((There is a lot of disjoint information in these notes; the notes themselves do not closely follow the lecture slides for today. I strongly recommend reading Chapter 3 of "TCP/IP Sockets in C: Practical Guide for Programmers / Edition 2" by Michael J. Donahoo. It is extremely informative; today's lecture was more of an explanation of that reading than an actual stand-alone lecture, and the notes will not make much sense unless you have read the chapter first.))

When transmitting information, the data move along layers. Among these seven layers, TCP / UDP are on level 4, the transport layer; IPv4 / IPv6 are on level 3, the network layer; the device driver and hardware are on levels 2 and 1, the datalink and physical layer; everything else (session, presentation, application layer) is lumped together under an "application" layer.

Sockets are between layers 4 and 5. The sockets are there so that our applications do not need to concern themselves with the transmission of data. Everything below the sockets is concerned with data transmission and interpretation; everything above sockets is concerned with user-end information.

Sockets use an API to link between protocol layers. These API links are called buffers; these buffers do not always work well together.

Protocols are linked together as protocol families. In a computer family, a protocol family will always have a "PF_" prefix. There are 42 protocol families, and they include PF_BLUETOOTH, PF_UNIX, PF_INET, and others.

Address families also exist, with the prefix "AF_". These used to be different to protocol families, but in modern times, the address family and the protocol family are the same thing. For example, AF_INET is exactly the same as PF_INET, and the two are interchangeable.

Remember! In a UNIX world, everything is a "file", including sockets. If sockets are left open, they will eat up system resources. Once a socket is done being used, always call a close() on the socket to shut it down and free up its memory space. Failing to do so creates a memory leak.

Inside of <sys/socket.h> is a structure called a "sockaddr". This is a Socket Address struct. The socket address contains two unsigned characters for sa_len and sa_family (socket address length and socket address family) and a char array of size 14, sa_data[14]

IPv4 used to specify the addresses of a computer as an unsigned 32 bit integer. Additionally, the socket address is an unsigned short integer; this allows for addresses in the range of 0 to 2^16 – 1, or 65536 possible addresses. This means that a computer only has 65536 addresses actually available. Unfortunately, in 2011, we ran out of IPv4 addresses, so we now use IPv6. (In this class, we will work with IPv4.)

An IP address, as described by the "sockaddr" struct, is a 16 bit number. The common "127.0.0.1" format of an IP address is a collection of four 8-bit numbers; this means that each part of an IP address can be, at most, 255. The highest IP address is 255.255.255.255 and the lowest is 0.0.0.0.

There are some additional structures available, and these have been described in the TCP/IP textbook in chapter 3. Again, I strongly advise reading the chapter.

As a socket illustration, suppose there is a mailbox. We are supposed to receive information in the mailbox, but we have to check the mailbox in order to receive the information. This is similar to a socket: if we intend to get information, we must have open sockets, and we must be listening on the port. Our mailbox is like a server. We don't need to know who is sending us mail, but the person who is sending mail must know the address. Likewise, a server does not need to know which clients want to contact it, but a client must know where the server is and what port it can talk to before it can send its information.

When a server receives a socket address on any interface at any port, the computer (if it is listening) will catch the address and use a bind() call to assign it to a port. The address, once received, is moved to two different socket address variables: one interprets it as a long variable, and one interprets it as a short. This is because the address is sometimes sent backwards, in little-endian format.

In TCP, there are receivefrom() and sendto() functions. These are used to send information to sockets, and receive information from sockets. UDP uses the send() function, NOT the sendto() function.

If we ping google.com, we may get about 16 different IP addresses. However, if we use the "dig" commands instead, we can learn more about google's IP addresses.

If we are on our own machine and want to learn our own IP address, there are many ways to do it! "ifconfig" works on Linux machines, while "ipconfig" is best for windows machines.