Socket Programming

Goal of Today’s Lecture:

• Learn how to build client-server Internet applications using the socket API
Client-Server Communication

- **Client** is “sometimes on”
  - Initiates a request to the server when interested
  - Needs to know the server’s address

- **Server** is “always on”
  - Services requests from many clients
  - Doesn’t initiate contact with the clients
Sockets

- A socket is an application-created, OS-controlled interface through which an application process can send/receive messages to/from another application process.
- The socket interface provides a syntax by which the transport layer services of the Internet can be invoked.
Identifying the Receiving Process

- Sending process identifies the receiving process by \(<\text{host, port}\>\) pair
  - The receiving host is identified by the 32-bit IP address
  - The receiving process is identified by the 16-bit port number
Port Numbers

- Servers of popular applications use well-known ports
  - Well known ports are between 0 and 1023
  - Examples: port 80 for Web server, port 25 for email server, port 53 for DNS name server

- Clients use ephemeral ports
  - Between 1024 and 65535
Typical Client Program Using TCP

1. Prepare to communicate
   1.1 Create a socket
   1.2 Determine server IP address and port number
   1.3 Initiate the connection to the server

2. Exchange data with the server
   2.1 Write data (i.e., request) to the socket
   2.2 Read data (i.e., response) from the socket
   2.3 Do stuff with the data (e.g., display a Web page)

3. Close the socket
Typical Server Program Using TCP

1. Prepare to communicate
   1.1 Create a socket \texttt{s\_listen} (i.e., the listening socket)
   1.2 Associate server’s IP address and port no. with the socket

2. Wait to hear from a client
   2.1 Indicate how many connections can be pending on the socket
   2.2 Accept an incoming connection from a client, create a \texttt{new socket} \texttt{s\_new} for the client

3. Exchange data with the client over the new socket \texttt{s\_new}
   3.1 Read data (i.e., client request) from the socket
   3.2 Handle the request
   3.3 Write data (i.e., server response) to the socket
   3.4 Close the socket \texttt{s\_new}

4. Repeat 2.2-3.4 with the next connection request
Socket Programming Using TCP: Flow Diagram

Establish TCP connection

Data (request)

Data (response)
#include <sys/socket.h>
int socket(int family, int type, int protocol)
// Create a socket (called by client and server)
- protocol family: AF_INET for Internet family
- socket type:
  - SOCK_STREAM: TCP socket
  - SOCK_DGRAM: UDP socket
- protocol: 0
- returns a socket descriptor on success, -1 on error.
bind()

#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen)
   // Bind a local address to a socket (called by server)
   sockfd: a socket descriptor
   struct sockaddr {
       uint8_t sa_len;  /* length of structure (16 bytes) */
       sa_family_t sa_family;  /* address family */
       char sa_data[14];  /* protocol-specific address */
   } //The generic socket address structure
   addrlen: size of the address structure
   Returns 0 on success, -1 on error.
The Internet Socket Address Structure

```
struct sockaddr_in {
    uint8_t sin_len;    /* length of structure */
    sa_family_t sin_family; /* AF_INET */
    in_port_t sin_port;   /* 16-bit port no., network byte ordered */
    struct in_addr sin_addr; /* 32-bit IP address, network byte ordered */
    char sin_zero[8];     /* unused */
};

struct in_addr {
    in_addr_t s_addr; /* 32-bit IP address, network byte ordered */
};
```
Host Byte Order

- Hosts differ in how they store data
- Little endian ("little end comes first")
  - Least significant byte stored at the lowest memory address
  - E.g., Intel x86 and x86-64 series of processors
- Big endian ("big end comes first")
  - Most significant byte stored at the lowest memory address
  - E.g., Motorola 6800 and 68k series of processors
Network Byte Order

- Internet protocols use big endian byte ordering (aka “network byte order”)
- Writing portable code requires conversion
  - htons() and htonl(): convert from host byte order to network byte order
  - ntohs() and ntohl(): convert from network byte order to host byte order
  - ‘s’ for 16-bit integer; ‘l’ for 32-bit integer
bind(): An Example

int sockfd;
struct sockaddr_in myaddr;

if ((sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0)
    { printf("socket error!\n"); exit(1); }

bzero(&myaddr, sizeof(myaddr));
myaddr.sin_family=AF_INET;
myaddr.sin_port=htons(5100);
myaddr.sin_addr.s_addr=htonl(INADDR_ANY); // allow the server to accept a
    client connection on any interface

if (bind(sockfd, (struct sockaddr *) &myaddr, sizeof(myaddr)) < 0)
    { printf("bind error!\n"); exit(1); }

You must cast the pointer to sockaddr_in into a pointer to sockaddr in bind()!
Listen(), Accept(), Connect()

int listen(int sockfd, int backlog)
// Listen for connections on a socket (called by server)
- backlog specifies the maximum number of pending connections the
  kernel should queue for the socket.
- Returns 0 if OK, -1 on error

int accept(int sockfd, struct sockaddr *fromaddr, socklen_t *addrlen)
// accept a new connection on a socket (called by server)
- Take the first connection off the queue for sockfd and create a new
  socket (the return value) for communicating with client
- The address of the client socket is returned in fromaddr
- addrlen is length of the socket address structure
- Returns a new socket descriptor if OK, -1 on error

int connect(int sockfd, struct sockaddr *toaddr, socklen_t addrlen)
// Establish a connection with a server (called by client)
- toaddr contains the IP address and port number of the server
- addrlen is length of the socket address structure
- Returns 0 if OK, -1 on error
Sending and Receiving Data

ssize_t read(int sockfd, void *buffer, size_t n)
  - Read up to n bytes from sockfd into buffer.
  - Returns the number of bytes read on success (0 indicates end of file), -1 on error.

ssize_t write(int sockfd, const void *buffer, size_t n)
  - Write up to n bytes from buffer to sockfd
  - Returns the number of bytes written on success, -1 on error.
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MYPORT 3490    // the port clients will be connecting to
#define BACKLOG 10     // how many pending connections queue will hold

int main()
{
    int sockfd, new_fd; // listen on sockfd, new connection on new_fd
    struct sockaddr_in myaddr; // my address information
    struct sockaddr_in cliaddr; // client's address information
    socklen_t len;
    char buff[80];
A Simple Server Using TCP Socket (cont’d)

```c
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    printf("socket error!\n");
    exit(1);
}

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

if (bind(sockfd, (struct sockaddr *)&myaddr, sizeof(myaddr)) == -1) {
    printf("bind error!\n");
    exit(1);
}

if (listen(sockfd, BACKLOG) == -1) {
    printf("listen error!\n");
    exit(1);
}

18
```
A Simple Server Using TCP Socket (cont’d)

```c
while(1) {
    len = sizeof(cliaddr);
    if ((new_fd = accept(sockfd, (struct sockaddr *)&cliaddr, &len)) == -1) {
        printf("accept error!\n");
        exit(1);
    }

    printf("Connection from: %s, Port: %d\n", 
            inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
            ntohs(cliaddr.sin_port)); // print client’s IP address and port number

    if (write(new_fd, "Hello, world!\n", 14) < 0) { //send “Hello, world!” to client
        printf("send error!\n");
        exit(1);
    }

    close(new_fd);
} //end of while loop
```
A Simple Client Using TCP Socket

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 3490 // the port client will be connecting to
#define MAXDATASIZE 100 // max number of bytes we can get at once

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in servaddr; // server's address information
```
A Simple Client Using TCP Socket (cont’d)

if (argc != 2) {
    printf("usage: client servername\n");
    exit(1);
}
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    printf("socket error!\n");
    exit(1);
}
if ((he=gethostbyname(argv[1])) == NULL) {  // get the host info of server
    printf("gethostbyname error!\n");
    exit(1);
}
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(PORT);
memcpy(&servaddr.sin_addr, he->h_addr, he->h_length); //copy server’s IP address to servaddr.sin_addr
A Simple Client Using TCP Socket (cont'd)

```c
if (connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1
{
    printf("connect error!\n");
    exit(1);
}
if ((numbytes = read(sockfd, buf, MAXDATASIZE)) < 0) {
    printf("read error!\n");
    exit(1);
}
printf("Received: %s",buf);
close(sockfd);
```
The hostent Structure

The hostent structure stores information about a given host.

```c
struct hostent {
    char  *h_name;  /* official name of host */
    char **h_aliases;  /* alias list */
    int h_addrtype;  /* host address type (AF_INET or AF_INET6) */
    int h_length;  /* length of address */
    char **h_addr_list;  /* list of addresses in network byte order*/
}

#define h_addr h_addr_list[0]  /* The macro h_addr is defined to be the first address in h_addr_list */
```