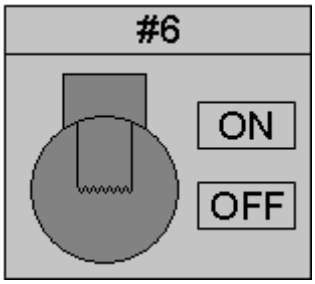
Network Protocol Primer

Author: Dave Feinberg

Layer 1: Light Signals

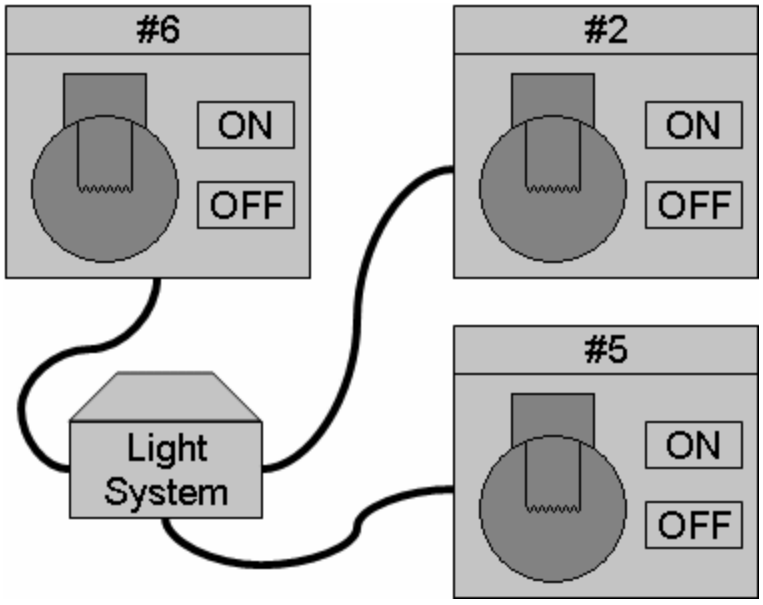
Some Light Reading

The basic problem of communicating over a computer network can be illustrated by analogy. Imagine that a dozen people know they will be imprisoned, one person per cell. Each cell has a single light panel, featuring a light bulb, on/off buttons, and an ID number.



light panel with ID #6

Pressing the "on" button on one light panel turns on the light bulbs in *every* cell. Likewise, the "off" button turns off everyone's lights. In other words, all light panels are connected to a single light system.

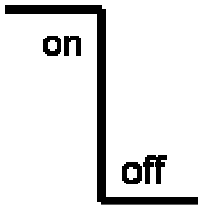


Knowing all this, the dozen people conspire to use the light system to communicate with each other. They know they can represent any information as a string of bits (0s and 1s). How can they use their light panels to transmit 0s and 1s, and what difficulties will they encounter?

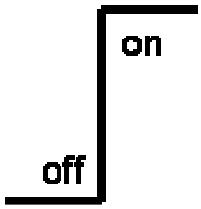
In a Bit

Let's first deal with the problem where only one person wishes to broadcast a string of bits to everyone. How can they use the light system to represent 0s and 1s? You might imagine we could turn the light off to represent a "0", and turn the the light on to represent a "1". Thus, to broadcase the message "101", we simply turn the light on, then off, then on again. But what about the message "1111111111", which requires us to leave the light on for a long period of time? How can we distinguish that message from the slightly longer message "11111111111"? If we leave the light off between messages, how will we distinguish silence from a long string of 0s?

One answer is to represent a "0" as a transition from "on" to "off", and a "1" as the opposite transition. (This is what the Ethernet protocol does, raising and lowering the voltage present on a wire shared by all machines on an Ethernet segment.) The following diagrams show how the the light's state changes over time in the course of representing a "0" or "1".

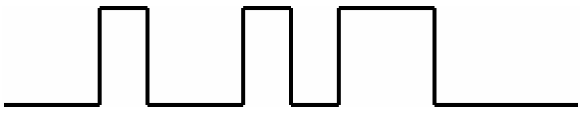


Binary "0"



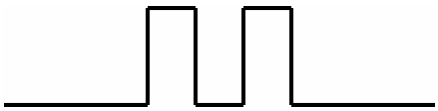
Binary "1"

Thus, the message "0110" is represented as follows. Do you see why?

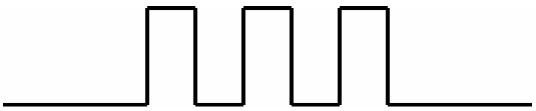


Every bit requires the light to transition between on and off. So, if we transmit 1 bit each second, then there will be at least one transition per second. This will make it relatively easy for anyone watching the light bulb to synch up to the timing of the person transmitting the message.

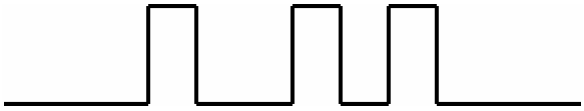
It should strike you that your curves for "00" and "11" are identical.



How will we know the difference? The solution we'll adopt is to add an extra "0" to the beginning of all messages. Thus, if we want to send the message "00", we'll use the signaling for "000". Likewise, if we want to send "11", we'll use the signaling for "011", and these are easily distinguished.



leading "0" before "00"

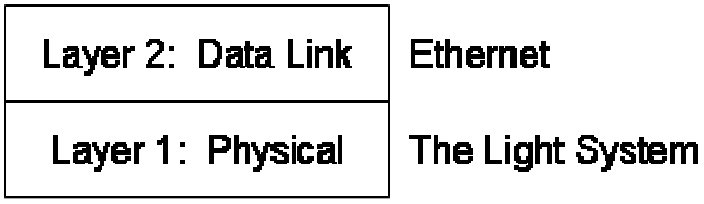


leading "0" before "11"

Layer 2: Ethernet Frames

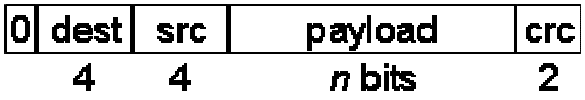
Frame of Reference

The light bulb system constitutes the *physical layer* of our simulated network protocols, much like the cables, hubs, wireless transmitters, etc., form the physical layer we use to connect to the Internet. This physical layer is often referred to as *layer 1*. The most widespread example of a *layer 2* protocol is Ethernet. In this lab, we'll implement a simplified version of the Ethernet protocol.



So far, we know how to broadcast meaningless strings of bits. When we receive such a string, how can we know if it was meant for us, or who it came from, or if we have correctly identified the same bits intended by the sender?

In the Ethernet protocol, a broadcast bit string is called a *frame*, and Ethernet answers these questions by requiring certain information to be provided in designated parts of the frame (similar to the way addresses must appear on an envelope used to mail a letter). To avoid getting caught up in the details of the protocols we'll be studying, we'll simplify them to allow us to focus on the essentials. In our simplified Ethernet protocol, a frame will appear as follows.



Our Simplified Ethernet Frame

The leading 0 is simply used for synchronization purposes, merely indicating the beginning of a broadcast. The destination address is a 4-bit binary number, and identifies the physical address of the intended recipient. In the real world, every Ethernet card has a unique serial number, called a MAC address ("media access control"). Our light panel IDs will play the role of a MAC address. Although each node on a LAN (local area network) can see every frame that is broadcast, it is expected that only the designated recipient will examine the contents of the frame. Occasionally, however, it will be useful to designate that a frame is intended for *all* nodes on the network. We'll use the special destination address 0000 for this purpose.

The next 4 bits identify the address of the source who transmitted the frame. Afterward, the actual data content, or *payload*, of the frame appears. In our simplified Ethernet protocol, this portion may consist of any number of bits (although real Ethernet frames have a minimum and maximum payload size).

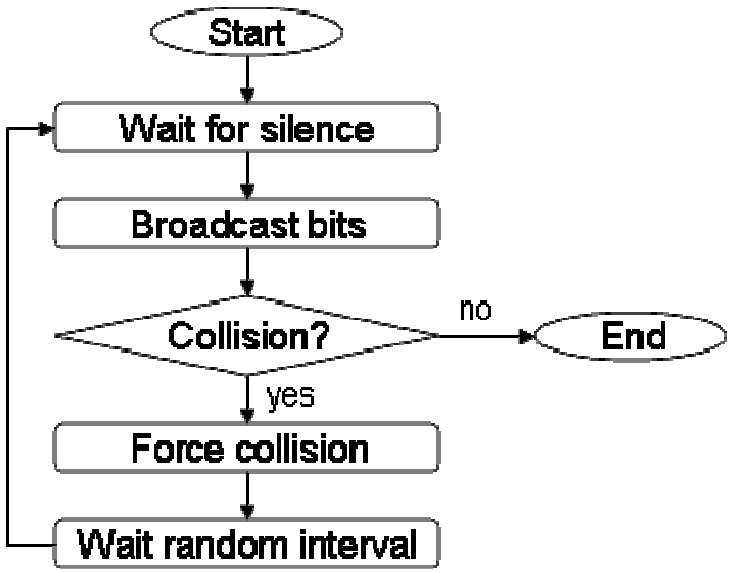
To determine that a frame has been transmitted and read correctly, some redundant information is tacked on to the end of a frame, called a *cyclic redundancy check* (or *checksum*). In our protocol, we'll find the total number of 1s in the frame's addresses and payload. The remainder when dividing this number by 4 will give us our CRC value, which will constitute the last 2 bits of our frame.

For example, if node #2 sent the payload "110000" to node #5, the resulting frame would appear as follows. Since there are five 1s in the frame's addresses and payload, the CRC value is "01" (the remainder when 5 is divided by 4). When #5 receives the frame, it will make sure that the number of 1s matches the CRC value. If the frame is corrupted on its way to #5, what are the odds that the frame will be misidentified as valid?

a frame sent from #2 to #5

Collision Coverage

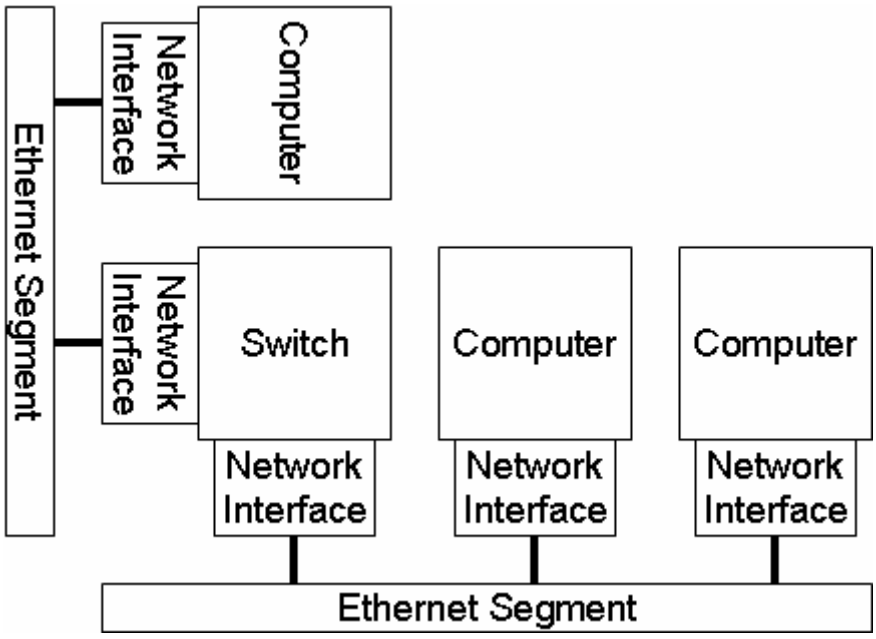
When sending a frame, it's important to wait until the network is silent first, so as not to corrupt someone else's message. But what's to stop two computers from attempting to send a message at the same time? This is called a *collision*. It's a fact of network protocols like Ethernet. Luckily, there are strategies for handling them. When a computer transmits a message, it can listen to the network. If what it detects doesn't match what it attempted to send, then a collision occurred. In this situation, the sender stops transmitting. (Actually, the sender will transmit a few extra bits before going silent. This helps ensure that all simultaneously transmitting computers detect the collision.) With any luck, anyone listening to the network at that time will notice the CRC for the interrupted frame doesn't check out, and will therefore discard the frame. When the network is silent again, the transmitting computers attempt to send their messages again (each waiting a random length of time to minimize the chance of another collision).



Listening to Either Net

An Ethernet network resembles a room full of people talking, each waiting to get a word in whenever the noise dies down. If a few people in the room generally only talk with each other, they may as well find another room to talk in. In networking terms, this other room is an *Ethernet segment*, also called a *collision domain* since collisions on the segment do not affect other segments. In our network simulation, each light system plays the role of an Ethernet segment. Any node wishing to join an Ethernet segment must have its own network interface (Ethernet card and associated drivers). For us, this network interface consists of a light panel and any equipment we use to transmit and receive bits with it.

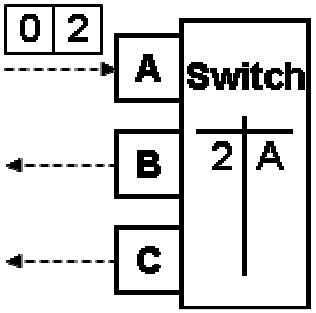
In our analogy, a person in one room may occasionally wish to converse with a person in another room. Such communication requires at least one person who is capable of speaking in both rooms (running back and forth relaying messages). In the Ethernet world, such a device is called a *switch* (essentially the same as a *bridge*). A switch has a different network interface connected to each of its Ethernet segments.



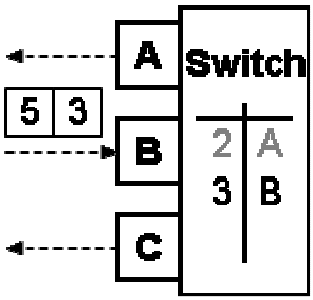
When an Ethernet frame is detected by one of the switch's network interfaces, the switch examines the destination address and retransmits the frame using the network interface capable of reaching that destination. But how does the switch know on which segment a particular MAC address is located?

A switch is actually a very clever device. It keeps a table in which each source MAC address it has seen is associated with the network interface that received a frame from that source. The following figures show how a switch behaves.

(1) The switch receives a frame from unknown source #2. The source and interface are recorded in the table. Because the frame uses broadcast address #0, it is forwarded to all other interfaces. (2) The switch receives a frame with unknown destination, and forwards it to all other interfaces.

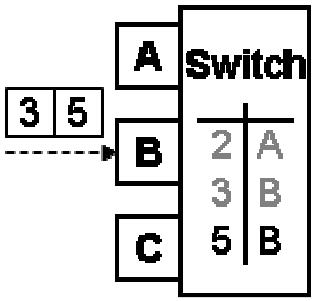


1. Broadcast Frame

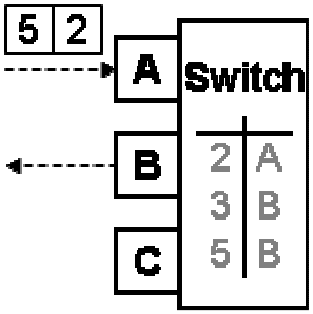


2. Unknown Destination

(3) The switch receives a frame addressed to #3, who is known to reside on the same segment. The frame is not forwarded. (4) The switch receives a frame addressed to #5, known to reside on segment B. The frame is forwarded to this segment.



3. Destination on Same Segment

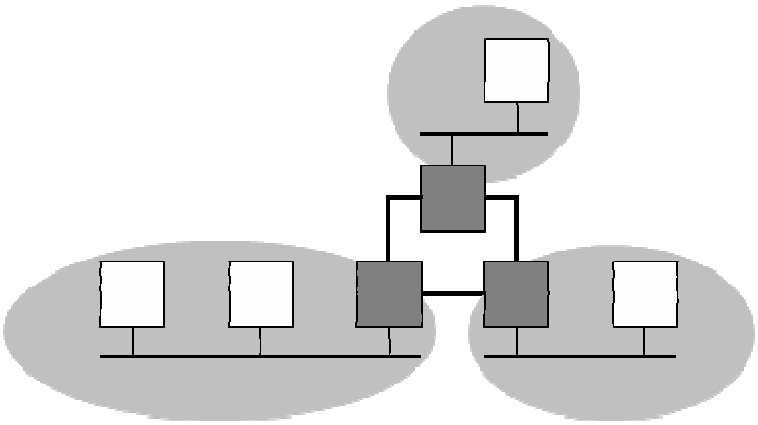


4. Destination on Different Segment

Layer 3: The Internet Protocol

Limitations of Ethernet

We know how Ethernet frames are exchanged on a LAN (local area network). We'll now consider issues that arise when we attempt to connect many such networks together to form an internet, as pictured below.



an internet, in which 3 switches connect machines on 3 different networks

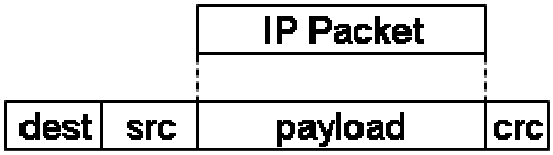
We could connect such networks exclusively through the use of the Ethernet protocol. This plan would work fine in principle, but it suffers from a couple of practical limitations.

- Each switch would need to memorize the locations of every MAC address on the Internet. This would require each switch to have an enormous amount of memory, and to engage in a time-consuming search before forwarding each frame, ultimately limiting the speed of the Internet.

- There are many other layer 2 protocols (Token Ring, PPP, X.25, Frame Relay, ISDN) that may be more appropriate than Ethernet for a particular network or for the infrastructure connecting networks together.

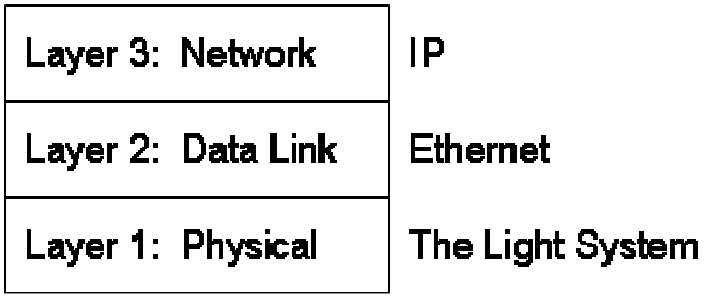
IP (Internet Protocol)

The Internet Protocol was designed to address these issues. As you probably know, IP is based on packets (similar to Ethernet frames). An IP packet can be transported as the payload of any layer 2 protocol, just as a package can be carried by a truck.



an IP packet embedded as the payload of an Ethernet frame

Thus, IP is a layer 3 protocol, where layer 3 is the *network layer*. (In fact, IP is *the* layer 3 protocol.)

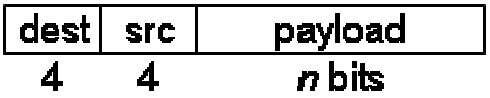


IP Addresses

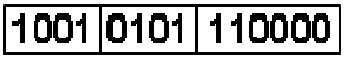
A MAC address is a *physical address*, and plays a role similar to a person's social security number. Why doesn't the telephone company let you call someone by dialing their social security number? That would require every telephone switch to know the location of everybody's telephone in the world! That's why the phone company assigns you a *logical address*—a telephone number. When you dial 1-202-456-1111, the telephone switch at the other end of your phone line has no idea where to find the phone you're calling, but it *does* know that the number starts with a 1, and is therefore a long distance call. Your call gets passed off to another switch, which looks at the 202 and connects you to a switch in Washington, DC. Another switch recognizes the 456 exchange and ultimately your call is connected to the White House by a series of switches, each of which knows only some of the information needed to route your call.

Routing telephone calls is made simple because telephone numbers are *hierarchical*. Likewise, IP addresses are hierarchical. IP addresses (such as `205.214.169.35`) consist of 4 octets (8-bit values), where the first octets identify a network, and the last octet identifies a particular *host* machine. Hence, IP routers can route packets without knowing the locations of all IP addresses.

In our simplified IP implementation, we'll use 4-bit IP addresses, consisting of two 2-bit values. For example, we'll represent the IP address `1.3` with the binary value `01.11`. We'll refer to the first part of the address as the *network*, and the second part as the *host*. Each of our IP packets will contain a destination IP address, source IP address, and payload, as shown below. (Real IP packets contain much more information.)



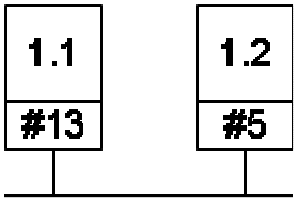
our simplified IP packet



`1.1` sends `110000` to `2.1`

Too Many Addresses

Suppose two machines on an Ethernet segment have been assigned the IP addresses `1.1` and `1.2`. Each of these machines has a network card, and therefore a MAC address, as shown below.



When `1.1` sends the payload `110000` in an IP packet to `1.2`, the resulting Ethernet frame will appear as follows.

Ethernet frame from #13 to #5, containing IP packet from `1.1` to `1.2`

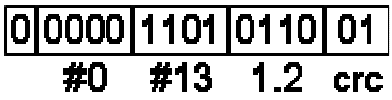
Notice that `1.1` must know `1.2`'s *MAC address* in order to send the packet. Yet, if you reflect on your network experience, you've probably found it's been sufficient just to know the target machine's IP address. How can that be?

ARP (Address Resolution Protocol)

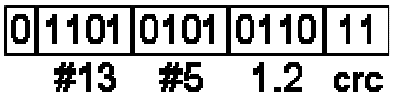
Every machine using IP over Ethernet maintains an ARP table, which associates the MAC address and IP address of other machines recently contacted on the LAN. (You can actually *see* this table by typing "`arp -a`" at a DOS prompt.) But how does the ARP table get populated in the first place? The answer is the Address Resolution Protocol (ARP) itself.

Again, suppose that `1.1` wishes to send an IP packet to `1.2`. Suppose also that `1.1` has not contacted `1.2` recently, and hence it has no entry for `1.2` in its ARP table. `1.1` must first broadcast an ARP request—an Ethernet frame requesting the MAC address of the machine with IP address `1.2`. Other machines ignore this request, but `1.2` sends an ARP reply with its MAC address (and IP address) back to `1.1`. Now `1.1` can add an entry to its ARP table, and can finally send the original IP packet to `1.2`.

The following figures show such an exchange, using the simplified ARP protocol we'll be implementing.

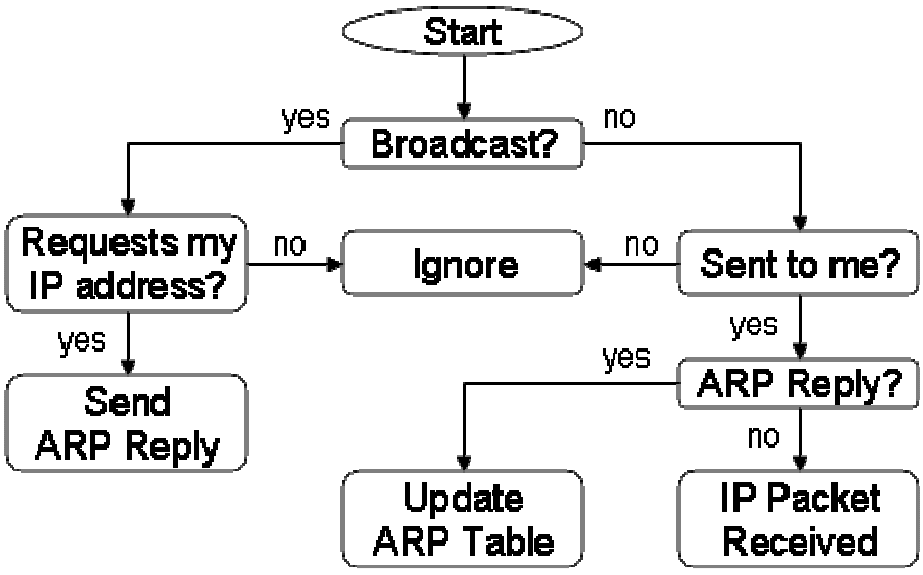


ARP Request for `1.2`'s MAC Address

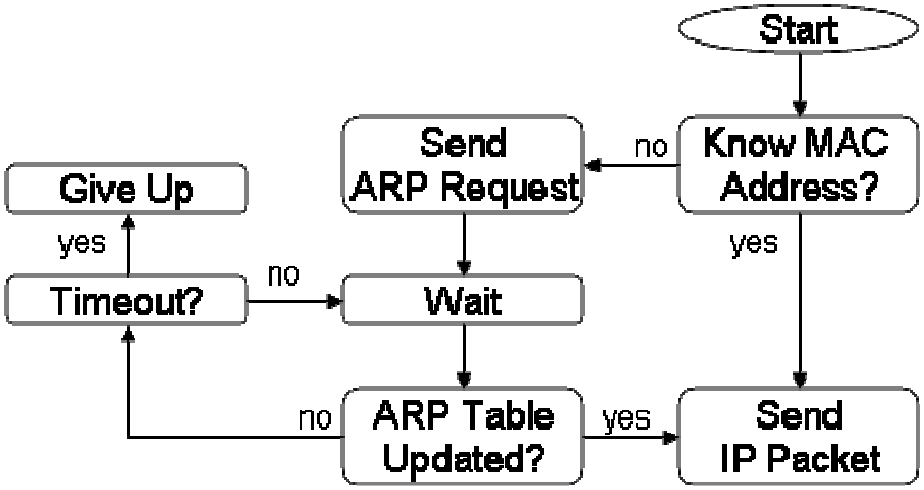


ARP Reply from `1.2`

A received frame might therefore be processed as follows.



Here are the steps we'll need to take when we send an IP packet.



Routing

It's now time to take advantage of those hierarchical IP addresses by considering a simple IP router. Think of an IP packet as a postcard you're sending to a friend, complete with the friend's address and a message. (This isn't going to be a perfect analogy.)

The roads that connect you to your friend are the physical layer. The mail truck that picks up your postcard is a bit like an Ethernet frame. It delivers your postcard to the post office. Note that the address on the post card is *not* the address of the post office (unless you're writing to a friend who lives at the post office).

The post office is like a router, and your local post office is like the gateway router on your LAN. A worker at the post office examines the address on your postcard, and identifies the next step along the route to the recipient. Your original postcard (with its original address) is put on another mail truck (another Ethernet frame or similar), destined for another mail sorting facility (router). After many such sorting facilities and mail trucks/planes, your postcard reaches your friend's local post office (his/her gateway), which places the postcard on a mail truck driving to your friend's house.

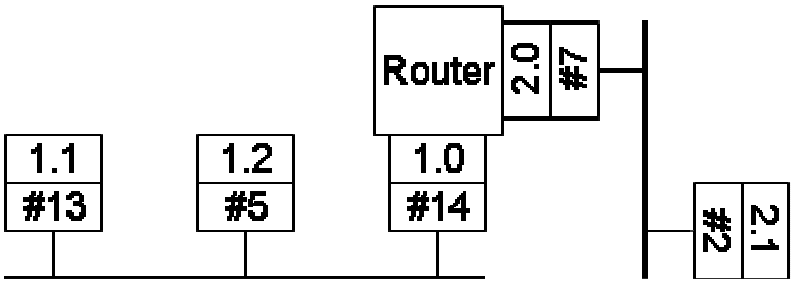
Gateways and Subnets

Like a switch (and unlike a post office), a router has two or more network cards, each connected to a different network segment. Recall that a switch is a layer 2 device, which learns the locations of MAC addresses and forwards frames accordingly. A router, on the other hand, is a layer 3 device, routing packets by destination IP address. To send a packet to another network, your computer needs to know your *gateway router's IP address*. When you need to send a packet to another network, you will send it in an Ethernet frame addressed to the gateway. But if the packet's recipient is on the same network, it would be silly to send it first to the router. Therefore, your computer is also configured to know which *network* it's on. Real computer's keep track of this information using a *subnet mask*. If your IP address is `192.168.1.2`, a subnet mask of `255.255.255.0` tells you that machines whose addresses begin with `192.168.1` are on the same subnet.

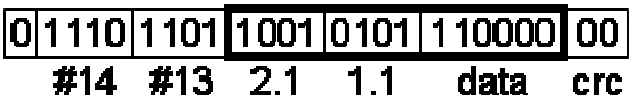
Now, when you want to send a packet to a computer on the same network, you'll use your ARP table and send a packet directly to the target machine's MAC address. But when you want to send a packet to a computer on another network, you'll use your ARP table to send a packet to your gateway.

A Simple Router

Consider the following diagram, and suppose that `1.1` wishes to send the message "110000" to `2.1`.



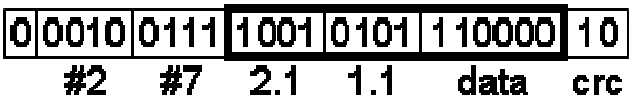
Since `2.1` is not on network 1, `1.1` sends the packet to the router in the following Ethernet frame.



Next, the router uses its routing table to decide which of its network interfaces to use in order to forward the packet. The simple routing table we'll be implementing will look something like this.

To reach this network ...	... Use this network interface
1	#14
2	#7

The packet is forwarded by network interface #7 as follows:

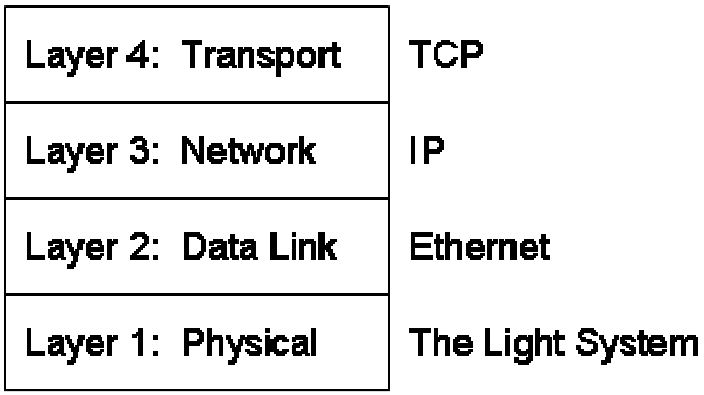


Layer 4: TCP

IP, our layer 3 protocol, provided us with the logical addressing scheme required to route data across the Internet. Routers (layer 3), switches (layer 2), hubs (layer 1), and other network elements generally do not concern themselves with the upper layers of the network protocol world. The remaining layers we'll be studying are typically handled only inside the host machine—in other words, your computer.

Layer 4 is the *transport layer*, and it addresses two essential but largely unrelated issues:

- inter-process communication
- reliable transport of large amounts of data



Inter-Process Communication

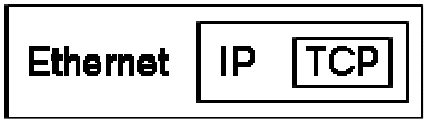
As you know, your computer lets you exchange email, access the web, transfer files, and run many other network applications all at once. Each application runs on your computer in the form of one or more processes. How can each of these processes access the network at the same time, when your network card can only transmit one packet at a time? Also, all packets your computer receives share the same destination IP address. When your computer receives one of these packets, how does it know which process needs it?

The answer to this last question is probably quite obvious. People living in an apartment complex share the same street address, yet each receives their own mail thanks to the magic of apartment numbers. Someone (mailman, apartment employee, or resident) has to examine each mail item's apartment number to place it in the appropriate apartment's mailbox. Notice that large postal sorting facilities couldn't care less about apartment numbers; these extra numbers are only meaningful when the mail actually reaches the apartment building.

Likewise, each process in your computer is assigned a unique number when it opens a network connection. This number is called a *port*. And it's the transport layer's task to use these port numbers to route received packets to the appropriate process (and to queue up and send packets to the network one at a time).

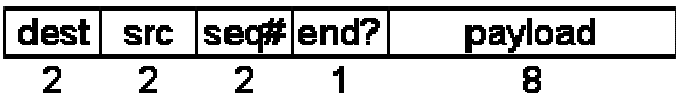
TCP Packets

We'll be studying the most common layer 4 protocol: TCP (Transmission Control Protocol). IP packets will now need to carry destination and source port numbers, along with other TCP-specific information. Therefore, we'll bundle all of this new information in a TCP packet, and send it as the payload of our IP packets.



a TCP packet embedded inside an IP packet, embedded in an Ethernet frame

Real port numbers range as high as 65535, and are therefore represented as 16-bit values. In our simplified TCP protocol, we'll use 2-bit values to identify ports. Therefore, our TCP packets will look like the following. (For now, we'll ignore the fields labelled "seq#" and "end?" in the picture below.) The payload of our simplified TCP packets will be a single character, represented by its 8-bit ASCII value.

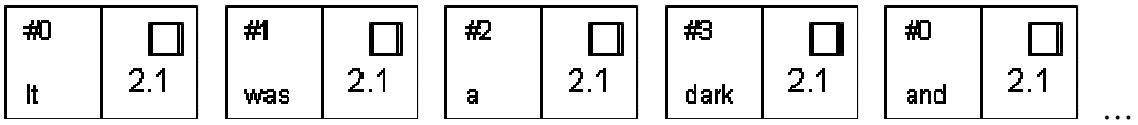


Long Messages, Small Packets

Although real Ethernet frames and IP packets can be quite large, there are a number of practical reasons why there must be a maximum packet size. First, a 100-megabyte packet would prevent other processes from accessing the network for an excessively long period of time. Second, the larger a packet is, the more likely an error occurs during its transmission. It would be unfortunate to have to re-send all 100 megabytes just because an error occurred in the last few bits. Therefore, if we wish to send a long message, we must break it down into smaller packet-sized segments.

But this means the receiving party will need to hold on to all those message segments it receives until the final segment arrives. Furthermore, the receiving party will need to know when it has received the last segment. For this reason, TCP packets contain a 1-bit *end-of-message flag*.

Using many IP packets to send a long message over the Internet is a bit like sending an entire book to a friend, writing each small part of the book on a different postcard, and mailing them one at a time. What if the postcards arrive out of order (as they almost certainly will)? What if some postcards don't arrive at all?



As you can see, one of the keys to the problem is to include a *sequence number* with each TCP packet, which will ensure that the receiver can put the message back in order. No matter how many bits we use for this field, there will always be someone who decides to send a message requiring higher sequence numbers. Therefore, TCP uses a fixed number of bits, and may re-use sequence numbers in the course of sending a single message. In our simplified TCP implementation, we'll use 2-bit sequence numbers, and re-use sequence number #0 after sending packet #3.

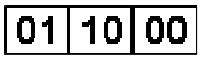
Acknowledgements

The use of TCP acknowledgement packets can help two parties establish that all packets are received. Whenever a regular payload-bearing packet is received, the receiver sends back an acknowledgement packet. If a sent packet is not acknowledged within some timeout period, the sender re-transmits the original packet. In our simplified protocol, we will have two kinds of TCP packets. Acknowledgement packets ("acks") will contain 2-bit destination and source port numbers, along with a 2-bit sequence number. Normal payload-bearing TCP packets will contain all this information, plus a 1-bit end-of-message flag and 8-bit payload representing a single character.

payload-bearing TCP packet *TCP ack*

The diagram below shows a TCP packet sent from port 1 of one machine to port 2, probably (but not necessarily) on some other machine. (Note that the sender and receiver could use the same port number, as long as they have different IP addresses.) This packet has sequence #0, which means it might be the first packet in the message, but could be the 5th, 9th, etc. The packet's end-of-message flag is 1, meaning this is the last packet in the message. The payload contains the ASCII value for the letter A. The smaller packet below is the corresponding acknowledgement packet sent back from the recipient, indicating that the original packet was received.

TCP packet #0, carrying the letter A



corresponding ack

Exchanging TCP Packets

In our simplified TCP implementation, the sender of a message string will send the first character in packet #0 and wait for ack #0. If none is received, or if the wrong ack is received, packet #0 is re-sent. When ack #0 is received, packet #1 is sent, and the sender waits for ack #1. (Note that a real TCP implementation would send a "window" of several packets before pausing to wait for any to be acknowledged.)

The receiver keeps track of the portion of the message received so far, and determines the next sequence number expected. If one character has been received, then the receiver has already seen packet #0, and is now expecting packet #1. When #1 arrives, the receiver adds the character to the message, sends ack #1, and waits for packet #2 (if the end-of-message flag was not set). If any other packet had arrived, the receiver would ignore it, and re-send ack #0. (Of course, real TCP software must simultaneously be able to receive data from multiple sources.)

TCP vs. UDP

Other than TCP, the most common layer 4 protocol is UDP (User Datagram Protocol). Like TCP, UDP handles inter-process communication through the use of port numbers. Unlike TCP, UDP does not aim to provide reliable communication. This means that UDP doesn't use the time-intensive exchange of acknowledgement packets, and is therefore ideal for applications that require high performance and can tolerate packet loss (such as streaming video).