Internet Programming & Protocols Lecture 6

TCP

TCP Sockets

TCP client/servers

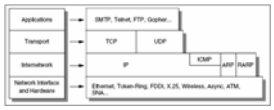
assignment 2

Plan of attack

- Network overview ✓
- BSD sockets and UDP ✓
- TCP
  - Socket programming
  - Reliable streams
  - Header and states
  - Flow control and bandwidth-delay
  - Measuring performance
  - Historical evolution
  - Congestion control
- Network simulation (ns)
- TCP accelerants
- TCP implementations

Transport layer

- end-to-end services to application
- API (BSD sockets, TLI)
- flow control
- error recovery
- ICMP, UDP, TCP
  - ICMP ping, traceroute
  - TCP ssh, www, ftp, mail, telnet, chat, print, finger, X…
  - UDP ntp/time, NFS, DNS, audio/video, RPC, snmp, DHCP



Ethernet	IP	TCP/UDP	Application

Transmission Control Protocol (TCP)

- TCP RFC 793 '81
- Provides a reliable stream of bytes on top of unreliable IP datagrams
- Connection oriented (circuit like)
- 16-bit port number (service)
- Statefull with timers, sequence numbers, flow control, congestion mgt.

4.4BSD lite (Stevens IP illustrated v2)
UDP: 9 functions, 800 lines of C code
TCP: 28 functions, 4500 lines of C code

Linux 2.6
UDP 1044 lines of C code
TCP 13050 lines of C code

Internet traffic

- 98% of packets are TCP, 2% UDP
  - Maybe more UDP on local net (NFS)
- Busiest TCP services
  - http 41%
  - ftp 20%
  - nntp 12%
  - nbftp 4%
  - Email 3%
- Busiest UDP services
  - RealPlayer 2%
  - DNS 0.2%-
- Encrypted services
  - ssh 7%
  - https 5%
  - IPsec 1%

Elephants & mice

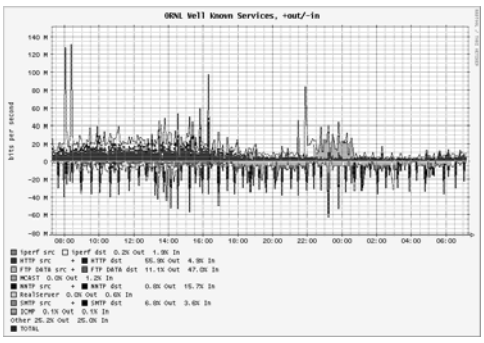
•A small percent of flows carry bulk of traffic

•Lots of tiny flow

78% < 10 pkts

95% < 50 pkts

Traffic volume for a day



socket calls

- **socket()** get a socket descriptor for given protocol family and type
- **bind()** associate name (address/port, etc.) with a server (usually) socket
- **connect()** client establishes a connection to a server
- **listen()** connection-oriented server tells system it's going to be passive.
- **accept()** server accepts incoming connection request and creates a new socket
- **close()** will try to deliver any unsent data
- Data transfers with **read(), write(), send(), recv()** or connectionless **sendto(), recvfrom()**

socket()

int socket(family, type, protocol)

- returns a socket descriptor which is then used in read/write/close
- family: AF_UNIX, AF_INET, AF_NS, AF_INET6
  - (actually should be PF_UNIX etc.)
- type: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
- protocol: usually 0
- fails: bad args, no fd's/memory
- just sets up kernel data structures
- You need

#include <sys/types.h>

#include <sys/socket.h>

bind()

int bind(sockfd, struct sockaddr *local, lth)

- binds local address and port to sockfd
- user fills struct sockaddr_in first providing port number
- required for server
- optional for client (usually not done by client)
- system will supply local address if client doesn't do bind
- lth of structure is required since struct sockaddr is different size for each protocol
- failures: bad args, port in use

port numbers

- Port numbers (UDP/TCP) provide a "process" address
  - Destination address, protocol (UDP or TCP), and port number define endpoint
  - Port number allows OS kernel to pass packets to appropriate process
  - For server process, bind() requests a port from the OS
  - In UNIX, ports < 1024 privileged
  - Well known (pre-defined) ports (services) listed in /etc/services
- bind() will fail if another server program on the machine is using the port
- bind() with port value of 0 tells OS to assign the port number
- bind() is optional for client (OS will assign a port number)

Well known TCP ports: echo (7), ftp (20/21), ssh (22), telnet (23), smtp/email (25), http (80), X (6000)

listen(int sockfd, int backlog)

- server call after bind() before accept()
- specify length of connection request queue (backlog)
- if queue is full, requests ignored (depends on OS)
- BSDs multiply backlog by 1.5
- backlog is limited (silently) to 5 on older SunOS
- don't use backlog of 0
- failures: bad args

accept(int sockfd, struct sockaddr *peer, int *lth)

- server accepts a connection request
- function returns NEW socket descriptor
- address info on peer is placed in *peer
- This function blocks til a connect() arrives
- Connecting host's IP address and port number stashed in peer struct
- failures: bad args, no mem

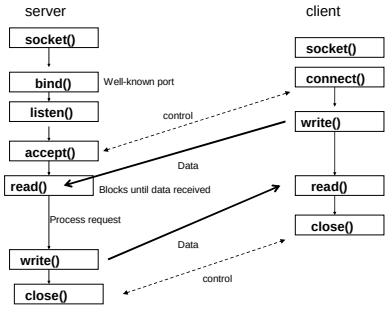
connect (int sockfd, struct sockaddr *server, socklen_t lth)

- connect() is optional for UDP client
- connect() is mandatory for TCP
  - Client attempts to establish a TCP connection on the with host and port specified in the server socket structure
  - OS assigns an ephemeral port for the client side
  - OS allocates buffers (SNDBUF/RCVBUF) and creates lots of other state info
  - Control packets exchanged to “establish” connection
  - A TCP “connection” is a five tuple {src IP, src port, dest IP, dest port, proto}
  - Connection is closed with **close()**
  - connection broken if client or server process dies or if no response (timeout)
- Failures:
  - Bad sockfd
  - ECONNREFUSED -- no process listening on that port at server
  - ETIMEDOUT -- server too busy
  - ENETUNREACH – network is unreachable

TCP data transfer

- Just use read() and write()
- read()/write() on streams will require looping to insure all data is read or written
- read()/write() returns number of bytes read or written
- Obviously, read() can block waiting for data, but write() can block too if other end is not reading (receiver RCVBUF full, and eventually sender’s SNDBUF fills) …
- fails: EOF, reset, interrupted, broken pipe (if connection broken)

TCP client/server



Example: get time from server

TCP server

```
/* daysrv [port] */
#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netinet/in.h>
void err_sys(char *msg) {perror(msg); exit(1);}

#define PORT 7654
#define BUFFSIZE        128
#define BACKLOG 5
char buff[BUFFSIZE];

main(argc, argv)
int     argc;
char    *argv[];
{
    int port = PORT;
    int reap();
    int n,sockfd, newsockfd, clilen;
    struct sockaddr_in serv_addr, cli_addr;

    if (argc > 1) port = atoi(argv[1]);
```



```
if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    err_sys("server: can't open stream socket");
/* setup struct for bind, so clients can find us */
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port        = htons(port);

if (bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
    err_sys("server: can't bind local address");

listen(sockfd, BACKLOG);
printf("server ready on port %d\n",port);
for(;;){
    int ticks;

    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
    ticks=time(0);
    strncpy(buff,ctime(&ticks),sizeof(buff));
    write(newsockfd,buff,strlen(buff));
    close(newsockfd);
}
```


TCP client

```
/* tcpday ipaddress simple tcp daytime client */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
void err_sys(char *msg) {perror(msg); exit(1);}
#define PORT 7654
char *host = "127.0.0.1"; /* localhost */
#define MAXBUF 128
main(argc, argv) char *argv[];
{
    int sd, n;
    struct sockaddr_in sin;
    char buff[MAXBUF + 1];
    if (argc > 1) host = argv[1];
    sd = socket(AF_INET,SOCK_STREAM,0);
    bzero(&sin,sizeof(sin));
    sin.sin_family = AF_INET;
    if ((sin.sin_addr.s_addr = inet_addr(host)) == -1 ) err_sys("inet_addr");
    sin.sin_port = htons(PORT); /* net byte order*/
    if (connect(sd, (struct sockaddr *)&sin, sizeof(sin)) < 0) err_sys("connect");
    while ((n = read(sd,buff,MAXBUF))>0){
          buff[n]=0;
          printf("%s",buff);
    }
}
```

Server strace

```
strace daysrv
...
socket (2, 1, 0) = 3
bind (3, ""..., 16) = 0
listen (3, 5) = 0
ioctl (1, 0x40125401, 0xf7ffee8c) = 0
write (1, "server ready on port 7654\n", 26) = server ready on port 7654
26
accept (3, 0xf7fffa48, 0xf7fffa64) =

accept (3, 0xf7fffa48, 0xf7fffa64) = 4
gettimeofday (0xf7fff9c0, 0) = 0
write (4, "Sun Sep  5 14:04:24 1999\n", 25) = 25
close (4) = 0
accept (3, 0xf7fffa48, 0xf7fffa64) =

netstat -a
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address           Foreign Address         (state)
tcp        0      0  *.7654                 *.*                     LISTEN
```


Client strace

```
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(7654),
 sin_addr=inet_addr("127.0.0.1")}, 16) = 0
read(3, "Mon Aug 29 17:42:07 2005\n", 128) = 25
write(1, "Mon Aug 29 17:42:07 2005\n", 25) = 25
read(3, "", 128) = 0
exit_group(0) = ?
```

- **Note that TCP returns a “stream” of bytes, not “messages”**
- **You may not actually write() the length specified … unusual**
- **Your read() may not return all that you requested!**

writen.c

```
writen(int fd, const void *ptr, int nbytes)
{
    int   nleft, nwritten;

    nleft = nbytes;
    while (nleft > 0) {
          nwritten = write(fd, ptr, nleft);
          if (nwritten <= 0) {
                if (errno == EINTR) nwritten=0;  /* do it again */
                else return(nwritten);    /* error */
          }
          nleft -= nwritten;
          ptr   += nwritten;
    }
    return(nbytes - nleft);
}
```


readn.c

```
readn(int fd, void *ptr, int nbytes)
{
    int   nleft, nread;

    nleft = nbytes;
    while (nleft > 0) {
          nread = read(fd, ptr, nleft);
          if (nread < 0) {
                if (errno == EINTR) nread = 0; /* do read again */
                else return(nread);    /* error, return < 0 */
          }else if (nread == 0) break;        /* EOF */

          nleft -= nread;
          ptr   += nread;
    }
    return(nbytes - nleft);    /* return >= 0 */
}
```


readline.c

```
int readline(int fd, char *ptr, int maxlen)
{
    int   n, rc;
    char  c;

    for (n = 1; n < maxlen; n++) {
    again:
          if ( (rc = read(fd, &c, 1)) == 1) {
                *ptr++ = c;
                if (c == '\n') break;
                } else if (rc == 0) {
                          if (n == 1)return(0);  /* EOF, no data read */
                else break;      /* EOF, some data was read */
          } else {
                if (errno == EINTR) goto again;
                return(-1); /* error */
          }
    }

    *ptr = 0;
    return(n);
}
```


tcpcli.c

```
main(argc, argv)
int   argc;
char  *argv[];
{
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port        = htons(SERV_TCP_PORT);
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
          err_sys("client: can't open stream socket");
    if (connect(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
          err_sys("client: can't connect to server");

    str_cli(stdin, sockfd);    /* do it all */

    close(sockfd);
    exit(0);
}
```

strcli.c

```
str_cli(fp, sockfd)
register FILE  *fp;
register int   sockfd;
{
    int   n;
    char  sendline[MAXLINE], recvline[MAXLINE + 1];

    while (fgets(sendline, MAXLINE, fp) != NULL) {
          n = strlen(sendline);
          if (writen(sockfd, sendline, n) != n)
                  err_sys("str_cli: writen error on socket");
          n = readline(sockfd, recvline, MAXLINE);
          if (n < 0) err_sys("str_cli: readline error");
          recvline[n] = 0;  /* null terminate */
          fputs(recvline, stdout);
    }

    if (ferror(fp)) err_sys("str_cli: error reading file");
}
```


tcpserv.c

```
main(argc, argv)
int   argc;
char  *argv[];
{
    int         sockfd, newsockfd, clilen, childpid;
    struct sockaddr_in   cli_addr, serv_addr;

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
          err_sys("server: can't open stream socket");
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port        = htons(SERV_TCP_PORT);

    if (bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
          err_sys("server: can't bind local address");

    listen(sockfd, 5);
    for ( ; ; ) { /* iterative server */
          clilen = sizeof(cli_addr);
          newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
          if (newsockfd < 0) err_sys("server: accept error");
          str_echo(newsockfd); /* process the request */
          close(newsockfd);    /* parent process */
    }
}
```


strecho.c

```
#define  MAXLINE  512

str_echo(sockfd)
int   sockfd;
{
    int   n;
    char  line[MAXLINE];

    for ( ; ; ) {
          n = readline(sockfd, line, MAXLINE);
          if (n == 0) return;     /* connection terminated */
          else if (n < 0)     err_sys("str_echo: readline error");

          if (writen(sockfd, line, n) != n) err_sys("str_echo: writen error");
    }
}
```


Some TCP socket options

- setsockopt(), getsockopt()
- must refer to open sockfd, issue before connect/bind

```
#include <sys/socket.h>
getsockopt(fd, level, optname, void *val, int *len)
setsockopt(fd, level, optname, void *val, int len)
```

- SO_KEEPALIVE – kernel sends probes on idle socket, early notification of broken connection
- SO_REUSEADDR – TCP close() can linger awhile, this allows you to restart your server with same port
- SO_SNDBUF SO_RCVBUF
  - Send and receive buffer sizes
  - Size to bandwidth-delay product
  - We'll have LOTS to say about these over the coming weeks

Things that go bump in the net

- TCP connect, and no server process
- TCP connect, server host down
- active TCP session, ctrl-c server
- inactive TCP session, ctrl-c server
- active TCP session, server computer crashes
- inactive TCP session, server computer crashes
- Inactive TCP session, several routers on the path crash and reboot
- inactive TCP session with KEEPALIVE, server computer crashes
- inactive TCP session, server computer crashes and reboots
- start 2nd copy of server
- server tries to bind to port < 1024
- A sends faster than B can receive

concurrent servers

- Iterative server OK for short requests. Need to fork (or multi-thread) for more complex services (http, mail, sshd)
- Template: mother process handles listen/accept and spawns children to do actual work
  - handle SIG_CHLD (child termination) and EINTR error in accept()

```
…
sd=socket(…)
bind(sd,…)
listen(sd,…)
while(1) {
          newsd = accept(sd,…)
          if (fork() == 0){ /* create child process*/
                    close(sd) /* inherits sockets */
                    child()  /* do the work */
                    exit      /* exit child */
          }
          close(newsd)
}
```

inetd xinetd

- Mother of all network servers
- Rather than start oodles of processes for all the network services offered (used to be a lot in old UNIX distributions), start one process to listen on the configured service ports, and spawn (fork) a child server process when a connection is requested
- Config file specifies services and executables (/etc/inetd.conf)
- Servers have to be written to “inherit” socket descriptor from inetd
- inted logic
  - Create socket descriptor for each config’d service: socket(), bind(), listen()
  - In infinite loop, await on connect (select()) on any of the socket descriptors
  - accept() and fork()/exec() the server with new sd dup’d to fd 0,1,2

Next time ...

- Reliable streams
- TCP header