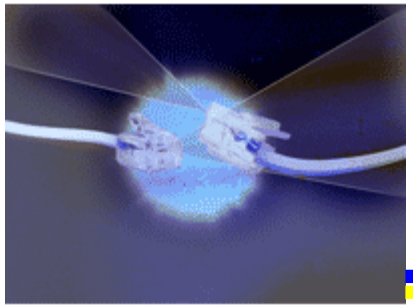
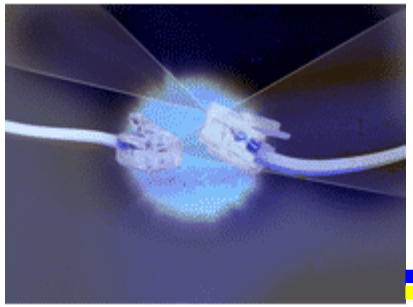


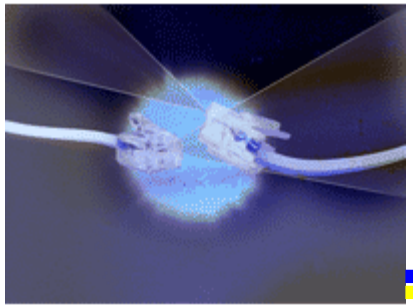
Network Software Implementations

- Number of computers on the Internet doubling yearly since 1981, nearing 200 million
- Estimated that more than 600 million people use the Internet
- Number of bits transmitted over the Internet surpassed volume in voice phone network in 2001
- Made possible by a good network architecture, plus an adaptable interface for new network applications



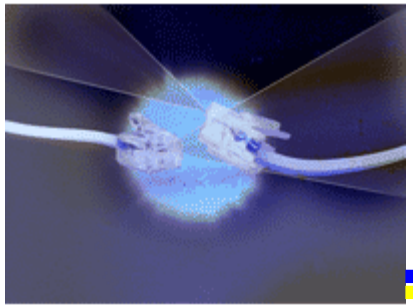
OS Network APIs

- Application programs interface with an OS networking subsystem through an API
- Some of these APIs have become widely adopted
  - Best example may be the *socket interface* introduced in Berkeley Unix
  - Implemented in many other operating systems, including most or all variants of Unix and Windows
  - Widespread availability makes it easy to port applications between different OSes (at least the networking portion of the app)



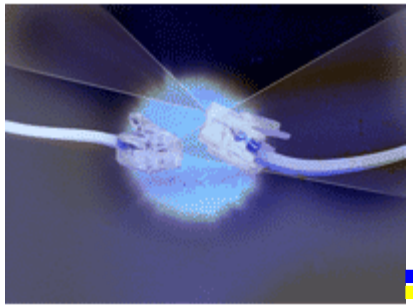
Protocols vs. APIs

- Each protocol supported by the OS provides a set of abstract *services*.
- The network API provides a *syntax* by which those services can be invoked.
- Desirable to choose a syntax that is applicable across different protocols



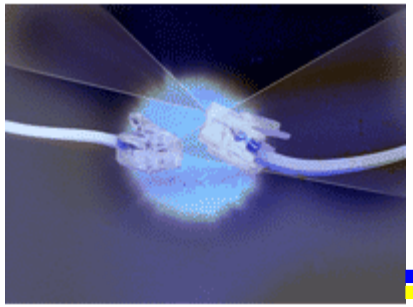
The socket API

- The *socket* is the “point” where a local application process attaches to the network
- Some common operations provided by APIs (see */usr/include/sys/socket.h)*:
  - Create a socket
    - int socket(int domain, int type, int protocol)
    - *domain* examples: PF_INET, PF_UNIX, PF_PACKET
    - *type* examples: SOCK_STREAM, SOCK_DGRAM
    - *protocol* examples: PF_INET, PF_BLUETOOTH
    - Returns a handle that refers to the socket, used on subsequent API calls



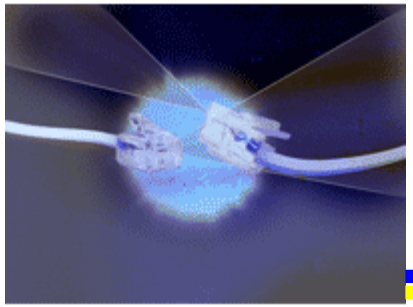
The socket API (cont.)

- Open the socket (TCP, server side)
  - int bind(int socket, struct sockaddr * address,
    int addr_len)
    Assigns a local address to the socket
  - int listen(int socket, int backlog)
    Indicates that the server is ready to accept connections on the socket; backlog indicates maximum number to buffer
  - int accept(int socket, struct sockaddr * address,
    int * addr_len)
    Waits for a new connection request. This returns a new socket that can be used to communicate with the client; the original socket can be used to continue to listen for new connections. Client's address is also returned



Socket addresses

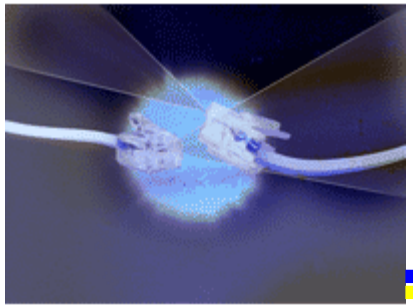
- Each socket is identified by an address that includes both the machine identifier and a *demux key*.
- *struct sockaddr* is actually a union of different socket address sub-structures, each with fields appropriate to the protocol
  - For the Internet protocol family (PF_INET), the *sockaddr_in* structure contains the IP address of the server, as four unsigned bytes, and a 16-bit *port* number that identifies the application.
  - Ports can be well-known pre-assigned numbers (like port 80 for HTTP - see */etc/services* for a list), chosen by agreement between the client and server, or can be chosen randomly and somehow communicated OOB.



Client socket open

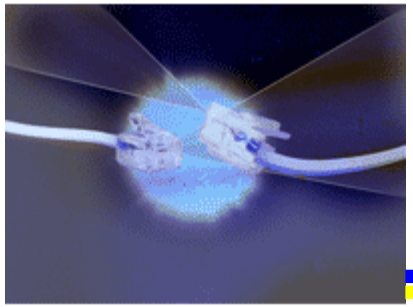
- Open the socket (TCP, client side)
  - int connect(int socket, struct sockaddr * address,
    int addr_len)

    Opens the socket and connects it to the specified server address. Blocks until connection established.
- Send data over the socket
  - int send(int socket, char * msg, int msg_len,
    int flags)
- Receive data from the socket
  - int recv(int socket, char * buffer, int buf_len,
    int flags)



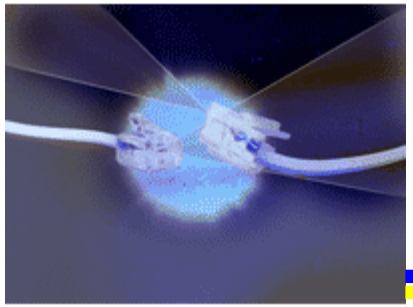
Example Application

- Implements a network-based *talk* (simple chat program)
- Client details
  - Uses *gethostbyname*() to translate from a host name (i.e. *esus.cs.montana.edu*) to an IP address (i.e. 153.90.199.47).
  - Uses a #defined constant port number – to use a well-known service, you can call *getservbyname*().



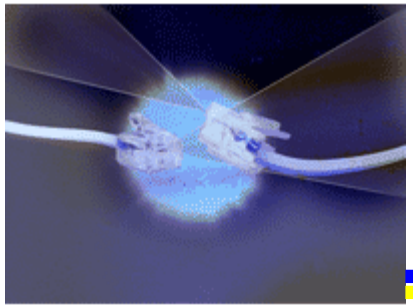
Example Application (cont.)

- Server details
  - Uses all zeroes for local IP address; causes OS to fill in value.
  - Specifies same #defined constant for port
  - After creating socket, binding address, and listening, it accepts a connection, reads data until the connection closes, then goes back to *accept*().
    - Not a very good server design – why?



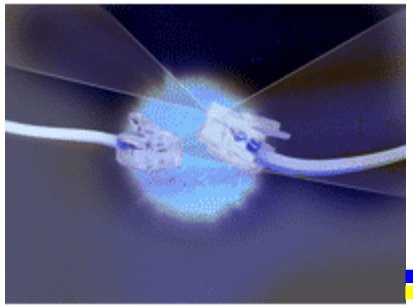
Byte Ordering

- Notice calls in example app to *htons*()
  - Abbreviation for “Host to Net Short”
  - API also includes *htonl*(), *ntohs*(), and *ntohl*()
  - Functions convert multi-byte numeric values between local byte order and network byte order (MSB first)
  - On big-endian machines, these are just macros that return the input value
  - Easy to forget, and hard to find errors!



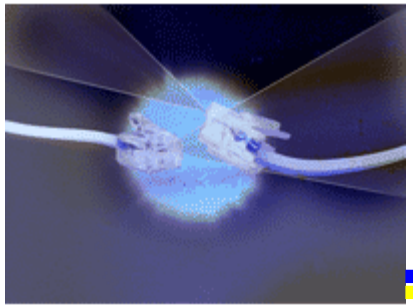
Protocol Implementation Issues

- Process models
  - *Process-per-protocol*
    - Conceptually simpler – isolates each protocol
  - *Process-per-message*
    - Better performance – fewer context switches
- Message buffering
  - Copying messages between buffers can be a relatively costly operation
    - Memory is still fairly slow compared to processor
  - Many APIs include a buffer abstraction to avoid copying buffers



Performance Measures

- Bandwidth (throughput) – measure of theoretical (practical) maximum data rate in units of bits/time (i.e. 10 Mbps for Ethernet)
- Latency (delay) – measure of end-to-end transit time
  - One-way or round-trip
  - Composed of propagation delay, time to transmit data, and queuing delays
  - Latency = Propagation + Transmit + Queue
    = distance/speed of light + size / bandwidth + queue



Response Time Graph

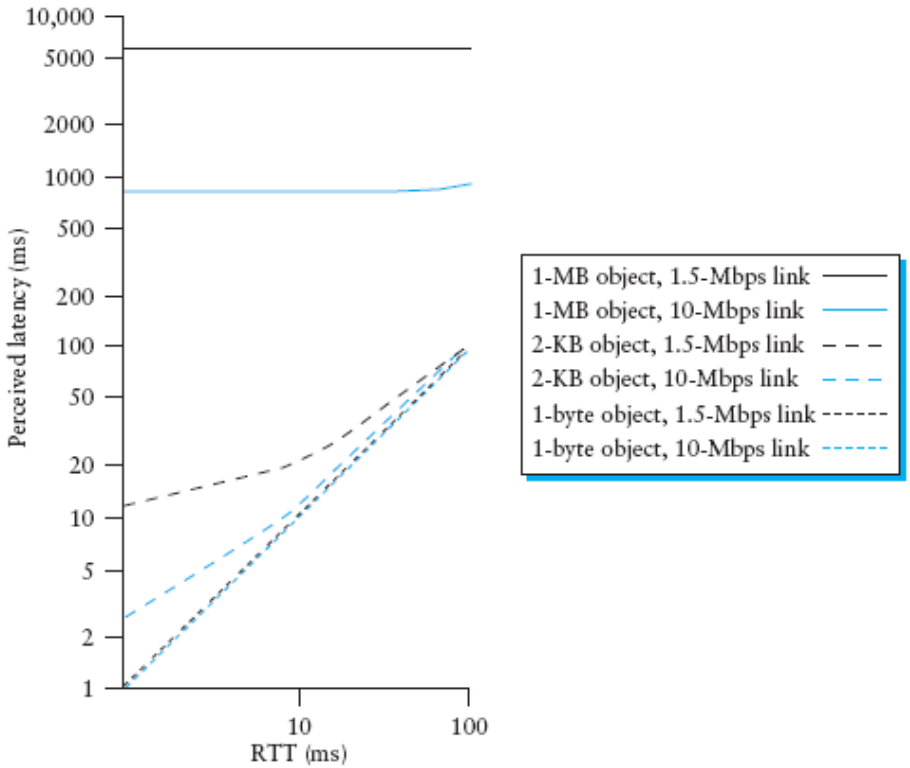
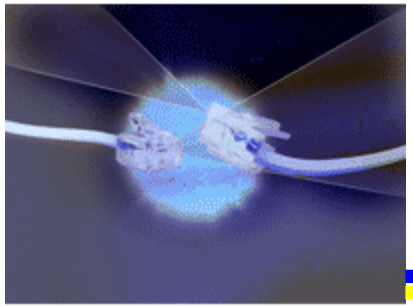
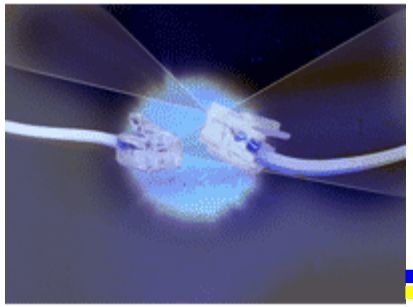


Figure 1.21 Perceived latency (response time) versus round-trip time for various object sizes and link speeds.



Delay x Bandwidth Product

- Measure of the amount of data the link can hold
  - One-way latency of 10 ms * bandwidth of 10 Mbps = 100,000 bits, or 12,500 bytes
- Also measures how many bits source must send before first bit reaches sink
- Multiply by 2 if sender needs to hear from receiver (for round-trip time, RTT) – number of bits “in flight”



High-Speed Networks

- Bandwidth continues to increase rapidly
- Propagation delay is fixed, regardless of bandwidth
- The RTT becomes much more significant as the time to actually transmit the data (size / bandwidth) shrinks