

Introduction to Network Programming

CSC 790

WAKE FOREST
UNIVERSITY

Department of Computer Science

Fall 2009

Socket Application Programming Interface (API)

- Introduced in 1981 Berkeley Software Distribution (BSD 4.1)
- Originally only Unix → WinSock *almost* the same
- Connects *sockets* on two hosts
- Data transfer similar to Unix file operations
- Two services provided
 - Datagram
 - Stream
- Socket interface is **generic**, it can be used with a variety of protocols (not just UDP/TCP/IP)

Review of Unix File Operations

- Unix file operations read/write to a *file descriptor*
 - An integer associated with an open *file*
 - A file can be data-file, device, network connection, ...

```
1 int fileDescriptor; // file descriptor
2 char buffer [256]; // buffer for read/write
3
4 fileDescriptor = open("file.dat", O_RDWR | O_CREAT);
5 // code storing info in buffer ...
6 write(fileDescriptor, buffer, 256);
7 close(fileDescriptor);
```

- Sockets are very similar, except for setup

Generic Data Types

Data types are needed to represent a socket and an address

- A socket is an `int` like a file descriptor
- `sockaddr` is a **generic** struct containing socket address information

```
1 struct sockaddr
2 {
3     unsigned short sa_family; // address family
4     char sa_data [14]; // protocol address
5 };
```

- `sa_family` identifies the address family (`AF_INET`)
- `sa_data` contains a network address
- Prefer something more IP specific, which is `sockaddr_in`

Internet Data Types

- `sockaddr_in` is a *parallel* struct containing Internet socket address information

```

1 struct sockaddr_in
2 {
3     short int          sin_family; // address family
4     unsigned short int sin_port;   // port number
5     struct in_addr     sin_addr;   // Internet address
6     char               sin_zero[8]; // filler, no data
7 };

```

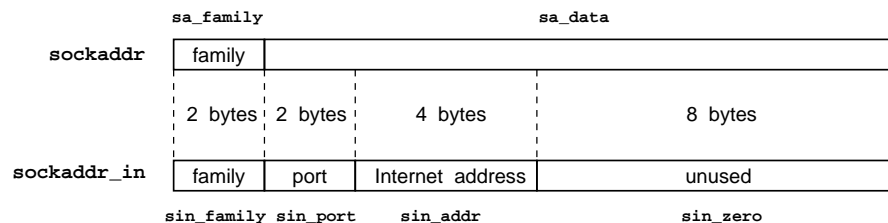
- `in_addr` is an Internet address

```

1 struct in_addr
2 {
3     unsigned long int s_addr; // 4 bytes, IPv4 address
4 };

```

What is parallel? Why is it important?



- Socket calls are *generic* (`sockaddr`) but we will use Internet addresses (`sockaddr_in`)
 - Since the structs are the same size (parallel) → **cast**

Byte Order

As you may recall...

- There are two types of byte orderings
 - Most significant byte first (also called **network byte order**)
 - Least significant byte first
- Computers (based on architecture) will use one **or** the other
 - Therefore we need to be consistent... remember to convert
- How do we convert? Use the following functions
 - `htons()` is *host to network* short
 - `htonl()` is *host to network* long
 - `ntohs()` is *network to host* short
 - `ntohl()` is *network to host* long

Address Example

Create an address for `www.cs.wfu.edu` web server

```
1 #include <arpa/inet.h> // for sockaddr_in and inet_addr
2 #include <string.h> // for memset
3
4 struct sockaddr_in wfuAddr; // www.cs.wfu.edu
5
6 wfuAddr.sin_family = AF_INET; // Internet address
7 wfuAddr.sin_port = htons(80); // port (network byte order)
8
9 // copy IP address (ascii to network byte order)
10 inet_aton("152.17.140.92", &(wfuAddr.sin_addr));
11
12 // zero the rest of the struct
13 memset(&(wfuAddr.sin_zero), '\0', 8);
14
15 // print the address to the screen
16 cout << inet_ntoa(wfuAddr.sin_addr) << '\n';
```

Creating a Socket

Before sending/receiving must create a socket

- Must specify, protocol family (PF_INET), type (SOCK_DGRAM or SOCK_STREAM), and protocol (0 for default)

```

1 #include <sys/socket.h> // for socket()
2
3 int sock; // stores the socket descriptor
4 if((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP))
5     == -1)
6 {
7     cerr << "Could not create socket\n";
8     exit(1);
9 }

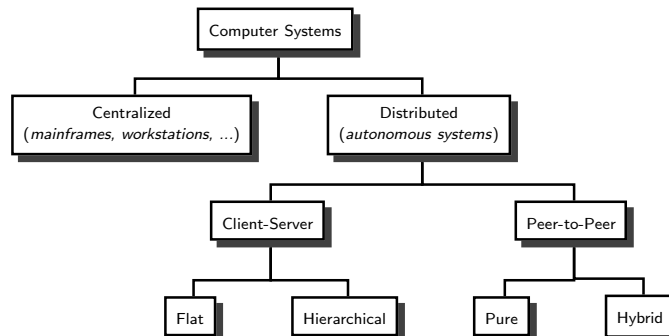
```

- Returns the socket number, -1 if error
- Once done with the socket, your program must close it (like a file)
close(sock);

Is the above socket UDP or TCP?

Communication Models

- Simple network programming typically uses a *client server* model
 - The server passively listens for communication
 - The client initiates the communication
- An alternative is peer-to-peer
 - Each entity can be a client or server



TCP or UDP

- There are two types of services available in the Internet
 - **User Datagram Protocol (UDP)** is an unreliable transmission
 - **Transport Control Protocol (TCP)** is a reliable stream
- Service must be specified once the socket is created
 - Creating the socket with `SOCK_DGRAM` uses UDP

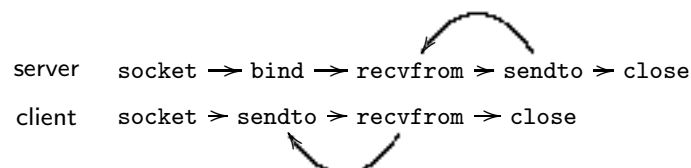

```
1 sock = socket(PF_INET, SOCK_DGRAM, 0)
```
 - Creating the socket with `SOCK_STREAM` used TCP


```
1 sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)
```
- Functions used to transmit data differ based on the service

UDP Sockets

Datagram sockets are connectionless and *do not have connection setup*

- Client
 - `socket`, create new end point
 - `sendto` and `recvfrom`, transmit data
- Server
 - `socket`, creates a new end point
 - `bind`, binds an *address* to the connection
 - `sendto` and `recvfrom`, transmit data



- Client does **not** establish a connection
 - connect is not needed
- Server does not *accept* a connection

What is the implication of no established connection?

Does the server still wait for datagrams?

Datagram sendto()

To sendto() the program must specify

- Socket, data, data length, flags, to address, to address length

```
1 int sendto(int sock, const void *msg, int len,  
2           unsigned int flags,  
3           const struct sockaddr *toAddr,  
4           int toLength);
```

- Assume we want to send a C-string

```
1 char msg[] = "What_time_is_it?";  
2 sendto(sock, msg, strlen(msg) + 1, 0,  
3       (struct sockaddr*) &srvAddr,  
4       sizeof(struct sockaddr));
```

- sendto() will return the number of data bytes sent, -1 if error

Datagram recvfrom()

Similarly, to `recvfrom()` the program must specify

- Socket, buffer (to store data), buffer length, flags, *from* address, and *from* address length

```
1 int recvfrom(int sock, void *buffer, int bufferLength,
2             unsigned int flags,
3             struct sockaddr *fromAddr,
4             int *fromLength);
```

- Assume we want to receive data from a datagram

```
1 char buffer [256];
2 sockaddr_in fromAddr;
3 unsigned int addrLength = sizeof(struct sockaddr);
4 recvfrom(sock, buffer, 256, 0, (struct sockaddr*)
5         &fromAddr, &addrLength);
```

- `recvfrom()` returns the number of data bytes, -1 if error
 - `fromAddr` stores the datagram address
 - `addrLength` stores the address length *must be initialized*

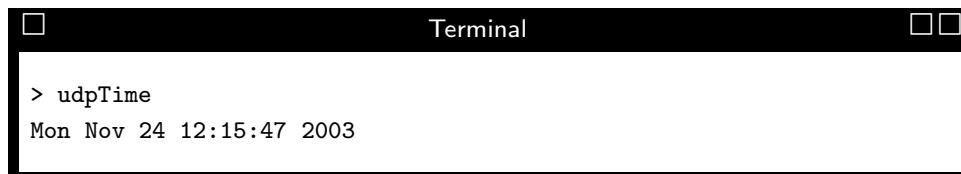
UDP Time Service

TCP/IP defines a service (port 13) that allows one machine to obtain the current data and time from another. The following program will obtain the time from 152.17.140.3

```
1 #include<iostream>
2 #include<sys/socket.h> // socket, sendto(), recvfrom()
3 #include<arpa/inet.h> // struct sockaddr
4 #include<unistd.h> // close()
5 #include<string.h> // memset()
6 using namespace std;
7
8 int main()
9 {
10     int sock; // socket for datagram communication
11     if((sock = socket(PF_INET, SOCK_DGRAM, 0)) == -1)
12     {
13         cerr << "Could not create socket\n";
14         exit(1);
15     }
```

```
1 struct sockaddr_in srvAddr; // address of server
2 srvAddr.sin_family = AF_INET; // Internet address
3 srvAddr.sin_port = htons(13); // port 13
4 srvAddr.sin_addr.s_addr = inet_addr("152.17.140.13");
5 memset(&(srvAddr.sin_zero), '\0', 8); // set to zero
6
7 // to receive the time, just ask the question...
8 char* msg = "What_time_is_time?";
9 if(sendto(sock, msg, strlen(msg) + 1, 0,
10 (struct sockaddr*) &srvAddr,
11 sizeof(struct sockaddr)) == -1)
12 {
13     cerr << "Could_not_send_to_socket\n";
14     exit(2);
15 }
```

```
1 char buffer[256]; // stores data received from socket
2 unsigned int addrLength = sizeof(sockaddr);
3
4 // will block until datagram received from socket
5 if(recvfrom(sock, buffer, 256, 0, (struct sockaddr*)
6 &srvAddr, &addrLength) == -1)
7 {
8     cerr << "Could_not_receive_from_socket\n";
9     exit(3);
10 }
11 close(sock);
12 cout << buffer << '\n';
13 return 0;
14 }
```



```
Terminal
> udpTime
Mon Nov 24 12:15:47 2003
```

Example UDP Client Server Program

Write a UDP client and a server that will send/receive C-strings

- Client will send the message "Hello from client" to the server
 - Assume the destination port is 1848
- Server will reply to the client "Hello *w.x.y.z*"
 - Listen for datagrams on port 1848
 - Must be started before client program

UDP C-String Client Program

```

1 #include<iostream>
2 #include<sys/socket.h> // socket, sendto(), recvfrom()
3 #include<arpa/inet.h> // struct sockaddr
4 #include<unistd.h> // close()
5 #include<string.h> // memset()
6 using namespace std;
7 int main()
8 {
9     int sock; // socket for datagram communication
10    if((sock = socket(PF_INET, SOCK_DGRAM, 0)) == -1)
11    {
12        cerr << "Could not create socket\n";
13        exit(1);
14    }
15    struct sockaddr_in srvAddr; // address of server
16    srvAddr.sin_family = AF_INET; // Internet address
17    srvAddr.sin_port = htons(1848); // port 1848
18    srvAddr.sin_addr.s_addr = inet_addr("152.17.140.17");
19    memset(&(srvAddr.sin_zero), '\0', 8); // set to zero

```

```

1  char* msg = "Hello_from_client"; // c-string to send
2  if(sendto(sock, msg, strlen(msg) + 1, 0,
3           (struct sockaddr*) &srvAddr,
4           sizeof(struct sockaddr)) == -1)
5  {
6      cerr << "Could_not_send_to_socket\n";
7      exit(2);
8  }
9
10 char buffer[256]; // stores data received from socket
11 unsigned int addrLength = sizeof(sockaddr);
12 if(recvfrom(sock, buffer, 256, 0,
13            (struct sockaddr*) &srvAddr,
14            &addrLength) == -1)
15 {
16     cerr << "Could_not_receive_from_socket\n";
17     exit(3);
18 }
19 cout << buffer << '\n';
20
21 close(sock);
22 return 0;
23 }

```

UDP C-String Server Program

```

1  #include<iostream>
2  #include<sys/socket.h> // socket, sendto(), recvfrom()
3  #include<arpa/inet.h> // struct sockaddr
4  #include<unistd.h> // close()
5  #include<string.h> // memset()
6  using namespace std;
7  int main()
8  {
9      int sock; // socket for datagram communication
10     if((sock = socket(PF_INET, SOCK_DGRAM, 0)) == -1)
11     {
12         cerr << "Could_not_create_socket\n";
13         exit(1);
14     }
15     struct sockaddr_in myAddr; // my (server) addr
16     myAddr.sin_family = AF_INET; // Internet address
17     myAddr.sin_port = htons(1848); // port 1848
18     myAddr.sin_addr.s_addr = INADDR_ANY; // my address
19     memset(&(myAddr.sin_zero), '\0', 8); // set to zero
20 }

```

```

21     if(bind(sock, (struct sockaddr*) &myAddr, sizeof(struct
22             sockaddr)) == -1)
23     {
24         cerr << "Could not bind to port\n";
25         exit(1);
26     }
27
28     struct sockaddr_in destAddr;    // dest (client) address
29     unsigned int destAddrLength = sizeof(struct sockaddr);
30
31     char buffer[256];                // data from socket
32     if(recvfrom(sock, buffer, 256, 0, (struct sockaddr*)
33             &destAddr, &destAddrLength) == -1)
34     {
35         cerr << "Could not receive from socket\n";
36         exit(3);
37     }
38
39     cout << buffer << '\n';
40
41     char *reply = new char[strlen("Hello ") +
42             strlen(inet_ntoa(destAddr.sin_addr)) + 1];
43     strcpy(reply, "Hello ");

```

E. W. Fulp

Fall 2009

```

44     strcat(reply, inet_ntoa(destAddr.sin_addr));
45     if(sendto(sock, reply, strlen(reply) + 1, 0,
46             (struct sockaddr*) &destAddr,
47             sizeof(struct sockaddr)) == -1)
48     {
49         cerr << "Could not send to socket\n";
50         exit(4);
51     }
52     close(sock);
53     return 0;
54 }

```

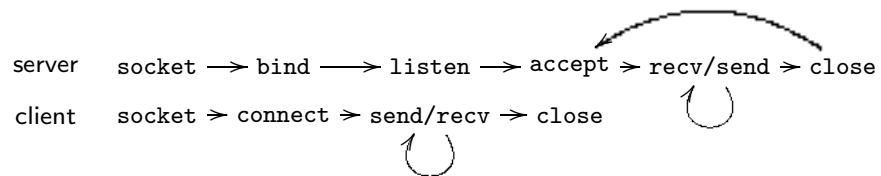
The image shows two terminal windows side-by-side. The left window, titled 'Terminal', shows a prompt '>' followed by the command 'udpStrSrv' (with a red box around '1'). The output is 'Hello from client' (with a red box around '3') followed by another prompt '>'. The right window, also titled 'Terminal', shows a prompt '>' followed by the command 'udpStrCli' (with a red box around '2'). The output is 'Hello 152.17.140.12' (with a red box around '4') followed by another prompt '>'.

E. W. Fulp

Fall 2009

TCP Client Server Model

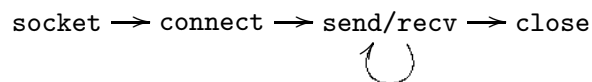
- Network programming typically uses a *client server* model
 - The server passively listens for communication
 - The client initiates the communication
- For TCP, once the server hears a connection request, handshake
 - Determine sequence numbers and buffer space
 - Once connection establish can send and receive data



What is the difference for UDP?

TCP Client

- TCP client must perform the following four steps
 1. Create a TCP socket using `socket`
 2. Establish connection to server using `connect`
 3. Communicate using `send` and `rcv`
 4. Close the connection with `close`



- Already discussed how to create a socket
 - Be certain the socket is `SOCK_STREAM` for TCP

TCP Client connect

To connect the client program must specify

- Socket, server address, and address length

```
1 int connect(int sock, struct sockaddr * serverAddr,
2            unsigned int addrLength);
```

- Assume we want to connect to a server

```
1 if(connect(sock, (struct sockaddr *) &serverAddr,
2           sizeof(struct sockaddr)) < 0)
3 {
4     cerr << "Could not connect to server\n";
5     exit(1);
6 }
```

- Connect will perform three-way handshake
- Once connected, client and server can send and recv data

Would UDP use the connect function?

TCP send

To send data (**TCP only**) the program must supply

- Socket, pointer to data, data length, and flags

```
1 int send(int sock, const void *msg,
2         unsigned int msgLength, int flags);
```

- Assume we want to send a C-string

```
1 char str[] = "Pluf still rules";
2 int strLength = strlen(str) + 1;
3 if(send(sock, str, strLength, 0) != strLength)
4 {
5     cerr << "Could not send all the data\n";
6     exit(1);
7 }
```

- send() will return the number of bytes sent

How does send know which address and port to send to?

TCP recv

To recv data (**TCP only**) the program must supply

- Socket, pointer to buffer, maximum buffer length, and flags

```
1 int recv(int sock, void *buffer,  
2         unsigned int bufferLength, int flags);
```

- Assume we want to receive data from a TCP socket

```
1 char buffer[256];  
2 int numBytes;  
3 if((numBytes = recv(sock, buffer, 256, 0)) <= 0)  
4 {  
5     cerr << "Could not recv from socket\n";  
6     exit(1);  
7 }
```

- recv() returns the number of bytes read
 - Blocks until data received or timeout
 - Data is placed in the buffer

Closing a Socket

Once communication over a socket is complete, you must close

- Specify the socket and use the close() function

```
1 #include <unistd.h>  
2  
3 close(sock);
```

- Cannot read/write to the socket after close()

Why is close necessary?

TCP Time Service

TCP/IP defines a service (port 13) that allows one machine to obtain the current data and time from another. The following **client** program will obtain the time from the **server** 152.17.140.3

```

1 #include<iostream>
2 #include<sys/socket.h> // socket(), send(), recv()
3 #include<arpa/inet.h> // struct socket sockaddr
4 #include<unistd.h> // close()
5 #include<string.h> // memset()
6 using namespace std;
7
8 int main()
9 {
10     int sock; // socket for datagram communication
11     if((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP))
12        == -1)
13     {
14         cerr << "Could not create socket\n";
15         exit(1);
16     }

```

```

1     struct sockaddr_in srvAddr; // time server
2     srvAddr.sin_family = AF_INET; // Internet addr
3     srvAddr.sin_port = htons(13); // port 13
4     srvAddr.sin_addr.s_addr = inet_addr("152.17.140.3");
5     memset(&(srvAddr.sin_zero), '\0', 8); // set to zero
6
7     if(connect(sock, (struct sockaddr *) &srvAddr,
8               sizeof(struct sockaddr)) == -1)
9     {
10         cerr << "Could not connect to server\n";
11         exit(2);
12     }
13
14     // to receive the time, just ask the question...
15     char* msg = "What time is it?";
16     if(send(sock, msg, strlen(msg) + 1, 0) == -1)
17     {
18         cerr << "Could not send to socket\n";
19         exit(3);
20     }

```

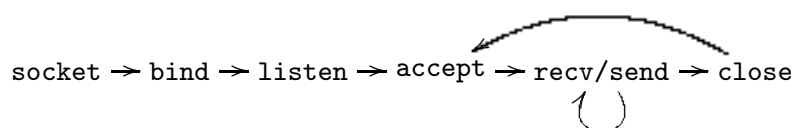
```
1 char buffer[256]; // stores data received from socket
2 // will block until datagram received from socket
3 if(recv(sock, buffer, 256, 0) == -1)
4 {
5     cerr << "Could not receive from socket\n";
6     exit(4);
7 }
8
9 cout << buffer << '\n';
10 close(sock);
11 return 0;
12 }
```



```
Terminal
> tcpTime
Mon Nov 3 12:15:47 2008
```

TCP Server

- TCP server must complete the following six steps
 1. Create a TCP socket using `socket`
 2. Assign a port number to the socket using `bind`
 3. Tell system to allow connections to port using `listen`
 4. Accept connection request using `accept`
 5. Communicate using `recv` and `send` via **new** socket
 6. Close the connection using `close`



- Already know about `socket`, `recv`, `send`, and `close`

TCP bind

To bind a port to an address the program must supply

- Socket, address, and address length

```
1  int bind(int sock, struct sockaddr *localAddr,
2          unsigned int addrLength);
```

- Assume we wanted to bind to port 1122

```
1  struct sockaddr_in localAddr;           // local addr
2  memset(&localAddr, 0, sizeof(localAddr)); // zero addr
3  localAddr.sin_family = AF_INET;        // Internet
4  localAddr.sin_addr.s_addr = htonl(INADDR_ANY); // any
5  localAddr.sin_port = htons(1122);     // local port
6
7  if(bind(sock, (struct sockaddr *) &localAddr,
8          sizeof(struct sockaddr)) < 0)
9  {
10     cerr << "Could not bind to socket\n";
11     exit(1);
12 }
```

TCP listen

To listen for a connection request the program must supply

- Socket and queue limit

```
1  int listen(int sock, int queueLimit);
   – queueLimit is the maximum outstanding requests
```

- Assume we wanted to listen for incoming connections

```
1  #define MAX_PENDING 5 // maximum outstanding req
2  if(listen(sock, MAX_PENDING) < 0)
3  {
4     cerr << "Could not listen on socket\n";
5     exit(1);
6  }
```

- listen() returns 0 on success, -1 otherwise
 - The queue is used by accept

TCP accept

To accept an incoming connection the program must supply

- Socket, client address, and address length

```
1  int accept(int sock, struct sockaddr *clientAddress,
2             unsigned int *addrLength);
```

- Assume we wanted to accept an incoming connection

```
1  int clientSock;
   // socket for connection
2  struct sockaddr_in clientAddr; // incoming address
3  unsigned int addrLength = sizeof(struct sockaddr_in);
4  if((clientSock = accept(sock, (struct sockaddr *)
5                          &clientAddr, &addrLength)) < 0)
6  {
7      cerr << "Could not accept connection\n";
8      exit(1);
9  }
```

- `accept()` returns a descriptor for a *new* socket
 - Dequeues the next connection on the socket queue
 - Creates a **new** socket for arriving connection
 - Sets the address and address length variables
 - If the queue is empty, then **block**
 - If error then `accept` returns -1
- After `accept()` program can `recv()` and `send()`
 - As described there are two sockets, one for receiving a connection request and another created for the connection

Which socket is used for sending and which for receiving?
 - The socket that has been bound to a port and marked *listening* is **never** used for sending and receiving in TCP

Example TCP Client Server Program

Write a TCP client and a server that will send/receive C-strings

- Client will send the message "Hello from client" to the server
 - Assume the destination port is 1848
- Server will reply to the client "Hello *w.x.y.z*"
 - Listen for a connection on port 1848
 - Must be started before client program

Why must the server be running before the client starts?

TCP C-String Client Program

```

1 #include<iostream>
2 #include<sys/socket.h> // socket, send(), recv()
3 #include<arpa/inet.h> // struct sockaddr
4 #include<unistd.h> // close()
5 #include<string.h> // memset()
6 using namespace std;
7
8 int main()
9 {
10     int sock; // socket for stream communication
11     if((sock = socket(PF_INET, SOCK_STREAM,
12                     IPPROTO_TCP)) == -1)
13     { cerr << "Could not create socket\n"; exit(1); }
14
15     struct sockaddr_in srvAddr; // address of server
16     srvAddr.sin_family = AF_INET; // Internet address
17     srvAddr.sin_port = htons(1848); // port 1848
18     srvAddr.sin_addr.s_addr = inet_addr("152.17.140.17");
19     memset(&(srvAddr.sin_zero), '\0', 8); // set to zero
20

```

```

21 // establish connection with server
22 if(connect(sock, (struct sockaddr *) &srvAddr,
23           sizeof(struct sockaddr)) < 0)
24 { cerr << "Could not connect\n"; exit(2); }
25
26 char* msg = "Hello from client"; // c-string to send
27 if(send(sock, msg, strlen(msg) + 1, 0) == -1)
28 { cerr << "Could not send to socket\n"; exit(3); }
29
30 char buffer[256]; // stores data received from socket
31 unsigned int addrLength = sizeof(sockaddr);
32 int numBytes = 0;
33 if((numBytes = recv(sock, buffer, 256, 0)) <= 0)
34 { cerr << "Could not receive from socket\n"; exit(4);
35 }
36
37 cout << buffer << '\n';
38
39 close(sock);
40 return 0;
41 }

```

TCP C-String Server Program

```

1 #include<iostream>
2 #include<sys/socket.h> // socket, send(), recv()
3 #include<arpa/inet.h> // struct sockaddr
4 #include<unistd.h> // close()
5 #include<string.h> // memset()
6 using namespace std;
7
8 int main()
9 {
10 int sock; // socket for datagram communication
11 if((sock = socket(PF_INET, SOCK_STREAM,
12                 IPPROTO_TCP)) == -1)
13 { cerr << "Could not create socket\n"; exit(1); }
14
15 struct sockaddr_in myAddr; // my (server) addr
16 myAddr.sin_family = AF_INET; // Internet addr
17 myAddr.sin_port = htons(1848); // port 1848
18 myAddr.sin_addr.s_addr = INADDR_ANY; // my address
19 memset(&(myAddr.sin_zero), '\0', 8); // set to zero

```

```

1  if(bind(sock, (struct sockaddr*) &myAddr,
2      sizeof(struct sockaddr)) == -1)
3  { cerr << "Could not bind to port\n"; exit(2); }
4
5  if(listen(sock, 0) < 0)
6  { cerr << "Could not listen to socket\n"; exit(3); }
7
8  int clientSock; // client socket
9  struct sockaddr_in clientAddr; // client addr
10 unsigned int clientAddrLength =
11     sizeof(struct sockaddr_in);
12
13 if((clientSock = accept(sock, (struct sockaddr *)
14     &clientAddr, &clientAddrLength)) < 0)
15 { cerr << "Could not accept connection\n"; exit(4); }
16
17 char buffer[256]; // data received from socket
18 if(recv(clientSock, buffer, 256, 0) == -1)
19 { cerr << "Could not receive from socket\n"; exit(3); }
20 }
21 cout << buffer << '\n';

```

E. W. Fulp

Fall 2009

```

1  char *reply = new char[strlen("Hello ") +
2      strlen(inet_ntoa(destAddr.sin_addr)) + 1];
3  strcpy(reply, "Hello ");
4  strcat(reply, inet_ntoa(clientAddr.sin_addr));
5  if(send(clientSock, reply, strlen(reply) + 1, 0) == -1)
6  { cerr << "Could not send to socket\n"; exit(4); }
7
8  close(sock);
9  return 0;
10 }

```

Terminal

```

> tcpStrSrv
Hello from client
>

```

Terminal

```

> tcpStrCli
Hello 152.17.140.12
>

```

E. W. Fulp

Fall 2009