**Chapter 11: BSD SOCKETS API**

**Topic – 1: Client-Server Model**

**Concept**

* Means **two processes** communicating with each other.
* **Client** connects to **server** for requesting information access.
* We only require **server’s address** to connect, though both can communicate with each other.
* **Socket:** An **end point** of an inter-process communication channel.
* So, both end of a channel can be called as **two different** sockets.
* Two processes can communicate **only** when their sockets are of **same type**.

**Establishing Client Socket**

* **Step 1:** **Create** a socket using ***socket()*** system call.
* **Step 2:** **Connect** socket to **server’s address** using ***connect()***.
* **Step 3:** **Send and receive** data using ***read()*** and ***write()*** system calls respectively.

**Establishing Server Socket**

* **Step 1:** **Create** a socket using ***socket()*** system call.
* **Step 2:** **Bind** socket to an address (**port number** of host’s machine) using ***bind()*** system call.
* **Step 3:** **Listen** (**look**) for connections using ***listen()*** system call.
* **Step 4:** **Accept** a connection using ***accept()*** system call, which **can’t** be called again during the time a client is connecting to the server.
* **Step 5:** **Send & receive** data.

**Topic – 2: Domain**

**Types Of Domains**

* UNIX domain
* Internet domain

**Differences Between Two**

|  |  |
| --- | --- |
| **UNIX Domain** | **Internet Domain** |
| **Two processes present at common file system communicate with each other.** | **Two physically separate processes on internet communicate with each other.** |
| **Socket address is a character array (string), which provides entry to the file system.** | **Socket address consists of IP address of the host machine (32-bits).** |
| **Doesn’t require port number.** | **A port number (16-bits) is required.** |

**Note!**

**🡪 Lower numbers in address (port numbers) are reserved in UNIX for standard services.**

**🡪 Port number for FTP is 21.**

**🡪 Port numbers above 2000 are more likely to be available & not being reserved.**

**Topic – 3: Sockets**

**Socket Types**

* Stream sockets
* Datagram sockets

**Stream Sockets**

* Reads messages character by character.
* Uses **TCP** (transmission control protocol) which is reliable & stream-oriented protocol.

**Datagram Sockets**

* Reads messages all at once.
* Uses **UDP** (UNIX datagram protocol) which is unreliable & message-oriented protocol.

**Topic – 4: Creating Server**

**Basic Information**

* Keyword ***localhost*** is used to refer to our local machine.

**Headers**

***#include <sys/types.h>***

* Contains some special data types we will be using.
* Used in ***sys/socket.h*** & ***netinet/in.h***.

***#include <sys/socket.h>***

* Contains some **socket structures** definitions we will require.

***#include <netinet/in.h>***

* Contains constants & structures for **internet domain address**.

**Main Function**

***int main(int argc, char \*argv[])***

* These parameters are required because we deal with **user arguments**.

**Declaring Important Variables**

***int sockfd, newsockfd, portno, clilen, n;***

***char buffer[256];***

* ***sockfd*** and ***newsockfd*** are file descriptors.
* ***clilen*** is client address size.
* ***n*** is the number or characters that will be read or written using ***read()*** & ***write()***.

**Address Structure**

***struct sockaddr\_in server\_addr, client\_addr;***

* ***sockaddr\_in*** contains the internet address.

**Definition In netinet/in.h**

***struct sockaddr\_in***

***{***

***/\* sin = Socket internet \*/***

***short sin\_family; // Set to AF\_INET (uses IPv4)***

***u\_short sin\_port; // Port number being used***

***struct in\_addr sin\_addr; // Internet address***

***char sin\_zero[8]; // Must be set to zero***

***}***

* Notice that structure ***in\_addr*** is a different structure in structure ***sockaddr\_in***.
* ***in\_addr*** contains only one **unsigned long** called ***s\_addr***.
* Programmer must carefully code to check if all arguments are passed.

**Creating Socket**

***sockfd = socket(AF\_NET, SOCK\_STREAM, 0);***

***if (sockfd < 0) {perror("Problem creating socket");***

* First argument is **socket’s domain address**.
* We could have also used ***AF\_UNIX*** at first argument.
* Second argument is **type of socket**.
* We could have also used ***SOCK\_DGRAM*** at second argument.
* Third argument is the **protocol** being used.
* Here, **0** means OS will decide which protocol to use.
* Otherwise, we can use ***UDP*** or ***TCP***.

**Warning!**

**🡪 It is advised to keep the 3rd argument as 0 until really necessary to specify.**

* Function ***socket()*** returns a **integer** value to **file descriptor** **table**.
* This value is used to **refer** to our socket.
* In case the socket call fails, **-1** is returned.
* However, it is almost impossible for it to fail.

**Flushing The Buffer**

***bzero((char\*) &serv\_addr, sizeof(serv\_addr));***

* First argument is **pointer to the buffer**.
* Second argument is the **size of buffer**.

**Converting Port Number**

***portno = atoi(argv[1]);***

* User passed the argument as a **string**, we converted it to **integer**.

**Server Address Structure**

***serv\_addr.sin\_family = AF\_NET;***

* ***sin\_family*** member of our **server address** structure must always be set to ***AF\_NET***.

***serv\_addr.sin\_port = htons(portno);***

* Function ***htons()*** is used to convert an integer in **host byte number** to **network byte order**.
* This is a compulsory step to perform.
* ***htons()*** stands for **"host to network short"**.

***serv\_addr.sin\_addr.s\_addr = INADDR\_ANY;***

* The unsigned long int ***s\_addr*** contains the **IP address** of the **host**.
* Means the **IP address** of the **server itself**.
* ***INADDR\_ANY*** is the symbolic constant which fetches it.

**Binding Socket**

***if (bind(sockfd, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0)***

***perror("Problem binding the socket.")***

* System call ***bind()*** binds a socket to an **address**.
* This address is **IP address** & **port number** of host system.
* 1st argument is a **file descriptor**, which described each socket separately.
* 2nd argument is **pointer to structure** of type ***sockaddr***.
* This connection **fails** if the socket is already in use.

**Looking For Connection**

***listen(sockfd, 5);***

* 2nd argument is the size of ***backlog queue***.
* **Backlog queue:** Request in queue waiting for connection with client.
* So, ***5*** in code above means **maximum 5 requests** can exist there.

**Accepting Connection Request**

***clilen = sizeof(cli\_addr);***

***newsockfd = accept(sockfd, (struct sockaddr \*)&cli\_addr, &clilen);***

***if (newsockfd < 0)***

***perror("Error ACCEPTING connection!");***

* System call ***accept()*** accepts a connection request & is **blocked** to other requests while accepting connection request.
* It returns a **new file descriptor** which must be used to communicate with it in future.
* 2nd argument is **pointer to the address of client**.
* Note that 3rd argument is passed **by reference**.

**Initializing Buffer**

***bzero(buffer, 256);***

***n = read(newsockfd, buffer, 255); # Reads from socket***

***if (n<0)***

***perror("Error reading from socket!");***

***printf("Buffer message: %s", buffer);***

* The code above is only valid if our connection is successfully established.
* Note that it uses the new file descriptor ***newsockfd*** returned by ***accept()***.
* ***read()*** reads when client has executed ***write()*** system call.
* And while reading, other ***write()*** signals are **blocked**.

**Writing To Socket**

***n = write(newsockfd, "I got your message", 18);***

***if (n<0)***

***perror("Error writing to the socket!");***

***exit(EXIT\_FAILURE);***

* **Both** **ends** can read & write to the connection.
* 3rd argument (***18***) is the **size of the message**.

**Topic – 5: Client-Side Programming**

**Headers**

***#include <stdio.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <netdb.h>***

* ***netdb.h*** defines structure of ***hostent*** which we will be using soon.

**Main Function**

***int main(int argc, char \*argv[])***

***{***

***int sockfd, portno, n;***

***struct sockaddr\_in serv\_addr;***

***struct hostent \*server;***

***// Rest of the code…***

* ***serv\_addr*** will contain the address of the server we want to connect to.
* Variable ***server*** is a pointer to a **structure** of type ***hostent***.

**hostent Definition**

***struct hostent***

***{***

***char \*h\_name; // “Official” name of host***

***char \*\*h\_aliases; // 0 terminated array of aliases***

***int h\_addrtype; // Mostly AF\_NET***

***int h\_length; // In bytes***

***char \*\*h\_addr\_list; // Pointer to list***

***#define h\_addr h\_addr\_list[0]***

***};***

* ***hostent*** is for defining host computer on the internet.
* The last line is for **backward compatibility**.
* Means if someone uses ***h\_addr*** which is actually an **old** macro, it works same as ***h\_addr\_list[0]***, pointing to first address on list.
* These are returned in **network byte** order.

**Common Code**

* This code is present in **both** server & client program.

***char buffer[256];***

***if (argc < 3)***

***{***

***fprintf(stderr, "Enter portno for host %s", argv[0]);***

***exit(EXIT\_FAILURE);***

***}***

***portno = atoi(argv[2]);***

***sockfd = socket(AF\_NET, SOCK\_STREAM, 0);***

***if (sockfd < 0)***

***perror("Error opening socket!");***

**Confirming Host Name**

***server = gethostbyname(argv[1]);***

***if (server==NULL)***

***{***

***fprintf(stderr, "There is no host named %s", server);***

***exit(EXIT\_FAILURE);***

***}***

**Function gethostbyname Declaration**

***struct hostent \*gethostbyname(char \*name);***

* This returns information **related to host**.
* Field ***char \*name*** contains the **IP address**.
* Earlier, function ***gethostbyname()*** was used for finding **file systems** online.
* Now it is used for **querying** large **databases** around the country.

**Converting Port Number**

***bzero((char \*) &serv\_addr, sizeof(serv\_addr));  
serv\_addr.sin\_family = AF\_INET;***

***bcopy((char \*)server->h\_addr, // IP address  
 (char \*)&serv\_addr.sin\_addr.s\_addr, // Integer storing IP  
 server->h\_length); // Host address length***

***serv\_addr.sin\_port = htons(portno);***

* Function ***bcopy()*** copies **IP address** from buffer to our structure member ***s\_addr***.

**Function bcopy Definition**

***void bcopy(char \*s1, char \*s2, int length);***

**Connecting To Client Socket**

***if (connect(sockfd,&serv\_addr,sizeof(serv\_addr)) < 0)***

***perror("ERROR connecting");***

* This function **returns 0** if connected successfully, **else -1** if it fails.
* Client requires **port number** for the server but **not** for itself because that is **automatically assigned** to one during connection.

**Remaining Code**

***printf("Please enter the message: ");***

***bzero(buffer, 256); fgets(buffer, 255, stdin);***

***n = write(sockfd, buffer, strlen(buffer));***

***if (n < 0)***

***perror("ERROR writing to socket");***

***bzero(buffer, 256);***

***n = read(sockfd, buffer, 255);***

***if (n < 0)***

***perror("ERROR reading from socket");***

***printf("%s", buffer);***

***return 0;  
}***

**Topic – 6: Server Code Enhancement**

**Introduction**

* The server we created runs & dies after **running** **once**.
* Real world server must run **indefinitely** without dying & must be able to handle **multiple connections**.
* We can do this by **forking off** a **new** **process** to handle each connection.
* Let’s consider a dummy function ***dostuff(int sockfd)*** for doing this.

**Indefinitely Running Server**

* **Step 1:** Make an **infinite loop**.
* **Step 2:** **Establish** a connection.
* **Step 3:** Call ***fork()*** to create new process.
* **Step 4:** Call ***dostuff()*** to establish a new connection.
* **Step 5:** **Close** the child process if required, the new connection will **still remain**.

**The Zombie Problem**

* The code above might create ***zombie problem***.
* If a lot of connections are accepted, then these connections create a **zombie** when terminated.
* **Zombie process:** A process which **doesn’t** fully die.
* Parent process **may not** let their child die & rather put them into **wait** **state**.
* These processes in **wait state** clog the process table.
* But the code preventing zombie processes **varies** across architectures.
* A dying child sends a **SIGCHLD** signal to its parent.

***signal(SIGCHLD, SIG\_IGN); // AIX systems***

***signal(SIGCHLD, wait3(NULL,WNOHANG,NULL)); // SunOS***

**Topic – 7: Alternatives**

**Introduction**

* What we saw earlier was a **stream socket**.
* One major difference between **stream socket** & **datagram socket** is that **datagram socket** is used when **only one** message has to be sent from client to server or vice-versa.

**Stream Sockets v/s Datagram Sockets**

|  |  |
| --- | --- |
| **Datagram Sockets** | **Stream Sockets** |
| **Unreliable, as it uses UDP protocol which has absence of acknowledgement system.** | **Reliable as it has TCP protocol which detects lost packets through acknowledgement system & these messages are retransmitted.** |
| **Data has to reach the receiver all at once.** | **Receiver can read it in chunks of byte(s).** |
| **Uses sendto() & receivefrom() system calls.** | **Uses read() & write() system calls.** |
| **Less overhead due to no need of setting up & then disbanding connections, plus packets need not to be acknowledged.** | **High overhead due to various factors.** |
| **Used where short-term service is required.** | **Used where long-term service is required.** |

**Differences In Code**

***sock=socket(AF\_INET, SOCK\_DGRAM, 0);***

***fromlen = sizeof(struct sockaddr\_in);***

***while (1)***

***{***

***n = recvfrom(sock,buf,1024,0,(struct sockaddr \*)&from,&fromlen);***

***if (n < 0)***

***error(“recvfrom”);***

* ***recvfrom()*** is used to **read messages**.
* It will **block** other system calls until the message is received.
* First three arguments are **same** as in **SOCK\_STREAM**.
* **4th argument** is **integer** **for flags**, **0** by default.
* **5th** one is a **pointer** to ***sockaddr*** in structure.
* **6th** one is the **size required** for **structure** in **5th argument**, in number of bytes.
* The **6th argument** however returns **-1** instead if any **error** is caused.

**Differences In Code – II**

***n = sendto(sock, "Got your message", 17, 0, (struct sockaddr \*) &from,fromlen);***

***if (n < 0)***

***error("sendto");***

***}  
}***

* In ***sendto()***, first **three** arguments are **same** as it was in ***write()***.
* **3rd** argument is **number of bytes** to write.
* **4th** argument is **flag**, set to zero by default.
* **5th** argument is **pointer** to structure ***sockadd\_in***.
* **6th** (last) argument is **size** of the **structure**.
* We have to **retrieve** the values from the structure through the ***recvfrom()*** call.
* Notice the differences that there is no ***connect()*** call.

**Topic – 8: Sockets In The UNIX Domain**

***struct sockaddr\_un***

***{***

***short sun\_family; // AF\_UNIX***

***char sun\_path[108]; // path name (gag)***

***};***

* ***sun\_path*** has the **path name** in UNIX file system.
* So, the client & server must be running on **same** file system.
* A socket always remains until it is **manually** deleted.
* **Named pipes:** IPC supporting transfer of data using **FIFO** principle.

**Topic – 9: Designing Servers**

**Concurrent & Connection Oriented Servers**

* In general servers, a **stream socket** is created & a **new process** is forked each time to handle a connection when it is established.
* This model is good for **telnet** or **FTP** servers.
* **TCP** protocol is general has **high overhead** on kernel.
* Though having high overhead, the data transmission is **reliable**.

**Iterative & Connectionless Servers**

* Servers with single message to be delivered to a client **doesn’t** involve forking.
* These include ***finger daemon***, ***timeofday*** or an ***echo*** server.
* **Finger daemon:** A server model used earlier to retrieve **basic user information** from remote servers such as name, email etc.
* **Timeofday:** A server model used to **synchronize time** on computers connected over same server.
* **Echo:** A server that just **echoes** the message sent by the client.
* These methods have **less overhead** but are **unrealiable**.

**Single Process Concurrent Servers**

* Here only **one process** is required.
* The server listens for connections & replies them quickly before breaking it and listening for another connection.