Computer Networks Lab

Socket Programming

Prof. Nikita Singhal
Department of Computer Engineering

Contents

- □ Client-Server Communication
- □ Socket and Socket based communication
- □ Socket Characteristics
- □ Socket API
- □ Connection based communication
- □ Connection less communication
- □ Socket Programming

Client Server Communication

- Server
  - passively waits for and responds to clients
  - passive socket
- Client
  - initiates the communication
  - must know the address and the port of the server
  - active socket

Sockets and Socket-based Communication

- Sockets provide an interface for programming networks at the transport layer. Network communication using Sockets is very much similar to performing file I/O.
- The streams used in file I/O operation are also applicable to socket-based I/O.
- Socket-based communication is independent of a programming language used for implementing it.
- That means, a socket program written in Java language can communicate to a program written in non-Java (say C or C++) socket program.

Socket Characteristics

- Socket are characterized by their domain, type and transport protocol.
- Common domains are:
  - IAF UNIX: address format is UNIX pathname
  - IAF INET: address format is host and port number
- Common types are:
  - virtual circuit: received in order transmitted and reliably
  - datagram: arbitrary order, unreliable

Socket Characteristics(continued)

- Each socket type has one or more protocols. Ex:
  - TCP/IP (virtual circuits)
  - UDP (datagram)
- Use of sockets:
  - Connection–based sockets communicate client-server: the server waits for a connection from the client
  - Connectionless sockets are peer-to-peer: each process is symmetric.

Socket Family

Name	Purpose
AF_UNIX, AF_LOCAL	Local communication
AF_INET	IPv4 Internet protocols
AF_INET6	IPv6 Internet protocols
AF_IPX	IPX - Novell protocols
AF_NETLINK	Kernel user interface device
AF_X25	ITU-T X.25 / ISO-8208 protocol
AF_AX25	Amateur radio AX.25 protocol
AF_ATMPVC	Access to raw ATM PVCs
AF_APPLETALK	Appletalk
AF_PACKET	Low level packet interface

Socket APIs

- socket: creates a socket of a given domain, type, protocol (buy a phone)
- bind: assigns a name to the socket (get a telephone number)
- listen: specifies the number of pending connections that can be queued for a server socket. (call waiting allowance)
- accept: server accepts a connection request from a client (answer phone)
- connect: client requests a connection request to a server (call)
- send, sendto: write to connection (speak)
- recv, recvfrom: read from connection (listen)
- shutdown: end the call

Connection-based communication (TCP)

- Server performs the following actions
  - socket: create the socket
  - bind: give the address of the socket on the server
  - listen: specifies the maximum number of connection requests that can be pending for this process
  - accept: establish the connection with a specific client
  - send,recv: stream-based equivalents of read and write (repeated)
  - shutdown: end reading or writing
  - close: release kernel data structures

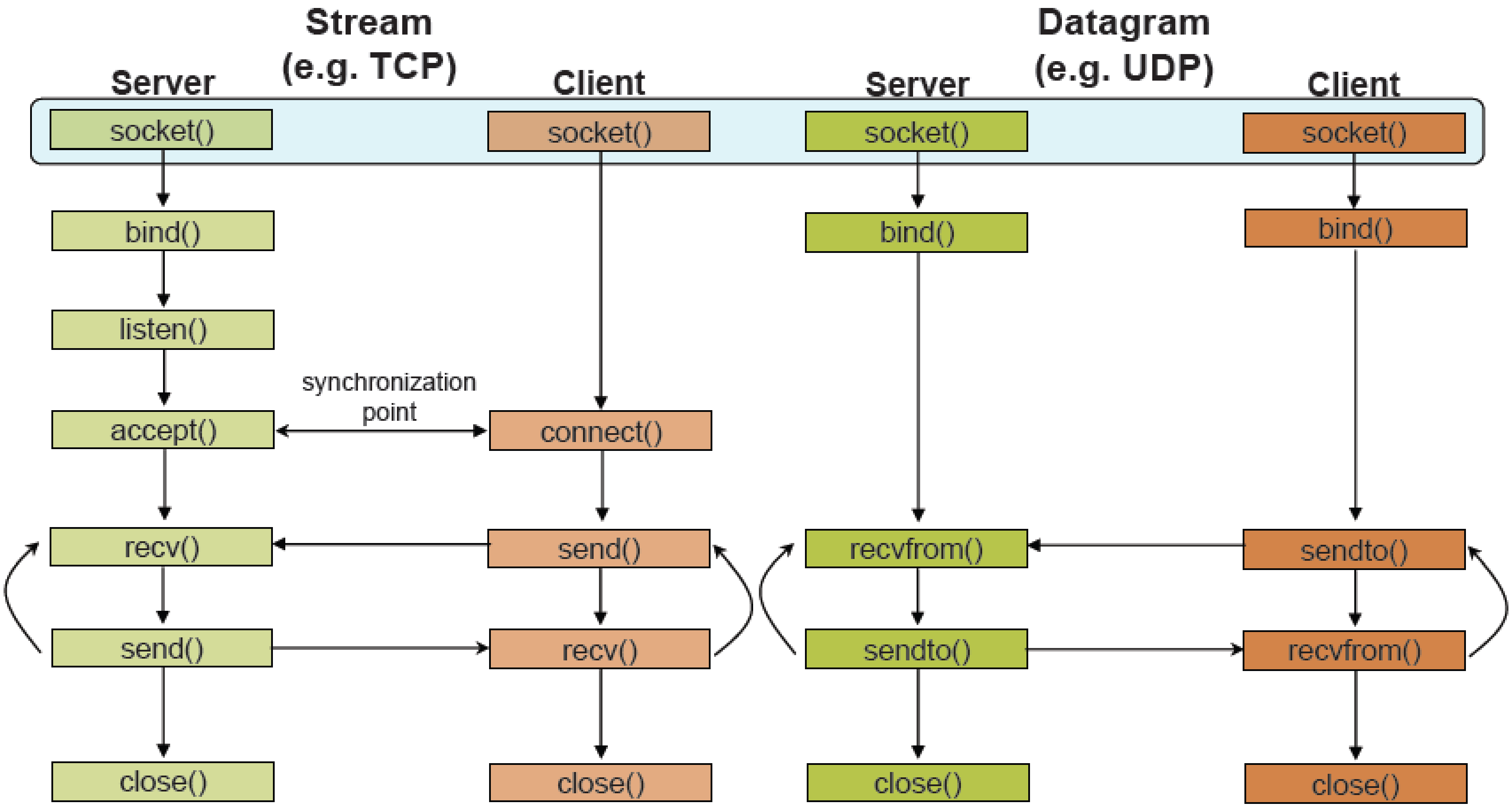
Connection-based communication (TCP)

- Client performs the following actions
  - socket: create the socket
  - connect: connect to a server
  - send, recv: (repeated)
  - shutdown
  - close

Connection-less communication (UDP)

- Communication is symmetric (peer-to-peer)
  - socket
  - bind: bind is optional for initiator
  - sendto, recvfrom (repeated)
  - shutdown
  - close

Client-Server Communication (Socket Programming)



Socket creation in C: socket()

-
  ```
  int sockid = socket(family, type, protocol);
  ```
  - sockid: socket descriptor, an integer (like a file-handle)
  - family: integer, communication domain, e.g.,
    - PF_INET, IPv4 protocols, Internet addresses (typically used)
    - PF_UNIX, Local communication, File addresses
  - type: communication type
    - SOCK_STREAM - reliable, 2-way, connection-based service
    - SOCK_DGRAM - unreliable, connectionless, messages of maximum length
  - protocol: specifies protocol
    - IPPROTO_TCP IPPROTO_UDP
    - usually set to 0 (i.e., use default protocol)
  - upon failure returns -1

☞ NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

Specifying Addresses

- Socket API defines a **generic** data type for addresses:

```
struct sockaddr {
   unsigned short sa_family;  /* Address family (e.g. AF_INET) */
   char sa_data[14];          /* Family-specific address information */
}
```

- Particular form of the sockaddr used for TCP/IP addresses:

```
struct in_addr {
   unsigned long s_addr;          /* Internet address (32 bits) */
}
struct sockaddr_in {
   unsigned short sin_family;     /* Internet protocol (AF_INET) */
   unsigned short sin_port;       /* Address port (16 bits) */
   struct in_addr sin_addr;     /* Internet address (32 bits) */
   char sin_zero[8];              /* Not used */
}
```

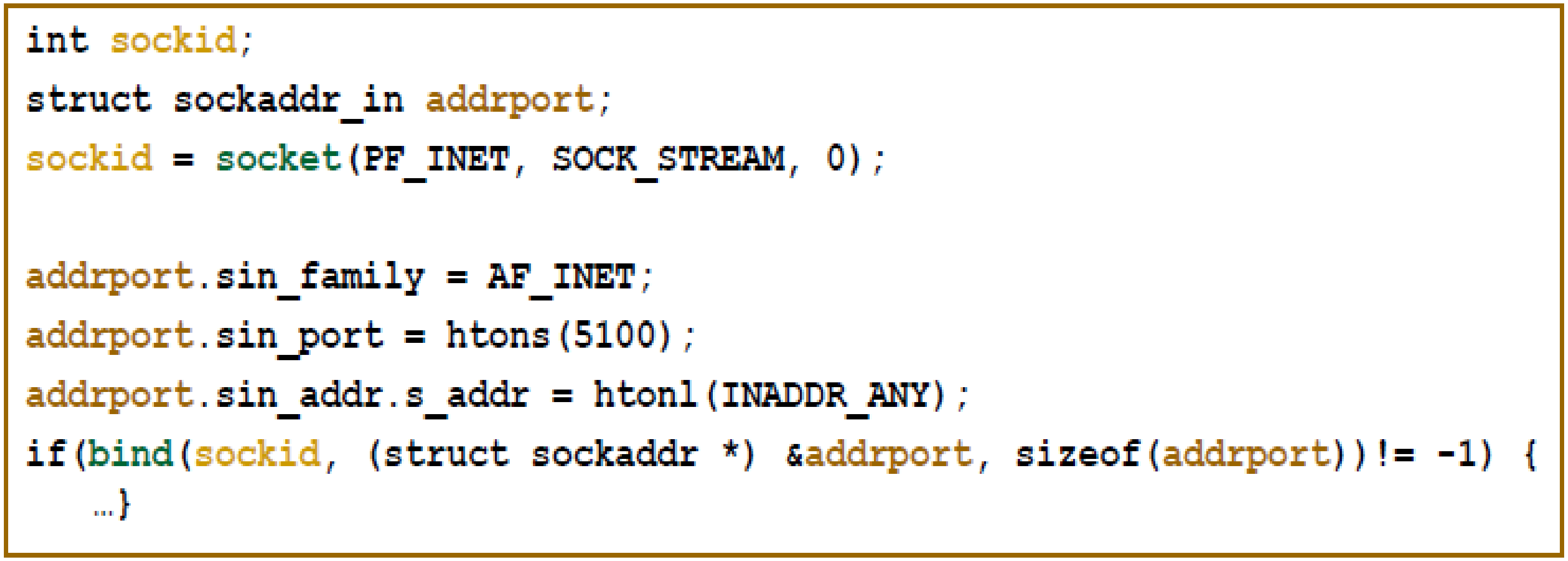
☞ **Important**: sockaddr_in can be casted to a sockaddr

Assign address to socket: bind()

- associates and reserves a port for use by the socket

-
  ```
  int status = bind(sockid, &addrport, size);
  ```
  - sockid: integer, socket descriptor
  - addrport: struct sockaddr, the (IP) address and port of the machine
    - for TCP/IP server, internet address is usually set to INADDR_ANY, i.e., chooses any incoming interface
  - size: the size (in bytes) of the addrport structure
  - status: upon failure -1 is returned

bind()-Example with TCP



```
int sockid;
struct sockaddr_in addrport;
sockid = socket(PF_INET, SOCK_STREAM, 0);

addrport.sin_family = AF_INET;
addrport.sin_port = htons(5100);
addrport.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sockid, (struct sockaddr *) &addrport, sizeof(addrport))!= -1) {
   …}
```

listen()

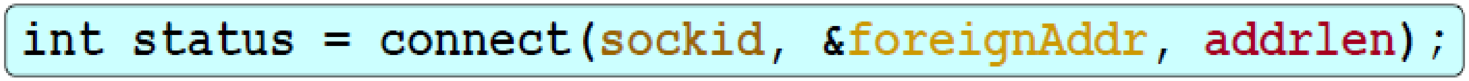
- Instructs TCP protocol implementation to listen for connections

- `int status = listen(sockid, queueLimit);`
  - **sockid**: integer, socket descriptor
  - **queuelen**: integer, # of active participants that can “wait” for a connection
  - **status**: 0 if listening, -1 if error
- `listen()` is **non-blocking**: returns immediately

- The listening socket (sockid)
  - is never used for sending and receiving
  - is used by the server only as a way to get new sockets

Establish Connection: connect()

- The client establishes a connection with the server by calling `connect()`

- 

  ```
  int status = connect(sockid, &foreignAddr, addrlen);
  ```

  - sockid: integer, socket to be used in connection
  - foreignAddr: struct sockaddr: address of the passive participant
  - addrlen: integer, sizeof(name)
  - status: 0 if successful connect, -1 otherwise

- `connect()` is **blocking**

Incoming Connection: accept()

- The server gets a socket for an incoming client connection by calling `accept()`
- `int s = accept(sockid, &clientAddr, &addrLen);`
  - s: integer, the new socket (used for data-transfer)
  - sockid: integer, the orig. socket (being listened on)
  - clientAddr: struct sockaddr, address of the active participant
    - filled in upon return
  - addrLen: sizeof(clientAddr): value/result parameter
    - must be set appropriately before call
    - adjusted upon return
- `accept()`
  - is **blocking**: waits for connection before returning
  - dequeues the next connection on the queue for socket (sockid)

Exchanging data with stream socket(TCP)

- `int count = send(sockid, msg, msgLen, flags);`
  - msg: const void[], message to be transmitted
  - msgLen: integer, length of message (in bytes) to transmit
  - flags: integer, special options, usually just 0
  - count: # bytes transmitted (-1 if error)
- `int count = recv(sockid, recvBuf, bufLen, flags);`
  - recvBuf: void[], stores received bytes
  - bufLen: # bytes received
  - flags: integer, special options, usually just 0
  - count: # bytes received (-1 if error)
- Calls are **blocking**
  - returns only after data is sent / received

Exchanging data with datagram socket (UDP)

-
  ```
  int count = sendto(sockid, msg, msgLen, flags,
  &foreignAddr, addrlen);
  ```
  - msg, msgLen, flags, count: same with `send()`
  - **foreignAddr**: struct sockaddr, address of the destination
  - **addrLen**: sizeof(foreignAddr)

-
  ```
  int count = recvfrom(sockid, recvBuf, bufLen,
  flags, &clientAddr, addrlen);
  ```
  - recvBuf, bufLen, flags, count: same with `recv()`
  - **clientAddr**: struct sockaddr, address of the client
  - **addrLen**: sizeof(clientAddr)

- Calls are **blocking**
  - returns only after data is sent / received

Socket close in C: close()

- When finished using a socket, the socket should be closed

-
  ```
  status = close(sockid);
  ```
  - sockid: the file descriptor (socket being closed)
  - status: 0 if successful, -1 if error

- Closing a socket
  - closes a connection (for stream socket)
  - frees up the port used by the socket

Example -Echo

- A client communicates with an "echo" server
- The server simply echoes whatever it receives back to the client

Example -Echo using stream socket

The server starts by getting ready to receive client connections…

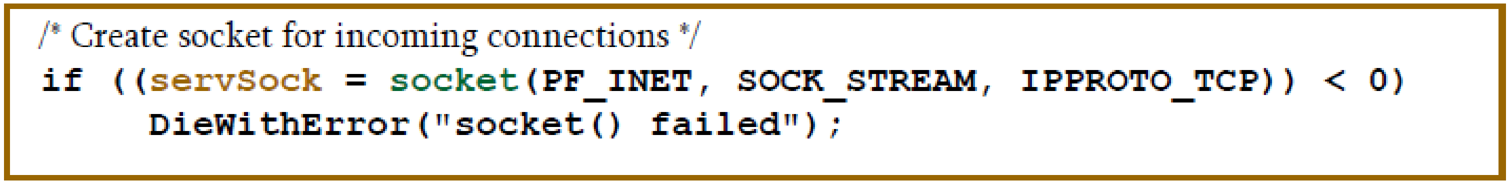
Client

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
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Example -Echo using stream socket



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

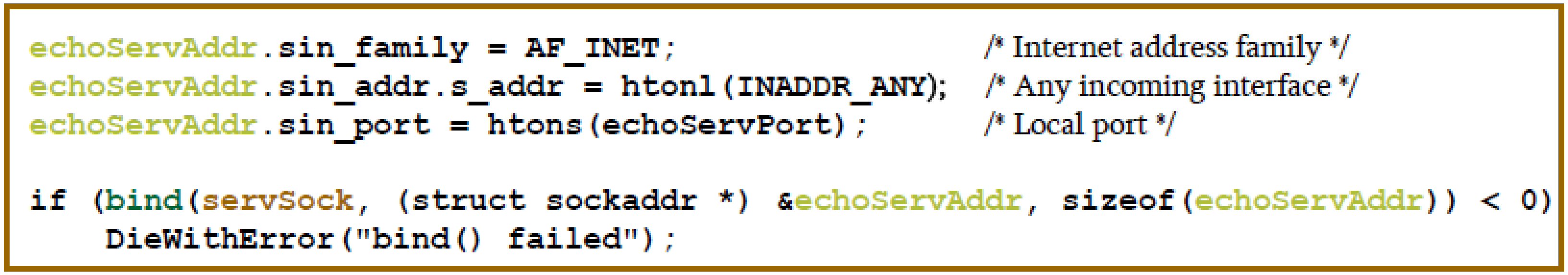
Client

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
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Example -Echo using stream socket



```
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 */

if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("bind() failed");
```

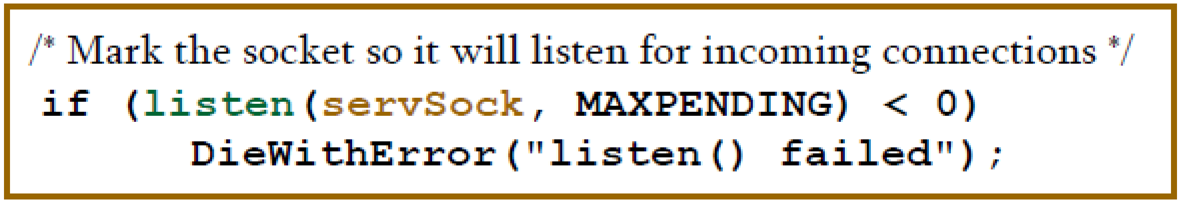
Client

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**
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Example -Echo using stream socket



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

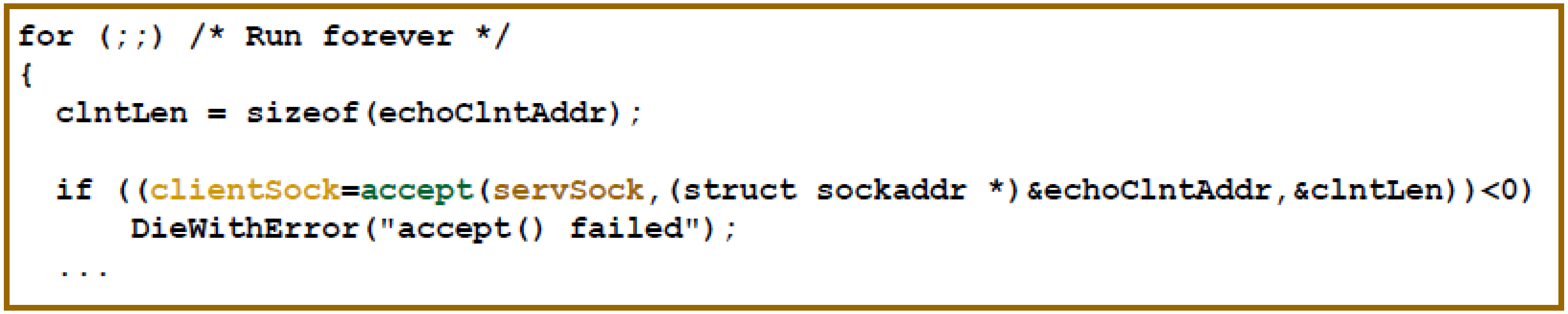
Client

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
3. **Set socket to listen**
4. Repeatedly:
    a. Accept new connection
    b. Communicate
    c. Close the connection

Example -Echo using stream socket



```
for (;;) /* Run forever */
{
  clntLen = sizeof(echoClntAddr);

  if ((clientSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen))<0)
      DieWithError("accept() failed");
  ...
```

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
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example -Echo using stream socket

Server is now blocked waiting for connection from a client

...

A client decides to talk to the server

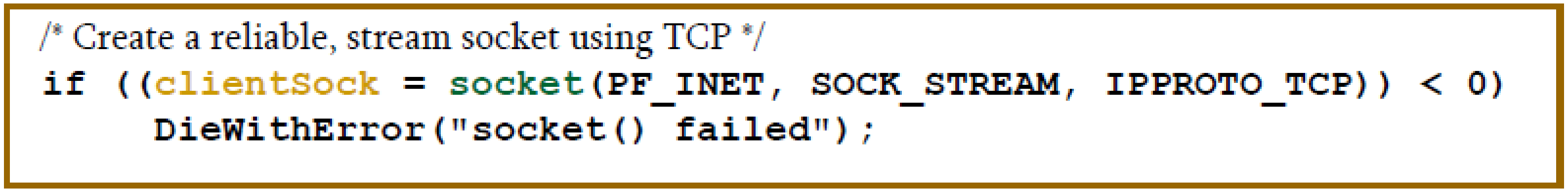
Client

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
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example -Echo using stream socket



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

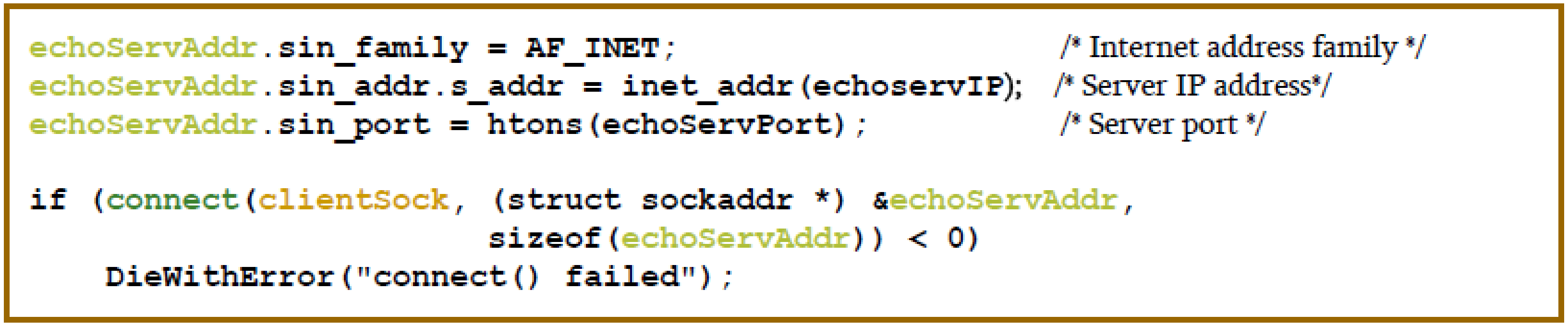
Client

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
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example -Echo using stream socket



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

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

Client

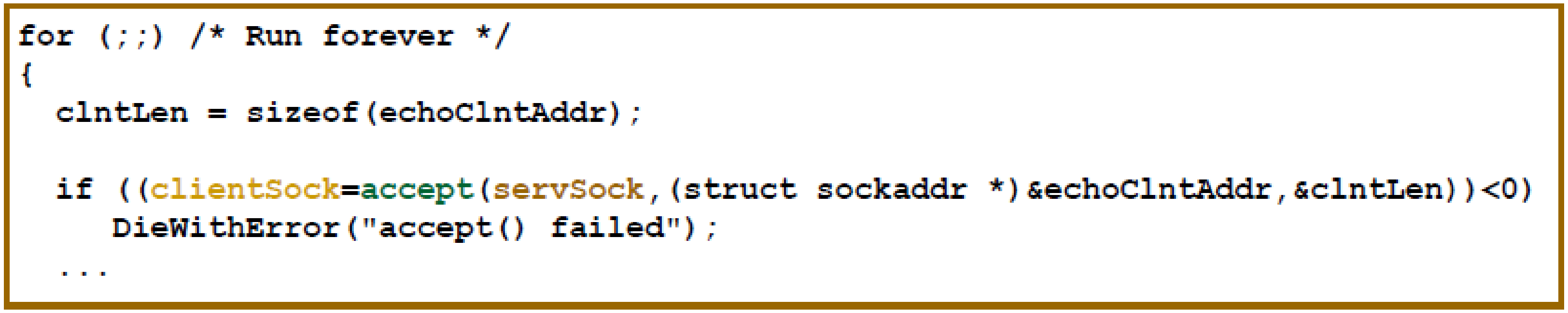
1. Create a TCP socket
2. **Establish connection**
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example -Echo using stream socket

Server's accept procedure in now unblocked and returns client's socket



```
for (;;) /* Run forever */
{
  clntLen = sizeof(echoClntAddr);

  if ((clientSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen))<0)
     DieWithError("accept() failed");
  ...
```

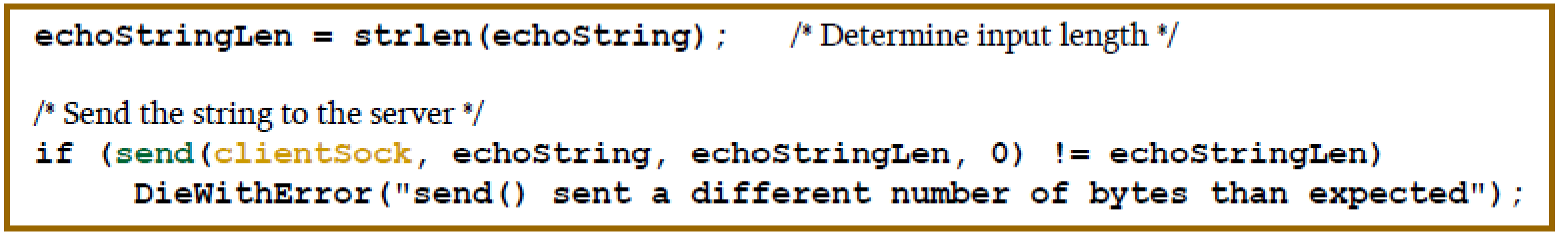
Client

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
3. Set socket to listen
4. Repeatedly:
   1. **Accept new connection**
   2. Communicate
   3. Close the connection

Example -Echo using stream socket



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

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

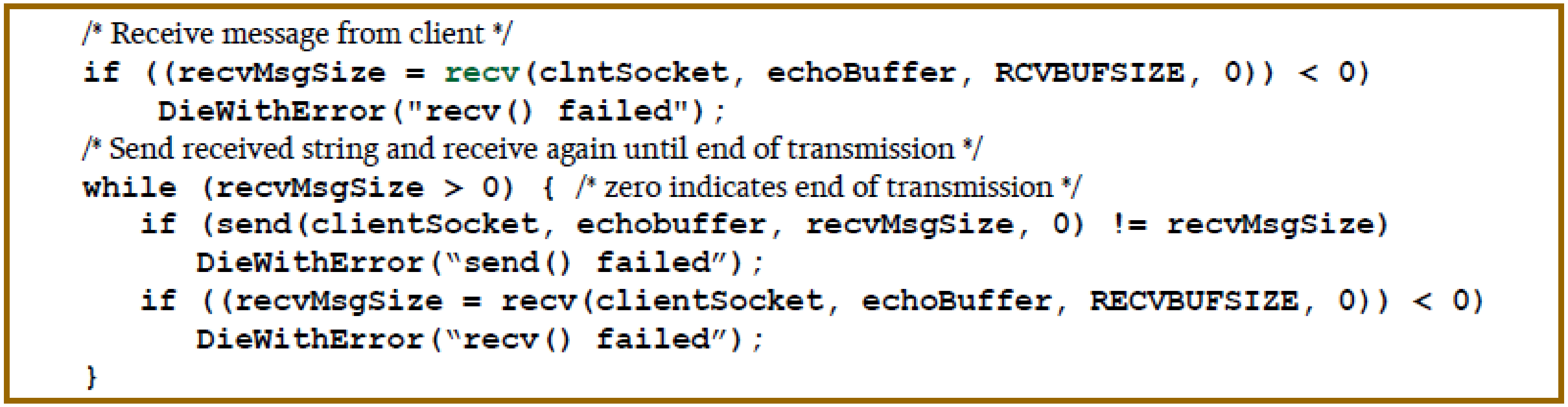
Client

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
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example -Echo using stream socket



```
/* Receive message from client */
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
    DieWithError("recv() failed");
/* Send received string and receive again until end of transmission */
while (recvMsgSize > 0) { /* zero indicates end of transmission */
   if (send(clientSocket, echobuffer, recvMsgSize, 0) != recvMsgSize)
      DieWithError("send() failed");
   if ((recvMsgSize = recv(clientSocket, echoBuffer, RECVBUFSIZE, 0)) < 0)
      DieWithError("recv() failed");
}
```

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
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. **Communicate**
   c. Close the connection

Example -Echo using stream socket

Similarly, the client receives the data from the server

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
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. **Communicate**
   c. Close the connection

Example -Echo using stream socket

```
close(clientSock);
```

```
close(clientSock);
```

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
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. **Close the connection**

Example -Echo using stream socket

Server is now blocked waiting for connection from a client

...

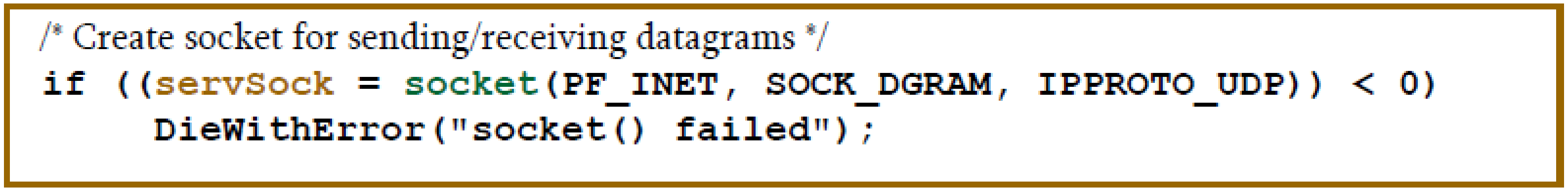
Client

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

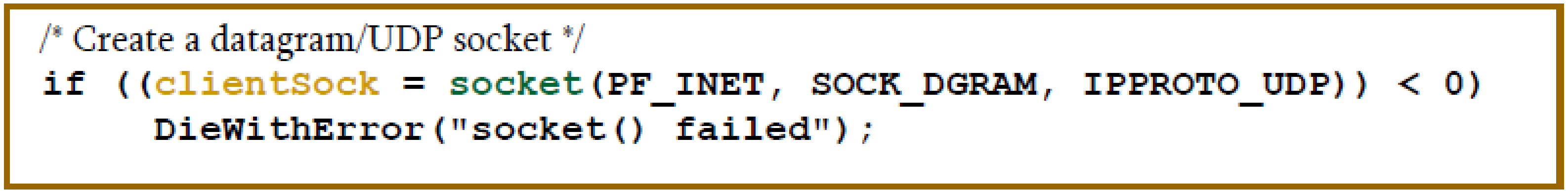
Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example -Echo using Datagram socket



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



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

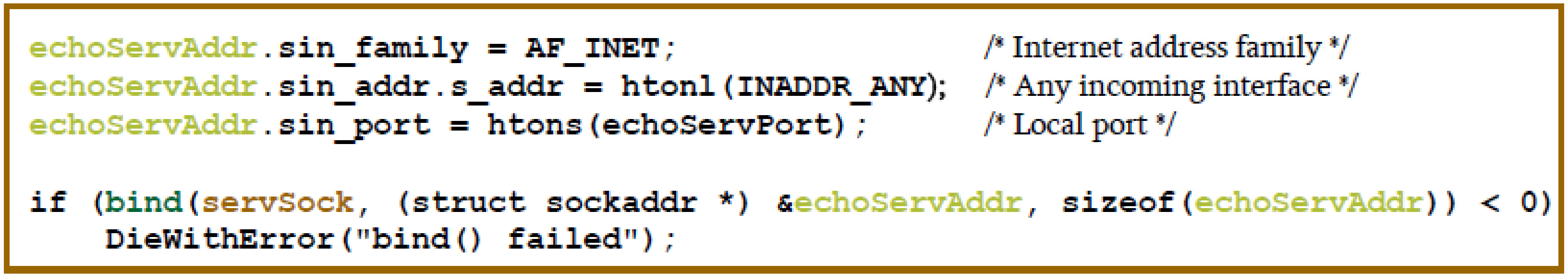
Client

1. **Create a UDP socket**
2. Assign a port to socket
3. Communicate
4. Close the socket

Server

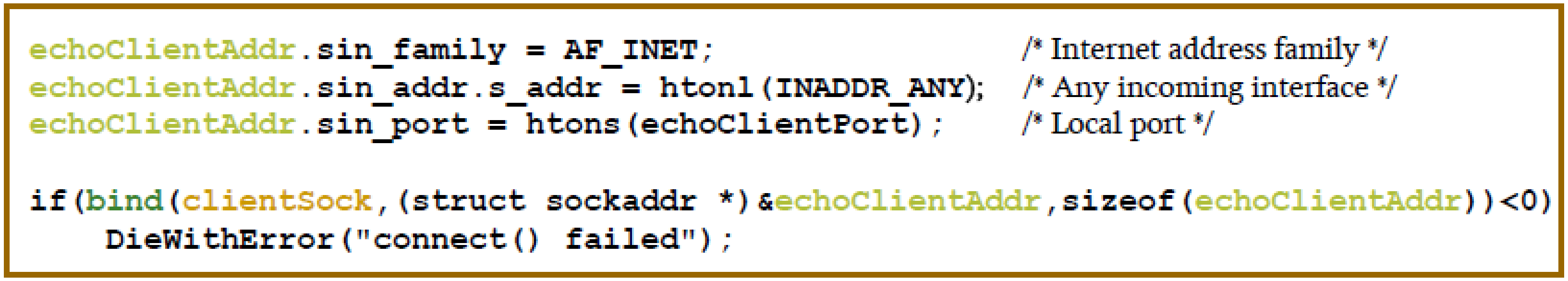
1. **Create a UDP socket**
2. Assign a port to socket
3. Repeatedly
   - Communicate

Example -Echo using Datagram socket



```
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 */

if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("bind() failed");
```



```
echoClientAddr.sin_family = AF_INET;                    /* Internet address family */
echoClientAddr.sin_addr.s_addr = htonl(INADDR_ANY);     /* Any incoming interface */
echoClientAddr.sin_port = htons(echoClientPort);        /* Local port */

if(bind(clientSock,(struct sockaddr *)&echoClientAddr,sizeof(echoClientAddr))<0)
    DieWithError("connect() failed");
```

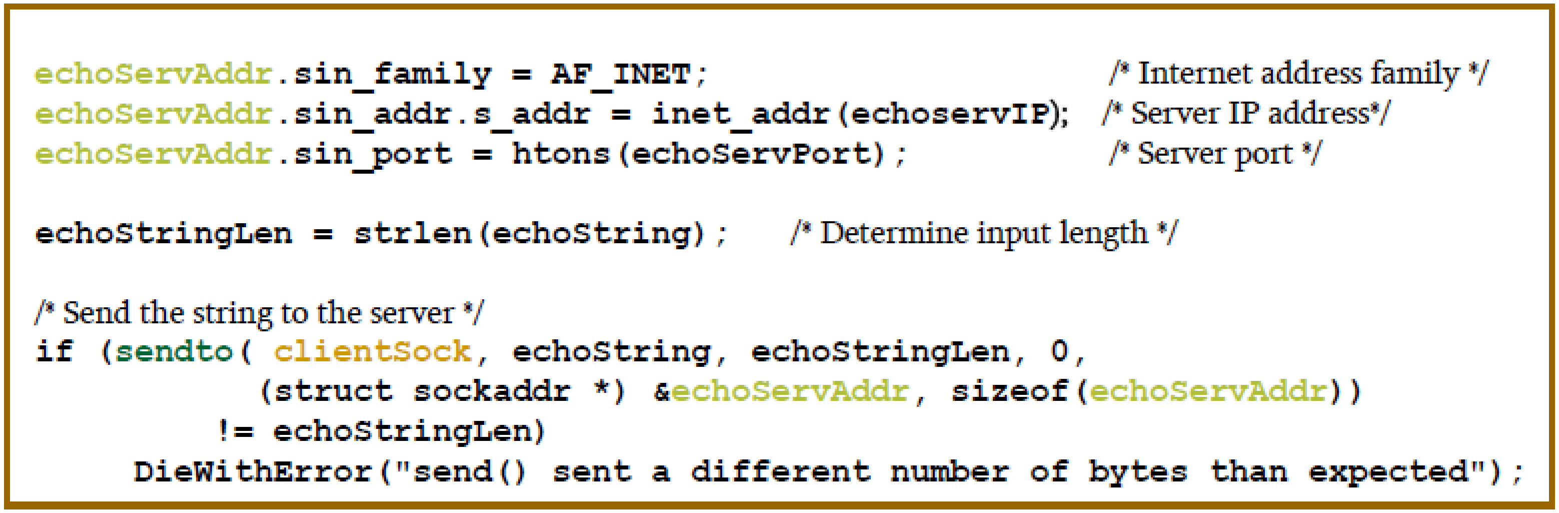
Client

1. Create a UDP socket
2. **Assign a port to socket**
3. Communicate
4. Close the socket

Server

1. Create a UDP socket
2. **Assign a port to socket**
3. Repeatedly
   - Communicate

Example -Echo using Datagram socket



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

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

/* Send the string to the server */
if (sendto( clientSock, echoString, echoStringLen, 0,
           (struct sockaddr *) &echoServAddr, sizeof(echoServAddr))
         != echoStringLen)
     DieWithError("send() sent a different number of bytes than expected");
```

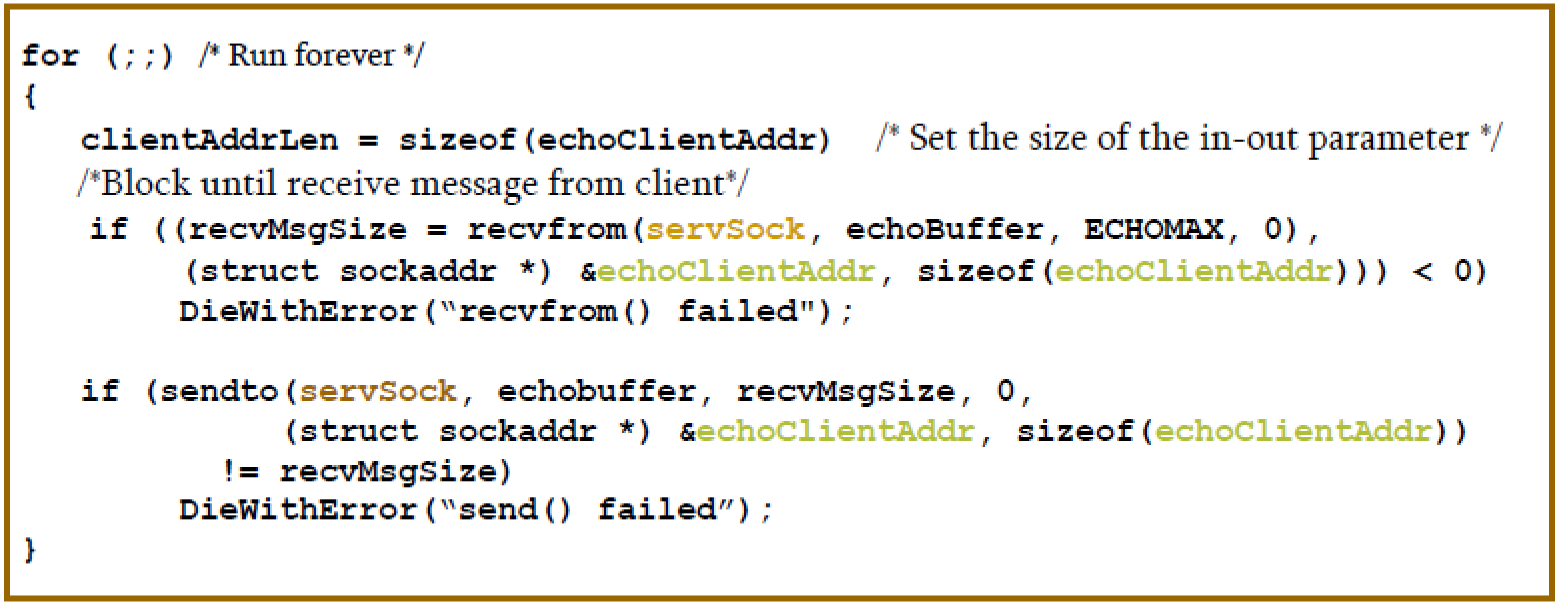
Client

1. Create a UDP socket
2. Assign a port to socket
3. **Communicate**
4. Close the socket

Server

1. Create a UDP socket
2. **Assign a port to socket**
3. Repeatedly
   - Communicate

Example -Echo using Datagram socket



```
for (;;) /* Run forever */
{
   clientAddrLen = sizeof(echoClientAddr)    /* Set the size of the in-out parameter */
   /*Block until receive message from client*/
    if ((recvMsgSize = recvfrom(servSock, echoBuffer, ECHOMAX, 0),
         (struct sockaddr *) &echoClientAddr, sizeof(echoClientAddr))) < 0)
        DieWithError("recvfrom() failed");

   if (sendto(servSock, echobuffer, recvMsgSize, 0,
              (struct sockaddr *) &echoClientAddr, sizeof(echoClientAddr))
          != recvMsgSize)
        DieWithError("send() failed");
}
```

Client

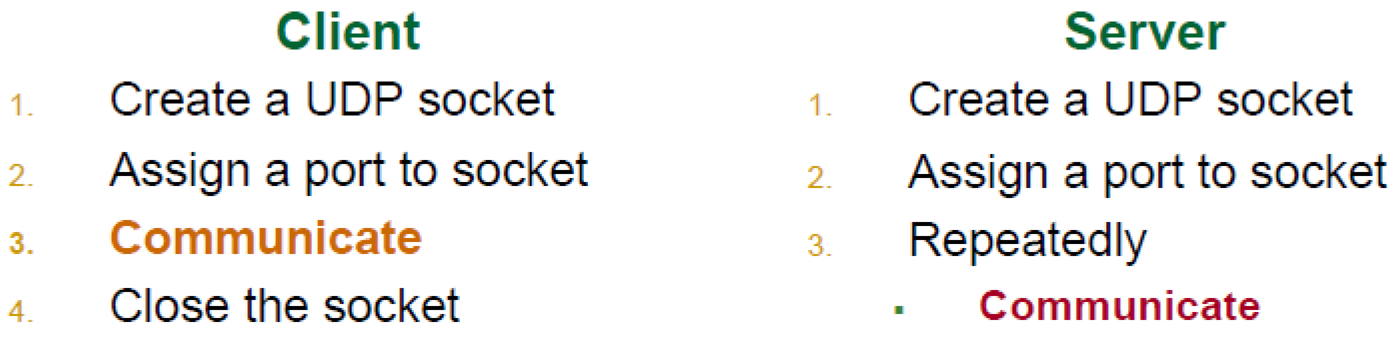
1. Create a UDP socket
2. Assign a port to socket
3. **Communicate**
4. Close the socket

Server

1. Create a UDP socket
2. Assign a port to socket
3. Repeatedly
   - **Communicate**

Example -Echo using Datagram socket

Similarly, the client receives the data from the server



Client

1. Create a UDP socket
2. Assign a port to socket
3. **Communicate**
4. Close the socket

Server

1. Create a UDP socket
2. Assign a port to socket
3. Repeatedly
   - **Communicate**

Example -Echo using Datagram socket

```
close(clientSock);
```

Client

1. Create a UDP socket
2. Assign a port to socket
3. Communicate
4. **Close the socket**

Server

1. Create a UDP socket
2. Assign a port to socket
3. **Repeatedly**
   - Communicate