browser displays the document, the screen changes to show the following:

You have displayed this page 2 times.

Click here to refresh the page.

The new document has argument *?2* embedded in the URL. Thus, if the user clicks the second sentence again, the browser will report that the page has been displayed 3 times. The dynamic document program does not know how many times the page has been refreshed - the program merely reports what the argument specifies. For example, if a user manually enters the URL:

http://www.nonexist.com/cgi/ex4?4023

The script will *incorrectly* report that the page has been displayed 4023 times! We can summarise:

> *When it generates a document, a dynamic document program can embed state information as arguments in URLs. The argument string is passed to the dynamic document program for the URL, enabling a program to pass state information from one invocation to the next.*

Forms and interaction

The notion of embedding state information in URLs has been generalized and extended. HTML allows a server to declare that a document is a *form* that contains one or more items that a user must supply. For each item, the form specifies an item name. When a user selects a hyperlink on the form, the browser appends an argument string to the URL that contains each of the named items. Thus, a dynamic program can obtain values that the user supplies. For example, suppose a form contains three items named AA, BB and CC, and suppose the user has supplied values *yes*, *no*, and *maybe*. When it collects the values into a URL, the browser creates a string:

> *?AA=yes,BB=no,CC=maybe*

which the browser appends to the specified URL. We can summarise:

> *An HTML form is a document that contains items that a user must supply. The information is encoded in a URL for transmission to another document.*

Copperbelt University
Computer Science Department

<u>Internet Technologies</u>

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

Java Technology for Active Web Documents

An early form of continuous update

Because information contained in a dynamic document is fixed when the document is created, a dynamic document can not update information on the screen as information changes and can not change graphic images to provide animation.

Two techniques have been invented to allow continuous update of a user's screen. The first technique places responsibility on the server. Called *server push*, the mechanism requires the server to periodically generate and send new copies of a document. In essence, the server must run the application program associated with a dynamic document continuously. As it runs, the program produces fresh output, which the server forwards to the browser for display. From the user's point of view, the contents of the page continue to change while being viewed.

The server push approach has two disadvantages: it causes excessive server overhead and introduces delay. When a user views a page that requires server push, the server must run the dynamic application program associated with the page. If many clients request documents which use server push, the server must run the application programs concurrently. Because the contents of a dynamic document can depend on the source of the request, a server must run a separate copy of a dynamic document application for each request. Delays arise from limits in available CPU and network bandwidth. When N application programs run concurrently on a time-sharing computer, each program receives at most 1/N of the available CPU power. Thus, the amount of CPU available for push applications decreases rapidly as the number of requests increases. If many browsers view pushed documents simultaneously, the CPU on the server's computer will become overloaded, and updates will be delayed.

Limited network bandwidth can also introduce delays. Server push requires each browser to maintain an active TCP connection over which the server sends updates continuously. Unfortunately, most computers that run servers have a single Internet connection over which all traffic must pass. If many applications attempt to send data, the network can become a bottleneck. To provide fairness, the operating system requires applications to take turns sending packets, as a result, an individual application is delayed waiting for access to the network.

Active documents and server overhead

Active documents, the second technique used to provide continuous update of information, arose as a way to avoid overloading servers. Instead of requiring a server to continuously compute updates for many clients, the active document approach moves computation to the browser. When a browser requests an active document, the server returns a computer program that the browser runs locally. Once it sends a copy of the document, the server has no further responsibility for execution or update - the browser handles all computation locally. Thus, unlike the server push approach, the active document approach does not require a server to use large amounts of CPU time. Furthermore, because it does not require the server to transmit continuous updates, an active document does not use as much network bandwidth as a pushed document.

Active document representation and translation

Because an active document is a program that specifies how to compute and display information, many representations are possible. Furthermore, no single representation is optimal for all purposes - various representations are needed. To achieve highest execution speed, computer hardware requires a program to be stored in binary form. Also a compressed representation minimises the delay and communication bandwidth required to transfer a program across the Internet. To satisfy all requirements, active document technologies use the same general approach to program representation as a conventional programming language: translation. In particular, most active document systems define multiple representations, where each representation is optimised for one purpose. Software is then created that can automatically translate among the document representations. Thus, a programmer can define and edit a document using one form, while a browser that runs the document can use another form. Figure 15 illustrates three popular active document representations, and shows the translation used to move from one to another.

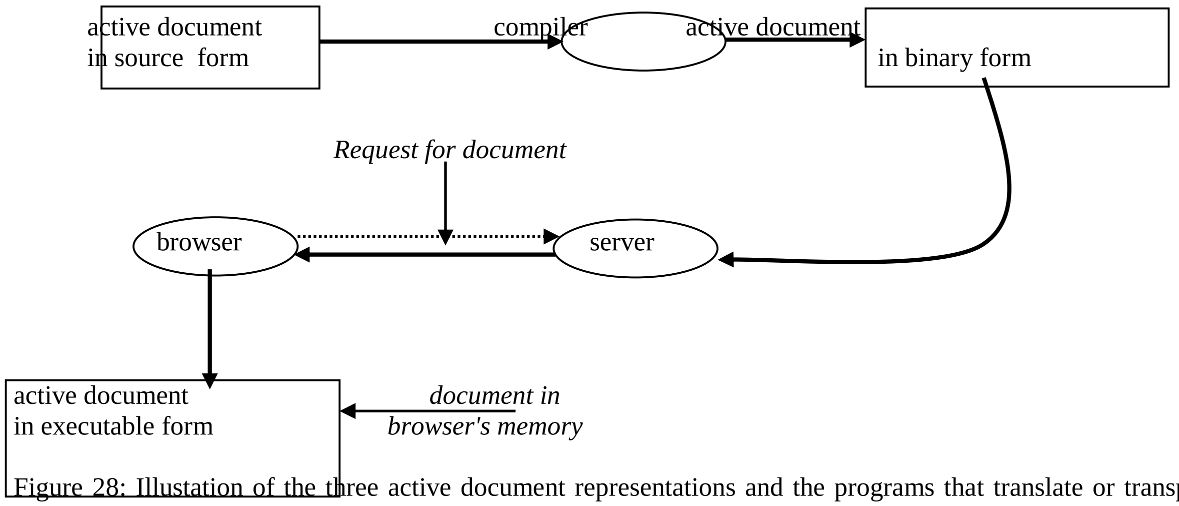


Figure 28: Illustation of the three active document representations and the programs that translate or transport the document. The darkened arrows show the direction a document moves.

As the figure shows, an active document begins in a *source representation*. The source representation contains declarations for local data and implementations of algorithms used to compute and display information. A *compiler* is used to translate a document from the source representation to a *binary representation* similar to the object representation used for the conventional program. In the binary representation, each executable statement is represented in binary form, and identifiers in the program are replaced by binary offsets. The binary representation includes a symbol table that records the relationships between names in the source representation and locations in the binary representation. After a browser obtains a copy of an active document in binary form, software in the browser translates the document to the executable representation, and loads the resulting program into the browser's memory. The browser must resolve the remaining references, and must link the program to library routines and system functions. Once linking is complete, the symbol table is no longer needed - the entire document has been translated.

Java technology

Devised by Sun Microsystems, Incorporated, *Java* is the name of a specific technology used to create and run active documents. Java uses the term *applet* to describe active document programs and to distinguish active documents from conventional computer programs. The Java technology includes three key components related to applets and these are the p*rogramming language, runtime environment and a class library*.

The Java Programming Language

When a programmer creates an active document, the programmer writes the program in a source language. The Java language can be characterised as:

High level. Like other high-level languages, Java is defined to hide hardware details and make it easy for a programmer to express computation.

General purpose. Although it was originally designed for writing applets, the language has sufficient expressive power to make it useful in other applications. There are also no limitations that prevent the Java language from being used for programs other than applets.

Similar to C++. Java descended from the C++ programming language. Thus both syntactic and semantic similarities exist between the two languages.
Object oriented. Java requires a programmer to define a set of objects that each contain data as well as methods that operate on the data.
Dynamic. An instance of an object is created dynamically at run-time. Furthermore, Java allows some bindings to be delayed until run-time.
Strongly typed. Java supports strong typing: each data item is declared to have a type, and the language prohibits operations from being applied to data other than the type for which the operation is intended.
Statically type checked. In Java, type checking does not occur at run-time. Instead, the type compatibility of all items in a program is determined when the program is compiled.
Concurrent. Java allows a program to have multiple threads of control. The threads execute concurrently (i.e., they appear to execute at the same time).

The Java run-time environment

Java defines a run-time environment in which Java programs execute. The run-time environment is characterised by:

Interpretative execution. Although a Java program can be compiled into code for a specific computer, the designers intended the language to be used with an interpreter. That is, unlike a conventional compiler that translates a source program into a binary object program for a specific architecture, a Java compiler translates a Java program into a machine-independent binary representation known as the Java *byte code representation*. A computer program called an *interpreter* reads the byte code representation and interprets the instructions.
Automatic garbage collection. Memory management in Java differs dramatically from memory management in languages like C and C++. Instead of requiring a program to call procedures like *malloc* and *free* to allocate and release memory, the Java run-time system provides automatic garbage collection. That is, a program can depend on the run-time system to find and reclaim memory that is no longer being used. In essence, when a program discards all references to an object, the garbage collector reclaims the memory allocated to that object.
Multithreaded execution. The Java run-time system provides support for concurrent thread execution. In essence, the run-time system contains parts of an operating system that handle scheduling and context switching. By switching the CPU among the set of currently active threads, the run-time system permits all threads to proceed. The run-time support for concurrent thread execution is sometimes called a *microkernel*.
Internet access. The Java run-time system includes a socket library that a program can use to access the Internet. In particular, a Java program can become a client - the Java program can use TCP or UDP to contact a remote server.
Graphic support. Because an applet needs to control the display, the Java run-time system includes basic mechanisms that allow a Java program to create windows on the screen that contain text or graphics.

Machine independence and portability

The Java language and run-time system are designed to make Java independent of computer hardware. For example, the language does not contain any implementation-defined features. Instead, the language reference manual defines the meaning of each operator and statement in the language unambiguously. In addition, the Java bytecode representation is independent of the underlying hardware. Consequently, once a Java program has been compiled into the bytecode representation, the program produces exactly the same output on any computer. Using a machine-independent binary representation and an interpreter makes Java programs portable across computer architectures. Keeping an applet independent of computer hardware is essential for the following reasons: First, in the Internet, users have many different computers. Machine independence allows a browser running on an arbitrary brand of computer to download and run a copy of the active document. Second, machine independence guarantees that the document will produce correct output on all browsers. Third, machine independence dramatically reduces document creation costs because a programmer can build and test one version of an active document instead of building one version for each type of computer.

The Java library

The third component of the Java technology is a library of class definitions. The library contains classes that span a variety of needs. For example, the library contains classes for:

Graphics manipulation. Graphics manipulation routines in the library allow a Java program to have precise control over the contents and appearance of the user's display screen (e.g., facilities in the library allow an applet to display text, graphics, images or dialog boxes).
Low-level network I/O. The library contains classes that provide access to the socket-level I/O facilities in the run-time system.
Interaction with a Web server. An applet may need to access static or dynamic Web documents or other applets. For example, an applet may follow a URL to retrieve and display a static document (e.g., an image). The library contains classes to handle such tasks.
Run-time system access. The Java library contains classes that an applet can use to access facilities in the run-time environment. For example, a running program can request the creation of a thread.
File I/O. An applet uses file I/O facilities to manipulate files on the local computer. File I/O is used by programs that save long-term state information.
Event capture. Events occur when a user depresses or releases a mouse button or key on the keyboard. The library contains classes that allow a Java program to capture and handle such events.
Exception handling. When an unexpected condition or an error occurs in a Java program, the run-time environment raises an *exception*. The Java library includes classes that make exception handling easier.

A graphics toolkit

The Java run-time system includes facilities that allow an applet to manipulate a user's display, and the Java library contains software that provides a high-level graphics interface. Together, the run-time graphics support and graphics library are known as a graphics *toolkit*. The specific graphics toolkit in Java is known as the *Abstract Window Toolkit (AWT)*. There are two reasons why Java needs a substantial graphics toolkit. First, the central purpose of an applet is complex display - an applet is used whenever a static display is inadequate. Second, a program that controls a graphics display must specify many details. For example, when a Java program needs to display a new item for the user, the program must choose between displaying the item in a sub-area of an existing window or creating a new, independent window. If a new window is created, the program must specify a heading for the window, the window size, the colours to use, and details such as where to place the window and whether the window has a scroll bar.

The Abstract Window Toolkit does not specify a particular style of graphics or level of interaction. Instead, the toolkit includes classes for both low-level and high-level manipulation. A programmer can choose whether to manage the details or invoke toolkit methods to do so. For example, the toolkit does not dictate the form of a window. On one hand, the toolkit includes low-level classes to create a blank rectangular panel on the screen, draw lines or polygons, or capture keystrokes and mouse events that occur in the panel. On the other hand, the toolkit includes high-level classes that provide a window complete with a heading, borders and vertical and horizontal scroll bars. A programmer can choose whether to use the window style provided or program all details. Similarly, a programmer must choose between high-level and low-level facilities for interaction with the user. For example, an applet can use high-level classes from the toolkit to create buttons, pull-down menus, or dialog boxes on the screen. Alternatively, an applet can create such items by using low-level toolkit classes to draw lines, specify shading, and control the front used to display text. Because an applet may need to interact with static documents, the toolkit includes classes that perform conventional Web browsing operations. For example, given a URL, an applet can use toolkit classes to fetch and display a static HTML document, fetch and display an image or fetch and play an audio clip.

Using Java graphics on a particular computer

Because the Abstract Window Toolkit is designed to be independent of a specific vendor's computer hardware and graphics system, Java does not need to use display hardware directly. Instead, Java can run on top of conventional window systems. For example, if a user runs a browser on a computer that uses the X Window System, a Java applet uses X to create and manipulate windows on the display. If another user runs a browser with a different window system, an applet will use that window system to create and manipulate the display. The Java applet is able to work with multiple window systems because of an important mapping built into the run-time environment and a set of intermediate functions. A Java run-time environment includes an intermediate layer of software. Each function in the intermediate layer uses the computer's window system to implement one specific Java graphics operation. Before an applet executes, the run-time environment establishes a mapping between each Java graphics method and the appropriate intermediate function. When an applet requests an AWT operation, control passes to a method in the Java library. The library method then forwards control to the appropriate intermediate function, which uses the computer's window system to provide the operation. The concept is known as a *factory*.

The advantages of using an intermediate layer of software are portability and generality. Because intermediate functions use the computer's window system, only one version of the AWT library is needed - the library does not contain code related to specific graphics hardware. Because the binding between Java graphics methods and the computer's window system software is established by the run-time system, a Java applet is portable. The chief disadvantage is the imprecision that arises because the intermediate layer of software can misinterpret operations. As a result, a Java program run on two different computers does not always produce identical results.

Java interpreters and browsers

A conventional browser contains an HTML interpreter that is used to display static or dynamic documents. A browser that runs Java contains two interpreters: an HTML interpreter and an additional interpreter for applets. A Java interpreter is a complex program. The heart of the interpreter is a simple loop that simulates the computer. The interpreter maintains an *instruction pointer* that is initialised to the beginning of an applet. At each interaction, the interpreter fetches the bytecode found at the address in the instruction pointer. The interpreter then decodes the instruction and performs the specified operation. For example, if the interpreter finds the *add* bytecode and two integer operands, the interpreter adds the two integers, updates the instruction pointer, and continues with the next operation. In addition to a basic instruction decoder, a Java interpreter must include support for the Java run-time environment. That is, a Java interpreter must be able to display graphics on the user's screen, access the Internet, and perform I/O. Furthermore, the interpreter must be designed to allow an applet to use facilities in the browser that retrieve and display static and dynamic documents. Thus, the Java interpreter in a browser must be able to communicate with the browser's HTTP client and HTML interpreter.

Compiling a Java program

A Java applet must be compiled and stored on a Web server before a browser can download and run the applet. The Java technology includes a compiler that is named *javac*. A programmer runs *javac* to compile a Java source program into the Java bytecode representation. Input to *javac* is a Java source program. The name of the input file must end with the suffix *.java*. *Javac* verifies that the source program is syntactically correct, translates the program into the bytecode representation, and places the output into a file with the suffix*.class*. A Java program consists of a sequence of declarations. An item that is declared public is exported to make it available to other applets. An item that is declared private is not exported and cannot be referenced externally. A source file must contain exactly one public class, which must use the same name as the source file prefix. That is, the public class in file *sussie.java* must be named *sussie*. Thus, a file that contains the compiled bytecode for class *sussie* has the name *sussie.class*. Consider the example below. Suppose that source file *aaa.java* contains the declarations:

```
public  class aaa{...}
        class bbb{...}
        class ccc{...}
```

The program declares class *aaa* to be public, while classes *bbb* and *ccc* are private. When *aaa.java* is compiled, *javac* produces output file *aaa.class*.

An example applet

The applet below begins with a sequence of import statements. Each import statement directs the computer to use a set of classes from the Java library. The example applet counts the number of times that a user clicks a button.

```
import java.applet.*;
importa java.awt.*;

public class clickcount extends Applet{
        int count;
        textField f;
public void init(){
        count = 0;
        add(new Button("Click Here"));
        f = new TextField("The button has not been clicked at all.");
        f.setEditable(false);
        add(f);
}

public boolean action(Event e, Object arg){
        if ((Button) e.target).getLabel() == "Click Here"){
                count += 1;
                f.setText("The button has been clicked " + count + " times.");
                }
                return true;
        }
}
```

An import statement contains a pattern that specifies a set of classes to be imported. In the pattern, most characters represent a literal match; an asterisk is used to denote a wildcard that matches an arbitrary string. Thus, the statement:

import java.applet.;*

specifies that the compiler should import all classes with names that begin with the string *java.applet*. In the example applet, the two import statements are used to import the standard *Applet* and *Abstract Window Toolkit* classes. As a result, the applet will have access to methods that create the applet environment and methods that manipulate the display.

The example applet defines a subclass of type *Applet* and named *clickcount*. In addition to the methods that the class inherits from *Applet*, class *clickcount* redefines two methods: *init* and *action*. When a browser invokes a reference to the applet, Java creates a new instance of the *clickcount* object. *Clickcount* defines two data items: an integer named *count* and a *TextField* named *f*. As part of the creation, the run-time system invokes the *init* method. *Init* performs several initialisation steps. After setting *count* to zero, *init* uses the *new* operation to create an instance of a *Button* object, and calls *add* to add the button to the applet's window. Finally, *init* uses *new* to create an instance of a *TextField* and assign it to variable *f*, method *setEditable* to make the text in *f* read-only, and *add* to associate *f* with the applet's window.

The heart of the example is method *action*. If the button labelled *Click Here* is selected, the applet increments variable *count* and changes the display. To do so, the applet invokes method *setText*, which replaces the message in f. Thus, each time a user clicks on the button, the display changes. From a user's point of view, the example applet appears to operate similar to the CGI script we looked at earlier. From an implementation's point of view, the two operate quite differently. Unlike a CGI script, an applet maintains state information locally. Thus, unlike the example CGI script, the example applet does not need to contact a server before updating the screen.

Invoking an applet

Two methods can be used to invoke an applet. First a user can supply the URL of an applet to a browser that understands Java. When the browser contacts the server specified in the URL, the server will inform the browser that the document is a Java applet. Second, an HTML document can contain an *applet tag* that refers to an applet. When a browser encounter the tag, the browser contacts the server to obtain a copy of the applet. In its simplest form, an applet tag specifies the location of a *.class* file to be fetched and executed. For example, suppose the Web server on machine *www.nonexist.ac* stores file *bbb.class* in directory *example*. The URL for the applet is:

http://www.nonexist.ac/example/bbb.class

The applet tag that specifies the location contains two items: a codebase that specifies the machine and path, and a code that gives the name of the class file. For example, the applet tag: <applet codebase = "www.nonexist.ac/example" code = "*bbb.class*"> contains the same information as the URL: *http://www.nonexist.ac/example/bbb.class*. A browser that encounters the tag will contact the server, obtain a copy of the class file, create an instance of the class *bbb* and call its *init* method.

Example of interaction with a browser

An applet can interact with both the HTTP client and the HTML interpreter in a browser. An applet uses the browser's HTTP client to retrieve documents and the browser's HTML interpreter to display pages of information. The example applet below interacts with the HTTP and HTML facilities in a browser.

```
import java.applet.*;
import java.net.*;
import java.awt.*;

public class buttons extends Applet{
        public void init(){
                add(new Button("Yin"));
                add(new Button("Yang"));
}
public boolean action(Event e, Object arg){
                if (((Button) e.target).getLabel() == "Yin"){
                        try{
                                getAppletContext().showDocument(new
                                URL("http://www.nonexist.com/yin"));
                        }
                        catch(Exception ex){
                        // note: code to handle the exception goes here
                        }
                }
                else if (((Button) e.target).getLabel() == "Yang"){
                        try{
                                getAppletContext().showDocument(new
                                URL("http://www.nonexist.com/yang");
                        }
                        catch(Exception ex){
                        // note: code to handle the exception goes here
                        }
                }
                return true;
}
}
```

As the example shows, Java provides high-level library facilities that simplify interaction between an applet and the browser. To use the facilities, the applet begins with three *import* statements. In addition to the applet and Abstract Window Toolkit definitions, the applet imports *java.net.** which contains definitions of classes related to the network. When a browser starts the applet, the browser creates an instance of the *buttons* class, and then invokes the *init* method. *Init* creates two *Button* objects and uses the *add* method to add them to the applet's display. One button is labelled *Yin*, while the other is labelled *Yang*. Code that interacts with the browser can be found in the *action* method. When the user clicks the button labelled *Yin*, the *action* method performs three tasks. First, the *if* statement tests the label on the button to determine which button has been clicked. Second, the applet retrieves the appropriate document. For example, if the user clicks the button labelled *Yin*, the applet invokes: *new URL("http://www.nonexist.com/yin")* to request that the browser's HTTP client retrieves a copy of the document associated with URL http://www.nonexist.com/yin. The *new* operation creates a local object to hold the document. Third, the Java code: *getAppletContext().showDocument* requests the HTML interpreter to display the document on the user's screen. The browser will replace its current display with the new document. The user must click the browser's *back* button to return to the applet.

Errors and exception handling

In Java all errors produce exceptions. An applet is required to handle such exceptions explicitly. The previous example uses the URL object to request that the HTTP client fetches a document. If the URL is incorrect, the network is down, or the server is unavailable, the invocation will result in an exception. Our example applet uses the try-catch construct to handle exceptions. The syntactic form: *try {S} catch (E) {H}* means: if exception E occurs while executing S then invoke handler H. In our example code, E is a variable ex declared to be of type Exception, and the handler is not specified, which means that all exceptions are to be ignored. Thus, the example simply ignores errors and forces the user to click a button to retry the operation.

Alternatives and variations

Several alternatives exist to the Java technology. For example, a variation called *JavaScript* is used for small applets that do not contain large or complex code. Instead of compiling an applet into the bytecode representation, JavaScript provides a scripting language and arranges for the browser to read and interpret a script in source form. JavaScript can be integrated with HTML - an HTML script can contain a JavaScript function that provides simple interaction with the user. For example, a JavaScript function can request a user to enter information and then verify that the information is in acceptable form for communicating with the server. A JavaScript function can also perform an action such as playing an audio file.

The chief advantages of the JavaScript are simplicity and ease of use. A small script can be embedded in a page. Because a script does not need to be compiled, the same tools can be used to create a script as an ordinary Web page. The chief disadvantages are speed and scalability - because the source representation is less compact than a bytecode representation, transporting a source program takes longer. Furthermore, a script takes longer to interpret than a program that has been translated into a bytecode representation.

Another variation allows programmers to write applets in a language other than the Java Programming Language. For example, the *AdaMagic* system consists of an *Ada* compiler that has been adapted to produce the Java bytecode representation. The compiler accepts programs written in the Ada Programming Language, and produces bytecode output compatible with the Java run-time environment. Thus, programmers who use the modified compiler can write applets in Ada.

Other companies have also begun to explore alternative technologies. For example, Lucent Bell Laboratories has developed an active document technology called *Inferno*. Although Inferno is conceptually similar to Java, the details differ dramatically.

Copperbelt University
Computer Science Department

<u>Internet Technologies</u>

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

Remote Procedure Call (RPC) and Middleware

Programming clients and servers
Programmers who build clients and servers must deal with the complex issues of communication. Although many of the needed functions are supplied by standard API such as the socket interface, programmers still face a challenge. The socket calls require the programmer to specify many low-level details such as names, addresses, protocols and ports. Small errors in the code that can easily go undetected when the program is tested can cause major problems when the program runs in production. Programmers who write client and servers realise that multiple programs often follow the same general structure and repeat many of the same details. That is, most client-server software systems follow a few basic architectural patterns. Furthermore, because implementations tend to use the same standard API (e.g., the socket interface), much of the detailed code found in one program is replicated in others. For example, all client programs that use a connection-oriented transport must create a socket, specify the server's endpoint address, open a connection to the server, send requests, receive responses and close the connection when interaction is complete. To avoid writing the same code repeatedly and to produce code that is both correct and efficient, programmers use automated tools to construct clients and servers. A *tool* is software that generates all or part of a computer program. The tool accepts a high-level description of the service to be offered, and produces much of the needed code automatically. The tool can not eliminate all programming - a programmer must supply code that performs the computation for the particular service. However, a tool can handle the communication details. As a result, the code contains fewer bugs.

The remote procedure call paradigm
One of the earliest facilities that was created to help programmers write client-server software is known generally as a Remote Procedure Call (RPC) mechanism. The idea of RPC arises from the observation that most programmers are familiar with procedure calls as an architectural construct. When programmers build a large program, they begin by dividing the program into major pieces. The most widely available programming language feature used for such divisions is the *procedure*. A programmer begins by associating a procedure with each major piece of the design. When implementing a program, the programmer uses procedures to keep the code manageable. Instead of defining a single, large procedure that performs many tasks, the programmer divides the tasks into sets and uses shorter procedures to handle each set. Thus the overall form of a program consists of a hierarchy of procedure calls. The procedural hierarchy of a program can be represented with a directed graph in which each node represents a procedure and an edge from node X to node Y means that procedure X contains a call to procedure Y. The graphical representation is known as a *procedure call graph*.

We can think of a main program that calls three procedures, A, B and C. Procedure A calls procedure D and procedure C calls two other procedures, E and F. To make procedures general and enable a given procedure to solve a set of related tasks, each procedure is parameterised. The procedure definition is given with a set of formal parameters. When the procedure is invoked, the caller supplies actual arguments that correspond to the formal parameters. Because parameterised procedures have worked well as a programming abstraction, researchers have examined ways to use a procedural abstraction to build clients and servers. Researchers have investigated ways to make client-server programming as much like conventional programming as possible. Unfortunately, most programming languages are designed to produce code for a single computer. Neither the language nor the compiler are designed for a distributed environment - the code that is generated restricts the flow of control and parameter passing to a single address space. Some researchers have investigated language modifications that embed client-server facilities in the language. For example, Modula-3 and Java Remote Invocation (Java RMI) both have facilities that allow a program to invoke procedures on other computers. Also, both have facilities that synchronise the computation from separate machines to ensure consistent results. In other words:

> *Because conventional programming languages do not allow procedure calls to pass from a program on one computer across a network to a program on another, tools are used to help programmers build client-server software using the procedure call abstraction.*

When using RPC, a programmer's attention is not focused on the computer network or communication protocols. Instead, the programmer is encouraged to think about the problem being solved. The programmer designs and builds the program using procedure calls as the basic architectural feature. Once a programmer has built a conventional program to solve the problem at hand, the programmer then considers how to divide the program into two pieces. In essence, the programmer must partition the call graph into two parts. The part that contains the main program and some of the procedures it calls becomes the client, while the remaining procedures become the server. When selecting a set of procedures that comprise the client and a set that comprise the server, a programmer must consider data - the global data that each procedure references must be located on the same computer as the procedure.

RPC extends the procedure call mechanism to allow a procedure in the client to call a procedure across the network in the server. That is, when such a procedure call occurs, the thread of control appears to pass across the network from the client to the server. When the called procedure returns, the thread of control appears to pass back from the server to the client. After deciding which procedures to place in the server and which to place in the client, the programmer is ready to use an RPC tool. To do so, the programmer creates a specification that describes the set of procedures that will be *remote* (i.e., the set of procedures that form the server). For each remote procedure, the programmer must specify the types of its parameters. The tool creates software that handles the necessary communication.

Communication stubs

A thread of control cannot jump from a program on one computer to a procedure on another. Instead, it uses client-server interaction. When a client calls a remote procedure, it uses conventional protocols to send a request message across the network to the server. The request identifies a procedure to invoke. After sending a request, the process on the client side blocks to await a response. When the remote program (i.e., the server) receives a request from a client, it invokes the specified procedure and then sends the result back to the client. Extra software must be added to each side of the program to implement the interaction. The extra software of the client side handles the details of sending a message across the network and waiting for a response. The extra software on the server side handles the details of receiving the incoming message, calling the specified procedure and sending a response.

Technically, each piece of added software is known as a *communication stub* or *proxy*. The two stubs, one in the client and one in the server, handle all communication details. In addition, the stubs are constructed to fit into the existing program; the rest of the program simply uses procedure calls as if all procedure calls were local. If a call is made to a procedure that is not local, a communication stub intercepts the procedure call, gathers values for the arguments (known as *argument marshalling*) and sends a message across the network to the communication stub on the server. The communication stub on the server side uses the conventional procedure call mechanism to invoke the specified procedure and then sends the results to the client stub. When the client stub receives the response, it returns the results to its caller exactly as if a local procedure was returning. Figure 16 shows how communication stubs are added to the clients and server sides of a program.

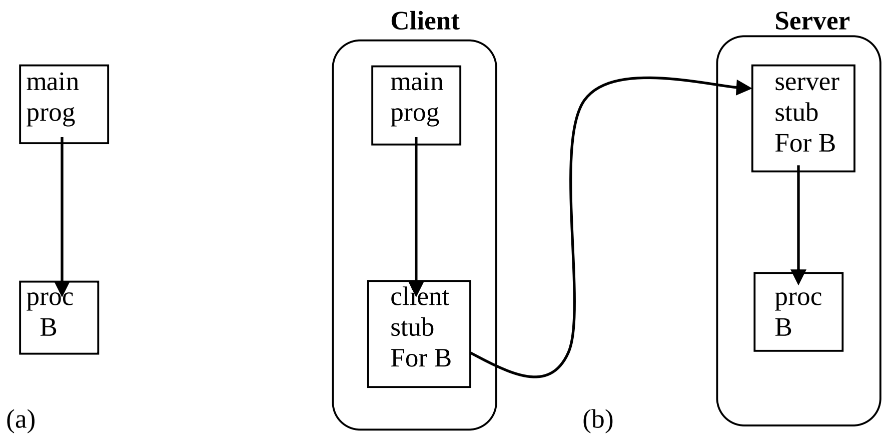


Figure 29: (a) The original call in a conventional program and (b) the remote version of the same call implemented with communication stubs.

Figure 53 (a) shows the procedure call in the original program before RPC stubs have been added. When the *main* procedure calls procedure *B*, the arguments it passes must agree exactly with the formal parameters of B. That is, the call must contain the correct number of arguments and the type of each argument must match the type declared for the formal parameters. Figure 53 (b) shows the communication stubs that must be added to the program when it is divided into client and server pieces. It is important to understand that the procedural interfaces in part b use the same number of and type of arguments as the original interface in (a). Thus, the call from *main* to the client stub and the call from the server stub to procedure B use exactly the same interface as the conventional call from *main* to procedure B. More important, client stubs can be given the same name as the procedure they replace. As a result, code from the original program need not be changed. The replacement that takes place here is actually the origin of the term *proxy* - each client stub acts as an exact replacement for one of the procedures in the original program.

External data representation

Computer systems do not all use the same internal data representation. For example, some computers store the most significant byte of an integer at the lowest address, and others store the least significant byte at the lowest address. Thus, when clients and servers send integer values between heterogeneous computers, they must agree on details such as whether the client or server will convert and which representation will be sent across the network. The term *external data representation* refers to the form that data is sent across the network. In addition to defining an interface language, a remote procedure call technology defines an external data representation. Some systems negotiate a representation - the client and server stubs exchange messages to determine which representation each will use and which needs to translate. Many systems choose another approach in which the external representation is fixed. In such cases, the sender always converts from the local representation to external representation and the receiver always converts from the external to the local representation.

Middleware and object-oriented middleware

To review, a tool can help a programmer to build a program that uses remote procedure call. The programmer specifies a set of procedures that will be remote by giving the interface details (i.e., the number and type of arguments). To do so, the programmer uses the tool's *Interface Definition Language (IDL*), also called *Interface Description Language*. The tool reads the IDL specification and generates the necessary client and server stubs. The programmer then compiles and links two separate programs. The server stubs are combined with the remote procedures to form the server program. The client stubs are combined with the main program and local procedures to form the client program. A variety of commercial tools have been developed that use the paradigm described above to help programmers construct client-server software. Such tools are described as *middleware* because they provide software that fits *between* a conventional application program and the network software.

In the 1990s, programming language research shifted focus from the procedure paradigm to the object-oriented paradigm. As a result, most new programming languages *are object-oriented languages.* The basic structuring facility in an object-oriented language is called an *object*, which consists of data items plus a set of operations for those data items, which are known as *methods*. The basic control mechanism in an object-oriented language is *method invocation*. In response to the change in languages, designers are creating new middleware systems that extend method invocation across computers in the same way that remote procedure call extended procedure call. Such systems are known as distributed object systems.

Some of the widely used middleware and object-oriented middleware technologies include *Open Network Computing Remote Procedure Call (ONC RPC)* from Sun Microsystems, Incorporated, *Distributed Computing Environment Remote Procedure Call (DCE RPC)* from Open Software Foundation (now called The Open Group), *Microsoft Remote Proedure Call (MSRPC) from Microsoft Corporation,* and also *MSRPC2* which is sometimes called *Object RPC (ORPC),* still from Microsoft. In 1994 Microsoft defined the *Component Object Model (COM),* which is an object-oriented software architecture. In 1998 Microsoft defined the *Distributed Component Object Model (DCOM),* which extends COM by creating an application-level protocol for object-oriented remote procedure calls. The combination of COM and DCOM is referred to as COM+.

Perhaps the best-known object-oriented middleware is named *Common Object Request Broker Architecture (CORBA)*. CORBA permits an entire object to be placed on a server, and extends method invocation using the same general approach as described earlier. One difference arises because proxies are instantiated at run-time like other objects. When a program receives a reference to a remote object, a local proxy is created that corresponds to the object. When the program invokes a method on the object, control passes to the local proxy. The proxy then sends a message across the network to the server, which invokes the specified method and returns the results. Thus, CORBA makes method invocation for remote and local objects appear identical. In addition to focusing on objects instead of procedures, CORBA differs from conventional RPC technologies because it is more dynamic. In conventional RPC technologies, the programmer uses a tool to create stub procedures when constructing the program. In CORBA, the software creates a proxy at run-time when it is needed (i.e., when a method is invoked on a remote object for which no proxy exists).

Copperbelt University
Computer Science Department

<u>Internet Technologies</u>

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

Network Management

Our society has come to depend on networks so much that the malfunctioning of an organisation's computer network can mean disruption of business and daily lives, resulting in, for example, frustrated users or customers, delays in receipt of critical data, possible loss of business revenue and image. Therefore the area of network management is critical in keeping computer networks in good working order and so reducing disruptions of business operations.

Network management is the process of supervising and controlling a computer network so as to maximise its efficiency and productivity. Network management includes collecting data for use in operating the network, analysing the data in order to detect any problems with the network and providing solutions for handling the problems. To make administration work easier, network management systems also produce various kinds of reports. To accomplish all this, network management consists of five functional areas:

- fault management,
- configuration management,
- security management,
- performance management and
- accounting management.

Fault Management

Fault management is the process of locating problems, or faults, on a computer network by discovering, isolating and reporting the problems. Where possible, a fault management system can fix a discovered problem automatically. Fault management increases network reliability by offering a variety of tools which can provide the necessary information about a network's current state, detect problems and help initiate recovery procedures. This increases the effectiveness and productivity of the network.

Configuration Management

The configuration of certain network devices controls the behaviour of a network. Configuration management is the process of finding and setting up (configuring) these critical devices. Configuration management provides tools that help keep track of an inventory of:

- what devices are being managed,
- what communications hardware and software these devices are using,
- their licences,
- their locations and
- how they are connected to the network.

Configuration management goes hand-in-hand with fault management because faulty conditions are often caused by misconfigurations. As a result, accurate and up-to-date configuration information can help to isolate and correct faulty situations.

Security Management

Security management is the process of controlling access to information on a computer network. Some information stored by computers attached to networks may be inappropriate for all users to view. Such sensitive information may include, for example, details about an organisation's new products, its customer base or even hospital patient data. Security management gives a way to monitor the access points on a critical device such as a server, for example, and record who is using the device. Security management also provides audit trails and alarms to alert the network administrator of potential security breaches. It also strives to provide confidentiality and integrity of data that resides on network devices and that which traverses networks. Network management also ensures availability of network resources to legitimate users of the network.

Performance Management

Performance management involves measuring the performance of network hardware, software, and media. Activities measured may be, for example,

- overall throughput,
- percentage utilisation,
- error rates or
- response times

This information can assist the network administrator to ensure that the network has enough capacity to accommodate the needs of users. Performance management tools can help the network administrator detect early, for instance, that a network link is nearing capacity, even before performance is impacted. This can help to reduce network overcrowding and inaccessibility and so providing a consistent level of service to users. Performance management also can aid capacity planning and network trend analysis.

Accounting Management

Accounting management involves tracking each individual and group users' utilisation of network resources so that the network administrator can better ensure that users are provided the quantity of resources they need. It also involves granting or removing permission for access to a network. In some cases, users are charged for the use of network resources.

Managing an internet

A *network manager* is a person responsible for monitoring and controlling the hardware and software systems that comprise an internet. A manager works to detect and correct problems that make communication inefficient or impossible and to eliminate conditions that will produce the problem again. Because either hardware or software failures can cause problems, a network manager must monitor both. Management of a network can be difficult for two reasons. First, most internets are heterogeneous. That is, the internet contains hardware and software components manufactured by multiple companies. Small mistakes by one vendor can make components incompatible. Second, most internets are large. In particular, the global Internet spans many sites in countries around the world. Detecting the cause of a communication problem can be especially difficult if the problem occurs between computers at two sites. However, failures that cause the most severe problems are often the simplest to diagnose. For example, if the coaxial cable in an Ethernet is cut or a LAN switch loses power, computers connected to the LAN will be unable to send or receive packets. Because the damage affects all the computers on the network, a manager can locate the source of the failure quickly and reliably. Similarly, a manager can also detect catastrophic failures in software such as an invalid route that causes a router to discard all packets for a given destination.

In contrast, intermittent or partial failures often produce a difficult challenge. For example, imagine a network interface device that corrupts bits infrequently, or a router that misroutes a few packets and then routes many correctly. In the case of an intermittent interface failure, the checksum or CRC in a frame may be sufficient to hide the problem - the receiver discards the corrupted frame and the sender must eventually retransmit it. Thus, from a user's perspective, the network appears to operate correctly because data eventually passes through the system. In other words, *Because protocols accommodate packet loss, intermittent hardware or software failures can be difficult to detect and isolate.*

The danger of hidden failures

Although they remain hidden, intermittent failures can affect network performance. Each retransmission uses network bandwidth that could be used to send new data. Worse still, hardware failures often become worse over time because the hardware deteriorates. As the frequency of errors increases, more packets must be retransmitted. Because retransmissions lower throughput and increase delay, overall network performance decreases. In a shared network, a failure associated with one computer can affect others. For example, suppose the interface begins to fail on a computer attached to an Ethernet. Because the failing hardware corrupts some of the packets the computer sends, the computer will eventually retransmit them. While the network is being used for a retransmission, other computers must wait to send packets. Thus, each retransmission reduces the bandwidth for all computers.

Network management software

Network management software allows a manager to monitor and control network components and so enable the network manager to find problems and isolate them. For example, network management software allows a manager to interrogate devices such as host computers, routers, switches and bridges to determine their status and to obtain statistics about the networks to which they attach. The software also allows a manager to control such devices by changing routes and configuring network interfaces.

Clients, servers, managers and agents

Network management is not defined as part of the transport or internet protocols. Instead, the protocols that a network manager uses to monitor and control network devices operate at the application level. That is, when a manager needs to interact with a specific hardware device, the management software follows the conventional client-server mode: an application program on the manager's computer acts as a client, and an application program on the manager's device acts as a server. The client on the manager's computer uses conventional transport protocols (e.g., TCP or UDP) to establish communication with the server. The two then exchange requests and responses according to the management protocol. To avoid confusion between application programs that users invoke and application programs that are reserved for network managers, network management systems avoid the terms *client* and *server*. Instead, the client application that runs on the manager's computer is called a *manager* and an application that runs on a network device is called an *agent*.

It may seem odd that conventional networks and conventional transport protocols are used for network management. After all, failures in either the protocol or underlying hardware can prevent packets from travelling to or from a device, making it impossible to control a device while failures are occurring. In practice, however, using an application protocol for network management works well for two reasons. First, in cases where a hardware failure prevents communication, the level of the protocol does not matter - a manager can communicate with those devices that remain operating and use success or failure to help locate the problem. Second, using conventional transport protocols means that a manager's packets will be subject to the same conditions as normal traffic. Thus, if delays are high, a manager will find out immediately.

The Simple Network Management Protocol (SNMP)

The standard protocol used to manage an internet is known as *the Simple Network Management protocol (SNMP)*. The SNMP protocol defines exactly how a manager communicates with an agent. For example, SNMP defines the format of requests that a manager sends to an agent and the format of replies that an agent returns. In addition, SNMP defines the exact meaning of each possible request and reply. In particular, SNMP specifies that an SNMP message is encoded using a standard known *as Abstract Syntax Notation.1 (ASN.1).* Consider sending an integer between an agent and a manager. To accommodate large values without wasting space on every transfer, ASN.1 uses a combination of length and value for each object being transferred. For example, an integer between 0 and 255 can be transferred in a single octet. Integers in the range 256 through 65535 require two octets, while larger integers require three or more octets. To encode an integer, ASN.1 sends a pair of values: a length, L, followed by L octets that contain the integer.

To permit encoding of arbitrarily large integers, ASN.1 also allows the length to occupy more than one octet. Extended lengths normally are not needed for the integers used with SNMP. The table in figure 17 illustrates the encoding.

decimal integer	**hexadecimal equivalent**	**length octet**	**octets of value (in hex)**
27	1B	01	1B
792	318	02	03 18
24,567	5FF7	02	5F F7
190,345	2E789	03	02 E7 89

Figure 30: An example of ASN.1 encoding for integers.

Fetch-store paradigm

The SNMP protocol does not define a large set of commands. Instead, the protocol uses a *fetch-store paradigm* in which there are two basic operations: *fetch*, used to obtain a value from a device and *store*, used to set a value in a device. Each object that can be fetched or stored is given a unique name; a command that specifies a fetch or store operation must specify the name of the object. A set of status objects must be defined and given names. To obtain status information, a manager fetches the value associated with a given object. For example, an object can be defined that counts the number of frames a device discards because the frame checksum is incorrect. The software must be programmed to increment the counter whenever a checksum error is detected. A manager can use SNMP to fetch the value associated with the counter to determine whether checksum errors are occurring.

Using the fetch-store paradigm to control a device may not seem obvious. Control operations are defined to be the side-effect of storing into an object. For example, SNMP does not include separate commands to *reset* the checksum error counter or to *reboot* the device. In the case of the checksum error counter, storing a zero into the object is intuitive because it resets a counter to zero. For operations like reboot, however, an SNMP agent must be programmed to interpret store requests and to execute the correct sequence of operations to achieve the desired effect. Thus, SNMP might define a reboot object, and specify that storing zero into the object should cause the system to reboot. In practice, however, most systems do not have a reboot counter - software in the agent must explicitly check for a store operation that specifies the reboot object, and must then execute the sequence of steps needed to reboot the system.

The MIB and object names

Each object to which SNMP has access must be defined and given a unique name. Furthermore, both the manager and agent programs must agree on the names and the meanings of fetch and store operations. Collectively, the set of all objects that SNMP can access is known as a *Management Information Base (MIB).* Objects in a MIB are defined with the ASN.1 naming scheme, which assigns each object a long prefix that guarantees the name will be unique. For example, an integer that counts the number of IP datagrams a device has received is named:
iso.org.dod.internet.mgmt.mib.ip.ipInReceives

Furthermore, when the object name is represented in an SNMP message, each part of the name is assigned an integer. Thus, in an SNMP message, the name of *ipInReceives* is:
1.3.6.1.2.1.4.3

The variety of MIB variables

Because SNMP does not specify a set of MIB variables, the design is flexible. New MIB variables can be defined and standardised as needed, without changing the basic protocol. More important, the separation of the communication protocol from the definition of objects permits groups of people to define MIB variables as needed. For example, when a new protocol is designed, the group that creates the protocol can also define MIB variables that are used to monitor and control the protocol software. Similarly, when a group creates a new hardware device, the group can specify MIB variables used to monitor and control the device. Many sets of MIB variables have been created. For example, there are MIB variables that correspond to protocols like UDP, TCP, IP and ARP as well as MIB variables for network hardware such as Ethernet, Token Ring and FDDI. In addition, groups have defined MIBs for hardware devices such as bridges, switches and printers.

MIB variables that correspond to arrays

In addition to simple variables such as integers that correspond to counters, a MIB can include variables that correspond to tables or arrays. Such definitions are useful because they correspond to the implementation of information in a computer system. For example, consider an IP routing table. In most implementations, the routing table can be viewed as an array of entries, where each entry contains a destination address and a next-hop used to reach that address. Unlike a conventional programming language, ASN.1 does not include an index operation. Instead, indexed references are implicit - the sender must know that the object being referenced is a table, and must append the indexing information on to the object name. For example, the MIB variable: *standard MIB prefix.ip.ipRouting Table*
Corresponds to the IP routing table, each entry of which contains several fields. Conceptually, the table is indexed by the IP address of a destination. To obtain the value of a particular field in an entry, a manager specifies a name of the form:

standard MIB prefix.ip.ipRouting Table.ipRouteEntry.field.Ipdestaddr

where *field* corresponds to one of the valid fields of an entry and *IPdestaddr* is a 4-octet IP address that is used as an index. For example, field *ipRouteNextHop* corresponds to the next-hop in an entry. When converted to the integer representation, the request for a next-hop becomes: *1.3.6.1.2.1.4.21.1.7.destination*

where *1.3.6.1.2.1* is the standard MIB prefix, 4 is the code for *ip*, 21 is the code for *ipRoutingTable*, 1 is the code for *ipRouteEntry*, 7 is the code for the field *ipRouteNextHop*, and *destination* is the numeric value for the IP address of a destination.

Copperbelt University
Computer Science Department

<u>Internet Technologies</u>

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

Network Security

In the wake of increased network connectivity, computer systems are becoming increasingly vulnerable to attacks. These attacks are aimed at compromising networks and the sensitive and confidential information that they contain. This scenario underscores the importance of network security. Network security sets out to protect information traversing networks and that, which is contained in network devices from attackers by preventing the theft, destruction, corruption, and introduction of unwanted pieces of information. Only authenticated users are granted access to the network and network resources. Integrity and confidentiality, as they apply to messages that transit networks, ensure that the information which arrives is identical to the one which was sent, and that only the intended recipients can access it and so affording legitimate users an uninterruptible availability of network resources.

Secure networks and policies

Although the concept of a secure network is appealing to most users, networks can not be classified simply as secure or not secure because the term is not absolute. Each organisation defines the level of access that is permitted or denied. For example, some organisations store data that is valuable. Such organisations define a secure network to be a system that prevents outsiders from accessing the organisation's computers. Other organisations need to make information available to outsiders, but prohibit outsiders from changing the data. Such organisations may define a secure network as one that allows arbitrary access to data, but includes mechanisms that prevent unauthorised changes. Still other groups focus on keeping communication private. They define a secure network as one in which no one other than the intended recipient can intercept and read a message. Finally, many organisations need a complex definition of security that allows access to selected data or services the organisation chooses to make public, while preventing access or modification to sensitive data and services that are kept private.

Because no absolute definition of *secure network* exists, the first step an organisation must take to achieve a secure system is to define the organisation's *security policy*. The policy does not specify how to achieve protection. Instead, it states clearly and unambiguously the items that are to be protected. Defining a network security policy is complex. The primary complexity arises because a network security policy cannot be separated from the security policy for computer systems attached to the network. In particular, defining a policy for data that traverses a network does not guarantee that data will be secure. For example consider data stored in a file that is readable. Network security cannot prevent unauthorised users who have accounts on the computer from obtaining a copy of the data. Thus, to be effective, a security policy must apply all the times. The policy must hold for the data stored on disk, data communicated over a telephone line with a dialup modem, information printed on paper, data transported on portable media such as a floppy disk and data communicated over a computer network.

Assessing the costs and benefits of various security policies also adds complexity. In particular, a security policy cannot be defined unless an organisation understands the value of its information. In many cases, the value of information is difficult to assess. Consider, for example, a simple payroll database that contains a record for each employee, the hours the employee worked and the rate of pay. The easiest aspect of value to assess is the replacement cost. That is, one can compute the man-hour required to recreate or verify the contents of the database (e.g., by restoring the data from an archive or by performing the work needed to collect the information). A second aspect of value arises from the liability an organisation can incur if the information is incorrect. For example, if an unauthorised person increases the pay rate in a payroll database, the company could incur arbitrary costs because employees would be overpaid. A third aspect of value arises from the indirect costs that can be incurred from security violations. If payroll information becomes public, competitors might choose to hire workers, which results in costs for hiring and training replacements as well as increased salaries needed to retain other employees.

Defining a security policy is also complicated because each organisation must decide which aspects of protection are most important, and often must compromise between security and ease of use. For example, an organisation can consider several facets of security:

Facets of Network Security

The primary facets of network security are:

- authentication,
- access control,
- integrity,
- confidentiality,
- non-repudiation
- auditing and
- availability.

An elaboration of each of these items follows below.

Authentication

Authentication is establishing proof of identity. Usually this involves one or a combination of

(a) something the user is,
(b) something the user knows and
(c) something the user has.

What one knows could be an account name and a password. What one has could be a hardware authentication device (e.g. smart cards or token cards which provide one-time passwords). Authentication differentiates one entity or user from another. Without authentication, all users would be treated equally, or each blindly trusted to his own unsupported claim of identity.

Access control

Access control relates to who (or what) may have access to some network resource. In other words, access control strives to regulate access to the network and its resources. Implementing access control based on identity requires some form of authentication. With an accurate authentication mechanism access controls can be applied to provide authorisation, access rights, and privileges.

Integrity

Integrity refers to protection from change: is the data that arrives exactly the same as the data that was sent? So, integrity refers to the current condition of some data as compared to their pure and original state. A message or file that traverses a network is at risk of having data added, removed, or modified along the way. A message that undergoes this experience loses its integrity. To ensure integrity, information should be protected from being deleted or altered in any way without the permission of the owner of that information.

Confidentiality and Privacy

Confidentiality and privacy refer to protection against snooping or wiretapping: So the question to ask here is - is data protected against unauthorised access? Confidentiality is protecting information from being read or copied by anyone who has not been explicitly authorised by the owner of that information. Most of the data that traverses computer networks can claim no pretence whatsoever of confidentiality. Protocols such as Simple Mail Transfer Protocol

(SMTP), File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP) send their data in clear text and so anyone that lays hands on the data can read it. However, solutions are now available so that sensitive data can be encrypted into an unintelligible format before transmission, and decrypted after delivery. Encryption provides more confidentiality because only intended recipients can decrypt the messages. Encryption solutions include, among others, programs like Pretty Good Privacy (PGP) and Privacy Enhanced Mail (PEM) for securing e-mails.

Non-repudiation

Non-repudiation is mostly achieved through the use of digital signatures. Non-repudiation means that after one has signed and sent a message, one cannot later claim that one did not sign the original message. One cannot repudiate one's signature, because the message was signed with one's private key, for instance (which, presumably only the owner knows). This is an important factor when exchanging sensitive information over the network.

Auditing

As well as worrying about unauthorised users, authorised users sometimes make mistakes, or even commit malicious acts. In such cases, one needs to determine what was done, by whom, and what was affected. Auditing can be of great help in such cases since it provides some incorruptible record of activity on a system that positively identifies the actors and their actions. Extensive logging is prerequisite to successful auditing.

Availability

Availability refers to protection against disruption of service: The question to ask is does data remain accessible for legitimate uses? Networks aim at making their services available to their authorised users. Hence a top priority when running a network is to ensure that it is well protected and that its services are not degraded or made unavailable (e.g. a system crash) without authorisation. An unavailable system denies its legitimate and authorised users the needed services. This state of affairs can lead to disruption of production.

Responsibility and control

Many organisations discover that they cannot design a security policy because the organisation has not specified how responsibility for information is assigned or controlled. The issue of responsibility for information has several aspects to consider:

- *Accountability*. Accountability refers to how an audit trail is kept: which group is responsible for each item of data? How does the group keep records of access and change?
- *Authorisation*. Authorisation refers to responsibility for each item of information and how such responsibility is delegated to others: who is responsible for where information resides and how does a responsible person approve access and change?

The critical issue underlying both accountability and authorisation is control - an organisation must control access to information analogous to the way the organisation controls access to physical resources such as offices, equipment and supplies.

Who are the Threats?

Among those posing threats to networks and network resources are:

- curious crackers,
- vandals,
- industrial spies and
- mere accidental disclosure of sensitive data.

Curious crackers poke around just to see what they can get into and just to satisfy their selfish interests. Vandals on the other hand, inflict various sorts of damage to networks and network resources. They cause system down-times, network outages and steal network bandwidth usage. These incidents can cause tremendous losses to the targeted organisations. Industrial spies target organisations so as to steal trade secrets and such information, which might put the target organisation at a disadvantage regarding its competitors. In some cases, they aim at harming the reputation of the targeted organisation. However, in some cases sensitive information may accidentally be disclosed to the people in whose hands the information is not supposed to land.

What are the Threats?

The threats to network systems manifest themselves as attacks targeted at resources of a system. These attacks often exploit flaws in either the operating system, network protocols or application programs. The general goal of such attacks is to subvert the traditional security mechanisms on the systems and execute operations in excess of the attacker's authorisation. These operations could include reading protected or private data or simply doing malicious damage to the systems or user files. Threats to network and computing resources could also be due to disasters like floods, fires or even electrical surges and spikes. Confidential information can reside in two states on a network.

- It can reside on physical storage media, such as a hard drive or main memory or
- It can reside in transit across the physical network wire in the form of network packets.

These two information states present multiple opportunities for attacks from internal as well as external attackers of a network. Some general attacks are discussed below:

Man-in-the Middle Attacks

A *man-in-the-middle* attack requires that the attacker have access to network packets that come across the networks. Network packet filters are the tools that attackers use for this type of attacks. Most network protocols distribute network packets in *clear text* and so the network packets can be processed and understood by any application that can pick them off the network and process them. A packet filter is a software application that uses a network adapter card in promiscuous mode to capture all network packets that are sent across a network. A packet filter can provide its user with sensitive information, such as user account names, passwords and the actual data carried in the captured packets. Packet filters can also provide information about the topology of a network, such as what computers run which services, how many computers are on a network and which computers have access to others

If an attacker gains access to a re-usable password for a system-level user account, he can use it to create a new account that can be used at anytime as a *back-door* to get into the network and its resources. Once in the target system, the attacker can modify critical system resources such as the password for the system administrator account, the list of services and permissions on file servers, and the login information for other computers that contain confidential information. In addition, a network packet filter can be modified to interject new information or change existing information in a network packet. By doing so, the attacker can cause network connections to shut down prematurely, as well as change critical information within the network packets. This can cause immense damage to the affected systems and consequently the affected organisations. When an attacker successfully gains access to a resource, he has the same rights as the user whose account has been compromised to gain access to that resource.

IP Address Spoofing

An IP Address Spoofing attack occurs when an attacker outside a network pretends to be a trusted computer either by using an IP address that is within the range of IP addresses for the attacked network or using an authorised external IP address that the attacked network trusts.

An attacker who manages to change the routing tables of the attacked network to point to the spoofed IP address, can receive all the network packets that are addressed to the spoofed address. In such a case, an attacker can monitor all the network's traffic, effectively becoming a *man in the middle*. IP spoofing can be used by both internal and external attackers of a network. In this way, IP spoofing can also yield access to user accounts and passwords. An attacker can also emulate an internal user in ways that can prove embarrassing for an organisation. For example, the attacker could send irresponsible e-mail messages to an organisation's business partners that appear to have originated from someone within the partner organisation.

Denial-of-Service Attacks (DOS)

Denial-of-service attacks focus on making a network service unavailable to legitimate users for normal use, and this is typically accomplished by exhausting some resource limitation on the network or within an operating system or application.

When involving specific network server applications, such as HTTP or FTP, these attacks can focus on acquiring and keeping open all the available connections supported by that server, effectively locking out valid users of the server or service. Most denial-of-service attacks exploit a weakness in the overall architecture of the system being attacked. Some attacks compromise the performance of a network by flooding the network with undesired network packets (e.g., ping floods) and by providing false information about the status of network resources.

Application Layer Attacks

Application layer attacks usually exploit well-known weaknesses in software commonly found on network servers, such as sendmail (e.g., the Internet Worm of 1988). By exploiting these weaknesses, attackers can gain access to a computer system with the permissions of the account running the application, which is usually a privileged system-level account.

Trojan horses are commonly used as application layer attacks. Trojan horses are programs that may provide all the functionality that a normal program offers, but they also include other features that are known only to the attacker, such as monitoring login attempts to capture user account and password information. These programs can capture sensitive information and distribute it back to the attacker. They can also modify application functionality, such as applying a blind carbon copy to all e-mail messages so that the attacker can read all the e-mails of an organisation.

Some application layer attacks exploit the openness of such technologies as the Hypertext Markup Language (HTML) specification, Web browser functionality, and the HTTP protocol. These attacks include Java applets, whereby harmful programs are passed across the network and are then loaded through a user's browser. These are then used as trojan horses to cause malicious damage, e.g. overwriting files or executing other malicious programs.

What are the Defences?

Several methods and technologies have been devised to protect network resources from malicious damage. Most of these defensive mechanisms and technologies are based on:

- protection from attacks
- detection of successful attacks (and also attacks in progress) and
- recovery from successful attacks.

Network Security Policy

The network security policy, is a collection of security rules, conventions, and procedures governing communications into and out of a protected network, allowing only approved services and packets to pass through between internal and external networks. The policy also spells out the access rights of users to information and information resources. Furthermore, a network security policy identifies the vital resources of a network, which require protection. A network security policy is the first step that should be taken as one embarks on securing a network.

Physical Security

Physical security is paramount in any intention to secure a network and its resources. A network whose computing equipment is physically accessible to anybody can never be well secured regardless of the effort and capital spent on security. Besides, computer equipment is sensitive to many types of environmental conditions such as dust, fire, smoke and humidity and should consequently be protected against them. Dust can shorten the life span of magnetic media, tape and optical drives. Surge suppression devices should be used to prevent computer and communication equipment from surges. Physical access to computing resources should be denied to unauthorised people. By controlling physical access to computers and communication equipment, it becomes difficult for vandals to steal or damage either data or equipment. Access policies for computing facilities should be established and the affected users educated on security.

Backups

A reliable and current backup can be used as a first line of defence against a network disaster. Backups are vital for the following reasons:

- If a site's network is compromised or undergoes any serious damage, the administrator can use the latest backup to restore the data and make the network operational again.
- If a network is compromised and one does not know the extent of the damage, backups can help the administrator to determine what changes were made to the system and so enable the administrator to restore the data with confidence.
- Backups are also useful for archiving seldom used data that must be saved for legal or historical purposes.

Backups should be updated from time to time. There are several types of backups such as:

- Incremental backups: where only data that has changed since the last backup is backed up.
- Full backup: where the entire specified data is backed up.

Which type of backup to carry out and the frequency at which it is carried out depends on the security policy of the site. To ensure the smooth restoration of data even in cases of catastrophes like floods or fires that befall an operational site, it is advisable to keep a standby backup copy in a water- and fire-proof safe off-site.

The three generation system could also be employed to give a big window for data recovery. In very sensitive organisations where system outages should not be tolerated, hardware backup systems could also be installed. These are operated on standby basis and become operational immediately the main system stops working due to corruption or break-down.

User Education

User-awareness is a very powerful tool in security. Trained and educated users are less likely to fall for scams and social engineering attacks. They are also more likely to participate in security measures without reservations if they understand why such security measures are in place. In security, education should be a continuous process since there are always new tools and new threats, new techniques, and new information to be learned.

Operating Systems Security

Operating systems such as 4.4-BSD UNIX and Linux, provide some security mechanisms that can be used to protect system and network resources from misuse. Such mechanisms include the use of file flags that restrict changes made to files, they also restrict who accesses which file or system resource. This gives control of access to files and system resources. Operating systems also keep system logs about the activities registered on the system. Used carefully, these logs can be a very handy tool in monitoring what goes on in the system. In the case of a successful attack, the logs can be used to ascertain the extent of the damage caused and even to find out who did the damage and at what time.

Integrity mechanisms

We have already discussed techniques that ensure the integrity of data against accidental damage. These are the parity bits, checksums and cyclic redundancy checks (CRCs), among others. A checksum or CRC cannot absolutely guarantee data integrity for two reasons. First, if malfunctioning hardware changes the value of a checksum as well as the value of the data, it is possible for the altered checksum to be valid for the altered data. Second, if data changes result from a planned attack, the attacker can create a valid checksum for the altered data. Several mechanisms have been used to guarantee the integrity of messages against intentional change. In general, the methods encode transmitted data with a *message authentication code (MAC)* that an attacker cannot break or forge. Typical encoding schemes use *cryptographic hashing* mechanisms. For example, one cryptographic hashing scheme uses a secret key known only to the sender and receiver. When a sender encodes a message, the cryptographic hash function uses the *secret key* to scramble the position of bytes within the message as well as to encode the data. Only the receiver can unscramble the data; an attacker who does not have the secret key, cannot decode the message without introducing an error. Thus, the receiver knows that any message that can be decoded correctly is authentic.

Access control and passwords

Many computer systems use a password mechanism to control access to resources. Each user has a password, which is kept secret. When a user needs to access a protected resource, the user is asked to enter the password. A simple password scheme works well for a conventional computer system because the system does not reveal the password to others. In a network, however, a simple password mechanism is susceptible to eavesdropping. If a user at one location sends a password across a network to a computer at another location, anyone who wiretaps the network can obtain a copy of all traffic. In such situations, additional steps must be taken to prevent passwords from being reused - ONE TIME PASSWORDS.

Authentication Servers

Like firewalls, authentication servers are a network's first line of defence against attackers as they limit access to network resources by selectively permitting or denying access based on specific characteristics such as phone numbers or passwords. This helps to keep attackers (intruders) off an organisation's network. Dial-back modems, for instance, call users back at predefined telephone numbers, thus ensuring that only authorised users are allowed access to networks. In such call-back (dial-back) systems, the user phones the main system and provides a valid username and password. The dial-back system then connects to a centrally held database to find out what telephone number is associated with the password. The call-back system then phones back the caller and allows the user's connection. In this way, even if an attacker manages to compromise a password, the system would not grant access because the call from the dial-back system would not reach him. By validating users before they are allowed to dial in (log on) to the network, authentication servers ensure that only legitimate users are allowed access.

Encryption and privacy

To ensure that the contents of a message remain confidential despite wiretapping, the message must be *encrypted*. In essence, encryption scrambles bits of the message in such a way that only the intended recipient can unscramble them. Someone who intercepts a copy of the encrypted message will not be able to extract information. Several technologies exist for encryption. In some technologies, a sender and receiver must both have a copy of an *encryption key*, which is kept secret. The sender uses the key to produce an encrypted message, which is then sent across a network. The receiver uses the key to decode the encrypted message. That is, the *encrypt* function used by the sender takes two arguments: a key, K, and a message to be encrypted, M. The function produces an encrypted version of the message, E.

E = encrypt (K, M)

The *decrypt* function reverses the mapping to produce the original message:

M = decrypt (K, E)

Mathematically, *decrypt* is the inverse of *encrypt*: *M = decrypt (K, encrypt (K, M))*

Public key encryption

In many encryption schemes, the key must be kept secret to avoid compromising security. One encryption technique assigns each user a pair of keys. One of the user's keys, called the *private key*, is kept secret, while the other, called the *public key*, is published along with the name of the user, so everyone knows the value of the key. The encryption function has the mathematical property that a message encrypted with the public key cannot be easily decrypted except with the private key, and a message encrypted with the private key can not be decrypted except with the public key. The relationships between encryption and decryption with the two keys can be expressed mathematically. Let *M* denote a message, *pub-u1* denote user1's puplic key and *prv-u1* denote user1's private key. Then

M = decrypt (pub-1, encrypt (prv-u1, M))

and

M = decrypt (prv-u1, encrypt (pub-u1, M))

Revealing a public key is safe because the functions used for encryption and decryption have a *one way property*. That is, telling someone the public key does not allow the person to forge a message that appears to be encrypted with the private key. Public key encryption can be used to guarantee confidentiality. A sender who wishes a message to remain private uses the receiver's public key to encrypt the message. Obtaining a copy of the message as it passes across the network does not enable someone to read the contents because decryption requires the receiver's private key. Thus, the scheme ensures that data remains confidential because only the receiver can decrypt the message.

Authentication with digital signatures

An encryption mechanism can also be used to authenticate the sender of a message. The technique is known as a *digital signature*. To sign a message, the sender encrypts the message using a key known only to the sender. The recipient uses the inverse function to decrypt the message. The recipient knows who sent the message because only the sender has the key needed to perform the encryption. To ensure that encrypted messages are not copied and resent later, the original message can contain the time and date that the message was created.

Let us consider how a public key can be used to provide a digital signature. To sign a message, a user encrypts the message using his or her private key. To verify the signature, the recipient looks up the user's public key and uses it to decrypt the message. Because only the user knows the private key, only the user can encrypt a message that can be decoded with the public key. Two levels of encryption can be used to guarantee that a message is both authentic and private. First, the message is signed by using the sender's private key to encrypt it. Second, the encrypted message is encrypted again using the recipient's public key. Mathematically, double encryption can be expressed as:

X = encrypt (pub-u2, encrypt (prv-u1, M))

where *M* denotes a message to be sent, *X* denotes the string that results from the double encryption, *prv-ul* denotes the sender's private key, and *pub-u2* denotes the recipient's public key.

At the receiving end, the decryption process is the reverse of the encryption process. First, the recipient uses his or her private key to decrypt the message. The decryption removes one level of encryption, but leaves the message digitally signed. Second, the recipient uses the sender's public key to decrypt the message again. The process can be expressed as

M = decrypt (pub-u1, decrypt (prv-u2, X))

where *X* denotes the encrypted string that was transferred across the network, *M* denotes the original message, *prv-u2* denotes the recipient's private key and *pub-u1* denotes the sender's public key. If a meaningful message results from the double decryption, it must be true that the message was confidential and authentic. The message must have reached its intended recipient because only the intended recipient has the correct private key needed to remove the outer encryption. The message must have been authentic, because only the sender has the private key needed to encrypt the message so the sender's public key will correctly decrypt it.

Firewalls

A firewall is a computer system, which is used to separate two networks from each other. The firewall only allows certain types of network traffic to pass through it from one network to the other. With the boom of the Internet, many organisations connect their local area networks (intranets) to the Internet. This allows them to share the vast amount of information that is available on the Internet. Besides, these organisations can also advertise themselves through the Internet. However, being a public network, the Internet, is full of security risks. An organisation that connects to the Internet exposes its local data and information to all the risks that exist on the Internet. Much as many organisations wish to connect to the Internet, they still want their private data and information to remain untampered with by intruders. To ensure that all access from external networks (such as the Internet) to the local network is controlled, a firewall is placed in between the local network and the Internet. The firewall will then ensure that only particular types of network traffic that satisfy the network security policy of the organisation is allowed through. This controlled access tremendously reduces the risks of unauthorised access to the local network from external networks. Unauthorised access from the Internet could lead to private and sensitive information on the local network being changed, deleted or copied to other areas. In performing its task to control access to the local network, the firewall uses network policy rules that are applied on the network traffic packets, using one of the following two principles:

- Only the type of network traffic that has been specified in the policy rules should be allowed and all other network traffic should be prohibited.
- Only the type of network traffic that has been specified in the policy rules should be prohibited and all other network traffic should be allowed.

The first principle is quite strict and allows the firewall to offer maximum security. However, it requires much work to keep on changing the rules as new services appear which an organisation wishes to use. The second principle offers little security but is much more flexible. The second principle is also powerless in case of new attacks. To achieve tight security, the first principle should be preferred.

Firewall Designs

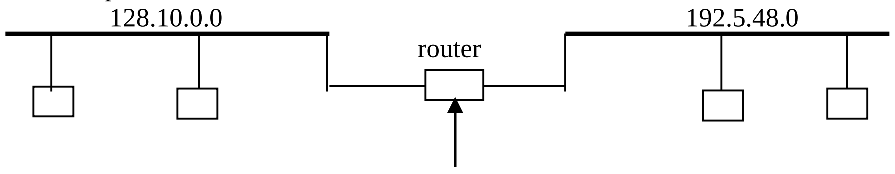
There are many different designs of firewalls. These include packet filtering routers, Bastion hosts, Dual Homed hosts and combinations of these.

Packet Filtering Routers (Screening Routers)

Screening routers are normally store-and-forward routers, which have been extended to include some packet filtering capabilities. Apart from their usual function of routing packets, they filter these packets according to the network security policy and so function as firewalls. Such filtering is done at the Internet and Transport Layers of the TCP/IP model using IP addresses and sometimes port numbers. Screening routers are the simplest type of firewalls and hence offer limited security. However, filtering routers are completely transparent to the users on the local network.

To prevent each computer on a network from accessing arbitrary computers or services, many sites use a technique known as *packet filtering*. As figure 18 shows, a packet filter is a program that operates in a router. The filter consists of software that can prevent packets from passing through the router on a path from one network to another. A manager must configure the packet filter to specify which packets are permitted to pass through the router and which should be blocked. *(Please note: Some LAN switches also provide a similar form of filtering that allows a manager to configure which frames are permitted to pass from one computer to another and which should be blocked)*. A packet filter operates by examining fields in the header of each packet. The filter can be configured to specify which header field to examine and how to interpret the value. To control which computers on the network can communicate with computers on another, a manager specifies that a filter should examine the *source* and *destination* fields in each packet header. In the figure, to prevent a computer with IP address 192.5.48.27 on the right-hand network from communicating with any computer on the left-hand network, a manager specifies that the filter should block all packets with a source address equal to 192.5.48.27. Similarly, to prevent a computer with address 128.10.0.32 on the left-hand network from receiving any packets from the right-hand network, a manager specifies that the filter should block all packets with a destination address equal to 128.10.0.32.

In addition to using the source and destination addresses, a packet filter can examine the protocol in the packet or the high-level service to which the packet corresponds. The ability to selectively block packets for a particular service means that a manager can prevent traffic to one service, while allowing traffic to another. For example, a manager can configure a packet filter to block all packets that carry World Wide Web communication, while allowing packets that carry e-mail traffic. A packet filter mechanism allows a manager to specify complex combinations of source and destination addresses and services. Typically, the packet filter software permits a manager to specify Boolean combinations of source, destination and service type. Thus, a manager can control access to specific services on specific computers. For example, a manager might choose to block all traffic destined for FTP service on computer 128.10.2.14, all WEB traffic leaving computer 192.5.48.33 and all e-mail from computer 192.5.48.34. The filter blocks only the specified combinations - the filter passes traffic destined for other computers and traffic for other services on the specified computers.



packet filter in router

Figure 31: Illustration of the location of a packet filter.

Bastion host

In a Bastion hot set-up, the internal and external networks are separated by a screening router. The internal network has a special and well-secured host - the Bastion host. The screening router only allows packets to and from the Bastion host. This forces all traffic to pass through the firewall (Bastion host) and so making the Bastion host design ideal for proxying. The Bastion host has no routing capability. Any attacker who intends to attack the local network from the external network has to go through the Bastion host. As a result, the more secure the Bastion host, the more secure the local network as well. For internal users, the Bastion host set-up of a firewall is less transparent. The local network hosts also need to be configured specifically to suit the Bastion host set-up.

Dual-Homed host

The Dual Homed host set-up of a firewall uses two routers. One of the routers has a connection to the external network while the other has a connection to the internal network. The area in between the two routers is called the Demilitarised Zone (DMZ). The Demilitarised Zone consists of at least one host, the firewall. The external router (the one connected to the external network) only allows packets through if they are addressed to hosts on the Demilitarised Zone. As a result, all network traffic from the outside is made to go through the firewall where it is analysed for suitability using the firewall rules, before being allowed into the internal network. Traffic from the inside, too, is forwarded to the firewall by the internal router. This subjects all traffic to and from the internal network to be checked against the network policy on the firewall. This set-up ensures that there is no direct connection between the external and the internal network. The internal hosts are totally invisible from the external network. The Dual Homed host is therefore suitable for running proxying software and also for providing Network Address Translation (NAT) so as to hide the internal network structure and addresses. The firewall on the Demilitarised Zone has two interfaces (homes), one for the external and the other for the internal connection, hence the name Dual Homed host.

Firewall Filtering Principles

The filtering of packets depends on whether the firewall is a:

- packet filter,
- circuit-level firewall,
- application-level firewall or
- hybrid firewall.

When filtering, each network packet that passes through the firewall will be evaluated against the following rules:

(a) If no matching rule is found, then the network packet is dropped.
(b) If a matching rule is found that permits the communication, then a peer-to-peer communication is allowed.
(c) If a matching rule that denies the communication is found then the network packet is dropped.

Static Packet Filter Firewalls

A packet filter firewall analyses network traffic at the network and transport protocol layers. Flags within the protocol headers can also be checked, for example, to determine if a packet is part of an established TCP session. Each IP network packet is examined to see if it matches one of a set of rules defining what data flows are allowed, depending on the following:

(a) the physical network interface that the packet arrives on
(b) the address the packet is supposedly coming from (e.g., source IP address)
(c) the address the packet is going to (e.g., destination IP address)
(d) the transport layer protocol (e.g., TCP, UDP)

(e) the source port and
(f) the destination port of the packet.

Packet filter firewalls are stateless. Each packet is examined in isolation from previous packets. Logging of packets is also done without regard to previous packets.

Advantages of packet filtering

(a) It is relatively easy to add new protocols since this requires only modifying filtering rules. As a result new services can be taken advantage of as they become available.
(b) They are generally faster than other firewall technologies because they perform relatively fewer evaluations.
(c) By using network address translation, they can shield internal IP addresses from external users.
(d) They are transparent to users as they require no client computers to be specifically configured.

Disadvantages of packet filtering

(a) They do not necessarily understand application layer protocols. As a result they may allow malicious commands in the payload data to enter the protected network.
(b) They are stateless in that they do not keep information about a session or application-derived information. Thus, they may not be in position to correlate events.
(c) The filtering languages may not usually be easy to implement correctly.
(d) Since they do not provide application-layer awareness, logging may also not be all that extensive. This can lead to ineffective network auditing.

Dynamic Packet Filter Firewalls

A dynamic packet filter is a firewall system that can monitor the state of active connections and use this information to determine which network packets to allow through the firewall. The firewall accomplishes its functional requirements by tracking and matching requests and replies and so it can screen for replies that don't match a request. This makes dynamic packet filtering firewalls ideal for filtering User Datagram Protocol (UDP) traffic. When a request is received, the dynamic packet filter records session information such as IP addresses and port numbers, and then it opens up a virtual connection so that only the expected data reply is let back through this virtual connection. Once the reply is received, the virtual circuit is closed. The state information associated with the virtual connection is typically remembered for a short period of time, and if no response packet is received within a given period of time then the virtual connection times out and is consequently invalidated. The ability to remember the state of the virtual circuit enables dynamic packet filters to have increased security capabilities than static packet filters. Dynamic packet filter firewalls have almost the same advantages and disadvantages associated with static packet filter firewalls, except that they do not allow unsolicited UDP packets on the local network.

Circuit-level Firewalls

A circuit level firewall, validates the fact that a packet is either a Transmission Control Protocol (TCP) connection request or a data packet belonging to a TCP connection between two peer transport layers. To validate a session, a circuit level firewall examines each connection set-up to ensure that it follows a legitimate handshake. Data packets are not forwarded until the handshake is complete. The firewall maintains a table of valid connections that includes complete session state and sequencing information. Data packets matching an entry in the table are allowed to pass through the firewall. When the connection is terminated then the corresponding table entry is also removed from the table and the virtual circuit between the two peer transport layers is closed. When a connection is set up, the circuit level firewall typically stores the following information about the connection:

(a) a unique session identifier for the connection, which is used for tracking purposes
(b) the state of the connection (e.g., handshake, established, or closing)
(c) the sequencing information

(d) the source IP address
(e) the destination IP address
(f) the physical network interface through which the packet arrives and
(g) the physical network interface through which the packet departs.

Advantages of Circuit-level Firewalls

(a) They are generally faster than application layer firewalls because they perform fewer evaluations.
(b) They can be used to prohibit connections between specific external hosts and internal hosts.
(c) By using network address translation, they can shield internal IP addresses from external users.

Disadvantages of Circuit-level Firewalls

(a) They can only restrict access to TCP.
(b) They cannot perform strict security checks on higher-level protocols. As a result, packets with malicious commands in their payloads may be allowed through the protected network.
(c) They offer limited logging and auditing as they are not application-layer aware.

Application-level Firewalls (proxies)

Application-level firewalls (also called proxies) evaluate network packets for valid data through all the layers up to the application layer before allowing a connection through the firewall. Proxies control connection establishment and user authentication based on source/destination addresses and port numbers and they maintain complete connection state and sequencing information. They mediate traffic by paying attention to particular protocol data and commands in the data payload. Since proxies are protocol-specific, they can provide increased access control, detailed checks for valid data, and generate detailed audit records about the traffic that they transfer. Proxy services never allow direct connections between internal and external hosts, hence they force all network packets to be examined and filtered for suitability.

Advantages of Application-level Firewalls

(a) Since they are end-points for communications, they guarantee that untrusted users and services communicate only to the proxy. This secures the addressed services and hosts. Also proxies are usually written with security in mind.
(b) Since they understand the application layer protocols, they can offer flexible security policy controls, authentication, increased access controls, detailed checks for valid data, extensive logging and auditing.
(c) New proxies are usually easy to implement.
(d) They offer Network Address Translation (NAT) and so shielding internal IP addresses from external users.

Disadvantages of Application-level Firewalls

(a) New application layer protocols that need to be added to the firewall will require new proxies.
(b) They may be less transparent, requiring users to have to connect to the firewall, which can expose the firewall to undesirable manipulations. Application-level firewalls may also require modified client software.
(c) Due to the extensive processing that they have to do on network packets, they may have lower performance as compared to firewalls, which operate at lower layers.
(d) Because proxy servers listen on the same port as network servers, one can not run such network servers on the firewall server.

Hybrid firewalls

Hybrid firewalls combine the characteristics and techniques of two or more other firewall types. A hybrid firewall possesses all the advantages and disadvantages of the firewall types that it combines.

Virtual Private Networks (VPN)

Virtual Private Networks (VPN) may be included into firewalls and so adding value to firewall systems. A virtual private network is a temporary and secure connection over a public network, usually the Internet, for the purpose of exchanging private information. Traffic over public networks is vulnerable to eavesdropping (snooping) attacks. Hence, a virtual private network encrypts data over a connection on a public network to protect the information from being revealed if intercepted. This makes virtual private networks ideal for securing on-line communications over public networks. Since they use public networks for communications, virtual private networks offer significant cost savings, greater flexibility, and easier management as compared to private networks like leased lines and dial-up remote access.

Virtual private networks use strong authentication, encryption and access control. Strong authentication technologies like token cards, smart cards, digital certificates, biometrics (fingerprints and iris scanning) enable the verification of individual identities and hence their activities on the network.

Virtual Private Network Implementations

Although there are many types of VPN implementations, they can be grouped into three main categories:

(a) Intranet VPN: between an organisation's branch offices.
(b) Remote Access VPN: between an organisation's remote or travelling employees.
(c) Extranet VPN: between an organisation and its associations, such as partners, customers, suppliers, and investors.

Intranet VPN

Intranet virtual private networks are temporary LAN-to-LAN connections that link branches of the same organisation, which are geographically separated from each other. This scenario is commonplace in today’s globalisation. Since the sites belong to one organisation and therefore have one and the same security policy, the branches have some trusted relationship to each other. Hence an intranet virtual private network establishes encrypted bi-directional tunnels between trusted local area networks across the Internet. Highly secured intranet virtual private networks can also ensure that only certain individuals at one branch office have access to the resources of the other branch, and that each individual user has a different set of permissions. All data transferred across the Internet is completely encrypted and authenticated all the way to the endpoints.

Remote Access VPN

Remote access virtual private networks are mostly used by mobile and telecommuting employees to enable them to have access to the organisations resources from a remote location. Apart from strong encryption methods, a remote virtual private network also requires strong authentication methods. Once an employee has authenticated to the organisations virtual private network server, a certain level of access is granted depending on the profile of the particular employee. If authentication fails then access is denied. During an authenticated session, all data is encrypted from one endpoint to the other.

Extranet VPN

Extranet virtual private networks are intended to reach networks of varying security policies such as those of an organisation's partners, customers, and suppliers. Hence, extranet virtual private networks provide a hierarchy of security, with access to the most sensitive data being nested under the tightest security control. An extranet virtual private network filters access to network resources based on several parameters including:

(a) source IP address,
(b) destination IP addresses,
(c) application usage,
(d) type of encryption and authentication used, and
(e) individual, group and subnet identity.

To be able to identify individual users, strong authentication methods such as token cards, smart cards, digital certificates, biometrics (fingerprints and iris scanning) are employed.

Virtual Private Network Technologies} Various technologies are used to implement virtual private networks. These include, among others, the

The Internet firewall concept – The Summary

A packet filter is often used to protect an organisation's computers and networks from unwanted Internet traffic. As figure 19 illustrates, the filter is placed in the router that connects the organisation to the rest of the Internet. A packet filter configured to protect an organisation against traffic from the rest of the Internet is called an Internet firewall. The term is derived from the fireproof physical boundary placed between two structures to prevent fire from moving between them. An Internet firewall is designed to keep problems in the Internet from spreading to an organisation's computers. Firewalls are the most important tools used to handle network connections between two organisations that do not trust each other. By placing a firewall on each external network connection, an organisation can define a secure perimeter that prevents outsiders from interfering with the organisation's computers. In particular, by limiting access to a small set of computers, a firewall can prevent outsiders from probing all computers in an organisation, flooding the organisation's networks with unwanted traffic, or attacking a computer by sending a sequence of IP datagrams that is known to cause the computer system to misbehave (e.g., to crash).

A firewall can lower the cost of providing security. Without a firewall to prevent access, outsiders can send packets to arbitrary computers in an organisation. For example, an outsider can guess the IP address of the computers in an organisation by finding the set of network numbers that the organisation has been assigned and then trying each of the possible hosts on those networks. Consequently, to provide security, an organisation must make all of its computers secure. With a firewall, however, a manager can restrict incoming packets to a small set of computers. In the extreme case, the set can contain a single computer. Although computers in the set must be secure, other computers in the organisation do not need to be. Thus, an organisation can save money because it is less expensive to install a firewall than to make all computer systems secure.

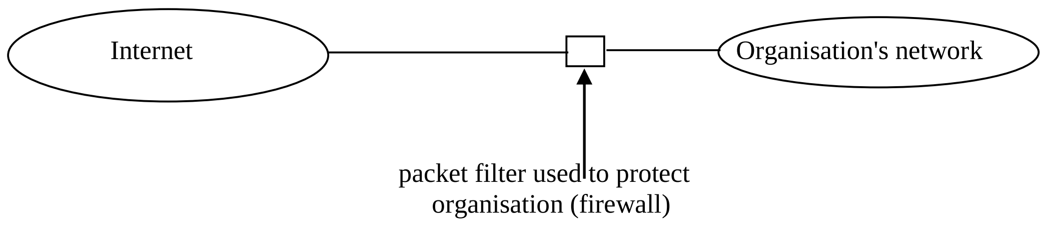


Figure 32: Illustration of a packet filter used as the primary part of a firewall that protects an organisation against unwanted traffic from the Internet

Copperbelt University
Computer Science Department

Internet Technologies

By Dr Derrick Ntalasha

CS 460: INTERNET TECHNOLOGIES

Initialisation (Configuration)

In all our discussions so far, we have assumed that the host computer systems and routers are already running. That is, each computer has been powered on, the operating system has started, protocol software has loaded and values such as entries in the routing table have been initialised. The question to ask is - how does the protocol software in a host or router begin operation? In particular, what steps must a computer system take before protocol software is ready for use?

Bootstrapping

What happens when a computer first begins operation? The process is known as *bootstrapping*. Sometimes this process is called *booting*. The term bootstrapping is derived from the phrase *"pulling oneself up by one's own bootstraps"*. When a user turns on the computer, the hardware searches permanent storage devices, usually disks, until it finds a device that contains a special program called a *boot program* at location zero. The hardware then copies the boot program into memory and branches to it. When the boot program runs, it accesses the storage device to read and load additional software (e.g., the operating system). Finally, after all the software has been loaded, the operating system allows a user to run application programs. The operating system software remains resident in memory, making it possible for an application to invoke operating system services at any time.

Starting protocol software

When is protocol software loaded during bootstrapping? The answers can be varied. On a computer that uses dialup telephone modem for communication, protocol software can be embedded in applications. When a user invokes an application that needs to communicate, the application manages the modem - only one application can use the modem at any time. More sophisticated computer systems, especially those with permanent network connections, embed protocol software in the computer's operating system where it is shared by all applications. In such systems, the bootstrap process loads protocol software into memory along with the operating system. Thus, like other operating system functions, the protocol software is ready to use before any application program runs. The question of how protocol software starts execution is further complicated because some systems use a computer network as part of the bootstrap process. For example, some computer systems use a network to download a copy of the operating system from a remote server. In such cases, a computer must have basic protocol capabilities built into the hardware or a bootstrap program.

Protocol parameters

Protocol software must understand many details. For example, the software must know such details as the exact packet format, the header size and the location of fields in the header. Furthermore, the software must recognise addresses and must know how to route packets to their destination. More important, the protocol software running on a specific computer must know many details about the computer. For example, the protocol must know the protocol address that has been assigned to the computer. To make protocol software general and portable, programmers who implement protocol software do not fix all details in the source code. Instead, they *parameterise* protocol software to make it possible to use a single binary image on many computers. Each detail that differs from one computer to another is encoded in a separate parameter that can be changed. Before the software runs, a value must be supplied for each parameter. For example, most protocol software does not have a computer's address compiled into the code. Instead, the software has an *address parameter* that can be changed. Before the software can send or receive packets, a value must be supplied for the address.

The act of supplying values for parameters in protocol software is known as *protocol configuration*. After a value has been supplied for each parameter, the protocol software is said to be *configured*. Protocol software must be configured before it can be used.

Examples of items that need to be configured

The configuration information that protocol software needs can be divided into two general classes: internal and external. Internal information pertains to the computer itself (e.g., the computer's protocol address). External information pertains to the environment that surrounds a computer (e.g., the location of printers that can be reached over the network). The exact details of configuration information depend on the protocol stack. For example, the items that TCP/IP protocol software needs to configure include:

- *IP address*. Each computer must have a unique IP address for each interface. Protocol software places the address in the SOURCE ADDRESS field in the header of all outgoing datagrams and uses the address to recognise datagrams sent to the computer.

- *Default router address*. To communicate with computers on remote networks, a computer must know the address of at least one router. Protocol software places the address in its routing table as the next hop for a default route.

- *Subnet mask*. Protocol software must be configured with a subnet mask before the software can know whether IP subnet addressing is being used on the network and if so, how many bits are used in the subnet portion of an address.

- *DNS server address*. Before application software on a computer can use the domain name system to translate a computer name to an IP address, the underlying protocol software must be configured with the address of a local DNS server.

- *Printer server address*. When an application specifies that data should be printed on a remote printer, protocol software must know the address of a server for that printer. One such address must be configured for each remote printer that can be accessed.

- *Other server addresses*. Most client programs specify the computer on which a server runs. However, some services are an integral part of the protocol software. For example, many computer systems expect protocol software to provide access to a time of day server that the system software can use to set the computer's clock. Protocol software must be configured with the address of a server for each such service.

Example configuration: using a disk file

Many computer systems use a file on the computer's disk. When the computer system begins execution, the protocol software reads the disk file and extracts values for parameters. To make the configuration file convenient for a human being to edit, many systems use a textual representation. For example, a configuration file might contain lines of the form:

parameter_name = value

where *parameter_name* is the name of a parameter and value is the *value* to be assigned. The protocol software must convert values from the form used in the configuration file to the form used internally (e.g., numeric values might be converted from ASCII text to binary). Once it has converted the value to internal form, the protocol software stores the value in an appropriate variable in memory. Configuration from a disk usually occurs once, when the protocol software begins execution. Thus, changing the contents of the disk file has no effect on software that has started running. To force changes in the configuration file to take effect, the system administrator must notify the protocol software that reconfiguration is needed. Some systems provide a special mechanism for notification (e.g., in UNIX, a signal is sent to the protocol software). Other systems require that the operating system be rebooted before the protocol software will read the configuration file and use the new values.

The need to automate protocol configuration

The disadvantage of storing protocol configuration information in a disk file becomes obvious when an administrator must handle many computers or when an administrator must manage computers that are moved from one network to another. When a computer is moved, for example, most of the internal configuration information changes, and the external configuration information can change as well. If the configuration information resides in a file, the administrator must manually change the file before the new configuration can be used. Consider laptop computers that can be moved easily. For example, imagine a student enrolled in a university carrying a portable computer. The student might attach a computer in his or her hostel each night and attach the same computer to the networks in one or more laboratories during the day (However, wireless technologies are making mobility among networks more common). Changing a configuration file two or more times a day is tiresome and someone entering such changes manually can make a mistake. More important, if several students move computers to new networks during the day, the effort required to enter configuration changes is overwhelming.

Methods for automated protocol configuration

Ways to automate protocol software configuration now exist. For example, some computer systems use a computer network as part of the bootstrap process. Instead of searching on local storage devices to find a bootstrap program, the computer hardware is designed to transmit a packet that contains a bootstrap request. A server on the network that is designed to respond to such requests returns a packet that contains the bootstrap program. Other computer systems are designed to boot the operating system from a local disk and to use the network to obtain protocol configuration. For example, TCP/IP protocols include *the Reverse Address Resolution Protocol (RARP)* that a computer can use to determine its own IP address. The computer sends a *RARP request* to a server. The server returns a *RARP reply* that contains the computer's IP address. The ICMP protocol provides other examples that show how protocol software can use a network to obtain configuration information. ICMP *includes Address Mask Request* and *Router Discovery* messages. A host broadcasts a Router Discovery message to locate routers on the local network. Routers respond by informing the host of their presence. A host sends an *Address Mask Request* to a router to request information about the address mask that is used on the network. The router receives the request and sends an *Address Mask Reply* message that specifies the IP subnet mask.

The address used to find an address

How can a computer send or receive packets before the protocol software has been configured? How can a computer communicate with a server before the computer knows the address of the server? In the case of RARP, the answer to the first question is simple. Although the computer does not know its IP address when it boots, the computer does know its hardware address. In fact, RARP can only be used on networks such as Ethernet, that have each computer's hardware address permanently fixed in the NIC hardware. RARP extracts the computer's hardware address from the NIC, places it in a RARP request message, and sends the request to a server. The server extracts the hardware address, consults a database that specifies the binding between a hardware address and an IP address and returns a RARP reply message that contains the computer's IP address. To make it possible for RARP to send and receive frames, the computer system must initialise the network hardware before using RARP to configure IP. In general, we can say:

> *In a layered protocol stack, layers are configured from lowest to highest, making it possible for a higher-layer protocol to use lower-layer protocols to obtain configuration information.*

The question of how a computer can communicate with a server before the computer knows the server's address is easy to answer. The computer simply broadcasts a request. For example, when sending a frame that contains a RARP request, a computer uses the hardware broadcast address as a destination address (RARP can only be used on networks that support broadcast). Similarly, a computer does not need to know the address of a router that will answer an ICMP Address Mask Request. The computer simply broadcasts the message and allows any router on the network to answer.

When a network contains multiple servers, broadcasting a request may result in multiple responses. Consider the following example. According to the ICMP protocol standard, any router that receives an ICMP Address Mask Request must return an ICMP Address Mask Reply. If a computer broadcasts a request on a network to which multiple routers attach, each router will receive a copy and each will return a response. Most implementations accept the first response and ignore others. However, because the network might also contain broadcast packets that are unrelated to the configuration request, each protocol must be designed so that a receiver can distinguish a valid response from other traffic on the network. For example, some protocols use the TYPE field in a hardware frame. Others place a randomly generated identification number in a request, and then wait for a response that contains the same value.

Sequence of protocols used during bootstrap

Using a separate protocol to obtain each piece of configuration information results in a sequence of configuration steps. Below are the first few steps that TCP/IP protocol software configuration requires.

Step 1. Broadcast a RARP Request message to obtain an IP address.
Step 2. Wait for a RARP Response message. If none arrives within T_1 seconds, return to Step 1.
Step 3. Broadcast an ICMP Address Mask Request message.
Step 4. Wait for an ICMP Address Mask Response message. If none arrives within T_2 seconds, return to Step 3.
Step 5. Use ICMP Router Discovery to find the IP address of a default router and add a default route to the routing table.

T_1 and T_2 denote timeout values.

Bootstrap protocol (BOOTP)

The chief advantage of using a separate step to obtain each item of configuration information is flexibility - each computer system can choose which items to obtain from a local file on disk and which to obtain over the network. The chief disadvantage becomes apparent when one considers the network traffic and delay. A given computer issues a series of small request messages. More important, each response returns a small value (e.g., a 4-octet IP address). Because networks enforce a minimum packet size, most of the space in each packet is wasted.

TCP/IP protocol designers observed that many of the configuration steps could be combined into a single step if a server was able to supply more that one item of configuration information. To provide such a service, the designers invented the *BOOTstrap Protocol (BOOTP)*. To obtain configuration information, protocol software broadcasts a *BOOTP Request* message. A BOOTP server that receives the request looks up several pieces of configuration information for the computer that issued the request, places the information in a single *BOOTP Reply* message and returns the reply to the requesting computer. Thus, in a single step, a computer can obtain information such as the computer's IP address, the server's name and IP address, and the IP address of a default router.

Like other protocols used to obtain configuration information, BOOTP broadcasts each request. Unlike other protocols used for configuration, BOOTP appears to use a protocol that has not been configured: BOOTP uses IP to send a request and receive a response. How can BOOPT send an IP datagram before a computer's IP address has been configured? The answer lies in a careful design that allows IP to broadcast a request and receive a response before all values have been configured. To sent a BOOTP packet, IP uses the all-1's broadcast address as a *destination address* and uses the all-0's address as a *source address*. If a computer uses the all-0's address to send a request, a BOOTP server either uses broadcast to return the response or uses the hardware address on the incoming frame to send a response via unicast. Thus, a computer that does not know its IP address can communicate with a BOOTP server. However, the server must be careful to avoid using ARP because a client that does not know its IP address can not answer ARP requests.

Figure 20 illustrates the BOOTP packet format. Each field in a BOOTP message has a fixed size. The first seven fields contain information used to process the message. The *OP* field specifies whether the message is a *request* or a *reply*, the *HTYPE* and *HLEN* fields specify the network hardware type and the length of a hardware address. The *HOPS* field specifies how many servers forwarded the request, and the *TRANSACTION IDENTIFIER* provides a value that a client can use to determine if an incoming response matches its request. The *SECONDS ELAPSED* field specifies how many seconds have elapsed since the computer began to boot. Finally, if a computer knows its IP address (e.g., the address was obtained using RARP), the computer fills in the *CLIENT IP ADDRESS* field in a request. If a computer does not know its address, the server uses field *YOUR IP ADDRESS* to supply the value. In addition, the server uses fields *SERVER IP ADDRESS* and *SERVER HOST NAME* to give the computer information about the location of a computer that runs servers. Field *ROUTER IP ADDRESS* contains the IP address of a default router.

In addition to protocol configuration, BOOTP allows a computer to negotiate to find a boot image. To do so, the computer fills in field *BOOT FILE NAME* with a generic request (e.g., the computer can request the UNIX operating system). The BOOTP server does not send an image. Instead, the server determines which file contains the requested image, and uses the field *BOOT FILE NAME* to send back the name of the file. Once a BOOTP response arrives, a computer must use a protocol like TFTP to obtain a copy of the image.

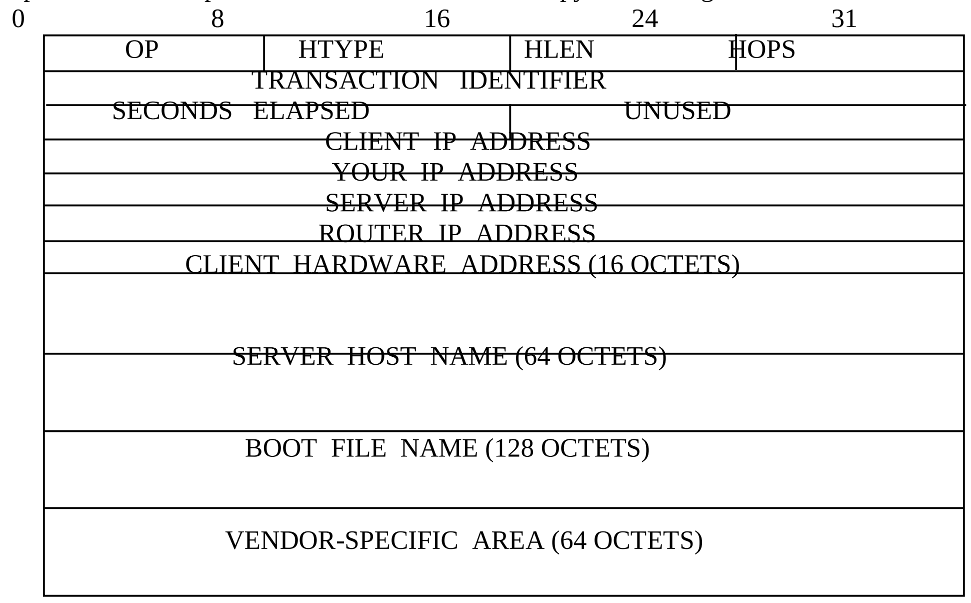


Figure 33: The format that BOOTP uses for request and response messages. The message is sent using UDP, which is encapsulated in IP.

Automatic address assignment

Although it simplifies loading parameters into protocol software, BOOTP does not solve the configuration problem completely. When a BOOTP server receives a request, the server looks up the computer in its database of information. Thus, even a computer that uses BOOTP can not boot on a new network until the administrator manually changes information in the database. Protocol software exists that allows a computer to join a new network without manual intervention. For example, IPX protocols generate a protocol address from the computer's hardware address. To make the IPX scheme work correctly, the hardware address must be unique. Furthermore, if the hardware address and protocol address are not the same size, it must be possible to translate the hardware address into a protocol address that is also unique.

The AppleTalk protocols use a *bidding* scheme to allow a computer to join a new network. When a computer first boots, the computer chooses a random address. For example, suppose computer C chooses address 17. To ensure that no other computer on the network is using the address, C broadcasts a request message and starts a timer. If no computer is using address 17, no reply will arrive before the timer expires. C can then begin using the address 17. If another computer is using address 17, the computer replies, causing C to choose a different address and begin again. Choosing an address at random works well for small networks and for computers that run client software. However, the scheme does not work well for servers. Remember that each server must be located at a well-known address. If a computer chooses an address at random when it boots, clients will not know which address to use when contacting the server on that computer. Because the address can change each time a computer boots, the address used to reach a server may not remain the same after a crash and reboot.

A bidding scheme also has the disadvantage that two computers can choose the same network address. In particular, assume that computer B sends a request for an address that another computer (e.g. A) is already using. If A fails to respond to the request for any reason, both computers will attempt to use the same address, with disastrous results. In practice, such failures can occur for a variety of reasons. A piece of network equipment such as a bridge can fail, a computer can be unplugged from the network when the request is sent, or a computer can be temporarily unavailable (e.g., in a hibernation mode designed to conserve power). Finally, a computer can fail to answer if the protocol software or operating system is not functioning correctly.

The Dynamic Host Configuration Protocol (DHCP)

To automate configuration, the IETF devised *the Dynamic Host Configuration Protocol (DHCP)*. Unlike BOOTP, DHCP does not require an administrator to add an entry for each computer to the database that a server uses. Instead, DHCP provides a mechanism that allows a computer to join a new network and obtain an IP address without manual intervention. The concept has been termed *plug-and-play networking*. More important, DHCP accommodates computers that run server software, as well as those that run client software:

- When a computer that runs client software is moved to a new network, the computer can use DHCP to obtain configuration information without manual intervention.

- DHCP allows non-mobile computers that run server software to be assigned a permanent address and the address will not change when the computer reboots.

To accommodate both types of computers, DHCP can not use a bidding scheme. Instead, it uses a client-server approach. When a computer boots, the computer broadcasts a *DHCP Request* to which a DHCP server sends a *DHCP Reply*. An administrator can configure a DHCP server to have two types of addresses: permanent addresses that are assigned to server computers and a pool of addresses to be allocated on demand. When a computer boots and sends a request to DHCP, the DHCP consults its database to find configuration information. If the database contains a specific entry for the computer, the server returns the information from the entry. If no entry exists for the computer, the server chooses the next IP address from the pool and assigns the address to the computer. In fact, addresses assigned on demand are not permanent. Instead, DHCP issues a *lease* on the address for a finite period of time. When the administrator establishes a pool of addresses for DHCP to assign, the administrator must also specify the length of the lease for each address. When the lease expires, the computer must negotiate with DHCP to extend the lease. Normally, DHCP will approve the lease extension. However, a site may choose an administrative policy that denies the extension (e.g., a university that has a network in a classroom might choose to deny extension on lease at the end of a class period to allow the next class to reuse the same addresses). If DHCP denies an extension request, the computer must stop using the address.

Optimisations in DHCP

If the computers on a network use DHCP to obtain configuration information when they boot, an event that causes all computers to restart at the same time can cause the network or server to be flooded with requests. To avoid the problem, DHCP uses the same technique as BOOTP: each computer waits a random time before transmitting or re-transmitting a request. The DHCP protocol has two steps: one in which the computer broadcasts a DHCP *Discover message* to find a DHCP server and another in which the computer selects one of the servers that responded to its message and sends a request to that server. To avoid having a computer repeat both steps each time it boots or each time it needs to extend the lease, DHCP uses *caching*. When a computer discovers a DHCP server, the computer saves the server's address in a cache on permanent storage (e.g., a disk file). Similarly, once it obtains an IP address, the computer saves the IP address in a cache. When a computer reboots, it uses the cached information to revalidate its former address. Doing so saves time and reduces network traffic.

DHCP message format

DHCP is designed as an extension of BOOTP. As figure 21 illustrates, DHCP uses a slightly modified version of the BOOTP message format. Most of the fields in a DHCP message have the same meaning as in BOOTP. DHCP replaces the 16-bit *UNUSED* field with a *FLAGS* field and uses the *OPTIONS* field to encode additional information. For example, as in BOOTP, the OP field specifies either a *Request* or a *Response*. To distinguish among various messages that a client uses to discover servers or request an address, or that a server uses to acknowledge or deny a request, DHCP uses a *message type option*.

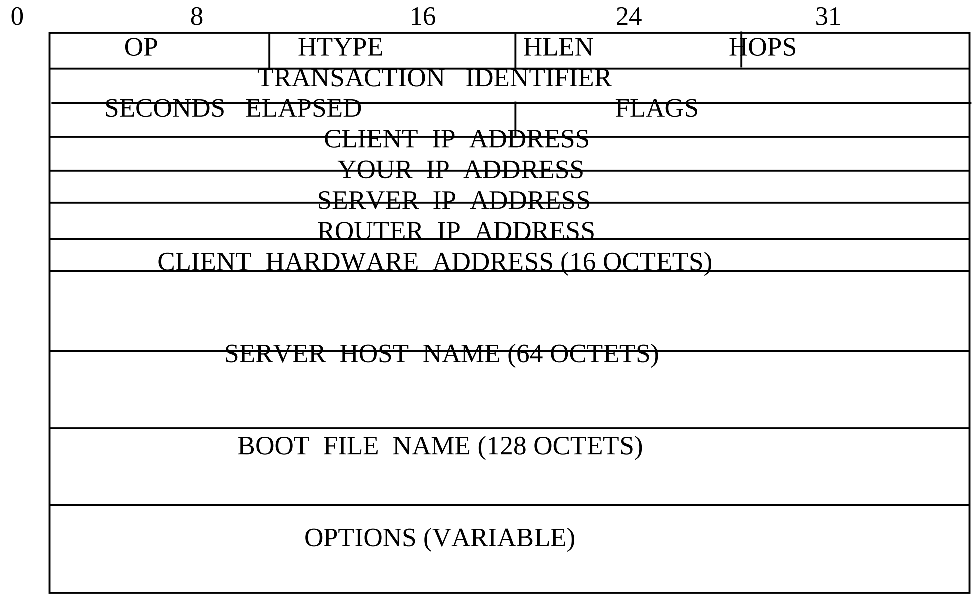


Figure 34: The DHCP message format, a slightly modified version of the BOOTP format.

DHCP and domain names

Although DHCP makes it possible for a computer to obtain an IP address without manual intervention, DHCP does not interact with the Domain Name System (DNS). As a result, a computer can not keep its name when it changes addresses. The computer does not need to move to a new network to have its name change. For example, suppose a computer obtains IP address 192.5.48.195 from DHCP and suppose the Domain Name System contains a record that binds the name x.y.z.com to the address. Now consider what happens if the owner turns off the computer and takes a two-month vacation during which the address lease expires. DHCP may assign the address to another computer. When the owner returns and turns on the computer, DHCP will deny the request to use the same address. Thus, the computer will obtain a new address. Unfortunately, the DNS continues to map the name *x.y.z.com* to the old address. Research is currently on to consider how DHCP should interact with the DNS. Until a protocol has been standardised, sites that use DHCP must either use a non-standard mechanism to change the DNS database or be prepared for computer names to change when an address changes.