Introduction to Network Programming

CSC 343·643

WAKE FOREST
UNIVERSITY
Department of Computer Science

Fall 2013

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, ...

```
int fileDescriptor;  // file descriptor
char buffer[256];      // buffer for read/write

fileDescriptor = open("file.dat", O_RDWR | O_CREAT);
// code storing info in buffer ...
write(fileDescriptor, buffer, 256);
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

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

  - `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

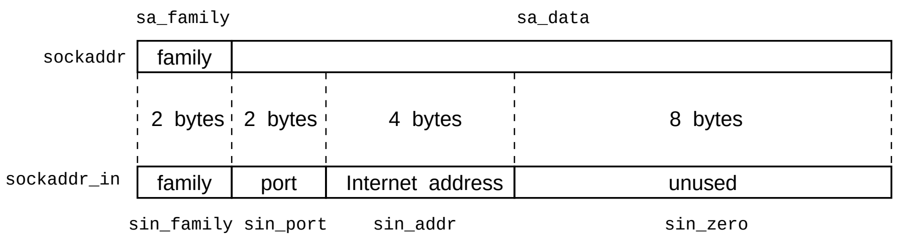
- sockaddr_in is a *parallel* struct containing Internet socket address information

```
struct sockaddr_in
{
    short int           sin_family;  // address family
    unsigned short int  sin_port;    // port number
    struct in_addr      sin_addr;    // Internet address
    char                sin_zero[8]; // filler, no data
};
```

- in_addr is an Internet address

```
struct in_addr
{
    unsigned long int s_addr;   // 4 bytes, IPv4 address
};
```

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

  How are numbers spoken in different languages, for example 21?

- 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

```
#include<arpa/inet.h>    // for sockaddr_in and inet_addr
#include<string.h>       // for memset

struct sockaddr_in wfuAddr;     // www.cs.wfu.edu

wfuAddr.sin_family = AF_INET;   // Internet address
wfuAddr.sin_port = htons(80);   // port (network byte order)

// copy IP address (ascii to network byte order)
inet_aton("152.17.140.92", &(wfuAddr.sin_addr));

// zero the rest of the struct
memset(&(wfuAddr.sin_zero), '\0', 8);

// print the address to the screen
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)

```
#include <sys/socket.h>  // for socket()

int sock;  // stores the socket descriptor
if((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP))
     == -1)
{
    cerr << "Could␣not␣create␣socket␣\n";
    exit(1);
}
```

- 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?

TCP or UDP

- There are two types of services available in the Internet
  - **User Datagram Procotol** (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

    ```
    sock = socket(PF_INET, SOCK_DGRAM, 0)
    ```

  - Creating the socket with `SOCK_STREAM` used TCP

    ```
    sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)
    ```

- Functions used to transmit data differ based on the service

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

$$\begin{tikzcd}
\text{server} & \texttt{socket} \arrow[r] & \texttt{bind} \arrow[r] & \texttt{listen} \arrow[r] & \texttt{accept} \arrow[r] & \texttt{recv/send} \arrow[r] \arrow[l, bend right] \arrow[loop below] & \texttt{close} \\
\text{client} & \texttt{socket} \arrow[r] & \texttt{connect} \arrow[r] & \texttt{send/recv} \arrow[r] \arrow[loop below] & \texttt{close}
\end{tikzcd}$$

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 `recv`
  4. Close the connection with `close`

$$\begin{tikzcd}
\texttt{socket} \arrow[r] & \texttt{connect} \arrow[r] & \texttt{send/recv} \arrow[r] \arrow[loop below] & \texttt{close}
\end{tikzcd}$$

- 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

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

- Assume we want to connect to a server

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

- 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

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

- Assume we want to send a C-string

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

- `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

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

- Assume we want to receive data from a TCP socket

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

- `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

```
#include<unistd.h>

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

```
#include<iostream>
#include<sys/socket.h>    // socket(), send(), recv()
#include<arpa/inet.h>     // struct socket sockaddr
#include<unistd.h>        // close()
#include<string.h>        // memset()
#include<cstdlib>         // exit()
using namespace std;

int main()
{
   int sock;  // socket for datagram communication
   if((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP))
       == -1)
   {
      cerr << "Could␣not␣create␣socket␣\n";
      exit(1);
   }
```

```
   struct sockaddr_in srvAddr;               // time server
   srvAddr.sin_family = AF_INET;             // Internet addr
   srvAddr.sin_port = htons(13);             // port 13
   srvAddr.sin_addr.s_addr = inet_addr("152.17.140.3");
   memset(&(srvAddr.sin_zero), '\0', 8);   // set to zero

   if(connect(sock, (struct sockaddr *) &srvAddr,
                 sizeof(struct sockaddr)) == -1)
   {
      cerr << "Could␣not␣connect␣to␣server␣\n";
      exit(2);
   }

   // to receive the time, just ask the question...
   char* msg = "What␣time␣is␣it?";
   if(send(sock, msg, strlen(msg) + 1, 0) == -1)
   {
      cerr << "Could␣not␣send␣to␣socket␣\n";
      exit(3);
   }
```

```
    char buffer[256];  // stores data received from socket
    // will block until datagram received from socket
    if(recv(sock, buffer, 256, 0) == -1)
    {
       cerr << "Could␣not␣receive␣from␣socket␣\n";
       exit(4);
    }

    cout << buffer << '\n';
    close(sock);
    return 0;
}
```

Terminal

```
> tcpTime
Mon Nov 8 12:15:47 2010
```

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`

$$\begin{tikzcd}
\texttt{socket} \arrow[r] & \texttt{bind} \arrow[r] & \texttt{listen} \arrow[r] & \texttt{accept} \arrow[r] & \texttt{recv/send} \arrow[r] \arrow[loop below] & \texttt{close} \arrow[ll, bend right]
\end{tikzcd}$$

- 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

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

- Assume we wanted to bind to port 1122

```
 struct sockaddr_in localAddr;                 // local addr
 memset(&localAddr, 0, sizeof(localAddr)); // zero addr
 localAddr.sin_family = AF_INET;               // Internet
 localAddr.sin_addr.s_addr = htonl(INADDR_ANY); // any
 localAddr.sin_port = htons(1122);             // local port

if(bind(sock, (struct sockaddr *) &localAddr,
           sizeof(struct sockaddr)) < 0)
{
   cerr << "Could not bind to socket \n";
   exit(1);
}
```

TCP `listen`

To `listen` for a connection request the program must supply

- Socket and queue limit

```
   int listen(int sock, int queueLimit);
```

  - queueLimit is the maximum outstanding requests

- Assume we wanted to `listen` for incoming connections

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

- `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

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

- Assume we wanted to accept an incoming connection

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

- `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?

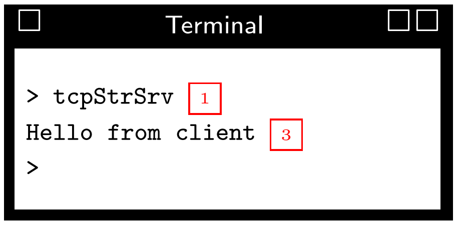
TCP C-String Client Program

```
#include<iostream>
#include<sys/socket.h>    // socket, send(), recv()
#include<arpa/inet.h>     // struct sockaddr
#include<unistd.h>        // close()
#include<string.h>        // memset()
#include<cstdlib>         // exit()
using namespace std;

int main()
{
   int sock;  // socket for stream communication
   if((sock = socket(PF_INET, SOCK_STREAM,
                     IPPROTO_TCP)) == -1)
   { cerr << "Could not create socket \n"; exit(1); }

   struct sockaddr_in srvAddr;      // address of server
   srvAddr.sin_family = AF_INET;    // Internet address
   srvAddr.sin_port = htons(1848); // port 1848
   srvAddr.sin_addr.s_addr = inet_addr("152.17.140.17");
   memset(&(srvAddr.sin_zero), '\0', 8);   // set to zero
   // establish connection with server
   if(connect(sock, (struct sockaddr *) &srvAddr,
              sizeof(struct sockaddr)) < 0)
   {  cerr << "Could␣not␣connect␣\n"; exit(2);  }

   char* msg = "Hello␣from␣client";    // c-string to send
   if(send(sock, msg, strlen(msg) + 1, 0) == -1)
   {  cerr << "Could␣not␣send␣to␣socket␣\n"; exit(3);  }

   char buffer[256];  // stores data received from socket
   unsigned int addrLength = sizeof(sockaddr);
   int numBytes = 0;
   if((numBytes = recv(sock, buffer, 256, 0)) <= 0)
   {  cerr << "Could␣not␣receive␣from␣socket␣\n"; exit(4);  }
   cout << buffer << '\n';

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

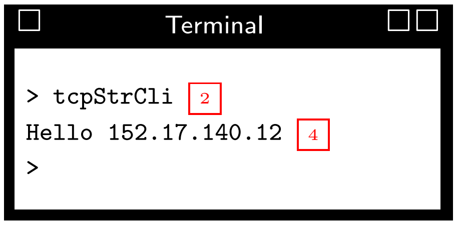
TCP C-String Server Program

```
#include<iostream>
#include<sys/socket.h>   // socket, send(), recv()
#include<arpa/inet.h>    // struct sockaddr
#include<unistd.h>       // close()
#include<string.h>       // memset()
#include<cstdlib>        // exit()
using namespace std;

int main()
{
   int sock;  // socket for datagram communication
   if((sock = socket(PF_INET, SOCK_STREAM,
                     IPPROTO_TCP)) == -1)
   {  cerr << "Could␣not␣create␣socket␣\n"; exit(1);  }

   struct sockaddr_in myAddr;            // my (server) addr
   myAddr.sin_family = AF_INET;          // Internet addr
   myAddr.sin_port = htons(1848);        // port 1848
   myAddr.sin_addr.s_addr = INADDR_ANY; // my address
   memset(&(myAddr.sin_zero), '\0', 8); // set to zero
   if(bind(sock, (struct sockaddr*) &myAddr,
           sizeof(struct sockaddr)) == -1)
   {  cerr << "Could␣not␣bind␣to␣port␣\n"; exit(2);  }

   if(listen(sock, 0) < 0)
   {  cerr << "Could␣not␣listen␣to␣socket␣\n"; exit(3);  }

   int clientSock;                        // client socket
   struct sockaddr_in clientAddr;         // client addr
   unsigned int clientAddrLength =
                        sizeof(struct sockaddr_in);

   if((clientSock = accept(sock, (struct sockaddr *)
                   &clientAddr, &clientAddrLength)) < 0)
   { cerr << "Could␣not␣accept␣connection␣\n"; exit(4); }

   char buffer[256];          // data received from socket
   if(recv(clientSock, buffer, 256, 0) == -1)
   {  cerr << "Could␣not␣receive␣from␣socket␣\n"; exit(3);  }
   cout << buffer << '\n';
```

```
   char *reply = new char[strlen("Hello␣") +
              strlen(inet_ntoa(clientAddr.sin_addr)) + 1];
   strcpy(reply, "Hello␣");
   strcat(reply, inet_ntoa(clientAddr.sin_addr));
   if(send(clientSock, reply, strlen(reply) + 1, 0) == -1)
   {  cerr << "Could␣not␣send␣to␣socket␣\n"; exit(4);  }

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



```
> tcpStrSrv
Hello from client
>
```



```
> tcpStrCli
Hello 152.17.140.12
>
```

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

$$\begin{tikzcd}
\text{server} & \texttt{socket} \arrow[r] & \texttt{bind} \arrow[r] & \texttt{recvfrom} \arrow[r] & \texttt{sendto} \arrow[r] \arrow[l, bend right] & \texttt{close} \\
\text{client} & \texttt{socket} \arrow[r] & \texttt{sendto} \arrow[r] & \texttt{recvfrom} \arrow[r] \arrow[l, bend left] & \texttt{close} &
\end{tikzcd}$$

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

What is the implication of no established connection?

Does the server still wait for datagrams?

In TCP `accept` assigned a new socket to the incoming data, how will communication take place in UDP?

Datagram `sendto()`

To `sendto()` the program must specify

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

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

- Assume we want to send a C-string

```
char msg[] = "What␣time␣is␣it?";
sendto(sock, msg, strlen(msg) + 1, 0,
       (struct sockaddr*) &srvAddr,
        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

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

- Assume we want to receive data from a datagram

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

```
#include<iostream>
#include<sys/socket.h>    // socket, sendto(), recvfrom()
#include<arpa/inet.h>     // struct sockaddr
#include<unistd.h>        // close()
#include<string.h>        // memset()
#include<cstdlib>         // exit()
using namespace std;

int main()
{
   int sock;  // socket for datagram communication
   if((sock = socket(PF_INET, SOCK_DGRAM, 0)) == -1)
   {
      cerr << "Could␣not␣create␣socket␣\n";
      exit(1);
   }
```

```
   struct sockaddr_in srvAddr;         // address of server
   srvAddr.sin_family = AF_INET;       // Internet address
   srvAddr.sin_port = htons(13);       // port 13
   srvAddr.sin_addr.s_addr = inet_addr("152.17.140.13");
   memset(&(srvAddr.sin_zero), '\0', 8);  // set to zero

   // to receive the time, just ask the question...
   char* msg = "What␣time␣is␣time?";
   if(sendto(sock, msg, strlen(msg) + 1, 0,
                (struct sockaddr*) &srvAddr,
                 sizeof(struct sockaddr)) == -1)
   {
      cerr << "Could␣not␣send␣to␣socket␣\n";
      exit(2);
   }
```

```
    char buffer[256];  // stores data received from socket
    unsigned int addrLength = sizeof(sockaddr);

    // will block until datagram received from socket
    if(recvfrom(sock, buffer, 256, 0, (struct sockaddr*)
                &srvAddr, &addrLength) == -1)
    {
       cerr << "Could␣not␣receive␣from␣socket␣\n";
       exit(3);
    }
    close(sock);
    cout << buffer << '\n';
    return 0;
}
```

Terminal

```
> udpTime
Mon Nov 8 12:15:47 2010
```

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

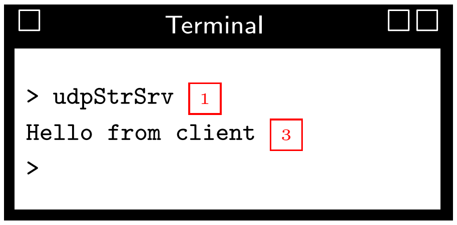
```
#include<iostream>
#include<sys/socket.h>   // socket, sendto(), recvfrom()
#include<arpa/inet.h>    // struct sockaddr
#include<unistd.h>       // close()
#include<string.h>       // memset()
#include<cstdlib>        // exit()
using namespace std;
int main()
{
   int sock;  // socket for datagram communication
   if((sock = socket(PF_INET, SOCK_DGRAM, 0)) == -1)
   {
      cerr << "Could␣not␣create␣socket␣\n";
      exit(1);
   }
   struct sockaddr_in srvAddr;         // address of server
   srvAddr.sin_family = AF_INET;       // Internet address
   srvAddr.sin_port = htons(1848);     // port 1848
   srvAddr.sin_addr.s_addr = inet_addr("152.17.140.17");
   memset(&(srvAddr.sin_zero), '\0', 8);   // set to zero
```

```
   char* msg = "Hello␣from␣client";      // c-string to send
   if(sendto(sock, msg, strlen(msg) + 1, 0,
             (struct sockaddr*) &srvAddr,
             sizeof(struct sockaddr)) == -1)
   {
      cerr << "Could␣not␣send␣to␣socket␣\n";
      exit(2);
   }

   char buffer[256];  // stores data received from socket
   unsigned int addrLength = sizeof(sockaddr);
   if(recvfrom(sock, buffer, 256, 0,
               (struct sockaddr*) &srvAddr,
               &addrLength) == -1)
   {
      cerr << "Could␣not␣receive␣from␣socket␣\n";
      exit(3);
   }
   cout << buffer << '\n';

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

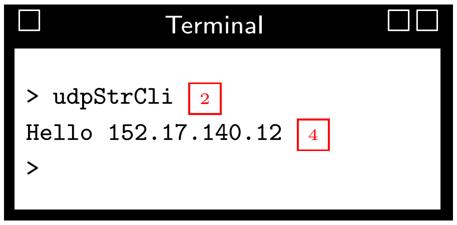
UDP C-String Server Program

```
#include<iostream>
#include<sys/socket.h>    // socket, sendto(), recvfrom()
#include<arpa/inet.h>     // struct sockaddr
#include<unistd.h>        // close()
#include<string.h>        // memset()
#include<cstdlib>         // exit()
using namespace std;
int main()
{
   int sock;  // socket for datagram communication
   if((sock = socket(PF_INET, SOCK_DGRAM, 0)) == -1)
   {
      cerr << "Could␣not␣create␣socket␣\n";
      exit(1);
   }
   struct sockaddr_in myAddr;           // my (server) addr
   myAddr.sin_family = AF_INET;         // Internet address
   myAddr.sin_port = htons(1848);       // port 1848
   myAddr.sin_addr.s_addr = INADDR_ANY; // my address
   memset(&(myAddr.sin_zero), '\0', 8); // set to zero
```

```
   if(bind(sock, (struct sockaddr*) &myAddr, sizeof(struct
           sockaddr)) == -1)
   {
      cerr << "Could␣not␣bind␣to␣port␣\n";
      exit(1);
   }

   struct sockaddr_in destAddr;   // dest (client) address
   unsigned int destAddrLength = sizeof(struct sockaddr);

   char buffer[256];              // data from socket
   if(recvfrom(sock, buffer, 256, 0, (struct sockaddr*)
               &destAddr, &destAddrLength) == -1)
   {
      cerr << "Could␣not␣receive␣from␣socket␣\n";
      exit(3);
   }

   cout << buffer << '\n';

   char *reply = new char[strlen("Hello␣") +
                 strlen(inet_ntoa(destAddr.sin_addr)) + 1];
   strcpy(reply, "Hello␣");
   strcat(reply, inet_ntoa(destAddr.sin_addr));

   if(sendto(sock, reply, strlen(reply) + 1, 0,
             (struct sockaddr*) &destAddr,
              sizeof(struct sockaddr)) == -1)
   {
      cerr << "Could not send to socket \n";
      exit(4);
   }
   close(sock);
   return 0;
}
```



```
> udpStrSrv
Hello from client
>
```



```
> udpStrCli
Hello 152.17.140.12
>
```