Sockets Overview

Network Programming

Overview

- UDP (User Datagram Protocol) is unreliable
  - Potentially delivered out of order (or not at all!)
  - Connectionless
- TCP (Transmission Control Protocol) provides reliability
  - In-order delivery of packets
  - Connection-oriented
- Both sit on top of another protocol

Sockets

- Berkeley sockets implementation, originally from 4.2BSD (1983!)
  - Effectively became POSIX sockets
- Building blocks for modern network-enabled programs
- Simple API
  - Typical server / client sequence to follow…

Server

- Acquire socket with socket() call
- Bind socket to address (or wildcard) with bind()
- Cause socket to enter listening state with listen()
- accept(): block until connection request

socket

- `#include <sys/socket.h>`
- `int socket(int domain, int type, int protocol);`

- Just creates an endpoint, nothing more!
- domain typically PF_INET / AF_INET
- type: SOCK_[STREAM,DGRAM,RAW]
- protocol: just use 0 for system default for given domain / type

bind

```
int bind(int fd, struct
   sockaddr *addr, socklen_t
   len);
```

- fd: must be returned by socket()
- sa: sockaddr containing IP / port
- len: length of passed-in sockaddr

- Servers call bind upon startup

listen

- `int listen(int fd, int p_log);`

- Cause fd to enter listening state (after call to bind)
- Connections aren't instantaneous. Allows for p_log pending connections

accept

```
int accept(int fd, struct
   sockaddr *addr, socklen_t
   *addr_len);
```

- fd: accept incoming connections
- addr: filled in with remote host info
- addr_len: initially should contain amount of space in addr, on return will contain length of filled-in addr data

Client

- Acquire socket with socket() call
- Call connect() to connect to the server

connect

- `int connect(int fd, struct sockaddr *sa, socklen_t len);`

- fd: must be returned by socket()
- sa: sockaddr containing IP / port
- len: length of passed-in sockaddr

- Clients call connect() and not bind()
- TCP: three way handshake, more later

close

- `#include <unistd.h>`
- `int close(int fildes);`

- Initiate a shutdown of connection
  - In-flight data typically finishes

- Free up resources

struct sockaddr_in

```
struct sockaddr_in {
    __uint8_t         sin_len;
    sa_family_t       sin_family;
    in_port_t         sin_port;
    struct  in_addr   sin_addr;
    char              sin_zero[8];
};
```

```
struct in_addr {
    in_addr_t         s_addr;
};
```

Typical usage

```
struct sockaddr_in saddr;

/* Zero out the memory */
bzero(&saddr, sizeof(saddr));

saddr.sin_family = PF_INET;
saddr.sin_port = htons(1234);
saddr.sin_addr.s_addr =
  htonl(INADDR_ANY);
```

send/recv data

- #include <sys/types.h>
  #include <sys/uio.h>
  #include <unistd.h>
- **`ssize_t read(int fd, void *buf, size_t sz);`**
- Read sz bytes from fd into buf
- Similar prototype for write
- Pretty much the same for send/recv, includes additional flags parameter

Other Useful Functions

- `inet_aton()` and `inet_pton()`
  - `“127.0.0.1” => sockaddr format`
- `gethostbyname()` and `gethostbyaddr()`
  - `“www.google.com” => hostent`
  - `“172.217.10.4” => “www.google.com”`
- `getaddrinfo()` and `getnameinfo()`
  - `IPv6-friendly versions`

- Use the book / man pages / google for more information

daytimetcpcli.c

```c
#include    "unp.h"

int
main(int argc, char **argv)
{
    int                 sockfd, n;
    char                recvline[MAXLINE + 1];
    struct sockaddr_in  servaddr;

    if (argc != 2)
        err_quit("usage: a.out <IPaddress>");

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        err_sys("socket error");

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port   = htons(13);    /* daytime server */
    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
        err_quit("inet_pton error for %s", argv[1]);

    if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
        err_sys("connect error");

    while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
        recvline[n] = 0;    /* null terminate */
        if (fputs(recvline, stdout) == EOF)
            err_sys("fputs error");
    }
    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

daytimetcpsrv.c

```c
#include    "unp.h"
#include    <time.h>

int
main(int argc, char **argv)
{
    int                 listenfd, connfd;
    struct sockaddr_in  servaddr;
    char                buff[MAXLINE];
    time_t              ticks;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(13);   /* daytime server */

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

    for ( ; ; ) {
        connfd = Accept(listenfd, (SA *) NULL, NULL);

        ticks = time(NULL);
        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
        Write(connfd, buff, strlen(buff));

        Close(connfd);
    }
}
```

Client / Server Model

Network Programming

Clients

- Fairly simple
  - Typically connects to centralized server
- Only concerned with single connection (yours)
- Ex: a web browser
  - One of thousands of users per website
  - Should not impact you at all

daytimetcpcli.c

```c
#include    "unp.h"

int
main(int argc, char **argv)
{
    int                 sockfd, n;
    char                recvline[MAXLINE + 1];
    struct sockaddr_in  servaddr;

    if (argc != 2)
        err_quit("usage: a.out <IPaddress>");

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        err_sys("socket error");

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port   = htons(13);    /* daytime server */
    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
        err_quit("inet_pton error for %s", argv[1]);

    if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
        err_sys("connect error");

    while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
        recvline[n] = 0;    /* null terminate */
        if (fputs(recvline, stdout) == EOF)
            err_sys("fputs error");
    }
    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Servers

- Typically far more complicated than clients
- Have to juggle resources, incoming connections, etc.
  - But often have better CPUs, faster disks, more RAM
- Maintain high uptime (aim for 5 9's)

Daemons

- Background processes
  - Often initialized during boot
- Network services often under *inetd*
  - Replaces telnet, ftp, www, etc. each having to run their own server
  - Could replace dozens of services + redundant startup code
  - Each would take slot in process table (remember OS?)

daemon_init.c

```c
#include    "unp.h"
#include    <syslog.h>

#define MAXFD   64

extern int  daemon_proc;    /* defined in error.c */

int
daemon_init(const char *pname, int facility)
{
    int     i;
    pid_t   pid;

    if ( (pid = Fork()) < 0)
        return (-1);
    else if (pid)
        _exit(0);           /* parent terminates */

    /* child 1 continues... */

    if (setsid() < 0)           /* become session leader */
        return (-1);

    Signal(SIGHUP, SIG_IGN);
    if ( (pid = Fork()) < 0)
        return (-1);
    else if (pid)
        _exit(0);           /* child 1 terminates */
```

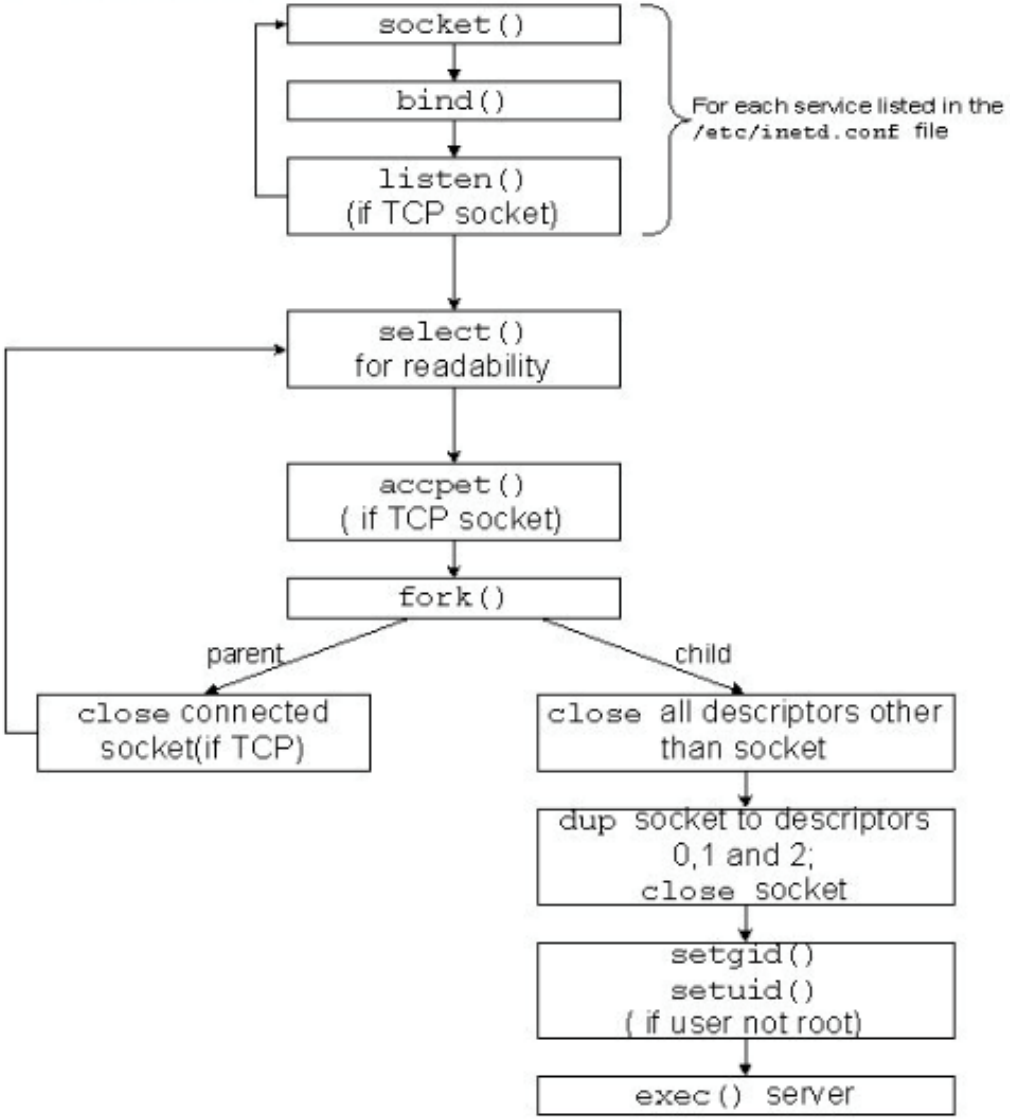
daemon_init.c pt. 2

```
    /* child 2 continues... */

    daemon_proc = 1;             /* for err_XXX() functions */

    chdir("/");                  /* change working directory */

    /* close off file descriptors */
    for (i = 0; i < MAXFD; i++)
        close(i);

    /* redirect stdin, stdout, and stderr to /dev/null */
    open("/dev/null", O_RDONLY);
    open("/dev/null", O_RDWR);
    open("/dev/null", O_RDWR);

    openlog(pname, LOG_PID, facility);

    return (0);                  /* success */
}
```

inetd/daytimetcpsrv2.c

```c
#include        "unp.h"
#include        <time.h>

int
main(int argc, char **argv)
{
        int listenfd, connfd;
        socklen_t addrlen, len;
        struct sockaddr *cliaddr;
        char buff[MAXLINE];
        time_t ticks;

        if (argc < 2 || argc > 3)
                err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");

        daemon_init(argv[0], 0);

        if (argc == 2)
                listenfd = Tcp_listen(NULL, argv[1], &addrlen);
        else
                listenfd = Tcp_listen(argv[1], argv[2], &addrlen);

        cliaddr = Malloc(addrlen);

        for ( ; ; ) {
                len = addrlen;
                connfd = Accept(listenfd, cliaddr, &len);
                err_msg("connection from %s", Sock_ntop(cliaddr, len));

                ticks = time(NULL);
                snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
                Write(connfd, buff, strlen(buff));

                Close(connfd);
        }
}
```

inetd flow diagram



Server Designs

- Iterative: completely finish work before handling next request
- Concurrent (fork): one process / client
- Concurrent (thread): one thread / client
- Server using **`select()`**
- Pre-forking: create a pool of processes ahead of time
- Pre-threading: create a pool of threads ahead of time

Threads

Ned Batchelder
@nedbat

Follow

Some people, when confronted with a problem, think, "I know, I'll use threads," and then two they hav erpoblesms.

RETWEETS	LIKES
1,701	388

8:47 AM - 23 Apr 2012

32 1.7K 388

daytimetcpsrv.c

```c
#include    "unp.h"
#include    <time.h>

int
main(int argc, char **argv)
{
    int                 listenfd, connfd;
    struct sockaddr_in  servaddr;
    char                buff[MAXLINE];
    time_t              ticks;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(13);   /* daytime server */

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

    for ( ; ; ) {
        connfd = Accept(listenfd, (SA *) NULL, NULL);

        ticks = time(NULL);
        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
        Write(connfd, buff, strlen(buff));

        Close(connfd);
    }
}
```

tcpserv01.c

```c
#include    "unp.h"

int
main(int argc, char **argv)
{
    int                 listenfd, connfd;
    pid_t               childpid;
    socklen_t           clilen;
    struct sockaddr_in  cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(SERV_PORT);

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

        if ( (childpid = Fork()) == 0) {    /* child process */
            Close(listenfd);    /* close listening socket */
            str_echo(connfd);   /* process the request */
            exit(0);
        }
        Close(connfd);          /* parent closes connected socket */
    }
}
```

select() system call

- Enables handling reading, writing, exceptional cases for file descriptors
- Informs kernel of interest in those specific file descriptors

```
int select(int nfds, fd_set
   *readfds, fd_set *writefds,
   fd_set *errorfds, struct
   timeval *timeout);
```

Clearing / Toggling FDS

- `void FD_SET(int fd, fd_set *s)`
- `void FD_CLR(int fd, fd_set *s)`
- `int FD_ISSET(int fd, fd_set *s)`
- `void FD_ZERO(fd_set *s)`

Summary

- If server load not heavy, concurrent server with **`fork()`**
- Pool of servers can “pre-pay” the cost associated with clients connecting
- **`select()`** can do a lot of heavy lifting for you (more on this next time)

Lab

- Lab released today at noon, Friday = lab day
- fork() lab
  - Read entire handout
  - Extra Credit (1 lab point) on second page
  - Recommend solving the lab w/o Extra Credit first
- Ask questions when you have them
- Once you're done with the lab make sure to get checked off by a mentor/TA/me
  - Wednesday office hours **this week** end at 11:30am