Introduction

- Sockets are a local and remote OS IPC abstraction defined in 4.2 BSD UNIX and beyond
  - Now part of most major operating systems, including Windows and Win32 systems

- Sockets were originally developed for TCP/IP protocols
  - Later generalized to include other protocol families
    * e.g., Novell IPX/SPX, TP4, ATM

- Socket routine control communication between processes and protocols
  - Also provide buffering between synchronous application domain and the asynchronous kernel domain

The Socket Interface (cont’d)

- Originally, sockets were implemented as a set of system calls
  - For efficiency, they were tightly-coupled with the BSD networking architecture in the OS kernel

- Recent versions of sockets are implemented as a library of functions in user-space
  - e.g., SVR4 UNIX and Win32

- User-space implementations improve flexibility and portability at the expense of some performance

Network Programming with Sockets

ECE 255

Douglas C. Schmidt

http://www.ece.uci.edu/~schmidt/
schmidt@uci.edu

University of California, Irvine
Communication Domains (cont’d)

- **UNIX domain (PF_UNIX)**
  - Communicate only with a process on the same machine
  * Uses UNIX filenames for rendezvous between client and server processes
  - Really a form of intra-machine IPC, similar to SVR4 STREAM pipes
  * Supports both reliable (SOCK_STREAM) and unreliable (SOCK_DGRAM) local IPC
  * Used for local X-windows traffic...

- **Internet domain or TCP/IP (PF_INET)**
  - Communicate across network or on same machine (uses “dotted-decimal Internet addresses”)
    * e.g., ”128.195.1.1 @ port 21”
  - General-purpose addressing, but existing versions don’t scale well due to fixed-sized addressing
    * This is fixed in IPv6
  - e.g., TCP, UDP, IP, ftp, rlogin, telnet

- **Xerox XNS (later evolved into Novell IPX)**
  - SPP, PEX, IPD

- **ISO OSI**
  - e.g., TP4-TP1, CLNS, CONS

Communication Domains (cont’d)

- **UNIX domain (PF_UNIX) (cont’d)**
  - 4.3 BSD and SunOS 4.1.x implement pipes via “lobotomized” connection-oriented Unix domain socket protocol implementations
  - SVR4-based UNIX systems use the STREAMS facility
    * In general, UNIX domain sockets have been subsumed by STREAM-pipes and **connect** in SVR4
  - Not surprisingly, Win32 does not support UNIX domain sockets


Stream Socket

- **Type of service**
  - Reliable (i.e., sequenced, non-duplicated, non-corrupted) bi-directional delivery of byte-stream data

- **Metaphor**
  - A “network pipe”

- *e.g.*, 
  ```
  int s = socket (PF_INET, SOCK_STREAM, 0); /* Note, s is an internal id...*/
  ```

- **Note**, we’ll use `int` as the socket type, although Win32 uses SOCKET...

Reliably-delivered Message Socket

- **Type of service**
  - Reliable datagram

- **Metaphor**
  - Sending a registered letter

- *e.g.*, 
  ```
  int s = socket (PF_NS, SOCK_RDM, 0);
  ```

Socket Types

- There are five Types of Sockets
  1. Stream Socket
  2. Datagram Socket
  3. Reliably-delivered Message Socket
  4. Sequenced Packet Stream Socket
  5. Raw Sockets

- `SOCK_STREAM` and `SOCK_DGRAM` are the most common types of sockets...

Datagram Socket

- **Type of service**
  - Unreliable, unsequenced datagram

- **Metaphor**
  - Sending a letter

- *e.g.*, 
  ```
  int s = socket (PF_INET, SOCK_DGRAM, 0);
  ```
Raw Sockets

- **Type of service**
  - Allows user-defined protocols that interface with IP
  - Requires root access

- **Metaphor**
  - Playing with an erector set…:-)

- *e.g.*,
  ```c
  int s = socket(PF_INET, SOCK_RAW, 0);
  ```

Sequenced Packet Stream Socket

- **Type of service**
  - Reliable, bi-directional delivery of record-oriented data

- **Metaphor**
  - Record-oriented TCP (*e.g.*, TP4 and XTP)

- *e.g.*,
  ```c
  int s = socket(PF_NS, SOCK_SEQPACKET, 0);
  ```

Socket Addresses (cont’d)

- **General Format**
  ```c
  struct sockaddr { u_short sa_family; char sa_data[14]; }
  ```

- **UNIX Domain**
  ```c
  struct sockaddr_un {
    short sun_family; char sun_path[108];
  }
  ```

- **Internet Domain**
  ```c
  struct in_addr { unsigned long s_addr; }
  struct sockaddr_in {
    short sin_family; u_short sin_port;
    struct in_addr sin_addr; char sin_zero[8];
  }
  ```

Socket Addresses

- UNIX supports multiple communication domains, protocol families, and address families
  - The socket API provides a single address interface for all these families

- The type of `sockaddr` structure used with `accept`, `bind`, `connect`, `sendto`, and `recvfrom` differs according to the domain (UNIX vs. Internet vs. XNS)

- The addressing API has a somewhat confusing and error-prone design
  - Motivation was to save space for the “common case”…
Socket Operations

- Local context management

  ```c
  int socket (int domain, int type, int protocol);
  int bind (int fd, struct sockaddr *, int len);
  int listen (int fd, int backlog);
  int close (int fd);
  int getpeername (int fd, struct sockaddr *, int *len);
  int getsockname (int fd, struct sockaddr *, int *len);
  ```

- Connection establishment and termination

  ```c
  int connect (int fd, struct sockaddr *, int len);
  int accept (int fd, struct sockaddr *, int *len);
  int shutdown (int fd, int how);
  ```

- Option management

  ```c
  int ioctl (int fd, int request, char *arg);
  intfcntl (int fd, int cmd, int arg);
  int getsockopt (int, int, int, char *, int *);
  int setsockopt (int, int, int, char *, int);
  ```

Socket Addresses (cont’d)

- General usage for Internet-domain service:

  ```c
  struct sockaddr_in addr;
  ```

  ```c
  memset (&addr, 0, sizeof addr);
  addr.sin_family = AF_INET;
  addr.sin_port = htons (port_number);
  addr.sin_addr.s_addr = htonl (INADDR_ANY);
  ```

  ```c
  if (bind (sd, (struct sockaddr *) &addr, sizeof addr) == -1)
      ...;
  ```

- Note the use of a cast

  - In C++, this whole mess can be cleaned-up via inheritance and dynamic binding!

Socket Operations

- Data transfer

  ```c
  int read (int fd, void *buf, int len);
  int write (int fd, void *buf, int len);
  int send (int fd, void *buf, int len, int flags);
  int recv (int fd, void *buf, int len, int *flags);
  int readv (int fd, struct iovec [], int len);
  int writev (int fd, struct iovec [], int len);
  int sendto (int fd, void *buf, int len, int flags, struct sockaddr *, int len);
  int recvfrom (int fd, void *buf, int len, int flags, struct sockaddr *, int len);
  int sendmsg (int fd, struct msghdr *msg, int flags);
  int recvmsg (int fd, struct msghdr *msg, int flags);
  ```

- Event demultiplexing

  ```c
  int select (int maxdp, fd_set *rdfs, fd_set *wfds, fd_set *exfds, struct timeval *);
  ```

Connection-oriented Socket Usage

- 1: Passive

  ```c
  socket() bind() listen() accept() send()/recv() close()
  ```

- 2: Active

  ```c
  server
  socket() bind() connect() send()/recv() close()
  ```

- 3: Service Processing

  ```c
  server
  socket() bind() listen() accept() send()/recv() close()
  ```

  ```c
  client
  socket() bind() connect() send()/recv() close()
  ```
Client and Server Operations

- socket
  - Creates and opens a socket and returns a descriptor
  - `int s = socket (int domain, int type, int protocol);`
    * `domain` → PF_UNIX, PF_INET
    * `type of service` → SOCK_STREAM, SOCK_DGRAM
    * `protocol` → generally 0, but could be TCP, VMTP, NETBLT, XTP

- Note, this call only fills in the first part of the 5-tuple association

Client and Server Operations (cont’d)

- close
  - Close a socket
  - `int close (int s);`
    * Note, there are subtle semantics related to “grace termination...” of protocols

- shutdown
  - Shutdown part or all of full-duplex connection
  - `int shutdown (int s, int how);`
    * `how` is 0, then further receives will be disallowed
    * `how` is 1, then further sends will be disallowed
    * `how` is 2, then further sends and receives will be disallowed
    * Note, `shutdown` does not close the descriptor...

Connectionless Socket Usage

2: ACTIVE ROLE
socket()  bind() (optional)
sendto()  recvfrom()  close()

1: PASSIVE ROLE
socket()  bind()
sendto()  recvfrom()  close()

3: SERVICE PROCESSING

CLIENT  SERVER  NETWORK

Client and Server Operations (cont’d)

- bind
  - Associates a local address (e.g., an IP address, address family, and port number) to an unnamed socket
  - `int bind (int s, struct sockaddr *addr, int addrlen);`
    * `addr` → local address (e.g., points to an Internet addr or a UNIX domain addr)
    * `addrlen` → length of address
    * Note
      * `bind` is not necessary for clients (which implicitly allocate transient port numbers)

    * The address INADDR_ANY is a wildcard for any server host/network interface

    * Always “zero-out” the address structure before using it...
**Typical Client Operations**

- **connect**
  - Specify foreign/remote destination address (e.g., IP/port numbers) and joins two sockets for I/O:
  ```
  int connect (int s, struct sockaddr *addr, int addrlen);
  *
  addr  -> address of remote client
  * addrlen  -> length of address
  ```

- **getsockname**
  - Returns address info describing the local socket
  ```
  getsockname (int s, struct sockaddr *addr, int *addrlen);
  *
  addr  -> address of local binding
  * addrlen  -> length of address
  ```

**Typical Server Operations**

- **listen**
  - Set the length of a TCP passive open queue, places the socket into "passive-mode"
  ```
  listen (int s, int backlog);
  *
  backlog  -> specifies how many connection requests can be queued
  ```

- **accept**
  - Returns a unique descriptor to the next available completed connection from the connection queue
  ```
  int accept (int s, struct sockaddr *addr, int *addrlenptr);
  *
  addr  -> address of remote server
  * addrlenptr  -> ptr to length of address
  *
  Returns new socket descriptor specifying the full association
  ```

  - Notes:
    1. Server may decide to reject connection only after first accepting it!
    2. *addr and *addrlenptr may be 0...
Typical Server Operations

• select
  – Synchronous event demultiplexer that queries the status of a set of socket descriptors under timer control:

    ```c
    int select (int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
    ```

    * maxfdp1 → max file descriptor to consider plus 1
    * readfds → set of descriptors to check for reading and incoming connections
    * writefds → set of descriptors to check for writing and outgoing connections
    * exceptfds → set of descriptors to check for urgent data
    * timeout → length of time to wait for activity on the descriptors

• send
  – Send a message to a socket:

    ```c
    int send (int s, char *msg, int len, int flags);
    ```

    * msg → buffer of data to send
    * len → length of buffer
    * flags

  1. MSG_OOB → send out-of-band data on sockets that support this operation

• Note that neither write nor send are guaranteed to write all the bytes!

Data Transfer Operations

• write
  – Send a message to a socket:

    ```c
    int write (int s, char *msg, int len);
    ```

    * msg → buffer of data to send
    * len → length of buffer

• sendto
  – Send a datagram message from a UDP socket:

    ```c
    int sendto (int s, char *msg, int len, int flags, struct sockaddr *addr, int addrlen);
    ```

    * addr → address of remote server
    * addrlen → length of address

• recvfrom
  – Receive a datagram message from a UDP socket:

    ```c
    int recvfrom (int s, char *buf, int len, int flags, struct sockaddr *addr, int *addrlenptr);
    ```

    * addr → address of remote server
    * addrlenptr → ptr to length of address

• read
  – Receive a message from a socket:

    ```c
    int read (int s, char *buf, int len);
    ```

• recv
  – Receive a message from a socket:

    ```c
    int recv (int s, char *buf, int len, int flags);
    ```

    * flags

  1. MSG_OOB → read any out-of-band data present on the socket, rather than the regular in-band data

  2. MSG_PEEK → “Peek” at the data present on the socket; the data are returned, but not consumed, so that a subsequent receive operation will see the same data
Option Management (cont’d)

- Arguments for setsockopt and getsockopt
  - `level` → protocol level (e.g., IP, TCP, socket, etc.)
    * e.g., SOL_SOCKET, IPPROTO_TCP, IPPROTO_IP
  - `optname` → name of option
    * e.g., SO_REUSEADDR, SO_ERROR, SO_BROADCAST, SO_SNDBUF, SO_RCVBUF
  - `optval` → value of option
  - `optlen` → length of option

Option Management

- `setsockopt`
  - Sets options on a socket
    - `int setsockopt (int s, int level, int optname, void *optval, int optlen);`

- `getsockopt`
  - Gets options regarding a socket
    - `int getsockopt (int s, int level, int optname, void *optval, int *optlenptr);`

Internet Domain Stream Sockets

- Header file
  ```
  #include <stdio.h>
  #include <sys/types.h>
  #include <sys/socket.h>
  #include <string.h>
  #include <netinet/in.h>
  #include <netdb.h>
  ```
  ```
  #define SRV_PORT 7734 
  #define SRV_ADDR "128.195.13.4" 
  #define STDOUT 1
  #define STDIN 0
  ```
  ```
  int process_msg (int ifd, int ofd);
  ```

Auxiliary Networking Functions

- `gethostname`
  - Returns the primary name of the current host as an ASCII string
    ```
    int gethostname (char *name, int namelen);
    ```

- `gethostbyname/gethostbyaddr`
  ```
  struct hostent *gethostbyname (char *name);
  struct hostent *gethostbyaddr (char *, int len, int type);
  ```

- `struct hostent`
  ```
  struct hostent { 
    char *h_name; /* name of host */
    char **h_aliases; /* alias list */
    int h_addrtype; /* address type */
    int h_length; /* length of addr */
    char **h_addr_list; /* list of addr */
  };
  ```
  ```
  #define h_addr h_addr_list[0]
  ```

- Note, hostnames/host numbers are stored in /etc/hosts
  - Also accessible via DNS...
Internet Domain Stream Sockets

(cont'd)

• Become a passive-mode “server”

    ```c
    int s_server (unsigned short port) {
        struct sockaddr_in name;
        memset ((void *)&name, 0, sizeof name);
        name.sin_family = AF_INET;
        name.sin_port = htons (port);
        name.sin_addr.s_addr = htonl (INADDR_ANY);

        int s_fd = socket (PF_INET, SOCK_STREAM, 0);
        if (s_fd == -1)
            return -1;
        else if (bind (s_fd, &name, sizeof name) == -1)
            return -1;
        else if (listen (s_fd, 5) == -1)
            return -1;
        return s_fd;
    }
    ```

• Read a message with TCP (server)

    ```c
    #include "header.h"
    int main (int argc, char *argv[]) {
        int s_fd = s_server (SRV_PORT);

        if (s_fd == -1)
            perror ("s_server");
        for (;;) {
            int cli_fd = accept (s_fd, 0, 0);

            if (cli_fd == -1)
                perror ("accept");
            else if (process_msg (cli_fd, STDOUT) == -1)
                perror ("process_msg");
            else if (close (cli_fd) == -1)
                perror ("close");
        }
    } /* NOTREACHED */
    ```

Internet Domain Stream Sockets

(cont'd)

• Become an active-mode “client”

    ```c
    int s_client (u_short port, const char *addr) {
        struct sockaddr_in name;
        memset ((void *)&name, 0, sizeof name);
        name.sin_family = AF_INET;
        name.sin_port = htons (port);
        name.sin_addr.s_addr = inet_addr (addr);

        int s_fd = socket (PF_INET, SOCK_STREAM, 0);
        if (s_fd == -1)
            return -1;
        else if (connect (s_fd, (struct sockaddr *)&name,
                          sizeof name) == -1)
            return -1;
        return s_fd;
    }
    ```

• Write a message (client)

    ```c
    #include "header.h"
    int main (int argc, char *argv[]) {
        int status = 1;
        int s_fd = s_client (SRV_PORT, SRV_ADDR);

        if (s_fd == -1)
            perror ("s_client");
        else if (process_msg (STDIN, s_fd) == -1)
            perror ("process_msg");
        else
            status = 0;
        close (s_fd);
        return status;
    }
    ```
Concurrent Server using Select

- Single-threaded concurrent socket server

```c
int main (void) {
    // Create a server end-point.
    int s_fd = s_server (PORT_NUM);
    fd_set temp_fds;
    fd_set read_fds;
    int maxfdp1 = s_fd + 1;

    // Check for constructor failure.
    if (s_fd == -1)
        perror ("server"), exit (1);
    FD_ZERO (&temp_fds);
    FD_ZERO (&read_fds);
    FD_SET (s_fd, &read_fds);

    // Loop forever performing logging server processing.
    for (; ;) {
        temp_fds = read_fds; // Structure assignment.

        // Wait for client I/O events (handle interrupts).
        while (select (maxfdp1, &temp_fds, 0, 0, 0) == -1
            && errno == EINTR)
            continue;

        // Handle pending logging records first (s_fd + 1
        // is guaranteed to be lowest client descriptor).
        for (int fd = s_fd + 1; fd < maxfdp1; fd++)
            if (FD_ISSET (fd, &temp_fds)) {
                int n = handle_logging_record (fd);
                // Guaranteed not to block in this case!
                if (n == -1)
                    perror ("logging failed");
                else if (n == 0) {
                    // Handle client connection shutdown.
                    FD_CLR (fd, &read_fds);
                    close (fd);
                    if (fd + 1 == maxfdp1) {
                        // Skip past unused descriptors.
                        while (!FD_ISSET (-fd, &read_fds))
                            continue;
                        maxfdp1 = fd + 1;
                    }
                }
            }

        // Check for incoming connections.
        if (FD_ISSET (s_fd, &temp_fds)) {
            static struct timeval poll_tv = {0, 0};

            // Handle all pending connection requests
            // (note use of "polling" feature).
            while (select (s_fd + 1, &temp_fds, 0, 0, &poll_tv) > 0) {
                int cli_fd = accept (s_fd, 0, 0);

                if (cli_fd == -1) perror ("accept");
                else {
                    FD_SET (cli_fd, &read_fds);
                    if (cli_fd >= maxfdp1)
                        maxfdp1 = cli_fd + 1;
                }
            }
        }
    }
}
```

Internet Domain Datagram Sockets

- Uses UDP to return the current time of day from a specified list of Internet hosts

  e.g.,

  % hostdate tango mambo lambada merengue
  tango: timeout at host
  merengue: Tue Aug 20 15:56:00 1996

- Note the use of select to prevent hanging from hosts that are "down" or nonexistent

Concurrent Server using Select

- Single-threaded concurrent socket server

```c
int main (void) {
    // Create a server end-point.
    int s_fd = s_server (PORT_NUM);
    fd_set temp_fds;
    fd_set read_fds;
    int maxfdp1 = s_fd + 1;

    // Check for constructor failure.
    if (s_fd == -1)
        perror ("server"), exit (1);
    FD_ZERO (&temp_fds);
    FD_ZERO (&read_fds);
    FD_SET (s_fd, &read_fds);
```
Internet Domain Datagram
Sockets (cont’d)

- **E.g.**

```c
int do_service (int sfd, u_short port, const char *host) {
    struct hostent *hp = gethostbyname (host);
    if (hp == 0) return -1;
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;
    sin.sin_port = port;
    memset (&sin.sin_addr, 0, sizeof(sin));
    printf ("%s: %s", host); fflush (stdout);
    char buf[BUFSIZ];

    if (sendto (sfd, "", 0, /* Note zero size */ 0, &sin, sizeof(sin) < 0)
        return -1;

    struct timeval tv = {5, 0};
    int len = sizeof(sin);
    ssize_t n = timed_recv (&tv, sfd, buf, sizeof(buf),
        &sin, &len);
    if (n == -1) return n;
    printf ("%s\n", n, buf);
    return 0;
}
```

Advanced Socket Operations

- **Non-blocking connections**

- **Checking for invalid sockets**

- **Checking for terminated peers**

Internet Domain Datagram
Sockets (cont’d)

- **Main driver program**

```c
#define SERVICE "daytime"
int do_service (int u_short, const char *);

int main (int argc, char *argv[]) {
    int s = socket (PF_INET, SOCK_DGRAM, 0);
    if (s == -1)
        error ("argv[0]"); exit (1);

    struct servent *sp =
        getservbyname (SERVICE, "udp");
    if (sp == 0)
        fprintf (stderr, "%s/udp: unknown service.\n", SERVICE); exit (1);

    for (++argv; --argc; ++argv)
        if (do_service (s, sp->s_port, *argv) == -1)
            error (*argv);

    close (s);
    return 0;
}
```

Internet Domain Datagram
Sockets (cont’d)

- **Performed “timed receives” for datagrams**

```c
int timed_recv (struct timeval *tv, int fd,
    char buf[], int buf_size,
    struct sockaddr *sin, int *slen) {
    fd_set read_fd;
    FD_ZERO (&read_fd);
    FD_SET (fd, &read_fd);

    switch (select (fd + 1, &read_fd, 0, 0, tv)) {
    case 0: errno = ETIMEDOUT; /* FALLTHRU */
    case -1: return -1;
    default:
        return recvfrom (fd, buf, buf_size,
            0, &sin, &slen);
    }
```
Example of Non-Blo cking Connect

- This is easier in C++...

```c
int nblock_connect (int sfd, struct sockaddr *sin, int sinlen)
{
    struct timeval timeout = {1, 0};
    set_f1 (sfd, O_NONBLOCK);
    if (connect (sfd, sin, sinlen) == -1) {
        if (errno == EINPROGRESS) {
            fd_set write_fds;
            FD_ZERO (&write_fds);
            FD_SET (sfd, &write_fds);
            if (select(sfd + 1, 0, write_fds, 0, timeout) == 1) {
                if (FD_ISSET (sfd, &write_fds)) {
                    if (getpeername (sfd, &sin, &sinlen) < 0)
                        return -1; /* Connection failed */
                } else {
                    /* select() timed out, do something else here ... */
                } else return -1; /* connect failed unexpectedly */
            } else return sfd; /* Success, we’re connected! */
        }
    } else return -1; /* connect failed unexpectedly */
}
```

Checking for Invalid Sockets

- It is often useful to have the client test if a previously established socket is still active before trying to write to it
  - This avoids catching SIGPIPE and such...

- To do this, first try to read from the socket
  - If the client has closed the connection the read should return EOF

- To keep from hanging in read, first put the socket descriptor in non-blocking mode
  - Conversely, use select to find out whether read will block...

Creating a Non-blocking Socket

- Enable I/O descriptor flags
  - e.g. O_NONBLOCK

  ```c
  int set_fl (int flags)
  {
      int val = fcntl (fd, F_GETFL, 0);
      if (val == -1)
          return -1;
      val |= flags; /* turn on flags */
      if (fcntl (fd, F_SETFL, val) == -1)
          return -1;
      return 0;
  }
  ```

Non-blocking Connections

- connect may be used in non-blocking mode

- A combination of select, getpeernam e, and getsockopt may be used to determine when the connection setup is complete

- This is useful to avoid long timeouts if client may not be accessible
Checking for Terminated Peers

• A question that often arises is “how do I get the first write after the other end has terminated to generate SIGPIPE”

• The answer is “you can not”

• If you want to know as soon as the process at the other end of a connection terminates, use `select()`, testing for readability, then the `read` will return 0

Network Databases and Address Mapping

• `/etc/hosts` (supplanted by NIS and DNS)
  
  - List of Internet and local hosts accessible from local machine

  - Accessed via `gethostbyname`, `gethostbyaddr`

  - e.g.,

    ```
    # Subnet 3: Machines on CS subnet
    # Name Full name Address Full name Aliases
    128.252.165.140 tango.cs.wustl.edu le0-tango
    128.252.164.18 tango.cs.wustl.edu encip1-tango
    128.252.165.145 merengue.cs.wustl.edu le0-merengue
    128.252.165.142 lambada.cs.wustl.edu le0-lambada
    128.252.165.10 cs.wustl.edu cs nfs.cs.wustl.edu nfs
    ```

• `/etc/services`
  
  - List of available network services

  - Accessed via `getservbyname`, `getservbyport`

  - e.g.,

    ```
    # Service Name Port,Protocol Alias
    ftp-data 20/tcp
    ftp 21/tcp
    telnet 23/tcp
    tftp 69/udp
    http 80/tcp
    talk 617/udp
    uucp 540/tcp uucpd
    chforw 701/tcp chforwd
    exec 612/tcp exceserver
    login 513/tcp loginserver
    ```

• `/etc/protocols`
  
  - Information about preconfigured protocols

  - e.g.,

    ```
    # Internet (ip) protocols
    # Name Number Alias Comment
    ip 0 ip # internet protocol, pseudo protocol number
    icmp 1 icmp # internet control message protocol
    ggp 3 ggp # gateway/gateway protocol
    tcp 6 tcp # transmission control protocol
    pup 12 pup # parc universal packet protocol
    udp 17 udp # user datagram protocol
    ```

Network Databases and Address Mapping

• `/etc/networks`
  
  - List of local/Internet networks

  - Accessed via `getnetbyaddr`, `getnetbyname`

  - e.g.,

    ```
    # Net Name Net Number Alias
    uciics-net 128.195
    uciics-main 128.195.1 localnet
    uciicslab 128.195.3 uclabnet uci-labnet
    uciicsrsh 128.195.4 uclabnet uci-rshnet
    ```
Unix Domain Stream Sockets

- UNIX-domain socket reader header

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <signal.h>
#include <sys/un.h>
#include <string.h>

#define SOCK_NAME "/tmp/foo"
#define STDOUT 1
#define STDIN 0

int process_msg (int ifd, int ofd);
```

- sendn is a handy utility routine

```c
ssize_t sendn (int handle, const void *buf, size_t len) {
    size_t bytes_written;
    ssize_t n;

    for (bytes_written = 0;
         bytes_written < len;
         bytes_written += n)
        if ((n = write (handle, buf + bytes_written, len - bytes_written)) == -1)
            return -1;
    return bytes_written;
}
```

- Both of the following UNIX domain and Internet domain examples use the following library routine:

```c
int process_msg (int ifd, int ofd) {
    for (char msg[BUFSIZE];) {
        ssize_t len = read (ifd, msg, sizeof msg);
        if (len > 0) {
            if (send_n (ofd, msg, len) != len)
                return -1;
        } else return len;
    }
    return 0;
}

int s_fd = socket (PF_UNIX, SOCK_STREAM, 0);
if (s_fd == -1)
    return -1;
else if (bind (s_fd, (struct sockaddr *) &name, sizeof name.sun_family +
                   strlen (name.sun_path)) == -1)
    return -1;
else if (listen (s_fd, 5) == -1)
    return -1;
return s_fd;
```

- Become a passive-mode “server”

```c
int s_server (const char sockaddr[]) {
    struct sockaddr_un name;
    name.sun_family = AF_UNIX;
    strncpy (name.sun_path, sock_name, sizeof name.sun_path);

    int s_fd = socket (PF_UNIX, SOCK_STREAM, 0);
    if (s_fd == -1)
        return -1;
    else if (bind (s_fd, (struct sockaddr *) &name, sizeof name.sun_family +
                   strlen (name.sun_path)) == -1)
        return -1;
    else if (listen (s_fd, 5) == -1)
        return -1;
    return s_fd;
}
```

Unix Domain Stream Sockets (cont’d)

- UNIX-domain server

```c
#include "header.h"
void clean_up (void) { unlink (SOCK_NAME), exit (1); }

int main (int argc, char *argv[]) {
    signal (SIGINT, clean_up);

    int s_fd = s_server (SOCK_NAME);
    if (s_fd == -1)
        perror ("s_server");
    else {
        int cli_fd = accept (s_fd, 0, 0);
        if (cli_fd == -1)
            perror ("accept");
        else if (process_msg (cli_fd, STDOUT) == -1)
            perror ("process_msg");
        else if (close (cli_fd) == -1)
            perror ("close");
    }
    /* NOTREACHED */
}
```
Unix Domain Stream Sockets
(cont’d)

• Become an active-mode “client”

```c
int s_client (const char *sock_name[]) {
    struct sockaddr_un name;
    name.sun_family = AF_UNIX;
    strcpy (name.sun_path, sock_name);
    int s_fd = socket (PF_UNIX, SOCK_STREAM, 0);
    if (s_fd == -1)
        return -1;
    else if (connect (s_fd, (struct sockaddr *) &name,
        sizeof name.sun_family
        + strlen (name.sun_path)) == -1)
        return -1;
    return s_fd;
}
```

Unix Domain Stream Sockets
(cont’d)

• UNIX-domain socket sender

```c
#include "header.h"

int main (int argc, char *argv[]) {
    int s_fd = s_client (SOCK_NAME);
    int status = 1;
    if (s_fd == -1)
        perror ("s_client");
    else if (process_msg (STDIN, s_fd) == -1)
        perror ("process_msg");
    else
        status = 0;
    close (s_fd);
    return status;
}
```