15-441

Network Programming Introduction
Jan. 25, 2006

(Borrowing heavily from 15-213)

Topics

- Programmer's view of the Internet
- Sockets interface
- Writing clients and servers
- Concurrency with I/O multiplexing
About This Lecture

“Intro to writing client/server programs with TCP”

- Stolen from 15-213
- Should be “review”
- Will zoom through these slides
  - You may review at your leisure

Extensions to reach Project 1

- 15-213 “rio” package may not be advisable
- You'll use UDP, not TCP
  - Packet protocol rather than byte-stream
  - No “connections” (hence no “disconnections” aka EOFs)
  - You may find error reporting confusing at first
A Client-Server Transaction

Every network application is based on the client-server model:

- A server process and one or more client processes
- Server manages some resource.
- Server provides service by manipulating resource for clients.

Note: clients and servers are processes running on hosts (can be the same or different hosts).
Network Applications

Access to Network via Program Interface

- Sockets make network I/O look like files
- Call system functions to control and communicate
- Network code handles issues of routing, reliability, ordering, &c.
Clients

Examples of client programs
- Web browsers, ftp, telnet, ssh

How does a client specify a server?
- The IP address in the server socket address identifies the host *(more precisely, an adaptor on the host)*
- The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.
- Examples of well-known ports
  - Port 7: Echo server
  - Port 23: Telnet server
  - Port 25: Mail server
  - Port 80: Web server
Internet Connections (TCP/IP)

Clients and servers communicate by sending streams of bytes over connections.

Connections are point-to-point, full-duplex (2-way communication), and reliable.

Client socket address
128.2.194.242:3479

Server socket address
208.216.181.15:80

Client host address
128.2.194.242

Server host address
208.216.181.15

Note: 3479 is an ephemeral port allocated by the kernel

Note: 80 is a well-known port associated with Web servers
Using Ports to Identify Services

Service request for 128.2.194.242:80 (i.e., the Web server)

Service request for 128.2.194.242:7 (i.e., the echo server)
Servers

Servers are long-running processes (daemons).

- Created at boot-time (typically) by the init process (process 1)
- Run continuously until the machine is turned off.

Each server waits for requests to arrive on a well-known port associated with a particular service.

- Port 7: echo server
- Port 23: telnet server
- Port 25: mail server
- Port 80: HTTP server

A machine that runs a server process is also often referred to as a “server.”

See /etc/services for a list of semi-standard service to port bindings.
Sockets Interface

Created in the early 80’s as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.

Provides a user-level interface to the network.

Underlying basis for all Internet applications.

Based on client/server programming model.
Overview of the Sockets Interface

Client

- socket
- connect
- rio_readlineb
- rio_writen
- close

Server

- socket
- bind
- listen
- accept
- rio_readlineb
- rio_writen
- close

Client / Server Session

open_clientfd

Connection request

Await connection request from next client

15-441, Spring 2006
Sockets

What is a socket?

- To the kernel, a socket is an endpoint of communication.
- To an application, a socket is a file descriptor that lets the application read/write from/to the network.
  - Remember: All Unix I/O devices, including networks, are modeled as files.

Clients and servers communicate with each by reading from and writing to socket descriptors.

The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors.
Socket Address Structures

Generic socket address:

- For address arguments to connect, bind, and accept.
- Necessary only because C did not have generic (\texttt{void *}) pointers when the sockets interface was designed.

```c
struct sockaddr {
    unsigned short sa_family;    /* protocol family */
    char            sa_data[14];  /* address data. */
};
```

Internet-specific socket address:

- Must cast (\texttt{sockaddr\_in *}) to (\texttt{sockaddr *}) for connect, bind, and accept.

```c
struct sockaddr\_in  {
    unsigned short sin\_family;    /* address family (always AF_INET) */
    unsigned short sin\_port;      /* port num in network byte order */
    struct in\_addr sin\_addr;     /* IP addr in network byte order */
    unsigned char    sin\_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```
Reliable I/O (RIO) Summary

I/O Package Developed by David O’Hallaron
  - http://csapp.cs.cmu.edu/public/code.html (csapp.{h,c})
  - Allows mix of buffered and unbuffered I/O

Important Functions
  - rio_writen(int fd, void *buf, size_t n)
    - Writes n bytes from buffer buf to file fd.
  - rio_readlineb(rio_t *rp, void *buf, size_t maxn)
    - Read complete text line from file rp into buffer buf.
      - Line must be terminated by newline (\n) character
    - Up to maximum of maxn bytes
#include "csapp.h"

/* usage: ./echoclient host port */
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];
    rio_t rio;

    host = argv[1];
    port = atoi(argv[2]);

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
Echo Client: open_clientfd

```c
int open_clientfd(char *hostname, int port)
{
    int clientfd;
    struct hostent *hp;
    struct sockaddr_in serveraddr;

    if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1; /* check errno for cause of error */

    /* Fill in the server's IP address and port */
    if ((hp = gethostbyname(hostname)) == NULL)
        return -2; /* check h_errno for cause of error */
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    bcopy((char *) hp->h_addr, (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
    serveraddr.sin_port = htons(port);

    /* Establish a connection with the server */
    if (connect(clientfd, (SA *) &serveraddr, sizeof(serveraddr)) < 0)
        return -1;
    return clientfd;
}
```

This function opens a connection from the client to the server at hostname:port.
Echo Client: `open_clientfd (socket)`

`socket` creates a socket descriptor on the client.

- **AF_INET**: indicates that the socket is associated with Internet protocols
  - Not Xerox XNS, DEC DECnet, not machine-local “Unix sockets”
- **SOCK_STREAM**: selects a reliable byte stream connection.
  - `(AF_INET, SOCK_STREAM, 0) ⇒ TCP`
  - `(AF_INET, SOCK_DGRAM, 0) ⇒ UDP`

```c
int clientfd; /* socket descriptor */

if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1; /* check errno for cause of error */

... (more)
```
Echo Client: `open_clientfd`

(gethostbyname)

The client then builds the server’s Internet address.

```c
int clientfd;                  /* socket descriptor */
struct hostent *hp;            /* DNS host entry */
struct sockaddr_in serveraddr; /* server’s IP address */

...  

/* fill in the server's IP address and port */
if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
bcopy((char *)hp->h_addr,
     (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
serveraddr.sin_port = htons(port);  
```
Echo Client: `open_clientfd` (connect)

Finally the client creates a connection with the server.

- Client process suspends (blocks) until the connection is created.
- After resuming, the client is ready to begin exchanging messages with the server via Unix I/O calls on descriptor `sockfd`.

```c
int clientfd;              /* socket descriptor */
struct sockaddr_in serveraddr; /* server address */
typedef struct sockaddr SA;   /* generic sockaddr */

/* Establish a connection with the server */
if (connect(clientfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
    return -1;
return clientfd;
```


int main(int argc, char **argv) {
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    struct hostent *hp;
    char *haddrp;

    port = atoi(argv[1]); /* the server listens on a port passed
    on the command line */
    listenfd = open_listenfd(port);

    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                           sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        haddrp = inet_ntoa(clientaddr.sin_addr);
        printf("Fd %d connected to %s (%s:%s)\n",
               connfd, hp->h_name, haddrp, ntohs(clientaddr.sin_port));
        echo(connfd);
        Close(connfd);
    }
}
int open_listenfd(int port)
{
    int listenfd, optval=1;
    struct sockaddr_in serveraddr;

    /* Create a socket descriptor */
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    /* Eliminates "Address already in use" error from bind. */
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval, sizeof(int)) < 0)
        return -1;

    ... (more)
Echo Server: open_listenfd (cont)

... 

/* Listenfd will be an endpoint for all requests to port on any IP address for this host */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port = htons((unsigned short)port);
if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
    return -1;

/* Make it a listening socket ready to accept connection requests */
if (listen(listenfd, LISTENQ) < 0)
    return -1;

return listenfd;
Echo Server: open_listenfd (socket)

socket creates a socket descriptor on the server.

- **AF_INET**: indicates that the socket is associated with Internet protocols.
- **SOCK_STREAM**: selects a reliable byte stream connection.

```c
int listenfd; /* listening socket descriptor */

/* Create a socket descriptor */
if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1;
```
Echo Server: open_listenfd (initialize socket address)

Next, we initialize the socket with the server’s Internet address (IP address and port)

```c
struct sockaddr_in serveraddr; /* server's socket addr */
...
/* listenfd will be an endpoint for all requests to port
   on any IP address for this host */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port = htons((unsigned short)port);
```

IP addr and port stored in network (big-endian) byte order

- `htonl()` converts longs from host byte order to network byte order.
- `htons()` converts shorts from host byte order to network byte order.
Echo Server: `open_listenfd (bind)`

**bind assigns a “name” to the socket**

- Internet-domain byte-stream sockets are TCP
- TCP socket names are (IP address, port)
  - `INADDR_ANY` means “all IP addresses of this machine”
- Once the socket is named clients can connect to it

```c
int listenfd;                 /* listening socket */
struct sockaddr_in serveraddr; /* server’s socket addr */

...                             /* listenfd will be an endpoint for all requests to port
if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)  
    return -1;
```
Echo Server: `open_listenfd (listen)`

`listen` indicates that this socket will accept connection (`connect`) requests from clients.

```c
int listenfd; /* listening socket */

... /* Make it a listening socket ready to accept connection requests */
    if (listen(listenfd, LISTENQ) < 0)
        return -1;
    return listenfd;
}
```

We’re finally ready to enter the main server loop that accepts and processes client connection requests.
Echo Server: Main Loop

The server loops endlessly, waiting for connection requests, then reading input from the client, and echoing the input back to the client.

```c
main() {
    /* create and configure the listening socket */
    while(1) {
        /* Accept(): wait for a connection request */
        /* echo(): read and echo input lines from client til EOF */
        /* Close(): close the connection */
    }
}
```
Echo Server: accept

accept () blocks waiting for a connection request.

```c
int listenfd; /* listening descriptor */
int connfd;   /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
```

accept returns a **connected descriptor** (connfd) with the same properties as the **listening descriptor** (listenfd)

- Returns when the connection between client and server is created and ready for I/O transfers.
- All I/O with the client will be done via the connected socket.

accept also fills in client’s IP address.
Echo Server: accept Illustrated

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`.

2. Client makes connection request by calling and blocking in `connect`.

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`. 
Connected vs. Listening Descriptors

Listening descriptor

- End point for client connection requests.
- Created once and exists for lifetime of the server.

Connected descriptor

- End point of the connection between client and server.
- A new descriptor is created each time the server accepts a connection request from a client.
- Exists only as long as it takes to service client.

Why the distinction?

- Allows for concurrent servers that can communicate over many client connections simultaneously.
Echo Server: Identifying the Client

The server can determine the domain name, IP address, and port of the client.

```c
struct hostent *hp;  /* pointer to DNS host entry */
char *haddrp;        /* pointer to dotted decimal string */

hp = gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                    sizeof(clientaddr.sin_addr.s_addr), AF_INET);

haddrp = inet_ntoa(clientaddr.sin_addr);
printf("Fd %d connected to %s (%s:%s)\n",
       connfd, hp->h_name, haddrp, ntohs(clientaddr.sin_port));
```
Echo Server: echo

The server uses RIO to read and echo text lines until EOF (end-of-file) is encountered.

- EOF notification caused by client calling close(clientfd).
- IMPORTANT: EOF is a condition, not a particular data byte.

```c
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", n);
        Rio_writen(connfd, buf, n);
    }
}
```

EOF notification caused by client calling close(clientfd).

IMPORTANT: EOF is a condition, not a particular data byte.
Running Echo Client/Server

[bryant@bryant echo]$ ./echoservers 15441
fd 4 connected to BRYANT-TP2.VLSI.CS.CMU.EDU
(128.2.222.198:3507)
Server received 12 (12 total) bytes on fd 4

[bryant@bryant-tp2 echo]$ ./echoclient bryant.vlsi.cs.cmu.edu 15441
hello world
hello world
Iterative Servers

Iterative servers process one request at a time.

call connect  
ret connect

call read
ret read
close

call accept
ret accept
write
close

call accept
ret accept
write
close

call connect
ret connect

call read
ret read
close

client 1  
server  
client 2
Fundamental Flaw of Iterative Servers

**Solution: use concurrent servers instead.**

- Concurrent servers use multiple concurrent flows to serve multiple clients at the same time.
Concurrent servers handle multiple requests concurrently.

User goes out to lunch

Client 1 blocks waiting for user to type in data

- 35 -
Possible Mechanisms for Creating Concurrent Flows

1. Processes
   - Kernel automatically interleaves multiple logical flows.
   - Each flow has its own private address space.

2. I/O multiplexing with `select()` (Our Focus)
   - User manually interleaves multiple logical flows.
   - Each flow shares the same address space.
   - Popular for high-performance server designs.

3. Threads
   - Kernel automatically interleaves multiple logical flows.
   - Each flow shares the same address space.
   - Hybrid of processes and I/O multiplexing!
Event-Based Concurrent Servers Using I/O Multiplexing

Maintain a pool of connected descriptors.

Repeat the following forever:

- Use the Unix `select` function to block until:
  - (a) New connection request arrives on the listening descriptor.
  - (b) New data arrives on an existing connected descriptor.
- If (a), add the new connection to the pool of connections.
- If (b), read any available data from the connection
  - Close connection on EOF and remove it from the pool.
The select Function

select() sleeps until one or more file descriptors in the set readset ready for reading.

```c
#include <sys/select.h>

int select(int maxfdp1, fd_set *readset, NULL, NULL, NULL);
```

readset

- Opaque bit vector (max FD_SETSIZE bits) that indicates membership in a descriptor set.
- On Linux machines, FD_SETSIZE = 1024
- If bit k is 1, then descriptor k is a member of the descriptor set.
- When call select, should have readset indicate which descriptors to test

maxfdp1

- Maximum descriptor in descriptor set plus 1.
- Tests descriptors 0, 1, 2, ..., maxfdp1 - 1 for set membership.

select() returns the number of ready descriptors and keeps on each bit of readset for which corresponding descriptor is ready.
Macros for Manipulating Set Descriptors

void FD_ZERO(fd_set *fdset);

- Turn off all bits in fdset.

void FD_SET(int fd, fd_set *fdset);

- Turn on bit fd in fdset.

void FD_CLR(int fd, fd_set *fdset);

- Turn off bit fd in fdset.

int FD_ISSET(int fd, *fdset);

- Is bit fd in fdset turned on?
Event-based Concurrent Echo Server

/*
 * echoservers.c - A concurrent echo server based on select
 */
#include "csapp.h"

typedef struct { /* represents a pool of connected descriptors */
    int maxfd;        /* largest descriptor in read_set */
    fd_set read_set;  /* set of all active descriptors */
    fd_set ready_set; /* subset of descriptors ready for reading */
    int nready;       /* number of ready descriptors from select */
    int maxi;         /* highwater index into client array */
    int clientfd[FD_SETSIZE]; /* set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* set of active read buffers */
} pool;

int byte_cnt = 0; /* counts total bytes received by server */
int main(int argc, char **argv)
{
    int listenfd, connfd, clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    static pool pool;

    listenfd = Open_listenfd(argv[1]);
    init_pool(listenfd, &pool);

    while (1) {
        pool.ready_set = pool.read_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set,
                              NULL, NULL, NULL);

        if (FD_ISSET(listenfd, &pool.ready_set)) {
            connfd = Accept(listenfd, (SA *)&clientaddr,&clientlen);
            add_client(connfd, &pool);
        }
        check_clients(&pool);
    }
}
/* initialize the descriptor pool */
void init_pool(int listenfd, pool *p) {
    /* Initially, there are no connected descriptors */
    int i;
    p->maxi = -1;
    for (i=0; i< FD_SETSIZE; i++)
        p->clientfd[i] = -1;

    /* Initially, listenfd is only member of select read set */
    p->maxfd = listenfd;
    FD_ZERO(&p->read_set);
    FD_SET(listenfd, &p->read_set);
}
void add_client(int connfd, pool *p) /* add connfd to pool p */
{
    int i;
    p->nready--;

    for (i = 0; i < FD_SETSIZE; i++) /* Find available slot */
        if (p->clientfd[i] < 0) {
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);

            FD_SET(connfd, &p->read_set); /* Add desc to read set */

            if (connfd > p->maxfd) /* Update max descriptor num */
                p->maxfd = connfd;
            if (i > p->maxi) /* Update pool high water mark */
                p->maxi = i;
            break;
        }

    if (i == FD_SETSIZE) /* Couldn't find an empty slot */
        app_error("add_client error: Too many clients");
}
void check_clients(pool *p) { /* echo line from ready descs in pool p */
    int i, connfd, n;
    char buf[MAXLINE];
    rio_t rio;

    for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        rio = p->clientrio[i];

        /* If the descriptor is ready, echo a text line from it */
        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            p->nready--;
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
                byte_cnt += n;
                Rio_writen(connfd, buf, n);
            } else { /* EOF detected, remove descriptor from pool */
                Close(connfd);
                FD_CLR(connfd, &p->read_set);
                p->clientfd[i] = -1;
            }
        }
    }
}
Pro and Cons of Event-Based Designs

+ One logical control flow.
+ Can single-step with a debugger.
+ No process or thread control overhead.
  - Design of choice for high-performance Web servers and search engines.

- Significantly more complex to code than process- or thread-based designs.

- Can be vulnerable to two forms of denial of service attacks
  - How?
Attack #1

**Overwhelm Server with Connections**

- Limited to FD_SETSIZE – 4 (typically 1020) connections

**Defenses?**
Attack #2: Partial Lines

- **Client gets attention of server by sending partial line**
- **Server blocks until line completed**

User types “Hello world\n”

Client sends “Hello world”

Server blocks waiting for “\n” from Client 1

Client 2 blocks waiting to complete its connection request until after lunch!
Flaky Client

while (Fgets(buf, MAXLINE, stdin) != NULL) {
    Rio_writen(clientfd, buf, strlen(buf)-1);
    Fgets(buf, MAXLINE, stdin); /* Read & ignore line */
    Rio_writen(clientfd, "\n", 1);
    Rio_readlineb(&rio, buf, MAXLINE);
    Fputs(buf, stdout);
}

- Sends everything up to newline
- Doesn’t send newline until user types another line
- Meanwhile, server will block
Implementing a Robust Server

Break Up Reading Line into Multiple Partial Reads

- Every time connection selected, read as much as is available
- Construct line in separate buffer for each connection

Must Use Unix Read

- `read(int fd, void *buf, size_t maxn)`
  - Read as many bytes as are available from file `fd` into buffer `buf`.
  - Up to maximum of `maxn` bytes

Cannot Use RIO Version

- `rio_readn(int fd, void *buf, size_t n)`
  - Read `n` bytes into buffer `buf`.
  - Blocks until all `n` read or EOF
Robust Server

```c
/*
 * echoserverub.c - A robust, concurrent echo server based on select
 */
#include "csapp.h"

typedef struct { /* represents a pool of connected descriptors */
    int maxfd;  /* largest descriptor in read_set */
    fd_set read_set; /* set of all active descriptors */
    fd_set ready_set; /* subset of descriptors ready for reading */
    int nready; /* number of ready descriptors from select */
    int maxi; /* highwater index into client array */
    int clientfd[FD_SETSIZE]; /* set of active descriptors */
    char clientbuf[FD_SETSIZE][MAXBUF]; /* set of read buffers */
    int clientcnt[FD_SETSIZE]; /* Count of characters in buffers */
} pool;

int byte_cnt = 0; /* counts total bytes received by server */
```
void check_clients(pool *p) {
    int i, connfd, n;
    for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        char *buf = p->clientbuf[i]; /* Private buffer */
        int cnt = p->clientcnt[i]; /* Number of chars read so far */
        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            p->nready--;
            if ((n = Read(connfd, buf+cnt, MAXBUF-cnt)) != 0) {
                byte_cnt += n; cnt += n;
                if (buf[cnt-1] == '\n') {
                    Write(connfd, buf, cnt); /* End of line */
                    p->clientcnt[i] = 0;
                } else
                    p->clientcnt[i] = cnt;
            } else { ... }  
        } else { ... }
    }
}
Robustness Principles

**Client**
- Nothing user does/types should make program crash
  - Must perform complete checking for user errors

**Server**
- Nothing a client does should cause server to malfunction
  - Possibly malicious clients

**Things to Worry About**
- Error return codes by system calls
- String overflows
- Malformed messages
- Memory/resource leaks
  - Especially for server
Echo Client Main Routine

```c
#include "csapp.h"

/* usage: ./echoclient host port */
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];
    rio_t rio;

    host = argv[1];
    port = atoi(argv[2]);

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

- No checking of command line arguments
- Wrappers exit on error
- fgets does not insert \n when string too long
Robust Version of Echo Client (1)

#include <limits.h>

/* To demonstrate truncation */
#define LINELEN 20

/* Maximum number of errors to tolerate before exiting */
int errlimit = 5;

void errcheck(char *message, int fatal)
{
    if (--errlimit == 0 || fatal) {
        fprintf(stderr, "Error: %s. Exiting\n", message);
        exit(1);
    }
    fprintf(stderr, "Error: %s. Continuing\n", message);
}

void usage(char *progname) {
    fprintf(stderr, "Usage: %s host port\n", progname);
    exit(0);
}
Robust Version of Echo Client (2)

```c
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[LNELEN];
    rio_t rio;

    if (argc != 3)
        usage(argv[0]);

    host = argv[1];
    port = atoi(argv[2]);

    if (port <= 0 || port > SHRT_MAX)
        errcheck("Invalid Port", 1);

    clientfd = open_clientfd(host, port);
    if (clientfd < 0)
        errcheck("Couldn't open connection to server", 1);

    rio_readinitb(&rio, clientfd);

    ...
... while (fgets(buf, LINELEN, stdin) != NULL) {
    int n;
    if (strlen(buf) == LINELEN-1 && buf[LINELEN-1] != '\n')
        strcpy(buf+LINELEN-5, "...
    /* Truncate string */
    if (rio_writen(clientfd, buf, strlen(buf)) < 0) {
        errcheck("Failed to send message", 0);
        continue;
    }
    if ((n = rio_readlineb(&rio, buf, LINELEN) <= 0)) {
        if (n == 0)
            errcheck("Unexpected EOF from server\n", 1);
        else
            errcheck("Failed to receive reply from server", 0);
    }
    if (fputs(buf, stdout) < 0)
        errcheck("Couldn't print reply\n", 0);
}
...
Robust Version of Echo Client (4)

```c
... 
if (close(clientfd) < 0)
    errcheck("Couldn't close connection to server", 1);
exit(0);
}
```
Design Issues

Error Classification & Recovery

- Fatal vs. nonfatal errors
  - Server code should only have fatal error when something is wrong on server machine
- What to do when encounter nonfatal error
  - Skip to next activity
  - Server might close connection to malfunctioning client

Other Types of Errors

- Client dormant too long
  - Add timeouts to code
  - Gets very messy
- Denial of service attacks
  - Difficult to detect and/or handle
Scripting Languages

General Features

- Easy to write “quick & dirty” code
  - Minimal type checking
  - Interpretive
- Good support for strings, regular expressions, invoking other programs
Scripting Languages

Examples

- awk, shell code
  - Developed originally at Bell Labs. Not very popular
- tcl
  - Developed by John Ousterhout (CMU PhD 1980)
  - Nice integration with tk graphics interface package
- perl
  - Developed by Larry Wall to aid system administration
  - Big & messy, but very powerful
- python
  - Developed by Guido van Rossum
  - Indentation is significant
    » (ouch)
#!/usr/bin/perl -w

use sigtrap;
use IO::Socket;

$host = $ARGV[0];
$port = $ARGV[1];

$socket = IO::Socket::INET->new("$host:$port") ||
    die("Couldn't connect to $host:$port: $!
"));

while (<STDIN>) {
    $line = $_;
    print $socket $line;
    $reply = <$socket>;
    print $reply;
}

---
Conceptual Model

**Maintain State Machine for Each Connection**
- First Version: State is just identity of connfd
- Second Version: State includes partial line + count of characters

**Select Determines Which State Machine to Update**
- First Version: Process entire line
- Second Version: Process as much of line as is available

**Design Issue**
- Must set granularity of state machine to avoid server blocking
For More Information


- THE network programming “bible”.

Complete versions of original echo client and server are developed in *Computer Systems: A Programmer’s Perspective*.

- Available from csapp.cs.cmu.edu
- You may compile and run them for yourselves to see how they work.
- Feel free to borrow any of this code.
- But be careful---it isn’t sufficiently robust for our programming assignments.
  - Most routines exit when any kind of error encountered