CS 33

More Network Programming
Client-Server Interaction

• Client sends requests to server
• Server responds
• Server may deal with multiple clients at once
• Client may contact multiple servers
Reliable Communication

- **The promise …**
  - what is sent is received
  - order is preserved
- **Set-up is required**
  - two parties agree to communicate
  - within the implementation of the protocol:
    » each side keeps track of what is sent, what is received
    » received data is acknowledged
    » unack’d data is re-sent
- **The standard scenario**
  - server receives connection requests
  - client makes connection requests
Streams in the Inet Domain (1)

- Server steps
  1) create socket

```c
sfd = socket(AF_INET, SOCK_STREAM, 0);
```
Streams in the Inet Domain (2)

• Server steps
  2) bind name to socket

```c
bind(sfd,
    (struct sockaddr *)&my_addr, sizeof(my_addr));
```

128.148.47.67
Some Details …

- Server may have multiple interfaces; we want to be able to receive on all of them

```c
struct sockaddr_in {
    sa_family_t sin_family;
    in_port_t sin_port;
    struct in_addr sin_addr;
} my_addr;

my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htons(port);
```

“Wildcard” address
Streams in the Inet Domain (3)

- **Server steps**
  3) put socket in “listening mode”

  ```c
  int listen(int sfd, int MaxQueueLength);
  ```
Streams in the Inet Domain (4)

- Client steps
  1) create socket

```java
    cfd = socket(AF_INET, SOCK_STREAM, 0);
```

`cfd`
Streams in the Inet Domain (5)

- Client steps
  2) connect to server

```c
connect(cfd, (struct sockaddr *)&server_addr,
        sizeof(server_addr));
```

128.137.23.6:43

cfd

128.148.47.67:7326

sfd

128.148.47.67:7326
Streams in the Inet Domain (6)

• Server steps
  3) accept connection

```c
fd = accept((int)sfd, (struct sockaddr *)&addr, (int *)&addrlen);
```

128.137.23.6:43

128.148.47.67:7326
Inet Stream Example (1)

• Server side

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[]) {
    struct sockaddr_in my_addr;
    int lsock;
    void serve(int);
    if (argc != 2) {
        fprintf(stderr, "Usage: tcpServer port\n");
        exit(1);
    }
}
```
Inet Stream Example (2)

    // Step 1: establish a socket for TCP
    if ((lsock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }
Inet Stream Example (3)

/* Step 2: set up our address */
memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htons(atoi(argv[1]));

/* Step 3: bind the address to our socket */
if (bind(lsock, (struct sockaddr *)&my_addr,
        sizeof(my_addr)) < 0) {
    perror("bind");
    exit(1);
}
Inet Stream Example (4)

/* Step 4: put socket into “listening mode” */
if (listen(lsock, 100) < 0) {
    perror("listen");
    exit(1);
}

while (1) {
    int csock;
    struct sockaddr_in client_addr;
    int client_len = sizeof(client_addr);

    /* Step 5: receive a connection */
    csock = accept(lsock,
        (struct sockaddr *)&client_addr, &client_len);
    printf("Received connection from %s#%hu\n",
        inet_ntoa(client_addr.sin_addr), client_addr.sin_port);
Inet Stream Example (5)

```c
switch (fork()) {
    case -1:
        perror("fork");
        exit(1);
    case 0:
        // Step 6: create a new process to handle connection
        serve(csock);
        exit(0);
    default:
        close(csock);
        break;
}
```
Inet Stream Example (6)

```c
void serve(int fd) {
    char buf[1024];
    int count;

    // Step 7: read incoming data from connection
    while ((count = read(fd, buf, 1024)) > 0) {
        write(1, buf, count);
    }
    if (count == -1) {
        perror("read");
        exit(1);
    }
    printf("connection terminated\n");
}
```
Inet Stream Example (7)

- **Client side**

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
// + more includes ...

int main(int argc, char *argv[]) {
    int s, sock;
    struct addrinfo hints, *result, *rp;

    char buf[1024];
    if (argc != 3) {
        fprintf(stderr, "Usage: tcpClient host port\n");
        exit(1);
    }
```
// Step 1: find the internet address of the server
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

if ((s=getaddrinfo(argv[1], argv[2], &hints, &result)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(1);
}
Inet Stream Example (9)

// Step 2: set up socket for TCP and connect to server
for (rp = result; rp != NULL; rp = rp->ai_next) {
    // try each interface till we find one that works
    if ((sock = socket(rp->ai_family, rp->ai_socktype,
        rp->ai_protocol)) < 0) {
        continue;
    }
    if (connect(sock, rp->ai_addr, rp->ai_addrlen) >= 0) {
        break;
    }
    close(sock);
}
if (rp == NULL) {
    fprintf(stderr, "Could not connect to %s
", argv[1]);
    exit(1);
}
freeaddrinfo(result);
Inet Stream Example (10)

// Step 3: send data to the server
while (fgets(buf, 1024, stdin) != 0) {
    if (write(sock, buf, strlen(buf)) < 0) {
        perror("write");
        exit(1);
    }
}

return 0;
}
Quiz 1

The previous slide contains
\texttt{write(sock, buf, strlen(buf))}

If data is lost and must be retransmitted
a) write returns an error so the caller can retransmit the data.

b) nothing happens as far as the application code is concerned, the data is retransmitted automatically.
Quiz 2

A previous slide contains
\texttt{write(sock, buf, strlen(buf))}

We lose the connection to the other party (perhaps a network cable is cut).

a) write returns an error so the caller can reconnect, if desired.

b) nothing happens as far as the application code is concerned, the connection is reestablished automatically.
Stream Relay

Source

Relay

Sink

Pipe

Pipe

Pipe

Pipe
Solution?

```c
while(...) {
    size = read(left, buf, sizeof(buf));
    write(right, buf, size);
    size = read(right, buf, sizeof(buf));
    write(left, buf, size);
}
```
Select System Call

```c
int select(
    int nfds,         // size of fd_sets
    fd_set *readfds,  // descriptors of interest
                     // for reading
    fd_set *writefds, // descriptors of interest
                     // for writing
    fd_set *excpfds,  // descriptors of interest
                     // for exceptional events
    struct timeval *timeout
                     // max time to wait
);
```
Relay Sketch

```c
void relay(int left, int right) {
    fd_set rd, wr;
    int maxFD = max(left, right) + 1;
    FD_ZERO(&rd); FD_SET(left, &rd); FD_SET(right, &rd);
    FD_ZERO(&wr); FD_SET(left, &wr); FD_SET(right, &wr);
    while (1) {
        select(maxFD, &rd, &wr, 0, 0);
        if (FD_ISSET(left, &rd))
            read(left, bufLR, BSIZE);
        if (FD_ISSET(right, &rd))
            read(right, bufRL, BSIZE);
        if (FD_ISSET(right, &wr))
            write(right, bufLR, BSIZE);
        if (FD_ISSET(left, &rd))
            write(left, bufRL, BSIZE);
    }
}
```
Quiz 3

40 bytes have been read from the left-hand source. Select reports that it is ok to write to the right-hand sink.

a) You’re guaranteed you can immediately write all 40 bytes to the right-hand sink

b) All that’s guaranteed is that you can immediately write at least one byte to the right-hand sink

c) Nothing is guaranteed
void relay(int left, int right) {
    fd_set rd, wr;
    int left_read = 1, right_write = 0;
    int right_read = 1, left_write = 0;
    int sizeLR, sizeRL, wret;
    char bufLR[BSIZE], bufRL[BSIZE];
    char *bufpR, *bufpL;
    int maxFD = max(left, right) + 1;
while(1) {
    FD_ZERO(&rd);
    FD_ZERO(&wr);
    if (left_read)
        FD_SET(left, &rd);
    if (right_read)
        FD_SET(right, &rd);
    if (left_write)
        FD_SET(left, &wr);
    if (right_write)
        FD_SET(right, &wr);

    select(maxFD, &rd, &wr, 0, 0);
if (FD_ISSET(left, &rd)) {
    sizeLR = read(left, bufLR, BSIZE);
    left_read = 0;
    right_write = 1;
    bufpR = bufLR;
}
if (FD_ISSET(right, &rd)) {
    sizeRL = read(right, bufRL, BSIZE);
    right_read = 0;
    left_write = 1;
    bufpL = bufRL;
}
if (FD_ISSET(right, &wr)) {
    if ((wret = write(right, bufpR, sizeLR)) == sizeLR) {
        left_read = 1; right_write = 0;
    } else {
        sizeLR -= wret; bufpR += wret;
    }
}
if (FD_ISSET(left, &wr)) {
    if ((wret = write(left, bufpL, sizeRL)) == sizeRL) {
        right_read = 1; left_write = 0;
    } else {
        sizeRL -= wret; bufpL += wret;
    }
}
return 0;
A Really Simple Protocol

• Transfer a file
  – layered on top of TCP
    » reliable
    » indicates if connection is closed

• To send a file
  P<null-terminated pathname><contents of file>

• To retrieve a file
  G<null-terminated pathname>
Server State Machine

RDY
- recv: P<pathname>
- send: G
- recv: !((P||G) || invalid pathname)
- send: B

GOOD IN
- send: G

GOOD OUT
- send: G

BAD

transfer IN
- recv: data

EOF

transfer OUT
- send: data

Done
typedef struct client {
    int fd;       // file descriptor of local file being transferred
    int size;    // size of out-going data in buffer
    char buf[BSIZE];
    enum state {RDY, BAD, GOOD, TRANSFER} state;
    /*
     * states:
     * RDY: ready to receive client's command (P or G)
     * BAD: client's command was bad, sending B response + error msg
     * GOOD: client's command was good, sending G response
     * TRANSFER: transferring data
     *
     */
    enum dir {IN, OUT} dir;
    /*
     * IN: client has issued P command
     * OUT: client has issued G command
     *
     */
} client_t;
Keeping Track of Clients

```c
client_t clients[MAX_CLIENTS];
for (i=0; i < MAX_CLIENTS; i++)
    clients[i].fd = -1; // illegal value
```
Main Server Loop

```c
while(1) {
    select(maxfd, &trd, &twr, 0, 0);
    if (FD_ISSET(lsock, &trd)) {
        // a new connection
        new_client(lsock);
    }
    for (i=lsock+1; i<maxfd; i++) {
        if (FD_ISSET(i, &trd)) {
            // ready to read
            read_event(i);
        }
        if (FD_ISSET(i, &twr)) {
            // ready to write
            write_event(i);
        }
    }
    trd = rd; twr = wr;
}
```
New Client

// Accept a new connection on listening socket
// fd. Return the connected file descriptor

int new_client(int fd) {
    int cfd = accept(fd, 0, 0);
    clients[cfd].state = RDY;
    FD_SET(cfd, &rd);
    return cfd;
}
Read Event (1)

// File descriptor fd is ready to be read. Read it, then handle
// the input

void read_event(int fd) {
    client_t *c = &clients[fd];
    int ret = read(fd, c->buf, BSIZE);
    switch (c->state) {
    case RDY:
        if (c->buf[0] == 'G') {
            // GET request (to fetch a file)
            c->dir = OUT;
            if ((c->fd = open(&c->buf[1], O_RDONLY)) == -1) {
                // open failed; send negative response and error message
                c->state = BAD;
                c->buf[0] = 'B';
                strncpy(&c->buf[1], strerror(errno), BSIZE-2);
                c->buf[BSIZE-1] = 0;
                c->size = strlen(c->buf)+1;
            }
        } else {
            // other states
            ...}
    }
}
else {
    // open succeeded; send positive response
    c->state = GOOD;
    c->size = 1;
    c->buf[0] = 'G';
}
// prepare to send response to client
FD_SET(fd, &wr);
FD_CLR(fd, &rd);
break;
}
if (c->buf[0] == 'P') {
    // PUT request (to create a file)
    c->dir = IN;
    if ((c->fd = open(&c->buf[1],
            O_RDWR|O_CREAT|O_TRUNC, 0666)) == -1) {
        // open failed; send negative response and error message
        ...
    } else {
        // open succeeded; send positive response
        ...
    }
    // prepare to send response to client
    FD_SET(fd, &wr);
    FD_CLR(fd, &rd);
    break;
}
case TRANSFER:
    // should be in midst of receiving file contents from client
    if (ret == 0) {
        // eof: all done
        close(c->fd);
        close(fd);
        FD_CLR(fd, &rd);
        break;
    }
    if (write(c->fd, c->buf, ret) == -1) {
        // write to file failed: terminate connection to client
        ...    
        break;
    }
    // continue to read more data from client
    break;
}
Write Event (1)

// File descriptor fd is ready to be written to. Write to it, then, // depending on current state, prepare for the next action.

```c
void write_event(int fd) {
    client_t *c = &clients[fd];
    int ret = write(fd, c->buf, c->size);
    if (ret == -1) {
        // couldn't write to client; terminate connection
        close(c->fd);
        close(fd);
        FD_CLR(fd, &wr);
        c->fd = -1;
        perror("write to client");
        return;
    }
    switch (c->state) {
```

Write Event (2)

case BAD:
   // finished sending error message; now terminate client connection
   close(c-&gt;fd);
   close(fd);
   FD_CLR(fd, &wr);
   c-&gt;fd = -1;
   break;
Write Event (3)

case GOOD:
    c->state = TRANSFER;
    if (c->dir == IN) {
        // finished response to PUT request
        FD_SET(fd, &rd);
        FD_CLR(fd, &wr);
        break;
    }
    // otherwise finished response to GET request, so proceed
Write Event (4)

case TRANSFER:
    // should be in midst of transferring file contents to client
    if ((c->size = read(c->fd, c->buf, BSIZE)) == -1) {
        ...
        break;
    } else if (c->size == 0) {
        // no more file to transfer; terminate client connection
        close(c->fd);
        close(fd);
        close(fd);
        FD_CLR(fd, &wr);
        c->fd = -1;
        break;
    }
    // continue to write more data to client
    break;
}
Problems

• Works fine as long as the protocol is followed correctly
  – can client (malicious or incompetent) cause server to misbehave?
• How can the server limit the number of clients?
• How does server limit file access?