COE 4DN4 - 2015
Advanced Internet Communications

Basic Sockets in C (with some Java and Python) – version 1, 2015

- Chapters 1 – 5 of book "TCP/IP Sockets in C"

Prof. Ted Szymanski
Dept. ECE. McMaster University
www.ece.mcmaster.ca/faculty/teds/COURSES

Reference Textbooks (2) :

* **TCP/IP Sockets in C**, M.J. Donahoo, K.L. Calvert, Morgan Kaufmann, ISBN 1-55860-826-5 (some figures & code are from this reference text)

* **TCP/IP Sockets in JAVA**, M.J. Donahoo, K.L. Calvert, Morgan Kaufmann, ISBN 9780123742551 (some figures & code are from this reference text)

Introduction - 2015

- In 2015, 4DN4 will undergo a big change in the class notes and labs, the biggest change in the last 5-10 years
- Previously, we have been using an excellent paperback book '*TCP/IP Sockets in C*' to guide the class notes
- Previously, we have been using an excellent paperback book '*TCP/IP Sockets in JAVA*' to guide the class notes
- Previously, we have also tried the paperback book '*TCP/IP Sockets in C#*' to guide the class notes
- Unfortunately, all these languages, and their software 'Integrated Development Environments' (IDEs), have proven to be not-too-easy to use
- In previous years, it could take a good fraction of the class several weeks just to install the IDEs (be it KDE IDE, Windows Visual C/C++, Netbeans, etc)

Introduction - 2015

• Last year, I used the Netbeans 7.2 IDE, which runs C and Java programs, on most operating systems (Windows, MAC OS X, Linux)

• In Jan. 2015, I spent a few hours trying to update my Netbeans 7.2 IDE to Netbeans 8.0 on my MAC computer running OS 10.6, before the start of 4DN4 classes

• It was unsuccessful, since Netbeans 8.0 needs to use Java 8.0, which only runs on MAC OS 10.8, which I cannot download for my laptop (I need to go and buy disks, which would take several days !)

• This experience prompted me to try the new language 'Python', which I have heard good things about (I have never programmed in Python)

• I managed in 1 hour to download the Python interpretter/compiler, complete several exercises, and managed to get a simple Client-Server program running, all in less time than it took me to try to install Netbeans 8.0

• The client-server software was much simpler too

Introduction - 2015

• Based on this brief but impressive 1 hour experience, I plan to add more Python to these 4DN4 notes in 2015

• Unfortunately, there is no concise book called "*TCP/IP Sockets in Python*" available, as the Python language is too new

• Therefore, I will continue to present and discuss C socket code from the textbook "*TCP/IP Sockets in C*"

• After examining the C code, I will present some Python code which we can analyse

• This will provide a good opportunity to learn about socket code in both the C and Python languages

• I will add more Python to these notes as we learn more about Python

• Hence, much of these notes are based on the book "*TCP/IP Sockets in C*"

Computer Communications

- How do we make computers talk?
  Internet Protocol (IP)

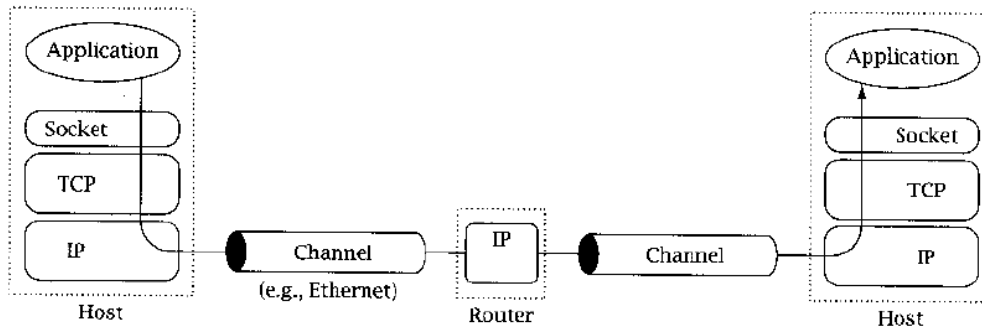


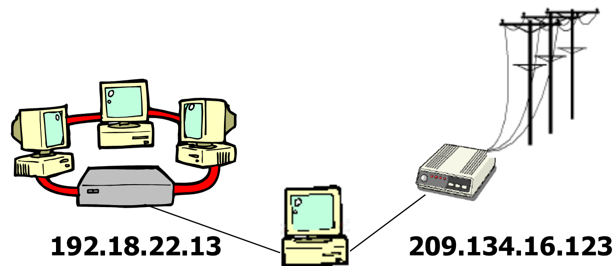
Figure 1.1: A TCP/IP network.

Internet Protocol (IP)

- Datagram (packet) protocol
- Best-effort service, no QOS guarantees
  - Loss allowed
  - Reordering "
  - Duplication "
  - Delay "
- Host-to-host delivery, not application program-to-program delivery

IPv4 Address

- 32-bit identifier (increased for IPv6)
- Dotted-quad notation: 192.118.56.25 (4 bytes)
- www.mkp.com -> 167.208.101.28
- Identifies a host (machine) interface (not a host)
- One host (machine) may have several interfaces (Ethernet, ATM, WiFi, WiMAX) and several IP addresses



Transport Protocols

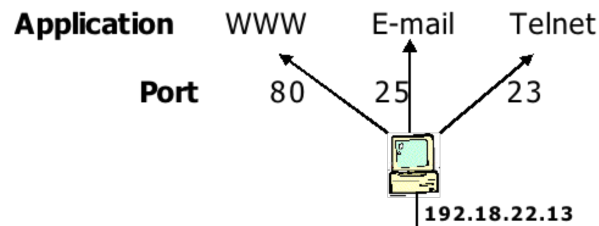
IP's Best-effort is not sufficient for most applications !

- TCP & UDP protocols (layer 4) add services on top of IP (layer 3)
- User Datagram Protocol (UDP)
  - Data checksum for error detection
  - Best-effort
  - Sends a message as one packet (<= 64KBytes) , best-effort
- Transmission Control Protocol (TCP)
  - Data checksum for error detection
  - Reliable byte-stream delivery using ARQ protocol
  - Sends a message, any size, as potentially many packets
  - Flow and congestion control
  - Send and receive buffers guaranteed to never overflow

Ports

Identifying the ultimate destination - an Application Program

- IP addresses identify host interfaces (machines)
- Host has many applications (email, web-browser, FTP, video-player)
- Ports (16-bit identifier) identify one target application on the host
- Internet Assigned Number Authority (IANA at http://www.iana.org/assignments/port-numbers) maintains assignment of default ports for common services such as FTP.
- Ports (1024-65535) reserved for user-developed programs; ports 1-1023 reserved for common web applications (email, ftp, etc)



Some Well known Port Numbers

Dix, Alan. Unix Network Programming with TCP/IP, Short Course Notes, 1996. Available from: http://www.hiraeth.com/alan/tutorials pp 21.

Service	Port no	Protocol	
echo	7	UDP/TCP	sends back what it receives
discard	9	UDP/TCP	throws away input
daytime	13	UDP/TCP	returns ASCII time
chargen	19	UDP/TCP	returns characters
ftp	21	TCP	file transfer
telnet	23	TCP	remote login
smtp	25	TCP	email
daytime	37	UDP/TCP	returns binary time
tftp	69	UDP	trivial file transfer
finger	79	TCP	info on users
http	80	TCP	World Wide Web
login	513	TCP	remote login
who	513	UDP	different info on users
Xserver	6000	TCP	X windows (N.B. >1023)

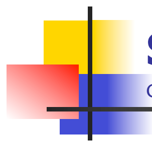
Client-Server Model (pg 7)

- Server application program runs on a host, waits for calls from clients
- Client programs initiate communications; needs to know server's host IP address & port number
- Universal Resource Locator (URL) such as http://www.dns.com identifies the server, and a name resolution service translates a URL to IP Host address
- The port number depends upon the application.
- Internet Assigned Number Authority (IANA at http://www.iana.org/assignments/port-numbers) maintains assignment of default ports for common services such as FTP and email. These are ports from 0-1K.

Sockets

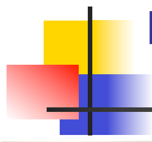
How does one speak TCP/IP?

- The Sockets "Application Programming Interface" (API) provides interface to TCP/IP
- Generic interface for many protocols
- Applications can "plug-in" to the Internet, hence the name 'sockets'



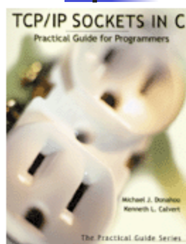
Sockets History (Shah,Linux Administration: A Beginner's Guide. Second Edition, McGraw Hill, 2000)

- Background
  - Berkeley Software Distribution (BSD) 1983
  - 15,000 Lines-of-Code, source code released 1994
  - Ported to Unix, Linux, and many non Unix Systems.
  - Ported to Apple Mac -> MacTCP
  - Ported to MS Windows –> **WinSock** (similar to sockets)
- API (Application Programmers Interface)
  - Not part of any standard
  - Depends on the platform:
    - UNIX TCP/IP API are 'kernel' system calls (to the OS)
    - Mac & Windows are extensions/drivers (external to the OS)



Downloading the Unix C Socket Code

http://cs.baylor.edu/~donahoo/practical/Csockets/textcode.html



Below is the example source code from "TCP/IP Sockets in C: Practical Guide for Programmers" by Michael J. Donahoo and Kenneth L. Calvert. This book can be ordered at your favorite local bookstore or online.

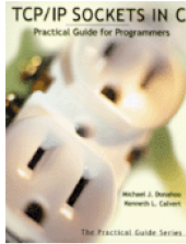
Official Book Website

Disclaimer: The purpose of this book is to provide general information about network programming as of the book's publication date. The authors have included sample code that is intended for the sole purpose of illustrating the use of the sockets API. Neither the authors nor the publisher are aware of any third party patents or proprietary rights that may cover any sample of any kind. The authors and the Publisher DISCLAIM ALL EXPRESS AND IMPLIED WARRANTIES, including warranties of merchantability and fitness for any particular purpose. Your use or reliance upon any sample code or other information in this book will be at your own risk. No one should use any sample code (or illustrations) from this book in any software application without first obtaining competent legal advice.

Example code:

- TCPEchoClient.c
- DieWithError.c
- TCPEchoServer.c
- HandleTCPClient.c
- UDPEchoClient.c
- UDPEchoServer.c

Winsock Adaptations of Example Code



Below is the example source code from "TCP/IP Sockets in C: Practical Guide for Programmers" by Michael J. Donahoo and Kenneth L. Calvert modified for use with WinSock. This book can be ordered at your favorite local bookstore or online.

Official Book Website

The code below demonstrates the minimal number of changes required to make the examples from the text execute under Winsock. Further changes can be made to make this code more Winsock complaint (e.g., test socket() failure return value as SOCKET_ERROR rather than < 0). If you want some details/justifications for the adaptations of the examples to Winsock, see Transitioning from UNIX to Windows Socket Programming by Paul O'Steen

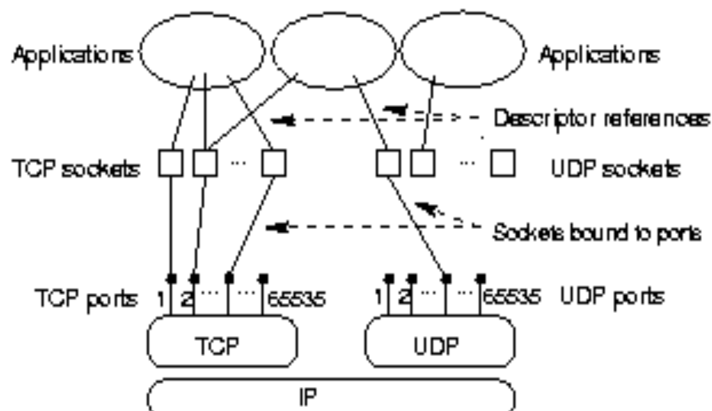
WinSock Example code:

- TCPEchoClientWS.c
- DieWithErrorWS.c
- TCPEchoServerWS.c
- HandleTCPClientWS.c

http://cs.baylor.edu/~donahoo/practical/Csockets/winsock.html

Sockets (Fig. 1.2., pg.8, ref. Text)

- Socket is a 5-tuple consisting of protocol identifier, and the local and remote IP address and port
- One Application may refer to many sockets
- Sockets can be accessed by many applications



Sockets

- 2 types of sockets: “datagram” for UDP and “stream” for TCP
- UDP (Datagram) sockets connect the application to IP directly, bypassing TCP
- TCP (Stream) sockets connect the application to TCP then to IP
- Each Datagram packet up to 64 Kbytes in length
- Stream sockets use smaller packets and acknowledgments (for ARQ protocol)

- In the UNIX OS, when you open a new IO file, you get a file descriptor
- In the Socket API, when you open a new socket, you get a socket descriptor (an integer), to be used to access that socket

Sockets - C Constants (pg 9)

- Sockets can have several protocol families; we will use the INTERNET protocol family, denoted by C constant **PF_INET**
- The INET protocol family has 2 protocols that we’ll use: **IPPROTO_TCP** and **IPPROTO_UDP**
- Sockets can have several address families ; we will use the INTERNET protocol address family, denoted by C constant **AF_INET**
- Note: The creators envisioned that one protocol family can have several address families: IPv4 and IPv6 is one example of 2 addressing schemes
- Sockets can have 2 types: **SOCK_STREAM** and **SOCK_DGRAM**

TCP/IP Socket Creation in C (pg 9)

- mySocket = socket(protocol family, type, protocol); % general form

- For TCP/IP sockets:
  - mySocket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

- For UDP sockets:
  - mySocket = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);

- These C functions return a
  - Socket descriptor (integer index into an array, 0, 1, 2 etc)
  - Socket "handle" in WinSock

SockAddr C data-structure (pg 10)

- The C sockets API defines a generic data structure, SockAddr, for specifying addresses associated with sockets (see next slide)
- The Internet address family (**AF_INET**) specifies a specific SockAddr format using a 32-bit IP address and 16-bit port # (see next slide)
- Once the SockAddr data structure is initialized, it can be passed to the socket creation/deletion functions, which look at the 1st field (protocol_family) to determine how to interpret the rest of the data structure

- Lets adopt a Class Terminology:
- A C data-structure has several fields
- A C program has several arguments (program.exe arg1 arg2 arg3…)
- A C function or procedure has several parameters

Specifying Addresses in C

- **Generic Data Structure (16 bytes):**
- **struct sockaddr**

```
{
        unsigned short sa_family;     /* Address family (e.g., AF_INET), 2 bytes */
        char sa_data[14];             /* Protocol-specific address information, 14 bytes */
};
```

IPv4 Specific data Structure (the same 16 bytes, interpretted this way):

- **struct sockaddr_in**

```
{
        unsigned short  sin_family;        /* Internet protocol (AF_INET) 2 bytes */
        unsigned short  sin_port;          /* Port (16-bits), 2 bytes */
        struct in_addr  sin_addr;          /* IP v4 address (32-bits) */
        char   sin_zero[8];                /* Not used, 8 bytes */
};
struct in_addr
{
        unsigned long s_addr;              /* IP v4 address (32-bits) */
};
```


Clients and Servers

- Client: Initiates the connection
- Server: Passively waits to respond

Client: Bob

Server: Jane

"Hi, Bob. I'm Jane"

TCP Client/Server Interaction

The steps involved at each end:

Client

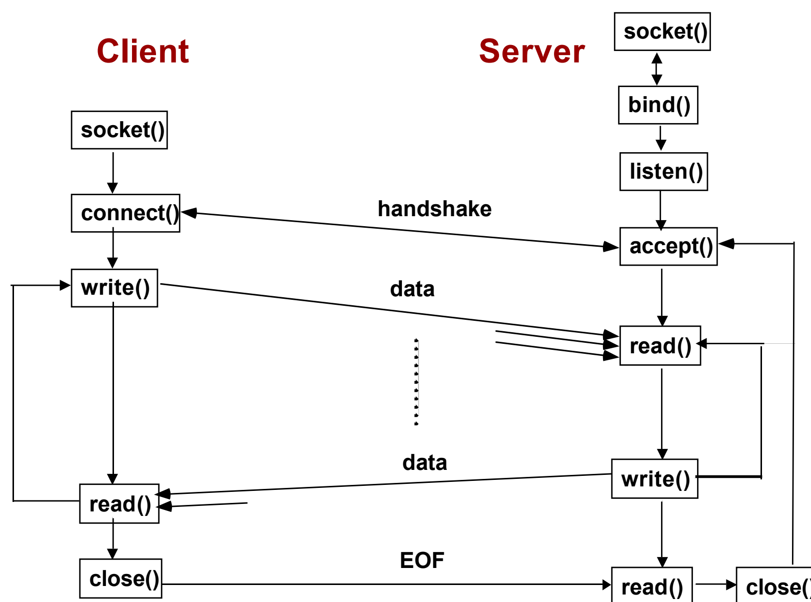
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket (using 'bind')
3. Set socket to listen
4. Loop:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Ted Szymanski, McMaster

TCP Client/Server Interaction



Ted Szymanski, McMaster

Client-Server Example: EchoClient (pg 13)

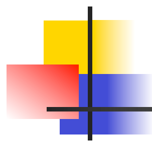
- Consider a simple server which simply echos whatever it gets
- the client code = **TCPEchoClient.c**, available at the Donahoo web site
- the server code = **TCPEchoServer.c**, available at the Donahoo web site

- EchoClient-Server useful for debugging code, so most systems provide such a server, using port 7

TCPEchoClient.c - UNIX C Headers (pg 13)

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */
#define RCVBUFSIZE 32   /* Size of receive buffer, 32 bytes for now */

void DieWithError(char *errorMessage);  /* Error handling function */
```

- These headers work for UNIX and the Mac OSX (which is Berkeley UNIX)
- The Windows API (WINSOCK) uses slightly different headers
- Linux probably uses the UNIX headers



Converting Unix to Winsock

- Please see the file 'Unix-to-Windows-Sockets' in our class web-site
- Below is an example of one change you need to make

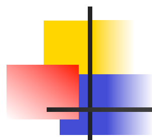
- Please see the file '4dn4_Visual_C_tutorial' in our class web-site for help getting started in Visual_C

UNIX vs. Windows Sockets

To demonstrate the key differences, we perform a side-by-side comparison between the UNIX and adapted Windows Socket code. We break the programs into separate sections, each of which deals with a specific difference. Comments have been removed from the programs to allow focus on the key differences.

UNIX	Windows

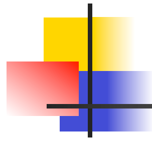
`#include <stdio.h>`
`#include <sys/socket.h>`
`#include <arpa/inet.h>`
`#include <stdlib.h>`
`#include <string.h>`
`#include <unistd.h>`
	`#include <stdio.h>`
	`#include <winsock.h>`
	`#include <stdlib.h>`



TCPEchoClient.c - Arguments

```
int main(int argc, char *argv[])
{
int sock;                         /* Socket descriptor */
struct sockaddr_in echoServAddr; /* Echo server address, 16 bytes */
unsigned short echoServPort;     /* Echo server port, 2 bytes */
char *servIP;                    /* Server IP address (dotted quad), pointer to  */
char *echoString;                /* String to send to echo server, pointer to */
char echoBuffer[RCVBUFSIZE];     /* Buffer for echo string */
unsigned int echoStringLen;      /* Length of string to echo */
int bytesRcvd, totalBytesRcvd;   /* Bytes read in single recv() and total bytes read */

if ((argc < 3) || (argc > 4))    /* Test for correct number of arguments */
{
        fprintf(stderr, "Usage: %s <Server IP> <Echo Word> [<Echo Port>]\n",argv[0]);
        exit(1);
}
```

- The first argument, **argv[0],** is supplied by the operating system: it is a string which identifies the program name (ie main)

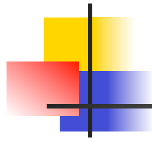


EchoClient.c - Socket creation

```
servIP     = argv[1]; /* First user arg: server IP address (dotted quad, string) */
echoString = argv[2]; /* Second user arg: string to echo */
if (argc == 4)
        echoServPort = atoi(argv[3]); /* Use given port, if specified */
else
        echoServPort = 7;  /* 7 is the well-known port for the echo service */


/* Create a TCP (reliable) stream socket */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");


/* Construct the 16-byte IPv4 server address structure */
memset(&echoServAddr, 0, sizeof(echoServAddr));     /* Zero out structure */
echoServAddr.sin_family      = AF_INET;             /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP);   /* Server IP address, 4 bytes */
echoServAddr.sin_port        = htons(echoServPort); /* Server port, 2 bytes */



/* htons() maps “host-format” numbers (big-endian or little-endian) to a universal
“network-format” numbers (big-endian only) */
```

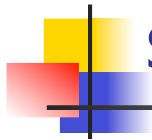
TCPEchoClient.c - Connect & Send

```
/* Establish the connection to the echo server */
if (connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
        DieWithError("connect() failed");


echoStringLen = strlen(echoString);          /* Determine input length */

/* Send the string to the server */
if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
        DieWithError("send() sent a different number of bytes than expected");




/* strlen() counts the number of bytes, until the delimiter ‘\n’ */
```



TCPEchoClient.c - Receive

```
/* Receive the same string back from the server */
totalBytesRcvd = 0;
printf("Received: ");   /* Setup to print the echoed string */


while (totalBytesRcvd < echoStringLen)
{
        /* Receive up to the buffer size (minus 1 to leave space */
        /* for a null terminator) bytes  from the sender */

        if ((bytesRcvd = recv(sock, echoBuffer, RCVBUFSIZE - 1, 0)) <= 0)
                DieWithError("recv() failed or connection closed prematurely");
        totalBytesRcvd += bytesRcvd;   /* Keep tally of total bytes */
        echoBuffer[bytesRcvd] = '\0';  /* Terminate the string! */
        printf(echoBuffer);            /* Print the echo buffer */
}

printf("\n");    /* Print a final linefeed */
close(sock);
exit(0);}

 /* send() and recv() are 'blocking' - process suspended until call completes */
```

Socket() function

```
/* Create socket descriptor for incoming connections */

if ((ServerSocket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");
```

- observe 3 parameters to "socket" call : Internet protocol family, type of socket - sock_stream in this case, and the TCP protocol

```
int connect(int socket, struct sockaddr *foreignAddress, unsigned int addresslength)
```

- socket is the socket descriptor created by socket()
- foreignAddress is declared to be a pointer to a sockaddr
- In our case, we always use the INET family, so foreignAddress is a pointer to a sockaddr_in element (16-bytes) containing the internet address and port of the remote server to connect to
- addressLength specifies the length of the address structure and this parameter is always sizeof(struct sockaddr_in) = 16 bytes
- When this call returns >= 0, the connection is established, and data transfer can proceed with calls to **send()** and **recv()**
- Process suspended until connect completes

```
struct  sockaddr_in  echoServAddr;                    /*  declare echoServAddr */

echoServAddr.sin_family = AF_INET;                    /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP);     /* server IP address */
echoServAddr.sin_port = htons(echoServPort);          /* server port */

if (connect(sock, (struct sockaddr *) & echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError(”connect() failed");
```

1. 1st parameter sock is the socket descriptor
2. 2nd parameter is a pointer to echoServAddr to be used:

   Note: (struct sockaddrs *) declares the type to be a pointer to sockaddr type

1. The “&” operator returns the address of ‘echoServAddr’
2. The 3rd parameter returns the number of bytes in ‘echoServAddr’

C : send() and rec() functions

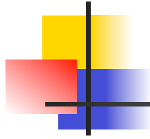
```
int send(int socket, const void *msg, unsigned int msgLength, int flags)
int  recv(int socket,  void *rcvBuffer, unsigned int bufferLength, int flags)
```

- socket is the socket descriptor (connected) created by socket()
- For send(), msg points to the message to send, and msgLength specifies the length in bytes of the message
- By default, send() will block until all of the data is sent (a blocking call to this routine)
- for recv(), rcvBuffer points to the buffer (a character array in memory), and bufferLength specifies the length in bytes of the buffer = maximum number of bytes which can be received at once
- By default, recv() will block until "some bytes are received"

Send to Server

```
echoStringLen = strlen(echoString);

If (send(sock, echoString, echoStringLen, 0) != echoStringLen)
        DieWithError("send() sent a different number of bytes than expected");
```

1. Send() returns the number of bytes sent



C : Receive echo

```
totalBytesRcvd = 0;
Printf("Received: ");
While (totalBytesRcvd < echoStringLen)
{
/* receive up to buffer size minus 1 data bytes from server, plus null terminator */
        if ((bytesRcvd = recv(sock, echoBuffer, RCVBUFSIZE -1, 0)) <= 0)
                DieWithError("recv() failed or connection closed prematurely ");
        totalBytesRcvd += bytesRcvd;
        echoBuffer[bytesRcvd] = '\0';        /* terminate string */
        printf(echoBuffer);                  /* print the echoBuffer */
}
```

1. Perform as many calls to **recv()** as necessary to receive the echo message
2. In TCP, a single **send()** can map to several **recv()**s; in TCP the mapping is not one-to-one

C : DieWithError() (pg 16)

```
#include <stdio.h>  /* for perror() */
#include <stdlib.h> /* for exit() */
void DieWithError(char *errorMessage)
{
    perror(errorMessage);
    exit(1);
}
```

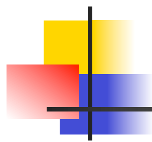
- Outputs a caller-specified error string, followed by an error description string from the system based on the value of errno—system call error.
- Uses the file designated by stderr

C : TCPEchoServer

- lets look at the Server C code

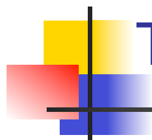
TCPEchoServer.c - header (pg 19)

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind(), and connect() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define MAXPENDING 5    /* Maximum outstanding connection requests */

void DieWithError(char *errorMessage);  /* Error handling function */
void HandleTCPClient(int clntSocket);   /* TCP client handling function */

int main(int argc, char *argv[])
{
    int servSock;                    /* Socket descriptor for server */
    int clntSock;                    /* Socket descriptor for client */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned short echoServPort;     /* Server port */
    unsigned int clntLen;            /* Length of client address data structure */

    if (argc != 2)     /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage:  %s <Server Port>\n", argv[0]);
        exit(1);
    }
```



TCPEchoServer.c - Bind & Listen

```
echoServPort = atoi(argv[1]);  /* First arg:  local port */

    /* Create socket for incoming connections */
    if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");

    /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));   /* Zero out structure */
    echoServAddr.sin_family = AF_INET;                /* Internet address family */
    echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
    echoServAddr.sin_port = htons(echoServPort);      /* Local port */

    /* Bind to the local PORT & IP address, ie initialize  part of the socket data struct */
    if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
        DieWithError("bind() failed");

    /* Mark the socket so it will listen for incoming connections */
    if (listen(servSock, MAXPENDING) < 0)
        DieWithError("listen() failed");
```

TCPEchoServer.c – Accept()

```
    for (;;) /* Run forever */
    {
        /* Set the size of the in-out parameter */
        clntLen = sizeof(echoClntAddr);

        /* Wait for a client to connect */
        if ((clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr, &clntLen)) < 0)
            DieWithError("accept() failed");

        /* clntSock is connected to a client! */

        printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));

        HandleTCPClient(clntSock);  /* CALL THIS FUNCTION TO DO SOMETHING */
    }
    /* NOT REACHED */
}


/* inet_ntoa() converts an internet address to ASCII test string, for IO */
```

int bind(int socket, struct sockaddr* localAddress, unsigned int addressLength)

Socket: descriptor returned by an earlier call to **socket()**

Sockaddr: for TCP/IP sockaddr will point to 'sockaddr_in' which contains IP address of the host interface and the port to listen on.

The IP address can be set to **INADDR_ANY** — connections to the specified port will be directed to this socket, regardless of which IP address they are sent to. Useful, if the host has multiple IP addresses.

addressLength: length of the address structure sizeof(struct sockaddr_in) = 16 bytes

Return 0 on success and –1 on failure.

After binding the socket to a port, with a specified IP address or a "wild-card" IP address, we can **listen()** on the socket for connection requests from a client.

Int listen(int socket, int queueLimit)

- QueueLimit—upper bound on the number of incoming connections waiting at any time. System dependent
- Return 0 on success and –1 on failure
- Used for getting *new* sockets, one for each client connection.
- The server gets a socket for an incoming client connection by calling **accept()**

accept() (pg 18)

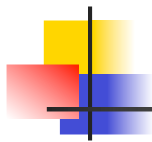
int accept(int socket, struct sockaddr* clientAddres, unsigned int* addressLength)

- De-queues the next connection request in the connection queue for the socket, blocks if connection queue is empty
- Fills in the socketaddr structure, associated with the socket, with the client's IP address.
- addressLength is the maximum size of the clientAddress structure-contains the number of bytes used by that structure.
- Returns a descriptor for a **new socket** that is **connected to the client**. This new socket is used for communication with this specific client using **send()** and **recv()**
- Return –1 on failure

HandleTCPClient.c (pg 21)

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for recv() and send() */
#include <unistd.h>     /* for close() */

#define RCVBUFSIZE 32   /* Size of receive buffer */

void DieWithError(char *errorMessage);  /* Error handling function */

void HandleTCPClient(int clntSocket)
{
   char echoBuffer[RCVBUFSIZE];        /* Buffer for echo string */
   int recvMsgSize;                    /* Size of received message */

   /* Receive message from client */
   if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
      DieWithError("recv() failed");
```

- This function Contains application-specific code, in this case, receive a message and echo it



HandleTCPClient.c (pg 21)

```
/* Send received string and receive again until end of transmission */
   while (recvMsgSize > 0)      /* zero indicates end of transmission */
   {
      /* Echo message back to client */
      if (send(clntSocket, echoBuffer, recvMsgSize, 0) != recvMsgSize)
         DieWithError("send() failed");

      /* See if there is more data to receive */
      if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
         DieWithError("recv() failed");
   }

   close(clntSocket);    /* Close client socket */
}

/* Q: recv() is blocking - process will suspend indefinitely until a message received */
/* Q: if sender calls close(), my guess is the receiver OS will close the socket, resulting */
/* in recv() returning -1:  Nonblocking sockets are discussed in Chapter 5.3, Donahoo  */
```

TCP Client/Server Interaction

TCP Tidbits

- Client knows server address and port
- No correlation between `send()` and `recv()`

Client	Server
send("Hello Bob")	
	recv() -> "Hello "
	recv() -> "Bob"
	send("Hi ")
	send("Jane")
recv() -> "Hi Jane"	

Closing a Connection

- close() used to delimit communication
- Analogous to EOF

Echo Client

send(*string*)

while (not received entire string)
    recv(*buffer*)
    print(*buffer*)

close(*socket*)

Echo Server

recv(*buffer*)
while(client has not closed connection)
    send(*buffer*)
    recv(*buffer*) /*blocking */

close(*client socket*)

Socket State Transitions

- next diagrams show state transitions: large arrows show the events which cause the state transitions
- time proceeds left-to-right, message arrivals shown in lower part of figures
- clients IP address in A.B.C.D, server's IP address in W.X.Y.Z, and server's port is Q

Client-side TCP Connection Establishment

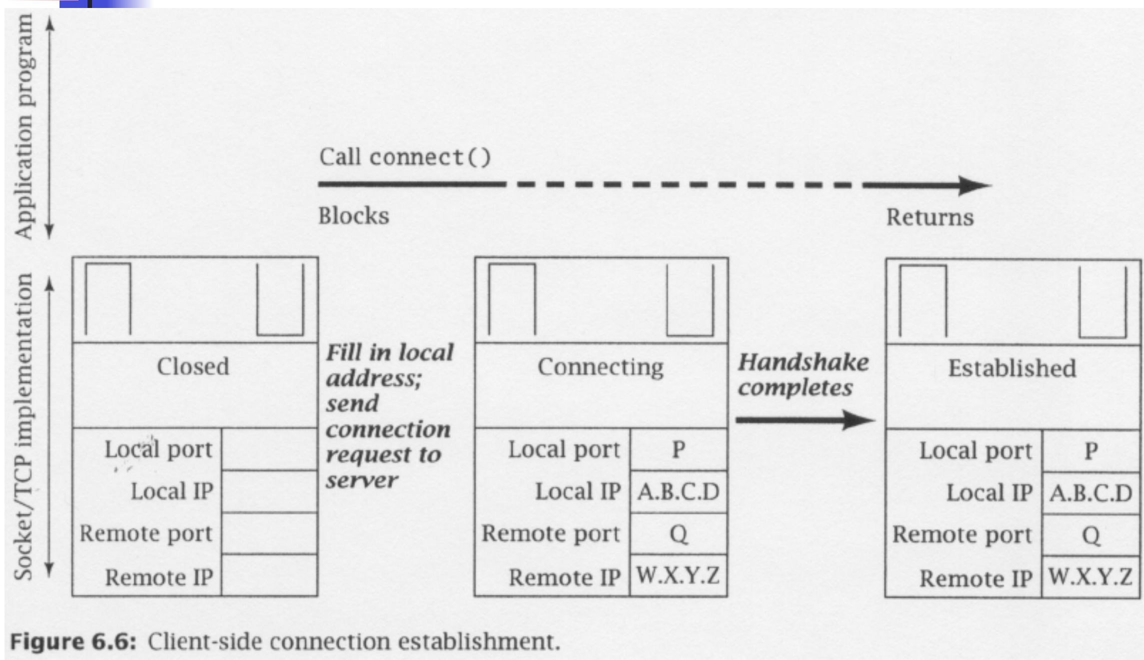


Figure 6.6: Client-side connection establishment.

Server-side Connection Establishment

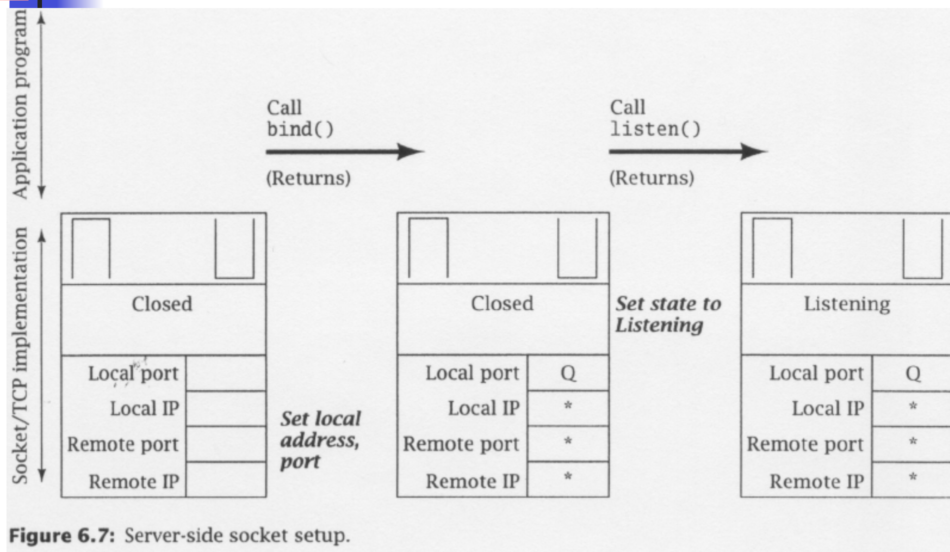


Figure 6.7: Server-side socket setup.

- Server-side socket setup: local IP address is the 'wild-card' **INADDR_ANY**, so the server can receive on any of its IP addresses (it has more than 1)

Incoming Connection

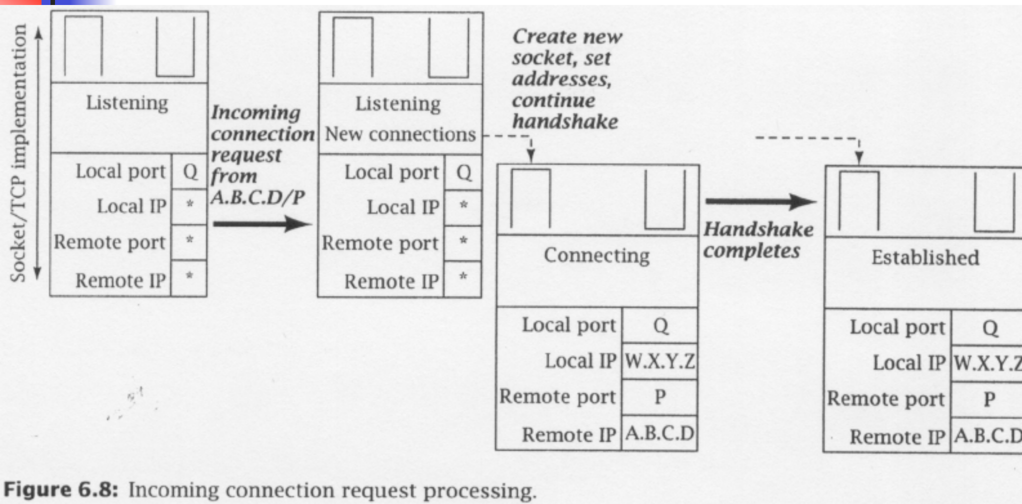


Figure 6.8: Incoming connection request processing.

- At the server host, when a connection request arrives from a client on a listening socket using INADDR_ANY, a new socket (lets call it a "**clone**") is created for the connection, and the client's IP address and port are entered. The local IP address which received the connection request is also entered (W.X.Y.Z) into the clone: NOTE: The established clone appears to receive an unused local port # exclusive to this client (not Q as shown in the textbook figure above; our in-class demo illustrated this phenomina)

Accept() Processing

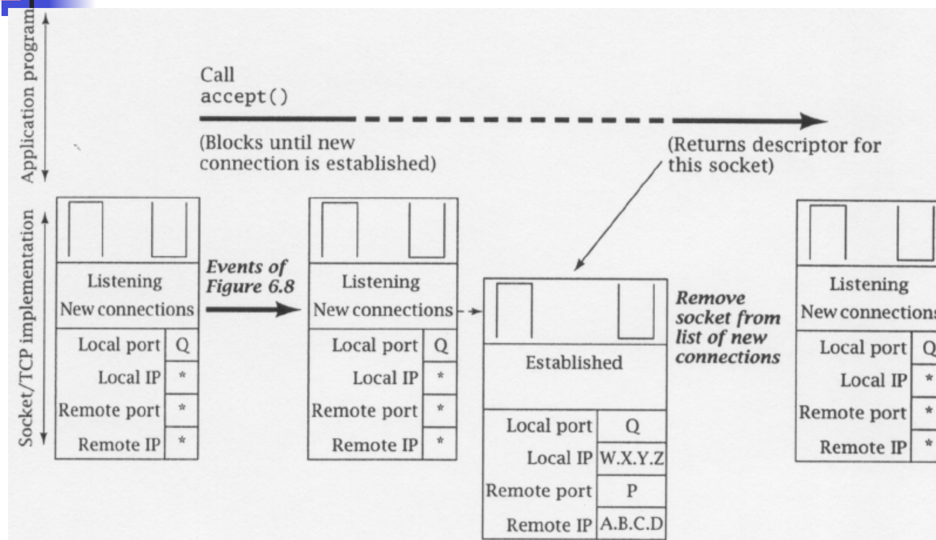


Figure 6.9: accept() processing.

- When the server calls blocking Accept(), it is blocked until the connection is established (Fig. 6.8): Accept() then returns a new socket descriptor, (the "clone" from the last slide)

Closing TCP Connection

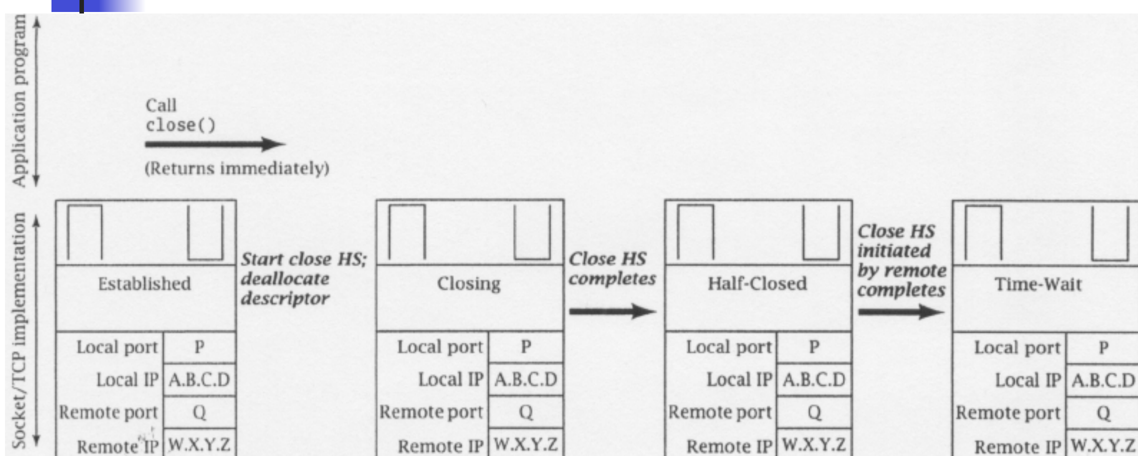


Figure 6.10: Closing a TCP connection first.

- Here is a sequence when one side calls close() first. One side labels its socket 'closing', and sends a CLOSE HAND SHAKE (HS) signal to the remote, and the call to close() returns.
- When the remote HS is received, the state changes to Time-Wait, and an Acknowledgement is sent back to the remote. The state will change to CLOSED when the remote application also handshakes (I believe).

Closing TCP Connection

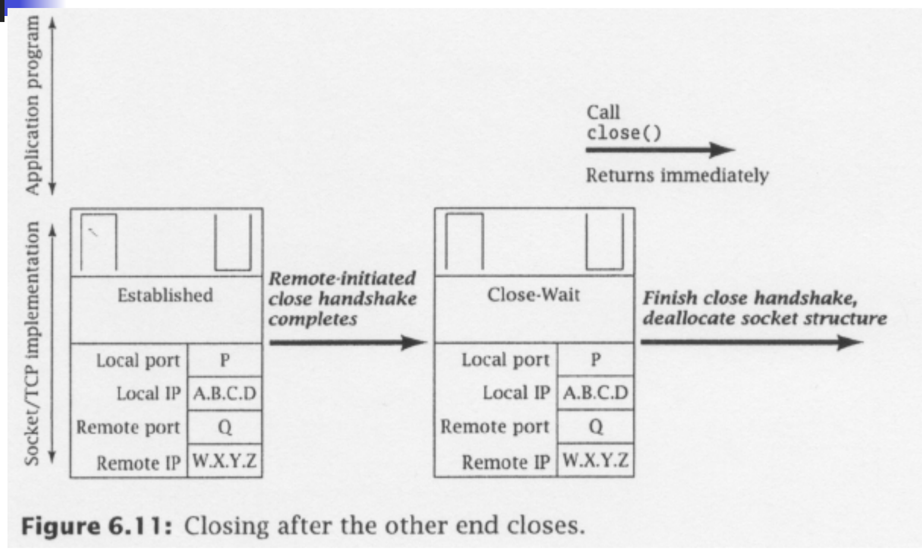


Figure 6.11: Closing after the other end closes.

- At the remote socket layer (which did not close first), the close HS is received, the state is changed to CLOSE-WAIT, and an acknowledgment is sent to the remote end. The socket now waits for the application program to call close(). When this is done, a final handshake ends the connection and deallocates the structures (I believe).

Demultiplexing connections at server

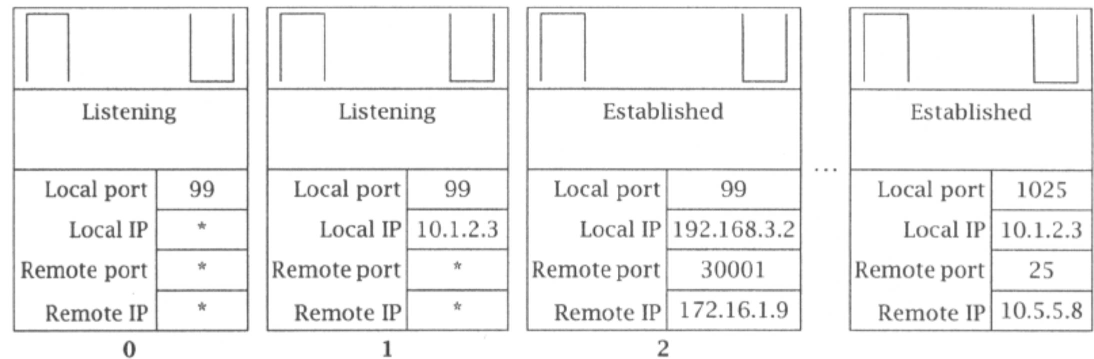
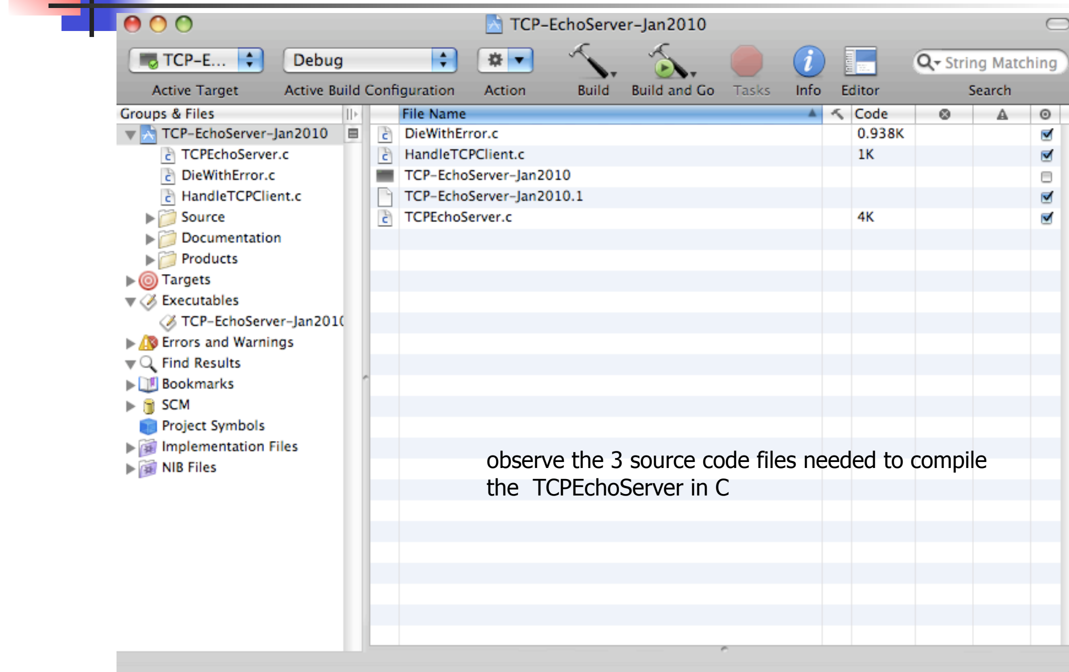


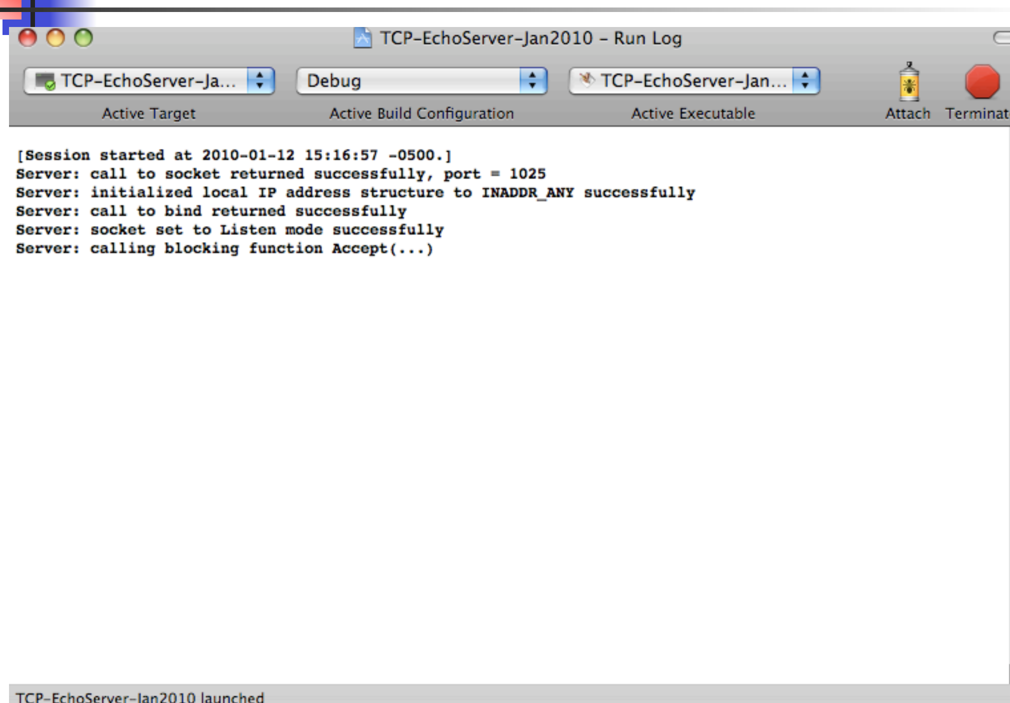
Figure 6.12: Demultiplexing with multiple matching sockets.

- In a server host with one IP address, every accepted connection on a listening socket using INADDR_ANY appears to result in a **cloned socket** with the same local IP address but a NEW (and apparently random) unused local port. In the example above, the last cloned socket for a connection is assigned port 1025. When packets are received at the server, the socket layer will automatically check the local port number, and deliver the packets to the right socket at the server.
- Above, the listening socket 0 is bound to port 99, with wildcard local & remote IP addresses. Listening socket 1 is listening on local IP address 10.1.2.3, and will accept only on port 99. Socket 2 belongs to a connection accepted and cloned from socket 0. (It should have a random local port #, not 99). Socket 3 (the last one) is cloned from listening socket 1, and it also gets the random local port 1025.

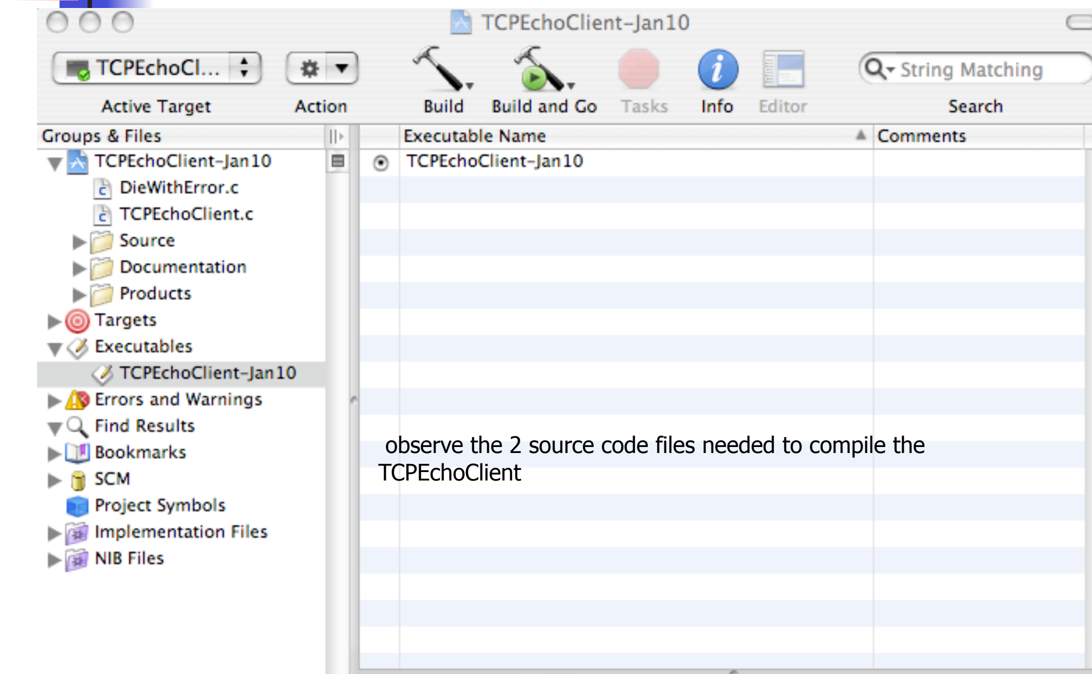
Example: TCPEchoServer Project (in Netbeans C on MAC OSX)



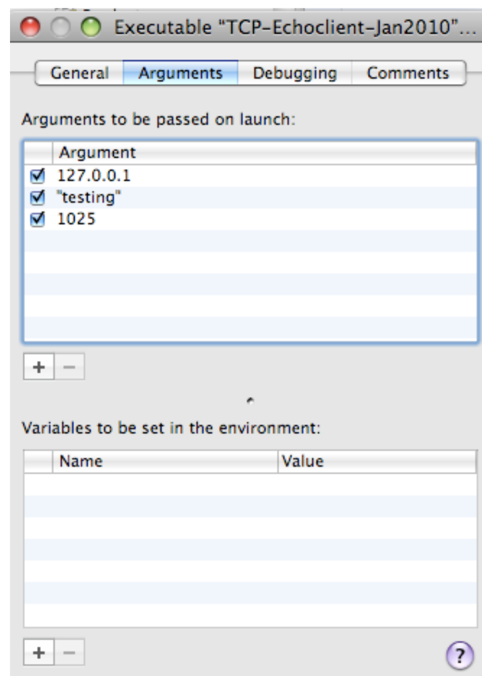
Example: TCPEchoServer Started



Example: Client Project (in Netbeans C on MAC OS X)



Example: Initializing Netbeans Command Line Arguments



Example: Client-Side Execution

TCP-Echoclient-Jan2010 – Run Log

TCP-Echoclient-Jan... | Debug | TCP-Echoclient-Jan2... | Attach | Run

Active Target | Active Build Configuration | Active Executable

```
[Session started at 2010-01-12 15:51:24 -0500.]

---------- TCPEchoClient: create client using port 1025 ------------
Client: call to socket returned successfully: using port 1025
Client: Calling blocking function connect() to IP address 127.0.0.1, port 1025
Client: Connected to server: sending string 'testing' to server
Client: Received : testing
Client: calling close(sock) and exitting

TCP-Echoclient-Jan2010 has exited with status 0.
```

TCP-Echoclient-Jan2010 exited normally.

Example: Server-Side Execution

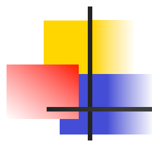
TCP-EchoServer-Jan2010 – Run Log

TCP-EchoServer-Ja... | Debug | TCP-EchoServer-Jan... | Attach | Terminate

Active Target | Active Build Configuration | Active Executable

```
[Session started at 2010-01-12 15:51:09 -0500.]

---------- TCPEchoServer: create server using port 1025 ------------
Server: call to socket returned successfully, port = 1025
Server: initialized local IP address structure to INADDR_ANY successfully
Server: call to bind returned successfully
Server: socket set to Listen mode successfully
Server: calling blocking function Accept(...)
Server: Handling client 127.0.0.1
Server: handleTCPClient: echo-ing string 'testing' and closing socket
Server: calling blocking function Accept(...)
```

TCP-EchoServer-Jan2010 launched



TCPEchoServer in Java

```
import java.net.*;  // for Socket, ServerSocket, and InetAddress
import java.io.*;   // for IOException and Input/OutputStream

public class TCPEchoServer {

  private static final int BUFSIZE = 32;   // Size of receive buffer

  public static void main(String[] args) throws IOException {

    if (args.length != 1)  // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Port>");
```

```
    int servPort = Integer.parseInt(args[0]);

    // Create a server socket to accept client connection requests
    ServerSocket servSock = new ServerSocket(servPort);

    int recvMsgSize;   // Size of received message
    byte[] byteBuffer = new byte[BUFSIZE];  // Receive buffer

    for (;;) { // Run forever, accepting and servicing connections
      Socket clntSock = servSock.accept();     // Get client connection


        System.out.println("Handling client at " +
          clntSock.getInetAddress().getHostAddress() + " on port " +
             clntSock.getPort());

        InputStream in = clntSock.getInputStream();
        OutputStream out = clntSock.getOutputStream();

        // Receive until client closes connection, indicated by -1 return
        while ((recvMsgSize = in.read(byteBuffer)) != -1)
          out.write(byteBuffer, 0, recvMsgSize);

        clntSock.close();  // Close the socket.  We are done with this client!
      }
      /* NOT REACHED */
    }
}
```

TCP EchoServer in Python

```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the port
server_address = ('localhost', 10000)
print >>sys.stderr, 'starting up on %s port %s' % server_address
sock.bind(server_address)

# Listen for incoming connections
sock.listen(1)

while True:
    # Wait for a connection
    print >>sys.stderr, 'waiting for a connection'
    connection, client_address = sock.accept()
```

TCP EchoServer in Python

```
while True:
    # Wait for a connection
    print >>sys.stderr, 'waiting for a connection'
    connection, client_address = sock.accept()

    try:
        print >>sys.stderr, 'connection from', client_address

        # Receive the data in small chunks and retransmit it
        while True:
            data = connection.recv(16)
            print >>sys.stderr, 'received "%s"' % data
            if data:
                print >>sys.stderr, 'sending data back to the client'
                connection.sendall(data)
            else:
                print >>sys.stderr, 'no more data from', client_address
                break

    finally:
        # Clean up the connection
        connection.close()
```

TCP EchoClient in Python

```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect the socket to the port where the server is listening
server_address = ('localhost', 10000)
print >>sys.stderr, 'connecting to %s port %s' % server_address
sock.connect(server_address)

try:

    # Send data
    message = 'This is the message.  It will be repeated.'
    print >>sys.stderr, 'sending "%s"' % message
    sock.sendall(message)
```

TCP EchoClient in Python

```
try:

    # Send data
    message = 'This is the message.  It will be repeated.'
    print >>sys.stderr, 'sending "%s"' % message
    sock.sendall(message)

    # Look for the response
    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print >>sys.stderr, 'received "%s"' % data

finally:
    print >>sys.stderr, 'closing socket'
    sock.close()
```

Additional References

[1] A. Leon-Garcia & I. Widjaja, Communication Networks, 2nd Ed., McGraw Hill

[2] Gary R. Wright and W. Richard Stevens, TCP/IP illustrated, Volume 2. Addison-Welsey, 1995

[3] Dix, Alan. Unix Network Programming with TCP/IP, Short Course Notes, 1996. Available from: http://www.hiraeth.com/alan/tutorials

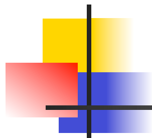
[4]Shah, Steve. Linux Adminstration: A Beginner's Guide. Second Edition, McGraw Hill, 2000

[5] Gary R. Wright and W. Richard Stevens, TCP/IP illustrated, Volume 1. Addison-Welsey, 1994

[6] Linux 2.4 Kernel Internals. Available from *http://www.moses.uklinux.net/patches/lki.sgml*

[7] Daneil P. Bovet and Marco Cestai. Understand the Linux Kernel, O'Reilly, 2001

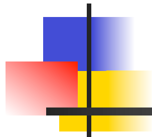
[8] David Rusling. The Linux Kernel. GNU General Public Licence, 1999.



COE 4DN4
Advanced Internet Communications

Messages - Chapter 3

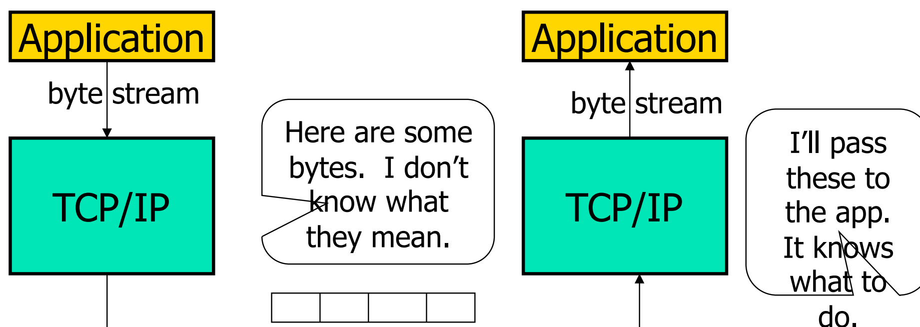
Prof. Ted Szymanski
Dept. EC E. McMaster University
www.ece.mcmaster.ca/faculty/teds/COURSES



Reference Textbook (1) :
TCP/IP Sockets IN C, M.J. Donahoo, K.L. Calvert,
Morgan Kaufmann, ISBN 1-55860-826-5

TCP/IP Byte Transport

- TCP/IP protocols transport byte streams



- Application protocol provides the semantics **

Application Protocol

- Encode information in bytes streams
- Lets call a sequence of bytes which are interpretted together a "**message**"
- Sender and receiver must agree ***the semantics*** (interpretation of the messages they send back and forth)
- Data encoding
  - **Primitive** types: character strings, integers,
  - **Composed** types: message data structure with fields

Primitive Types

- A string of ASCII characters or integers is a **primitive type**
- The string must be terminated
- A 'delimiter' character can be used, ie NULL character, white space character, or new line character
- Alternatively, the number of bytes in the string can be appended at the head of the string
- ASCII numbers from '0' to '9' are unsigned integers from 48-57
- Here is a string of ASCII integers, followed by a delimiter

49	55	57	57	56	55	48	10
'1'	'7'	'9'	'9'	'8'	'7'	'0'	\n

- Advantage: 1. Human readable, 2. Arbitrary size
- Disadvantage: 1. Inefficient,
- 2. Arithmetic manipulation is difficult

2 Universal Data Storage Formats

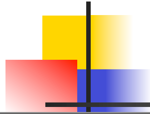
- Half of the world's computers use a storage format called "**Little-Endian**"
- The other half use a format called "**Big-Endian**"
- When the Internet allows these computers to exchange files, these differences must be resolved
- Every string that is stored could be interpretted 2 different ways - what a mess !

- Approach in the Internet: The "**Universal**" Network storage format is **Big-Endian**;
- When strings are exchanged over the Internet between computers, they should follow Big-Endian format
- A machine using Little-Endian may store the message in Little-Endian format locally, so the message will have to be converted to Big-Endian when it is transmitted or used on the web
- For example, the character string '**servIP**' which contains an ASCII representation of the Ipv4 address (using dotted quad notation) must be interpretted in the right order

Big-endian VS Little-endian

- In Big-endian, when storing 4 bytes in a 32-bit world, the most significant byte in the sequence is stored in the smallest byte address of the word (ie Big-End First)
- In Little-endian, when storing 4 bytes in a 32-bit world, the least significant value in the sequence is stored in the smallest byte address of the word (ie Little end first)
- For a character string, let the most significant value be the first value
- A Little-endian computer will store the string '**whats up doc'** as follows:
- (largest address) **whats up doc** (smallest address)
- A Big-endian computer will store the string 'whats up doc' as follows:
- (largest address) **cod pu stahw** (smallest address)

- Big-endian machines : IBM's 370, MIPS RISC chip, Motorola PowerPC
- Little-endian machines : DEC Alpha, Intel Pentium



Big-endian VS Little-endian
(wikipedia)

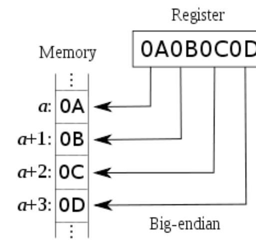
Big-endian
[edit]

With 8-bit atomic element size and 1-byte (octet) address increment [edit]

increasing addresses →

...	$0A_h$	$0B_h$	$0C_h$	$0D_h$	...

The most significant byte (*MSB*) value, which is $0A_h$ in our example, is stored at the memory location with the lowest address, the next byte value in significance, $0B_h$, is stored at the following memory location and so on. This is akin to Left-to-Right reading in hexadecimal order.



With 16-bit atomic element size [edit]

increasing addresses →

...	$0A0B_h$	$0C0D_h$	...

The most significant atomic element stores now the value $0A0B_h$, followed by $0C0D_h$.

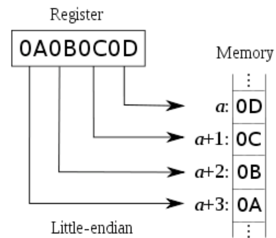
Little-endian
[edit]

With 8-bit atomic element size and 1-byte (octet) address increment [edit]

increasing addresses →

...	$0D_h$	$0C_h$	$0B_h$	$0A_h$	...

The least significant byte (*LSB*) value, $0D_h$, is at the lowest address. The other bytes follow in increasing order of significance.



With 16-bit atomic element size [edit]

increasing addresses →

...	$0C0D_h$	$0A0B_h$	...

Htonl() and Ntoh() (pg 29)

- To resolve the interpretation of integers, these 2 functions are used

- Hton_() = Host-to-Network number conversion (to Big-endian)
- Ntoh_() = Network-to-Host() number conversion (Big-endian to local machine format)

- The hton_() function is frequently used to convert Internet addresses from a host-specific byte order, to a universal network byte order (big-endian), for socket calls requiring an Internet address as a parameter.

- There are 2 versions of each function, for short integers (2 bytes) and long integers (4 bytes)
- Htonl() is for 32 bits, Htons() for 16 bits
- Ntohl() is for 32 bits, Ntohs() for 16 bits

Messages with Multiple Fields pg 30

- A Message can be composed of multiple fields, either fixed or variable length
- If fields are variable length, delimiters must be used
- Most compilers align data structures to start on word boundaries, ie they 'pad' fields with zeros, so that fields do not cross unnecessary word boundaries
- A 32-bit machine uses 4 byte words, a 64-bit machine uses 8 byte words
- Data Structures will use a different # of bytes on different machines, depending upon compiler padding !! Another potential mess on the Internet
- To avoid problems, messages must be defined so that all fields are aligned consistently between different machines
- Typically, we can reorder fields and then manually 'pad' them so they align onto word boundaries, when we define them on each machine
- The protocol designers need to keep track of how their data structures are created by the compiler, on each type of machine

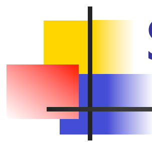
C : Allignment by compiler padding

```
struct tst {
  short x;        /* 2 bytes, padded by compiler to use a full 4-byte word */
  int y;          /* 4 bytes, aligned by compiler */
  short z;        /* 2 bytes, padded by compiler to use a full 4-byte word */
};
```

- If the structure was not aligned by the compiler, integer y will cross a word boundary, and every 'read" of integer y will cause 2 memory accesses, very slow operation !

- We can solve this problem by reorganizing the data structure to avoid padding:

```
struct tst {
  int y;          /* 4 bytes, uses a full 4-byte  word */
  short x;        /* next 2 'short' integers share a 4-byte word */
  short z;
};   /* this data structure avoids the padding problem */
```



Sending C Data Structures over IP

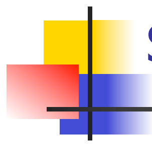
```
Struct {
int               dollars_depositted;
unsigned short    number_deposits;
int               dollars_depositted;
unsigned short    number_withdrawals;
}  Message_Buffer


Send(socket, &Message_Buffer, sizeof(MessageBuffer))
```

The above C code will probably result in a message of 14 bytes, maybe even 16 bytes, instead of 12 bytes, due to compiler padding !

C : Receiving Fixed Size Messages (pg 32)

Suppose your client and server exchange messages of known sizes. We can define a new receive function, ReceiveMessage(), where an entire message is received at once. We need to know how long the message is: Suppose the first 2 bytes specify the number of bytes in the message;

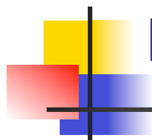
```
int ReceiveMessage(int socket, char *buf, int maxlength)
{ int received = 0;          /* number bytes received */
  int delimCount = 0;       /* number delimiters received */
  int rv;
  While ((received < maxLength) && (delimCount <= 2))   {
     rv = recv(socket, buf+received, 1, 0);              /* receive one byte at a time */
     if (rv < 0)
              DieWithError("recv() failed in ReceiveMessage\n");
    elseif (rv == 0)
              DieWithError("recv(): unexpected end of transmission in ReceiveMessage\n");
    if ( *(buf+received) == DELIMCHAR)
              delimCount += 1;
              received += 1;
   }          /* end while */
   return received;
}
```



Sending/Receiving a File (class exercise)

Suppose you want to open a file (on the disk), and send the contents over a socket. The file can be very large, so you should probably read in 'chunks' of the file into main memory, and send each chunk as a socket message, which can written to a file at the destination. Write a flow-chart for the SendFile() and ReceiveFile() functions.

(We will end up doing this later in the 4DN4 class and in the 4DN4 labs, so this slide is meant to get you thinking about how to do this.)

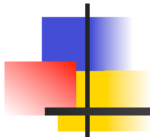


Notes

COE 4DN4
Advanced Internet Communications

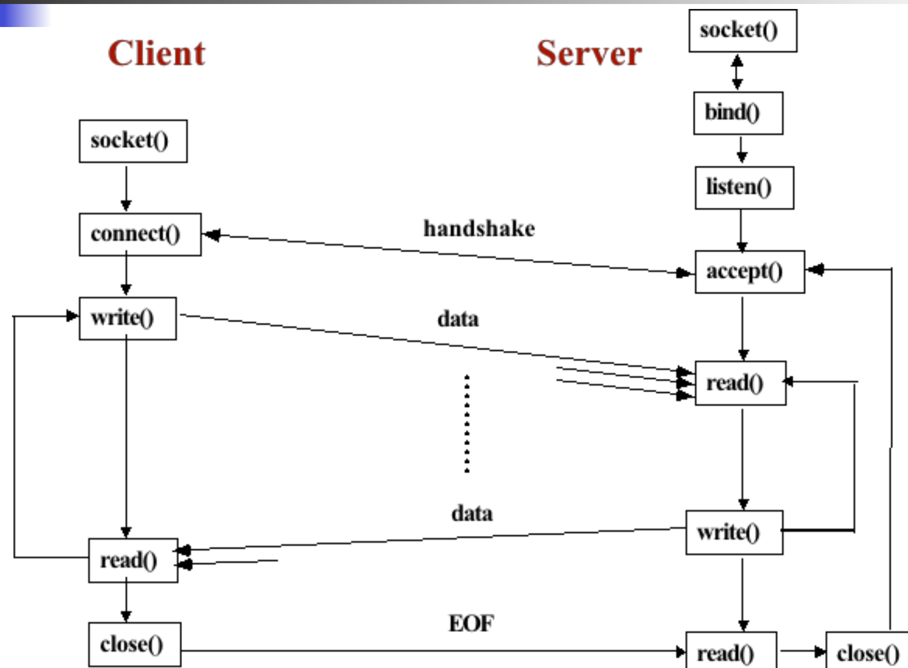
UDP Sockets - Chapter 4

Prof. Ted Szymanski
Dept. ECE. McMaster University
www.ece.mcmaster.ca/faculty/teds/COURSES



Reference Textbook (1) :
TCP/IP Sockets IN C, M.J. Donahoo, K.L. Calvert,
Morgan Kaufmann, ISBN 1-55860-826-5

TCP Client/Server Interaction



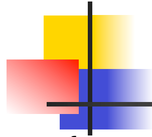
C : UDP Client-Server Example (pg 36)

- Consider a simple UDP server which simply echos whatever it gets
- the client code = UDPEchoClient.c, available at the Donahoo web site
- the server code = UDPEchoServer.c, available at the Donahoo web site

- Echo Client-Server useful for debugging code, so most systems provide TCP and/or UDP servers, using port 7

UDPEchoClient.c – C Headers

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), sendto(), and recvfrom() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define ECHOMAX 255     /* Longest string to echo */

void DieWithError(char *errorMessage);  /* External error handling function */

int main(int argc, char *argv[])
{
    int sock;                        /* Socket descriptor */
    struct sockaddr_in echoServAddr; /* Echo server address */
    struct sockaddr_in fromAddr;     /* Source address of echo */
    unsigned short echoServPort;     /* Echo server port */
    unsigned int fromSize;           /* In-out of address size for recvfrom() */
    char *servIP;                    /* IP address of server */
    char *echoString;                /* String to send to echo server */
    char echoBuffer[ECHOMAX+1];      /* Buffer for receiving echoed string */
    int echoStringLen;               /* Length of string to echo */
    int respStringLen;               /* Length of received response */

    if ((argc < 3) || (argc > 4))    /* Test for correct number of arguments */
    {
```



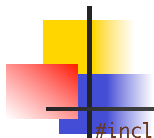
UDPEchoClient.c – C Arguments

```
{
        fprintf(stderr,"Usage: %s <Server IP> <Echo Word> [<Echo Port>]\n", argv[0]);
        exit(1);
    }

    servIP = argv[1];           /* First arg: server IP address (dotted quad) */
    echoString = argv[2];       /* Second arg: string to echo */

    if ((echoStringLen = strlen(echoString)) > ECHOMAX)  /* Check input length */
        DieWithError("Echo word too long");

    if (argc == 4)
        echoServPort = atoi(argv[3]);  /* Use given port, if any */
    else
        echoServPort = 7;         /* 7 is the well-known port for the echo service */

    /* Create a datagram/UDP socket */
    if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        DieWithError("socket() failed");

    /* Construct the server address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));    /* Zero out structure */
    echoServAddr.sin_family = AF_INET;                 /* Internet addr family */
    echoServAddr.sin_addr.s_addr = inet_addr(servIP);  /* Server IP address */
    echoServAddr.sin_port   = htons(echoServPort);     /* Server port */
```

UDPEchoClient.c - Send and Receive

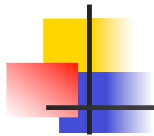
```
    /* Send the string to the server: UDP send must specify destination address */
   /* for every message sent; no notion of an established TCP connection to a peer */
    if (sendto(sock, echoString, echoStringLen, 0, (struct sockaddr *)
               &echoServAddr, sizeof(echoServAddr)) != echoStringLen)
        DieWithError("sendto() sent a different number of bytes than expected");

    /* Recv a response ; UDP receive will return IP address of sender */
    fromSize = sizeof(fromAddr);
    if ((respStringLen = recvfrom(sock, echoBuffer, ECHOMAX, 0,
         (struct sockaddr *) &fromAddr, &fromSize)) != echoStringLen)
        DieWithError("recvfrom() failed");

    if (echoServAddr.sin_addr.s_addr != fromAddr.sin_addr.s_addr)
    {
        fprintf(stderr,"Error: received a packet from unknown source.\n");
        exit(1);
    }

    /* null-terminate the received data */
    echoBuffer[respStringLen] = '\0';
    printf("Received: %s\n", echoBuffer);    /* Print the echoed arg */

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

2 new functions are underlined: sendto(), recvfrom()



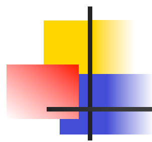
UDPEchoServer.c

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket() and bind() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define ECHOMAX 255     /* Longest string to echo */

void DieWithError(char *errorMessage);  /* External error handling function */

int main(int argc, char *argv[])
{
    int sock;                        /* Socket */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned int cliAddrLen;         /* Length of incoming message */
    char echoBuffer[ECHOMAX];        /* Buffer for echo string */
    unsigned short echoServPort;     /* Server port */
    int recvMsgSize;                 /* Size of received message */

    if (argc != 2)         /* Test for correct number of parameters */
    {
        fprintf(stderr,"Usage:  %s <UDP SERVER PORT>\n", argv[0]);
        exit(1);
    }
```

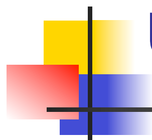
UDPEchoServer.c

```
    echoServPort = atoi(argv[1]);  /* First arg:  local port */

    /* Create socket for sending/receiving datagrams */
    if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        DieWithError("socket() failed");

    /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));   /* Zero out structure */
    echoServAddr.sin_family = AF_INET;                /* Internet address family */
    echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
    echoServAddr.sin_port = htons(echoServPort);      /* Local port */

    /* Bind to the local address */
    if (bind(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
        DieWithError("bind() failed");

    for (;;) /* Run forever */
    {
        /* Set the size of the in-out parameter */
        cliAddrLen = sizeof(echoClntAddr);

        /* Block until we receive a message from a client */
        if ((recvMsgSize = recvfrom(sock, echoBuffer, ECHOMAX, 0,
            (struct sockaddr *) &echoClntAddr, &cliAddrLen)) < 0)
            DieWithError("recvfrom() failed");
```



UDPEchoServer.c

```
for (;;) /* Run forever */
    {
        /* Set the size of the in-out parameter */
        cliAddrLen = sizeof(echoClntAddr);

        /* Block until receive message from a client */
        if ((recvMsgSize = recvfrom(sock, echoBuffer, ECHOMAX, 0,
            (struct sockaddr *) &echoClntAddr, &cliAddrLen)) < 0)
            DieWithError("recvfrom() failed");

        printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));

        /* Send received datagram back to the client */
        if (sendto(sock, echoBuffer, recvMsgSize, 0,
             (struct sockaddr *) &echoClntAddr, sizeof(echoClntAddr)) != recvMsgSize)
            DieWithError("sendto() sent a different number of bytes than expected");
    }
    /* NOT REACHED */
}
```

UDP Client-Server Example (pg 36)

- In TCP, a client and server established a connection; they could then communicate with messages, without having to re-identify the remote IP address or remote port repeatedly
- In UDP, there is no notion of an established connection; Every message is sent once as a datagram; Therefore, we need to specify a remote IP address and port, for every message sent in UDP.
- Similarly, when we receive a message in UDP, we need to know the remote IP address and port that we are receiving from, since the message could come from anyone.
- Question: which local port does the server receive on ? In the previous TCP socket examples, cloned sockets at the server appear to receive random unused local ports. What about the UDP server ? Can it receive from all 64K ports, or does it receive from only one port, and if so, which port ?

UDP Client-Server Example (pg 36)

- int sendto(int socket, const void *msg, unsigned int msgLength, int flags, struct sockaddr *destAddr, unsigned int addrLen)
- Note that we pass the remote IP address and port to sendto(), in the sockaddr structure; we also pass the length of the structure

- int recvfrom(int socket, void *msg, unsigned int msgLength, int flags, struct sockaddr *srcAddr, unsigned int addrLen)
- Note that we now receive the remote socket address in the sockaddr structure, along with each message received

- Recall that UDP does not fragment a message ; it is sent all at once; The receiver receives it all at once, if it is received.

COE 4DN4
Advanced Internet Communications

Socket Programming - Chapter 5

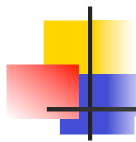
Prof. Ted Szymanski
Dept. EC E. McMaster University
www.ece.mcmaster.ca/faculty/teds/COURSES

Reference Textbook (1) : TCP/IP Sockets IN C, M.J. Donahoo, K.L. Calvert, Morgan Kaufmann, ISBN 1-55860-826-5

- TCP/IP C sockets have most parameters set at reasonable default values
- See Internet Engineering Task Force IETF (www.ietf.org) Request For Comments RFC 1122 and RFC 1123 for extremely detailed discussions on default values
- You can read socket option values and reset them to other values, as required
- In C, Use functions getsocketopt() and setsocketopt()

- Here is an example, to fetch and then increase the # of bytes in a socket's receive buffer

```
int rcvBufferSize;
int sockOptSize;
…

sockOptSize = sizeof(rcvBufferSize);
If (getsockopt(sock, SOL_SOCKET, SO_RCVBUF, &rcvBufferSize, &sockOptSize) < 0)
      DieWithError("getsockopt() failed");
Printf("Initial Receive Buffer Size = %d \n", rcvBufferSize);

rcvBufferSize *= 2;              /* double the size */
If (setsockopt(sock, SOL_SOCKET, SO_RCVBUF, &rcvBufferSize, sizeof(rcvBufferSize)) < 0)
      DieWithError("setsockopt() failed");
```



Socket Options

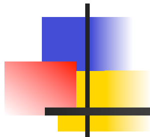
optName	Type	Values	Description
SOL_SOCKET Level			
SO_BROADCAST	**int**	0,1	Broadcast allowed
SO_KEEPALIVE	**int**	0,1	Keepalive messages enabled (if implemented by the protocol)
SO_LINGER	**linger**{}	time	Time to delay `close()` return waiting for confirmation (see Section 6.4.2)
SO_RCVBUF	**int**	bytes	Bytes in the socket receive buffer (see code on page 44 and Section 6.1)
SO_RCVLOWAT	**int**	bytes	Minimum number of available bytes that will cause `recv()` to return
SO_REUSEADDR	**int**	0,1	Binding allowed (under certain conditions) to an address or port already in use (see Section 6.4 and 6.5)
SO_SNDLOWAT	**int**	bytes	Minimum bytes to send a packet
SO_SNDBUF	**int**	bytes	Bytes in the socket send buffer (see Section 6.1)
IPPROTO_TCP Level			
TCP_MAX	**int**	seconds	Seconds between keepalive messages.
TCP_NODELAY	**int**	0,1	Disallow delay for data merging (Nagle's algorithm)

Socket Options

IPPROTO_IP Level			
IP_TTL	**int**	0–255	Time-to-live for unicast IP packets
IP_MULTICAST_TTL	**unsigned char**	0–255	Time-to-live for multicast IP packets (see `MulticastSender.c` on page 81)
IP_MULTICAST_LOOP	**int**	0,1	Enables multicast socket to receive packets it sent
IP_ADD_MEMBERSHIP	**ip_mreq**{}	group address	Enables reception of packets addressed to the specified multicast group (see `MulticastReceiver.c` on page 83)—set only
IP_DROP_MEMBERSHIP	**ip_mreq**{}	group address	Disables reception of packets addressed to the specified multicast group—set only

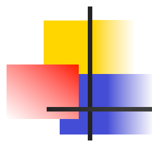
Table 5.1: Socket Options

5.4 MultiTasking (pg 60)



Multi-tasking (pg 60)

- TCP echo server handles 1 client at a time, using **blocking** calls; additional clients served **sequentially**
- Sequential service ok for numerous small jobs; inappropriate for time-consuming clients
- UNIX OS provides a solution: use "**processes**" or "**threads**" to create independently executing copies of the server, each copy serving one client, in a **blocking** mode
- Called "**concurrent servers**" ; effectively results in parallel execution



Multi-tasking in C (pg 60)

- "process" = independently executing program on the same host
- "server with process-per-client" - each client connection request creates a new process at the server
- UNIX **"fork()"** creates a process, returning -1 on failure; if successful, a copy of calling process is made, with new process ID number
- Execution begins after the **fork()** call
- **fork()** returns 0 for the child, returns process ID of child to the parent
- When a child process terminates, it does not automatically dissappear - its becomes a "**zombie**" in most UNIX systems; these zombies consume system resources until they are harvested by parent with call to '**waitpid()**'
- Example: TCPEchoServer-Fork.c (client code is unchanged)

Forking TCP Echo Server

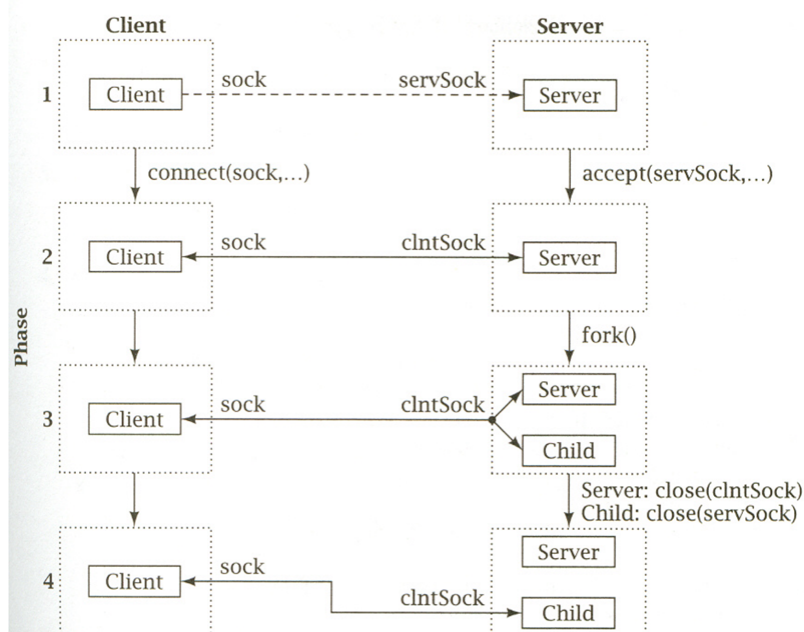
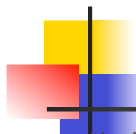
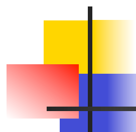


Figure 5.1: Forking TCP echo server.



TCPEchoServer-Fork.c (pg 61)

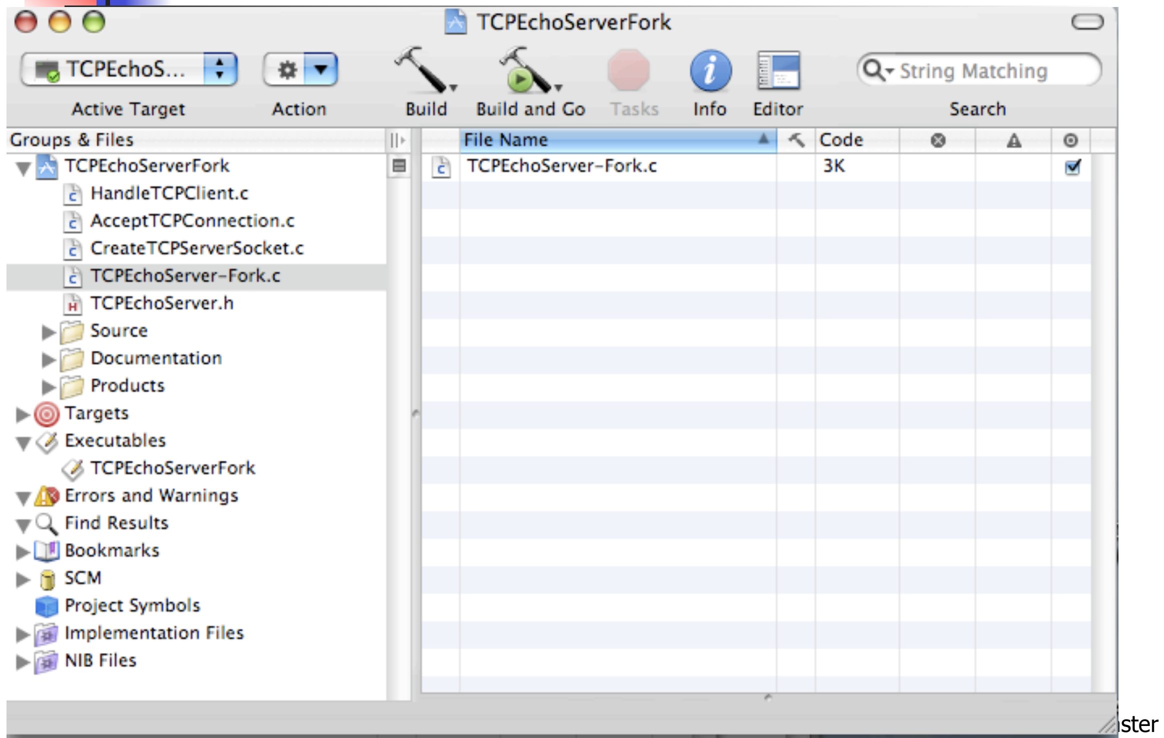
```
#include "TCPEchoServer.h"  /* TCP echo server includes */
#include <sys/wait.h>       /* for waitpid() */

int main(int argc, char *argv[])
{
    int servSock;                    /* Socket descriptor for server */
    int clntSock;                    /* Socket descriptor for client */
    unsigned short echoServPort;     /* Server port */
    pid_t processID;                 /* Process ID from fork() */
    unsigned int childProcCount = 0; /* Number of child processes */

    if (argc != 2)     /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage:  %s <Server Port>\n", argv[0]);
        exit(1);
    }

    echoServPort = atoi(argv[1]);  /* First arg:  local port */

    servSock = CreateTCPServerSocket(echoServPort);

    for (;;) /* Run forever */
    {
        clntSock = AcceptTCPConnection(servSock);
        /* Fork child process and report any errors */
        if ((processID = fork()) < 0)
```

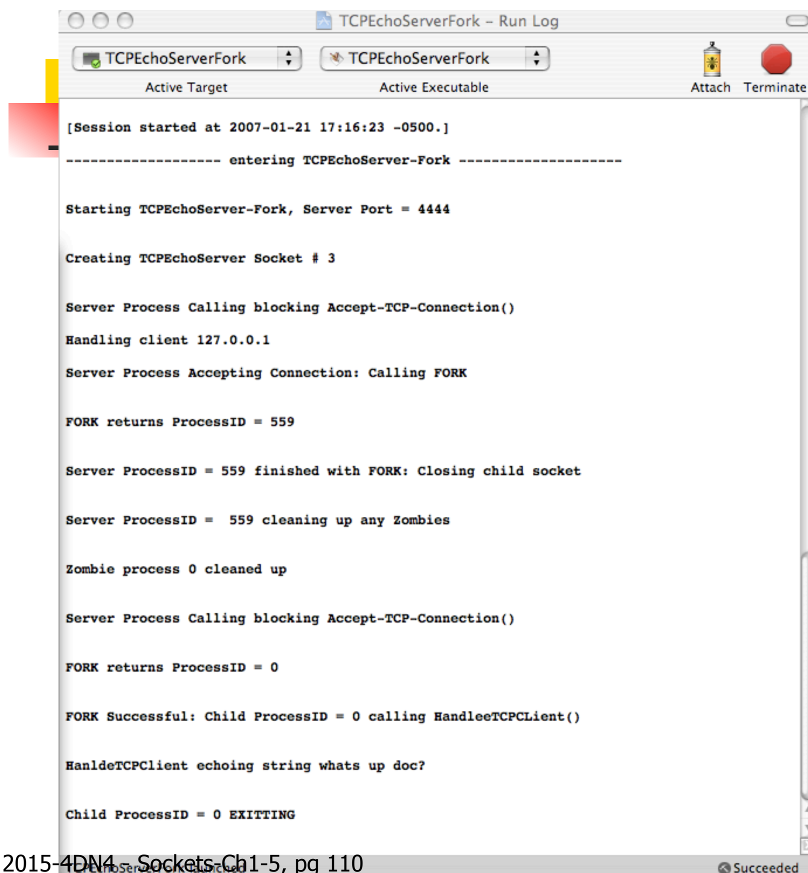



TCPEchoServer-Fork.c (pg 61)

```
    for (;;) /* Run forever */
    {
        clntSock = AcceptTCPConnection(servSock);
        /* Fork child process and report any errors */
        if ((processID = fork()) < 0)
            DieWithError("fork() failed");
        else if (processID == 0)  /* If this is the child process */
        {
            close(servSock);       /* Child closes parent socket */
              HandleTCPClient(clntSock);   /* handle client         */
            exit(0);                /* Child process terminates   */
        }

        printf("Parent processID = : %d\n", (int) processID);
        close(clntSock);       /* Parent closes child socket descriptor */
        childProcCount++;      /* Increment number of outstanding child processes */

        while (childProcCount > 0) /* Clean up any zombies */
        {
            processID = waitpid((pid_t) -1, NULL, WNOHANG); /* Non-blocking wait + clean up */
              if  (processID  <  0)       /*  waitpid()  error?  */
                DieWithError("waitpid() failed");
             else if (processID == 0)      /* No zombie to wait on */
                break;
            else
                childProcCount--;       /* call to waitpid() cleaned up after a child */
        }
    }
    /* NOT REACHED */
}
```

C Source files for EchoServer





Sample

Server

Activity

TCPEchoClient-Jan10 – Run Log

TCPEchoClient-Jan10 (Active Target) | TCPEchoClient-Jan10 (Active Executable) | Attach | Run

```
TCPEchoClient has sent string to EchoServer

TCPEchoClient has received a string from EchoServer:

whats up doc?

TCPEchoClient closing connection to EchoServer

TCPEchoClient-Jan10 has exited with status 0.
[Session started at 2007-01-21 17:16:30 -0500.]

-------------- entering TCPEchoClient ---------------------


TCPEchoClient using remote IP address 127.0.0.1
Using echoServerPort # = 4444

Socket created

Calling Connect()

TCPEchoClient has connected to EchoServer

TCPEchoClient has sent string to EchoServer

TCPEchoClient has received a string from EchoServer:

whats up doc?

TCPEchoClient closing connection to EchoServer

TCPEchoClient-Jan10 has exited with status 0.
```

TCPEchoClient-Jan10 exited normally. Succeeded

Sample Client Activity

System Resources per Forked Process

Activity Monitor

Quit Process | Inspect | Filter | Show: My Processes

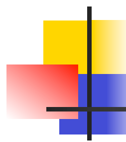
Process ID	Process Name	User	% CPU	# Threads	Real Memory	Virtual Memory
694	TCPEchoServerFor	tedszymanski	0.00	1	356.00 KB	27.40 MB
630	Activity Monitor	tedszymanski	1.80	2	11.38 MB	346.68 MB
431	Xcode	tedszymanski	0.00	7	30.65 MB	378.34 MB
429	Netscape	tedszymanski	9.40	12	120.24 MB	1.24 GB
427	Firefox	tedszymanski	1.00	12	110.16 MB	1.08 GB
418	Preview	tedszymanski	0.00	2	16.44 MB	361.36 MB
413	Adobe Reader	tedszymanski	0.80	10	42.51 MB	908.69 MB
409	PowerPoint	tedszymanski	0.40	11	157.86 MB	1.27 GB
398	Database Daemon	tedszymanski	0.00	3	6.58 MB	455.00 MB
397	Word	tedszymanski	3.40	8	45.08 MB	791.31 MB
333	Eudora	tedszymanski	0.10	7	33.62 MB	614.38 MB
323	sh	tedszymanski	0.20	1	460.00 KB	27.07 MB
322	lmgrd	tedszymanski	0.10	1	716.00 KB	27.74 MB
321	tcsh	tedszymanski	0.00	1	176.00 KB	31.06 MB
229	UniversalAccessApp	tedszymanski	0.00	1	1.39 MB	334.95 MB
218	XPSLauncher	tedszymanski	0.00	3	5.52 MB	408.69 MB
217	HPEventHandler	tedszymanski	0.00	2	836.00 KB	326.70 MB
215	HP Communications	tedszymanski	2.80	4	7.97 MB	447.92 MB
212	HP IO Classic Proxy	tedszymanski	0.20	2	5.18 MB	403.83 MB
211	CanonPS Helper	tedszymanski	0.00	2	4.05 MB	411.96 MB
210	iCalAlarmScheduler	tedszymanski	0.00	1	1.57 MB	328.86 MB
107	Finder	tedszymanski	0.10	3	16.70 MB	366.10 MB
106	SystemUIServer	tedszymanski	1.70	2	5.27 MB	353.41 MB
84	Dock	tedszymanski	0.00	2	7.66 MB	315.31 MB
77	pbs	tedszymanski	0.00	2	780.00 KB	56.13 MB
68	loginwindow	tedszymanski	0.00	3	2.04 MB	332.42 MB
67	[illegible]	tedszymanski	0.00	2	5.76 MB	66.00 MB

Per-Client Thread (pg 67)

- Forking processes is expensive; each child duplicates entire state of parent process, including code, memory, stack, file/socket descriptors, etc
- "**threads**" decrease cost by allowing multi-tasking within the same process; newly created threads **share** same address space (for code & data) with parent, negating need to duplicate state
- Example: TCPEchoServer-Thread.c, using POSIX threads

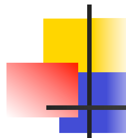
TCP-EchoServer-Thread.c (pg 67)

```
#include "TCPEchoServer.h"  /* TCP echo server includes */
#include <pthread.h>        /* for POSIX threads */

void *ThreadMain(void *arg);            /* Main program of a thread */

/* Structure of arguments to pass to client thread */
struct ThreadArgs
{
    int clntSock;                      /* Socket descriptor for client */
};

int main(int argc, char *argv[])
{
    int servSock;                     /* Socket descriptor for server */
    int clntSock;                     /* Socket descriptor for client */
    unsigned short echoServPort;      /* Server port */
    pthread_t threadID;               /* Thread ID from pthread_create() */
    struct ThreadArgs *threadArgs;    /* Pointer to argument structure for thread */

    if (argc != 2)     /* Test for correct number of arguments */
    {
        fprintf(stderr,"Usage:  %s <SERVER PORT>\n", argv[0]);
        exit(1);
    }

    echoServPort = atoi(argv[1]);  /* First arg:  local port */
```



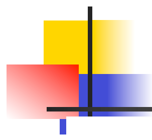
TCP-EchoServer-Thread.c (pg 67)

```
    servSock = CreateTCPServerSocket(echoServPort);

    for (;;) /* main program runs forever */
    {
         clntSock = AcceptTCPConnection(servSock);

        /* Create separate memory for client argument */
        if ((threadArgs = (struct ThreadArgs *) malloc(sizeof(struct ThreadArgs)))
               == NULL)
            DieWithError("malloc() failed");
        threadArgs -> clntSock = clntSock;

        /* Create client thread, using 'ThreadMain' function */
        if (pthread_create(&threadID, NULL, ThreadMain, (void *) threadArgs) != 0)
            DieWithError("pthread_create() failed");
        printf("Create client thread with threadID %ld\n", (long int) threadID);
    }
    /* NOT REACHED */
}

/* ThreadMain code for the newly created threads */
void *ThreadMain(void *threadArgs)
{
    int clntSock;                   /* Socket descriptor for client connection */
    /* next line guarantees that thread resources are deallocated upon return */
```

TCP-EchoServer-Thread.c (pg 67)

```
/* ThreadMain function for newly created threads */
void *ThreadMain(void *threadArgs)
{
    int clntSock;                   /* Socket descriptor for client connection */

    /* next line Guarantees that thread resources are deallocated upon return */
    pthread_detach(pthread_self());

    /* Extract socket file descriptor from argument passed to thread */
    clntSock = ((struct ThreadArgs *) threadArgs) -> clntSock;
    free(threadArgs);               /* Deallocate memory for argument */

    HandleTCPClient(clntSock);

    return (NULL);
 /* the return call deallocates all resources */
}
```



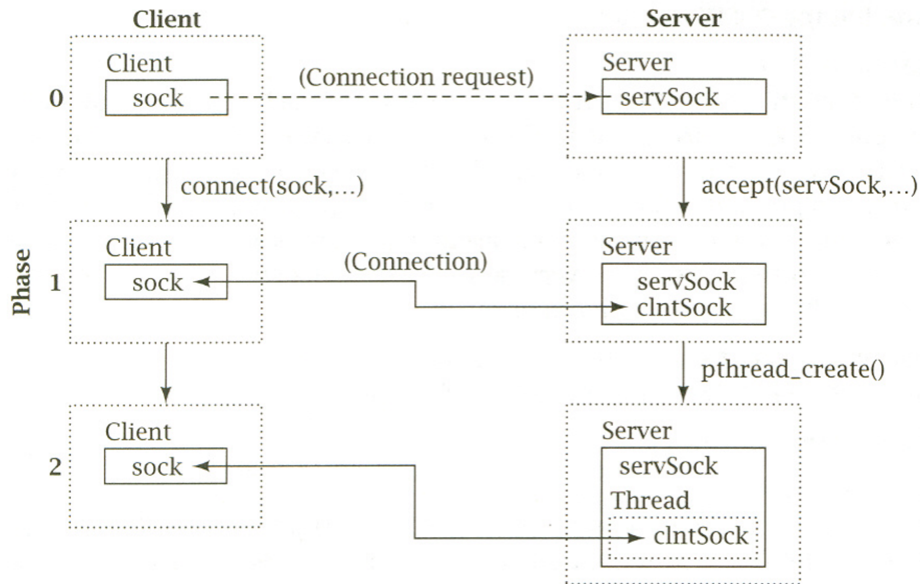
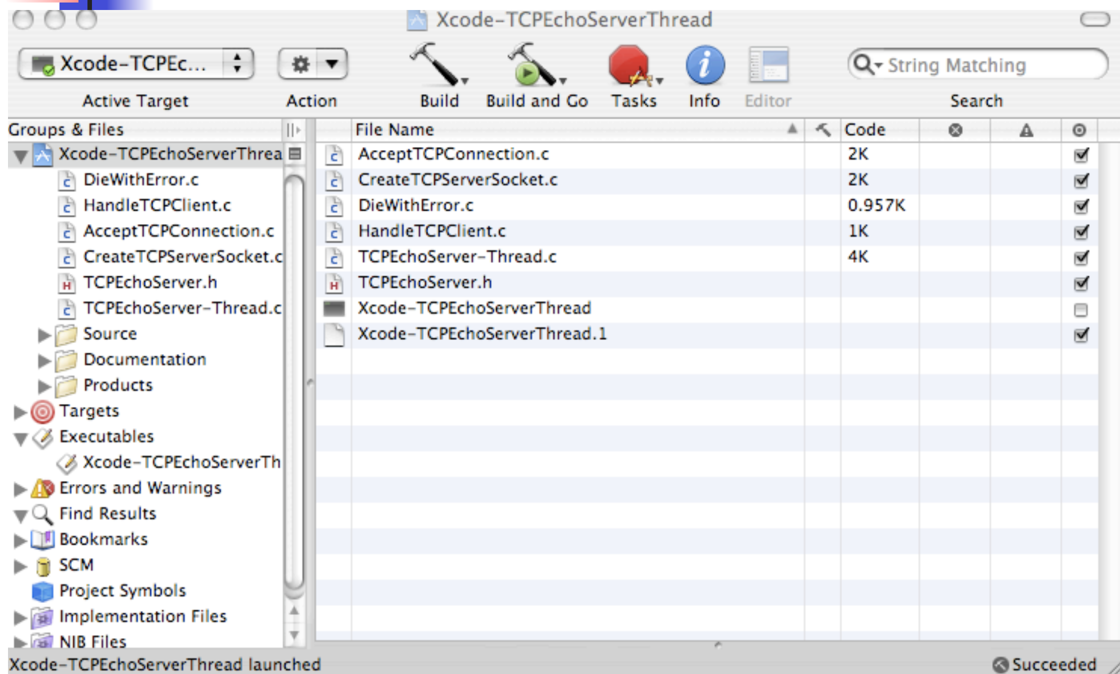


Figure 5.2: Threaded TCP echo server.

Source files for the EchoServer



Xcode-TCPEchoServerThread – Run Log

Xcode-TCPEchoServ... | Xcode-TCPEchoServ...

Active Target | Active Executable

```
Handling client 127.0.0.1

Server Process Accepting Connection: Creating Client thread

Server Process created thread ID = 25168384


Server Process Calling blocking Accept-TCP-Connection()


Thread calling HandleTCPClient


HandleTCPClient echoing string :  whats up doc?


Thread terminating

Handling client 127.0.0.1

Server Process Accepting Connection: Creating Client thread

Server Process created thread ID = 25167360


Server Process Calling blocking Accept-TCP-Connection()


Thread calling HandleTCPClient
```

Sample Server Activity

TCPEchoClient-Jan10 – Run Log

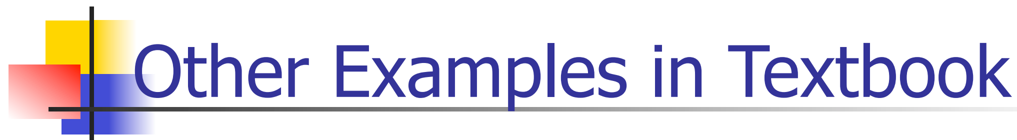
TCPEchoClient-Jan10 | TCPEchoClient-Jan10 | Attach | Run

Active Target | Active Executable

```
TCPEchoClient has connected to EchoServer

TCPEchoClient has sent string to EchoServer

TCPEchoClient has received a string from EchoServer:

whats up doc?

TCPEchoClient closing connection to EchoServer

TCPEchoClient-Jan10 has exited with status 0.
[Session started at 2007-01-21 19:37:01 -0500.]

-------------- entering TCPEchoClient ---------------------


TCPEchoClient using remote IP address 127.0.0.1

Using echoServerPort # = 4444

Socket created

Calling Connect()

TCPEchoClient has connected to EchoServer

TCPEchoClient has sent string to EchoServer

TCPEchoClient has received a string from EchoServer:

whats up doc?

TCPEchoClient closing connection to EchoServer

TCPEchoClient-Jan10 has exited with status 0.
```

TCPEchoClient-Jan10 exited normally. Succeeded

Sample Client Activity



Other Examples in Textbook

- The textbook contains discussions of several other aspects of multi-tasking, including:
- Signals: essentially interrupts to wake up blocked processes
- Forking servers with a maximum number of forked processes
- Threaded servers with a maximum number of theads
- Nonblocking servers, which can execute other tasks while waiting for an incoming connection(s)

5.6.1 Broadcasting (pg 77)

- So far, all sockets deal with 2 entities (server & client) - "unicast" communications
- Suppose an application requires multiple destinations ; we could create a unicast connection to each destination -> very inefficient
- Suppose 100K people in Toronto wanted to watch Deep-Blue vs. Kasparov chess game, streaming 1MByte/sec from IBM New Jersey; using unicast, there will be 100K unicast connections carrying the same data from NJ to TO over a single path over the Internet, using 100K Mbytes of link bandwidth !
- Solution: let the network IP routers duplicate data when appropriate
- 2 types of network duplication: "**broadcast**" & "**multicast**"
- Broadcast: all hosts on network receive a copy of the message
- Multicast: a subset of hosts receive a copy of the message
- In IP, only UDP sockets support broadcast & multicast

5.6.1 Broadcasting (pg 77)

- "**Local broadcast address**" **255.255.255.255** sends message to every host on the local network (ie local Ethernet); these messages are not forwarded by routers
- "**Directed broadcast**" allows broadcasts to all hosts on a specific network
- IPv4 32-bit addresses have 2 parts: typically 16 bits for network, 16 bits for host within the network
- A **directed broadcast** to all hosts over network 169.125 = 169.125.**255.255**
- Network-wide broadcasts over the entire internet are not allowed: consequences of misuse are too great, so these where omitted intentionally by the IETF
- Example: **BroadcastSender.c**, sends UDP broadcast of a string every 3 seconds to the specified broadcast address

BroadcastSender.c (pg 78)

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket() and bind() */
#include <arpa/inet.h>  /* for sockaddr_in */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

void DieWithError(char *errorMessage);  /* External error handling function */

int main(int argc, char *argv[])
{
    int sock;                         /* Socket */
    struct sockaddr_in broadcastAddr; /* Broadcast address (structure) */
    char *broadcastIP;                /* IP broadcast address (string) */
    unsigned short broadcastPort;     /* Server port */
    char *sendString;                 /* String to broadcast */
    int broadcastPermission;          /* Socket OPTION to set permission to broadcast */
    unsigned int sendStringLen;       /* Length of string to broadcast */

    if (argc < 4)                     /* Test for correct number of parameters */
    {
        fprintf(stderr,"Usage:  %s <IP Address> <Port> <Send String>\n", argv[0]);
        exit(1);
    }
```

BroadcastSender.c (pg 78)

```
broadcastIP = argv[1];            /* First arg:  broadcast IP address (string) */
broadcastPort = atoi(argv[2]);    /* Second arg:  broadcast port */
sendString = argv[3];             /* Third arg:  string to broadcast */

/* Create socket for sending/receiving datagrams */
if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    DieWithError("socket() failed");

/* Set socket OPTION to allow broadcasting */
broadcastPermission = 1;
if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST, (void *) &broadcastPermission,
      sizeof(broadcastPermission)) < 0)
    DieWithError("setsockopt() failed");

/* Construct local address structure */
memset(&broadcastAddr, 0, sizeof(broadcastAddr));   /* Zero out structure */
broadcastAddr.sin_family = AF_INET;                 /* Internet address family */
broadcastAddr.sin_addr.s_addr = inet_addr(broadcastIP);/* Broadcast IP address */
broadcastAddr.sin_port = htons(broadcastPort);         /* Broadcast port */
```


BroadcastSender.c (pg 78)

```
    sendStringLen = strlen(sendString);  /* Find length of sendString */
    for (;;) /* Run forever */
    {
         /* Broadcast sendString in datagram to clients every 3 seconds*/
         if (sendto(sock, sendString, sendStringLen, 0, (struct sockaddr *)
               &broadcastAddr, sizeof(broadcastAddr)) != sendStringLen)
             DieWithError("sendto() sent a different number of bytes than expected");

        sleep(3);   /* Avoids flooding the network */
    }
    /* NOT REACHED */
}
```

BroadcastReceiver.c (pg 78)

```c
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), sendto(), and recvfrom() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define MAXRECVSTRING 255  /* Longest string to receive */

void DieWithError(char *errorMessage);  /* External error handling function */

int main(int argc, char *argv[])
{
    int sock;                         /* Socket */
    struct sockaddr_in broadcastAddr; /* Broadcast Address (structure) */
    unsigned int broadcastPort;       /* Port */
    char recvString[MAXRECVSTRING+1]; /* Buffer for received string */
    int recvStringLen;                /* Length of received string */

    if (argc != 2)    /* Test for correct number of arguments */
    {
        fprintf(stderr,"Usage: %s <Broadcast Port>\n", argv[0]);
        exit(1);
    }

    broadcastPort = atoi(argv[1]);   /* First arg: broadcast port */
```

BroadcastReceiver.c (pg 78)

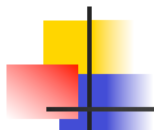
```c
    /* Create a best-effort datagram socket using UDP */
    if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        DieWithError("socket() failed");

    /* Construct bind structure */
    memset(&broadcastAddr, 0, sizeof(broadcastAddr));   /* Zero out structure */
    broadcastAddr.sin_family = AF_INET;                 /* Internet address family */
    broadcastAddr.sin_addr.s_addr = htonl(INADDR_ANY);  /* Any incoming interface */
    broadcastAddr.sin_port = htons(broadcastPort);      /* Broadcast port */

    /* Bind to the broadcast port */
    if (bind(sock, (struct sockaddr *) &broadcastAddr, sizeof(broadcastAddr)) < 0)
        DieWithError("bind() failed");

    /* Receive a single datagram from the server */
    if ((recvStringLen = recvfrom(sock, recvString, MAXRECVSTRING, 0, NULL, 0)) < 0)
        DieWithError("recvfrom() failed");

    recvString[recvStringLen] = '\0';       /* terminate with NULL symbol */
    printf("Received: %s\n", recvString);    /* Print the received string */

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

5.6.2 Multi-casting (pg 81)

- UDP **multi-cast** similar to **unicast**, main difference = **address**
- IPv4 allocates a range of address space for multi-casts, called '**class D'** addresses which range from 224.0.0.0 to 239.255.255.255
- A few multicast addresses are reserved
- A sender can send datagrams addressed to any **class D** address
- Example: MulticlassSender.c; sends string every 3 seconds to a specific multicast address
- Main differences: (1) multicast sender doesn't need to set the permission to multicast, (2) Time-to-Live (TTL) set for UDP datagrams;
- Each router decrements TTL, and when TTL = 0 packet is discarded (limits the # of routers a message passes through)

5.6.2 Multi-casting (pg 83)

- Receivers in a multi-cast session need to join a "**multicast**" group which has a **Class D** address
- A multiclass request message is sent by the socket interface to join a group
- Example: MulticlassReceiver.c
- Main difference from broadcast: (1) multicast receiver specifies multicast group to join using the **ip_mreq** structure
- **Imr_multiaddr** contains internet address for the group(ie 224.1.2.3)

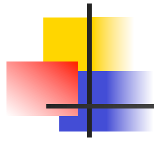


MulticastSender.c (pg 81)

```
#include <stdio.h>      /* for fprintf() */
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for sleep() */

void DieWithError(char *errorMessage);  /* External error handling function */

int main(int argc, char *argv[])
{
    int sock;                         /* Socket */
    struct sockaddr_in multicastAddr; /* Multicast address */
    char *multicastIP;                /* IP Multicast address (string) */
    unsigned short multicastPort;     /* Server port */
    char *sendString;                 /* String to multicast */
    unsigned char multicastTTL;       /* TTL of multicast packets */
    unsigned int sendStringLen;       /* Length of string to multicast */

    if ((argc < 4) || (argc > 5))         /* Test for correct number of parameters */
    {
        fprintf(stderr,"Usage:  %s <Multicast Address> <Port> <Send String> [<TTL>]\n",
                argv[0]);
        exit(1);
    }
```

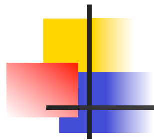
MulticastSender.c (pg 81)

```
    multicastIP = argv[1];             /* First arg:  multicast IP address string */
    multicastPort = atoi(argv[2]);     /* Second arg:  multicast port */
    sendString = argv[3];              /* Third arg:  String to multicast */

    if (argc == 5)                      /* Is TTL specified on command-line? */
        multicastTTL = atoi(argv[4]);   /* Command-line specified TTL */
    else
        multicastTTL = 1;               /* Default TTL = 1 */

    /* Create socket for sending/receiving datagrams */
    if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        DieWithError("socket() failed");

    /* Set TTL of multicast packet */
    if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, (void *) &multicastTTL,
          sizeof(multicastTTL)) < 0)
        DieWithError("setsockopt() failed");

    /* Construct local address structure */
    memset(&multicastAddr, 0, sizeof(multicastAddr));   /* Zero out structure */
    multicastAddr.sin_family = AF_INET;                 /* Internet address family */
    multicastAddr.sin_addr.s_addr = inet_addr(multicastIP);/* Multicast IP address */
    multicastAddr.sin_port = htons(multicastPort);         /* Multicast port */

    sendStringLen = strlen(sendString);  /* Find length of sendString */
    for (;;) /* Run forever */
```



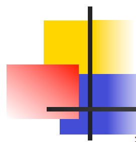
MulticastSender.c (pg 81)

```
    {
        /* Multicast sendString in datagram to clients every 3 seconds */
        if (sendto(sock, sendString, sendStringLen, 0, (struct sockaddr *)
              &multicastAddr, sizeof(multicastAddr)) != sendStringLen)
            DieWithError("sendto() sent a different number of bytes than expected");
        sleep(3);
    }
    /* NOT REACHED */
}
```

Inet_addr() function

- Pointer = inet_addr(address)
- Function receives a pointer to a character string of the multicast address, in dotted-quad notation
- Function returns a pointer to a binary version of the multicast address, in the network-byte order (big-endian)

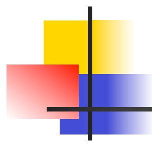


MulticastReceiver.c (pg 81)

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), sendto(), and recvfrom() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define MAXRECVSTRING 255  /* Longest string to receive */

void DieWithError(char *errorMessage);  /* External error handling function */

int main(int argc, char *argv[])
{
    int sock;                         /* Socket */
    struct sockaddr_in multicastAddr; /* Multicast Address */
    char *multicastIP;                /* IP Multicast Address (string) */
    unsigned int multicastPort;       /* Port */
    char recvString[MAXRECVSTRING+1]; /* Buffer for received string */
    int recvStringLen;                /* Length of received string */
    struct ip_mreq multicastRequest;  /* structure for Multicast address to join */

    if (argc != 3)    /* Test for correct number of arguments */
    {
        fprintf(stderr,"Usage: %s <Multicast IP addr> <Multicast Port>\n", argv[0]);
        exit(1);
    }
```

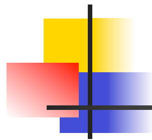
MulticastReceiver.c (pg 81)

```
multicastIP = argv[1];          /* First arg: Multicast IP address (dotted quad) */
multicastPort = atoi(argv[2]);/* Second arg: Multicast port */

/* Create a best-effort datagram socket using UDP */
if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    DieWithError("socket() failed");

/* Construct multiclass address structure */
memset(&multicastAddr, 0, sizeof(multicastAddr));   /* Zero out structure */
multicastAddr.sin_family = AF_INET;                 /* Internet address family */
multicastAddr.sin_addr.s_addr = htonl(INADDR_ANY);  /* Any incoming interface */
multicastAddr.sin_port = htons(multicastPort);      /* Multicast port */

/* Bind to the multicast port we chose*/
if (bind(sock, (struct sockaddr *) &multicastAddr, sizeof(multicastAddr)) < 0)
    DieWithError("bind() failed");

/* use the NEW structure: Specify the multicast group */
multicastRequest.imr_multiaddr.s_addr = inet_addr(multicastIP);  /* string to binary*/
/* Accept multicast from any interface IP address */
multicastRequest.imr_interface.s_addr = htonl(INADDR_ANY);
/* Join the multicast address using socket OPTIONs */

if (setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (void *) &multicastRequest,
      sizeof(multicastRequest)) < 0)
      DieWithError("setsockopt() failed");
```



MulticastReceiver.c (pg 81)

```
    /* Receive a single datagram from the server */
    if ((recvStringLen = recvfrom(sock, recvString, MAXRECVSTRING, 0, NULL, 0)) < 0)
        DieWithError("recvfrom() failed");

    recvString[recvStringLen] = '\0';
    printf("Received: %s\n", recvString);    /* Print the received string */

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

Summary

- Broadcasting works well if most hosts wish to receive a message
- According to textbook, most routers do not forward broadcast packets (to avoid flooding the internet)
- Disadvantage of multicast is that each receiver must know the address of a multicast group to join
- Any application could create a server to provide clients with multicast content and the multicast address to join to get that content
- However, broadcasting does not require any special address (just use 255.255.255.255), and most hosts set to receive broadcasts by default
- Therefore, it is easy to broadcast a message like "Where is the printer"