Sockets

- programming using sockets
- socket types
- socket operations
- sockets example (in C)

programming using sockets

- networking programs communicate (send and receive) over a special programming construct called a *socket*
- a socket is an endpoint of communication
- communication can only occur between two (or more) sockets
- a socket is an abstract data type (ADT): an opaque type that supports a set of operations

socket types

- in common usage, a socket is either a TCP socket or a UDP socket
  - TCP sockets are STREAM sockets
  - UDP sockets are DATAGRAM sockets
- there are more specialized (and less portable) socket types, including raw sockets (raw frames on the interface) and packet sockets (IP datagrams)

socket types: address families

- a socket, when first created, is not associated (*bound*) to any address
- a socket can only be bound to one type of address, e.g. IPv4 (INET) or IPv6 (INET6)
  - IPv6 sockets usually also support IPv4
- we may *bind* a socket to a local address and port number, and/or *connect* a socket to a remote address and port number
  - UDP is connectionless, but *connect* can be used to specify a remote address that is used by default

socket types: connection status

- a socket, when first created, is not connected
- *connect* works as a client to connect to a server
- *accept* works as a server to accept incoming connections
  - *accept* creates a new socket for each connection
  - the socket used in *accept* must be in the *listen* state
- once done, we should close connections
  - OS closes any open sockets when programs exit

socket operations: create and close

- *socket* creates a new socket
  - *accept* creates new sockets given a listen socket
  - *listen* puts an unconnected socket in listen state
- *close* closes a socket
- *shutdown* can “half close” a socket, by sending a FIN but allowing subsequent receives

socket operations: bind and connect

- *bind* specifies the local port number, and optionally the local address
- *connect*:
  - for TCP, does the 3-way handshake
    - *accept* on the server side responds
  - for UDP, specifies the remote address

socket operations: send and receive

- *send* sends a buffer of specified length
  - *sendto* is like *send*, but for unconnected sockets, so also specifies an address
- *recv* receives into a buffer up to the specified length
  - *recvfrom* is like *recv*, but for unconnected sockets, so reports the address from which the datagram was received.
- all return the number of bytes sent/received
  - or -1 for errors, or 0 for a closed connection or other special situations

sockets example: client

```
int s;
if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  error("socket");
hostentry = gethostbyname("example.com");
if ((hostentry == NULL) || (hostentry->h_addr_list == NULL))
  error("gethostbyname");
memset (&sin, 0, sizeof (sin));
sin.sin_family = AF_INET;
memcpy(&(sin.sin_addr), hostentry->h_addr_list[0],
       hostentry->h_length);
sin.sin_port = htons(portnumber);
if (connect(s, (struct sockaddr *)(&sin), sizeof(sin)) < 0)
  error("connect");
if (send(s, buf, sizeof(buf), 0) < 0) error("send");
if (close(s) < 0) error("close");
```

sockets example: server

```
int passive, session;
if ((passive = socket(PF_INET, SOCK_STREAM, 0)) < 0)
  error("socket");
memset (&sin, 0, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(portnumber);
sin.sin_addr.s_addr = INADDR_ANY;
if (bind(passive, (struct sockaddr *) (&sin), sizeof (sin)) != 0)
  error("bind");
if(listen(passive, 5) < 0) error("listen");
int adrsize = sizeof (sin);
while ((session = accept(passive, sap, &adrsize)) >= 0) {
    count = recv(session, buf, BUFSIZE - 1, 0);
    if (close(session) < 0) error("child close");
    adrsize = sizeof (sin);
}
```