Interprocess Communication
Dr. Xiaobo Zhou

Middleware Layers

- Applications, services
- RMI and RPC
- Request-reply protocol
- Marshalling and external data representation
- UDP and TCP

- RMI and RPC: integrating communication into a programming language paradigm
- Request-reply: message passing communication between processes and support for communication between objects
Interprocess Communication (IPC)

- **Lack of shared memory** → **Communication by sending and receiving messages**

- **Message passing:**
  - Process interaction by sending and receiving message operations

- **Remote procedure call (RPC):**
  - Process interaction at language level by procedure interaction

- **Remote method invocation (RMI):**
  - Process interaction at language level by object invocation

### Blocking vs. Non-Blocking Primitives

<table>
<thead>
<tr>
<th></th>
<th>blocking</th>
<th>non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>send</td>
<td>returns control to user only after message has been sent, or until acknowledgement has been received</td>
<td>returns control to user as soon as message is buffered</td>
</tr>
<tr>
<td>receive</td>
<td>returns only after message has been received</td>
<td>returns after signaling willingness to receive.</td>
</tr>
<tr>
<td>problems</td>
<td>reduces concurrency</td>
<td>need buffering; more complicated</td>
</tr>
</tbody>
</table>
Synchronous and Asynchronous Communication

- **Synchronous communication:**
  - Sending and receiving processes synchronize at every message
  - Both send and receive are blocking operations

- **Asynchronous communication:**
  - Messages need to be buffered
  - Send is non-blocking, and receive can be blocking or non-blocking
  - Non-blocking receive is unusual, which involves extra complexity and requires to separately receive notification that its buffer has been filled by polling or interrupt

Reliable vs. Unreliable Primitives

- **Transmission problems:**
  - Loss, duplication, ordering, corruption

- **Reliable transmission:**
  - Use acknowledgements for recovery
Message Destinations – Sockets and ports

- **Internet address, local port**
  - **Port:** a message destination within a computer, an 16-bit integer
  - **Socket:** an endpoint for communication between processes
  - **One port has one receiver (process), but can have many senders**
  - **One process can use multiple ports**

Locality Transparency

- **Service re-location**
  - Translate service name into server locations at run time; allowing services to be relocated but not to migrate

- **Service migration**
  - **Mach OS:** provides location-independent identifiers for message destinations, mapping them onto lower-level address in order to deliver message to ports
UDP Datagram Communication

UDP: The delivery of the message is not guaranteed
- Message size: up to $2^{16}$ B (usual restriction 8 KB)
- Non-blocking `send` and blocking `receive`
- Timeout: to avoid infinite wait of blocking `receive`
- Receive from any
- Ordering: messages can be delivered out of sender order
- Omission failures: send-omission, receive-omission, channel-omission

TCP Stream Communication

TCP: The delivery of the message is “guaranteed”;
- Message size: unlimited; hidden by the stream abstraction
- Loss: an acknowledgement scheme; timeout and retransmission
- Flow control: to match the speeds of the processes that read and write to a stream; if writing too fast, blocked
- Duplication and ordering: message identifiers associated with each packet; reordering provided
- Message destinations: explicit after connection is built
TCP Stream Communication Issues

Matching of data items: two ends agree on the contents (order) of the data; e.g., `int + double`

Blocking: reading blocks until data becomes available; TCP flow control protocol will block a writing process if reading is too slow

Threads: when accepting a connection, it generally creates a new thread with a socket to communicate to the new client; more concurrency; no thread? `select()` in Unix

UDP vs. TCP

Failure models:
- Packet loss
- Packet ordering

Overhead:
- Store state information at source and destination
- Transmission of connection messages
- Flow control (redundant if passing only small arguments and results)
- Acknowledgement (redundant is requests followed by replies)

Use of UDP/TCP
- Telnet, FTP, HTTP, RPC
External Data Representation and Marshalling

- **Heterogeneity:**
  - Computers store primitive values such as integers & FP in different orders, big-endian vs. little-endian
  - Different coding schemes for character representation, ASCII vs. Unicode

- **Who does marshalling and unmarshalling?**
  - Middleware for transparency, and efficiency

**CORBA’s CDR**

- **CDR: Common Data Representation**
  - Concerned with structures and primitive data types
  - Does not deal with Objects (Java Object Serialization does)
  - Requires a IDL (interface description language) to specify the order and the data items in a message
  - Values are transmitted in sender’s ordering, translated by receiver

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>length (unsigned long) followed by elements in order</td>
</tr>
<tr>
<td>string</td>
<td>length (unsigned long) followed by characters in order (can also have wide characters)</td>
</tr>
<tr>
<td>array</td>
<td>array elements in order (no length specified because it is fixed)</td>
</tr>
<tr>
<td>struct</td>
<td>in the order of declaration of the components</td>
</tr>
<tr>
<td>enumerated</td>
<td>unsigned long (the values are specified by the order declared)</td>
</tr>
<tr>
<td>union</td>
<td>type tag followed by the selected member</td>
</tr>
</tbody>
</table>

**CORBA CDR for constructed types**
A CORBA CDR Message

<table>
<thead>
<tr>
<th>index in sequence of bytes</th>
<th>4 bytes</th>
<th>notes on representation</th>
<th>length of string</th>
<th>'Smith'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4–7</td>
<td>&quot;Smit&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8–11</td>
<td>&quot;h_&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12–15</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16–19</td>
<td>&quot;Lond&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20–23</td>
<td>&quot;on &quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24–27</td>
<td>1934</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The flattened form represents a Person struct with value: {'Smith', 'London', 1934}

Struct Person {
    string name;
    string place;
    unsigned long year;
} // defined by CORBA IDL, known to both the sender and receiver

Extensible markup language (XML)

※ A markup language that was defined by WWW Consortium for general use on the Web.
Client-Server Communication

Operations of the Request-Reply Protocol

- Request-reply communication (one-to-one):
  - To support message exchanges in typical client-server interactions
  - Synchronous vs. asynchronous communication
  - Send and receive operations in UDP are lightweight
    - But TCP does not have limitations over the buffer size which UDP does

- HTTP is an example of a request-reply protocol
Request-Reply Message Structure

| messageType | int (0=Request, 1=Reply) |
| requestID | int |
| objectReference | RemoteObjectRef |
| methodID | int or Method |
| arguments | array of bytes |

Remote object references:

A unique identifier for a remote object in a distributed system

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet address</td>
<td>port number</td>
<td>time</td>
<td>object number</td>
</tr>
</tbody>
</table>

interface of remote object

How about failures?

- Timeout
- Discarding duplicate request messages
- Lost reply messages: idempotent
- History: retransmission vs. re-execution

Class RemoteObjectReference

Define a class whose instances represent remote object references. It should provide access methods needed by the request-reply protocol. Explain how each of the access methods will be used by that protocol.

class RemoteObjectReference{
    private InetAddress ipAddress;
    private int port;
    private int time;
    private int objectNumber;
    private Class interface;
    public InetAddress getIpAddress() { return ipAddress; }
    public int getPort() { return port; }
}

The server looks up the client port and IP address before sending a reply. The variable interface is used to recognize the class of a remote object when the reference is passed as an argument or result.
Communication Schemes

- One-to-one unicast
- One-to-one multicast
- Many-to-one
- Many-to-many

Group Communication

**Multicast (one-to-many):**
- Sends a single message from a process to each of the members of a group of processes; usually transparent group membership
- Failures and ordering are more complicated than one-to-one

**Case 1: fault tolerance**
- Client
- Replicated services

**Case 2: resource location**
- Client
- Spontaneous network
- Multicast address allocation
- Multicast router
- SetTimeToLive (TTL)

**Case 3: update replicas**
- Server
- Replicated servers

**Case 4: event notification**
- Server
- Group users
IPC in UNIX: UDP Sockets

- **IPCs primitives provided as system calls in BSD 4.x**
  - A socket call returns a descriptor (identifier)
  - **Binding**: bind the descriptor to a socket address before communication
    - Java API: socket creation and name binding are integrated

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)  
  bind(s, ClientAddress, size of CA)  
  sendto(s, "message", ServerAddress)  
```

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)  
  bind(s, ServerAddress, size of CA)  
  amount = recvfrom(s, buffer, from)  
```

- **ServerAddress** and **ClientAddress** are socket addresses (**struct sockaddr_in sa**)
- sendto() returns the actual number of bytes sent
- recvfrom() blocks if the queue is empty until a message arrives

IPC in UNIX: TCP Sockets

- **Establishing a connection before communication**
  - Call waiting in TCP: listen()
  - **Binding**: bind the descriptor to a socket address before communication
    - Java API: socket creation and name binding are integrated
  - A new socket will be created when a connection is accepted by the server

Requesting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)  
  bind(s, ServerAddress, size of SA)  
  connect(s, ServerAddress, size of SA)  
  write(s, "message", length)  
```

Listening and accepting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)  
  bind(s, ServerAddress, 0)  
  listen(s,5);  
  sNew = accept(s, ClientAddress, sizeofSA);  
  n = read(sNew, buffer, amount)  
```

**ServerAddress** and **ClientAddress** are socket addresses
### IPC in Unix BSD: Socket Primitives for TCP/IP

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>Create a new communication endpoint</td>
</tr>
<tr>
<td>Bind</td>
<td>Attach a local address to a socket</td>
</tr>
<tr>
<td>Listen</td>
<td>Announce willingness to accept connections</td>
</tr>
<tr>
<td>Accept</td>
<td>Block caller until a connection request arrives</td>
</tr>
<tr>
<td>Connect</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>Send</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>Receive</td>
<td>Receive some data over the connection</td>
</tr>
<tr>
<td>Close</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>

### BSD UDP Socket Communication Pattern

Connection-oriented communication pattern using sockets.
Example: UDP Echo Server

```
/* Echo server using UDP */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

/* Bind an address to the socket */
bzero((char *)&server, sizeof(server));
server.sin_family = AF_INET;
server.sin_port = htons(port);
server.sin_addr.s_addr = htonl(INADDR_ANY);

#define SERVER_UDP_PORT 5000
#define MAXLEN 4096

int main(int argc, char **argv)
{
    int sd, client_len, port, n;
    char buf[MAXLEN];
    struct sockaddr_in server, client;
    switch(argc) {
    case 1:
        port = SERVER_UDP_PORT;
        break;
    case 2:
        port = atoi(argv[1]);
        break;
    default:
        fprintf(stderr, "Usage: %s [port]\n", argv[0]);
        exit(1);
    }

    /* Create a datagram socket */
    if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket\n");
        exit(1);
    }

    /* Bind an address to the socket */
    if (bind(sd, (struct sockaddr *)&server, sizeof(server)) == -1) {
        fprintf(stderr, "Can't bind name to socket\n");
        exit(1);
    }

    while (1) {
        client_len = sizeof(client);
        if ((n = recvfrom(sd, buf, MAXLEN, 0,
            (struct sockaddr *)&client, &client_len)) < 0) {
            fprintf(stderr, "Can't receive datagram\n");
            exit(1);
        }
        if (sendto(sd, buf, n, 0,
            (struct sockaddr *)&client, client_len) != n) {
            fprintf(stderr, "Can't send datagram\n");
            exit(1);
        }
    }
    exit(1);
}
```

Example: UDP Echo Client

```
#define SERVER_UDP_PORT         5000
#define MAXLEN                  4096
#define DEFLEN                  64

long delay(struct timeval t1, struct timeval t2)
{
    long d;
    d = (t2.tv_sec - t1.tv_sec) * 1000;
    d += ((t2.tv_usec - t1.tv_usec + 500) / 1000);
    return(d);
}

int main(int argc, char **argv)
{
    int     data_size = DEFLEN, port = SERVER_UDP_PORT;
    int     i, j, sd, server_len;
    char    *pname, *host, rbuf[MAXLEN], sbuf[MAXLEN];
    struct  hostent         *hp;
    struct  sockaddr_in     server;
    struct  timeval         start, end;
    unsigned long address;
    pname = argv[0];
    argc--; 
    argv++; 
    if (argc > 0 && (strcmp(*argv, "-s") == 0)) {
        if (--argc > 0 && (data_size = atoi(*++argv))) {
            argc--; 
            argv++; 
        } 
        else { 
            fprintf(stderr,
"Usage: %s [-s data_size] host [port]\n", pname);
            exit(1);
        }
    }

    if (argc > 0) {
        host = *argv;
        if (--argc > 0)
            port = atoi(*++argv);
    }
    gethostname(host, 150); 
    if (argc > 0) {
        if (--argc > 0)
            port = atoi(*++argv);
    }

    gettimeofday(&start, NULL); /* start delay measurement */
    server_len = sizeof(server);
    if (sendto(sd, sbuf, data_size, 0, (struct sockaddr *)
        &server, server_len) == -1) {
        fprintf(stderr, "sendto error\n");
        exit(1);
    }

    if (recvfrom(sd, rbuf, MAXLEN, 0, (struct sockaddr *)
        &server, &server_len) < 0) {
        fprintf(stderr, "recvfrom error\n");
        exit(1);
    }

    gettimeofday(&end, NULL); /* end delay measurement */
    if (strncmp(sbuf, rbuf, data_size) != 0)
        printf("Data is corrupted\n");
    close(sd);
    return(0);
}
```
UDP Client in Java

```java
import java.net.*;
import java.io.*;

public class UDPClient {
    public static void main(String args[]) {
        // args give message contents and server DNS hostname
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket(); // client can choose a free local port
            byte[] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]); // API for encp. Internet address
            int serverPort = 6789; // server port, known to clients
            DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
            aSocket.send(request); // request is the instance of class DatagramPacket
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply); // receive does not specify an origin for messages
            System.out.println("Reply: " + new String(reply.getData()));
            // getPort(), getAddress access the port and IP address
        } catch (SocketException e) {System.out.println("Socket: " + e.getMessage());} // port in use or reserved
        catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        finally {if(aSocket != null) aSocket.close();}
    }
}
```

1/14/2008

Adapted from Distributed Systems: Concepts and Design  Edn. 4
© Addison-Wesley Publishers 2005

UDP Server in Java

```java
import java.net.*;
import java.io.*;

public class UDPServer {
    public static void main(String args[]) {
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket(6789); // 6789 is the fixed receiving port
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        } catch (SocketException e) {System.out.println("Socket: " + e.getMessage());}
        catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        finally {if(aSocket != null) aSocket.close();}
    }
}
```

1/14/2008

Adapted from Distributed Systems: Concepts and Design  Edn. 4
© Addison-Wesley Publishers 2005
BSD TCP Socket Communication Pattern

Connection-oriented communication pattern using sockets.

Example: TCP Echo Server

/* A simple echo server using TCP */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER TCP PORT
#define BUFLEN 256
int main(int argc, char **argv)
{
    int n, bytes_to_read;
    int sd, new_sd, client_len, port;
    struct sockaddr_in server, client;
    char *bp, buf[BUFLEN];
    switch(argc) {
    case 1:
        port = SERVER_TCP_PORT;
        break;
    case 2:
        port = atoi(argv[1]);
        break;
    default:
        fprintf(stderr, "Usage: %s \[port\]\n", argv[0]);
        exit(1);
    }
    /* queue up to 5 connect requests */
    listen(sd, 5);
    while (1) {
        client_len = sizeof(client);
        if ((new_sd = accept(sd, (struct sockaddr *)&client, &client_len)) == -1) {
            fprintf(stderr, "Can't accept client\n");
            exit(1);
        }
        bp = buf;
        bytes_to_read = BUFLEN;
        while ((n = read(new_sd, bp, bytes_to_read)) > 0) {
            bp += n;
            bytes_to_read -= n;
        }
        printf("Rec'd: %s
", buf);
        write(new_sd, buf, BUFLEN);
        printf("Sent: %s
", buf);
        close(new_sd);
    }
    close(sd);
    return(0);
}
Example: TCP Echo Client

```c
/* A simple TCP client */
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_TCP_PORT 3000
#define BUFLEN 256

int main(int argc, char **argv) {
    int n, bytes_to_read;
    int sd, port;
    struct hostent *hp;
    struct sockaddr_in server;
    char *host, *bp, rbuf[BUFLEN], sbuf[BUFLEN];
    switch(argc) {
    case 2:
        host = argv[1];
        port = SERVER_TCP_PORT;
        break;
    case 3:
        host = argv[1];
        port = atoi(argv[2]);
        break;
    default:
        fprintf(stderr, "Usage: %s host [port]\n", argv[0]);
        exit(1);
    }

    /* Create a stream socket */
    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket\n");
        exit(1);
    }

    /* Connecting to the server */
    if (connect(sd, (struct sockaddr *)&server, sizeof(server)) == -1) {
        fprintf(stderr, "Can't connect\n");
        exit(1);
    }

    printf("Connected: server's address is %s\n", hp->h_name);
    printf("Transmit:\n");
    gets(sbuf); write(sd, sbuf, BUFLEN);
    printf("Receive:\n");
    bp = rbuf;
    bytes_to_read = BUFLEN;
    while ((n = read(sd, bp, bytes_to_read)) > 0) {
        bp += n;
        bytes_to_read -= n;
    }
    printf("%s\n", rbuf);
    close(sd);
    return(0);
}
```

TCP Client in Java

```java
import java.net.*; // makes connection to server, and
import java.io.*; // sends request and receives reply
public class TCPClient {
    public static void main(String args[]) {
        // arguments supply message and DNS hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort); // DNS hostname, not IP address used;
            // the constructor connects to the server
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out = new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]); // UTF is a string encoding; see Section 4.3
            String data = in.readUTF(); // receives the reply of the previous sending
            System.out.println("Received: " + data);
        } catch (UnknownHostException e) {
            System.out.println("Sock: " + e.getMessage());
        } catch (EOFException e) {
            System.out.println("EOF: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("IO: " + e.getMessage());
        } finally {
            if (s != null) try { s.close(); } catch (IOException e) {
                System.out.println("close: " + e.getMessage());
            }
        }
    }
}
```

Adapted from Distributed Systems: Concepts and Design 4th Edition © 2005 Addison-Wesley
TCP Server in Java

```java
import java.net.*;
import java.io.*;
public class TCPServer { // makes a connection for each client
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            } // a new thread & socket will be created after a connection is built
        } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
    // this figure continues on the next slide
}
```

TCP Server in Java Continued

```java
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
            clientSocket = aClientSocket; // a new stream socket is created actually
            in = new DataInputStream( clientSocket.getInputStream());
            out =new DataOutputStream( clientSocket.getOutputStream());
            this.start();
        } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
    }
    public void run(){
        try {
            // an echo server which echoes the client’s request
            String data = in.readUTF();
            out.writeUTF(data);
            } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());}
            } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
                } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}
        }
    }
```
TCP Server Step-by-Step: Socket Establishment

```c
int establish(unsigned short portnum)
{
    char myname[MAXHOSTNAME+1];
    int s;
    struct sockaddr_in sa;
    struct hostent *hp;

    memset(&sa, 0, sizeof(struct sockaddr_in)); /* clear our address */
    gethostname(myname, MAXHOSTNAME); /* who are we? */
    hp = gethostbyname(myname); /* get our address info */
    if (hp == NULL) /* we don't exist!? */
        return(-1);
    sa.sin_family = hp->h_addrtype; /* this is our host address */
    sa.sin_port = htons(portnum); /* this is our port number */
    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) /* create socket */
        return(-1);
    if (bind(s, (struct sockaddr *)&sa, sizeof(struct sockaddr_in)) < 0)
        close(s);
    return(-1); /* bind address to socket */
}
```

TCP Server Step-by-Step: Connection Accept

```c
/* wait for a connection to occur on a socket created with establish() */
int get_connection(int s)
{
    int t; /* socket of connection */

    if ((t = accept(s, NULL, NULL)) < 0) /* accept connection if there is one */
        return(-1);
    return(t);
}
```
TCP Client Step-by-Step: Socket Est. and Connection

```c
int call_socket (char *hostname, unsigned short portnum) // how to call a server socket
{
    struct sockaddr_in sa;
    struct hostent *hp;
    int a, s;
    if ((hp = gethostbyname (hostname)) == NULL) { /* do we know the host's */
        errno= ECONNREFUSED; /* address? */
        return(-1); /* no */
    }
    memset (&sa,0,sizeof(sa)); /* clear our address */
    memcpy ((char *)&sa.sin_addr, hp->h_addr, hp->h_length); /* set address */
    sa.sin_family= hp->h_addrtype;
    sa.sin_port= htons ((u_short) portnum); // host byte order to network byte order
    if ((s= socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0) /* get socket */
        return(-1);
    if (connect (s, (struct sockaddr *)&sa, sizeof sa) < 0) { /* connect (dialing) */
        close(s);
        return(-1);
    }
    return(s);
}
```

TCP Server Step-by-Step: Multiple Connections

```c
#include <errno.h> // you may still accept calls while processing previous connections.
#include <stdio.h> // For this reason you usually fork off child jobs to handle each connection.
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <netdb.h>

PORTNUM 50000 /* random port number, we need something */
void fireman(void);
void do_something(int);

main()
{
    int s, t;
    if ((s= establish (PORTNUM)) < 0) { /* plug in the phone */
        perror("establish");
        exit(1);
    }
    /* as children die we should get catch their returns or else we get zombies, A Bad Thing.
    fireman() catches falling children. */
    signal (SIGCHLD, fireman); /* this eliminates zombies */
}
```
TCP Server Step-by-Step: Multiple Connections (cont.)

for (;;) { /* loop for phone calls */
    if (get_connection(s) < 0) { /* get a connection */
        if (errno == EINTR) /* EINTR might happen on accept(), */
            continue; /* try again */
        perror("accept"); /* bad */
        exit(1);
    }
    switch (fork()) { /* try to handle connection */
        case -1 : /* bad news. scream and die */
            perror("fork");
            close(s); close(t); exit(1);
        case 0 : /* we're the child, do something */
            close(s);
            do_something(t);
            exit(0);
        default : /* we're the parent so look for */
            close(t); /* another connection */
            continue;
    }
}
/* main() ends */

void fireman(void) { while (waitpid(-1, NULL, WNOHANG) > 0) ; }
/* this is the function that plays with the socket. it will be called after getting a connection. */
void do_something(int s) { /* do your thing with the socket here : : */

How to Talk between Sockets

int read_data (int s, /* connected socket */
    char *buf, /* pointer to the buffer */
    int n) /* number of characters (bytes) we want */
{
    int bcount; /* counts bytes read */
    int br; /* bytes read this pass */

    bcount = 0;
    br = 0;
    while (bcount < n) { /* loop until full buffer */
        if (((br = read (s, buf, n - bcount)) > 0) {
            bcount += br; /* increment byte counter */
            buf += br; /* move buffer ptr for next read */
        } else if (br < 0) /* signal an error to the caller */
            return(-1);
    }
    return (bcount);
}