TCP Server

Server pattern: create server socket, bind and listen on that socket. Then in a loop, accept a connection and process the accepted connection. Later we’ll look at using the fork() system call in this loop to handle multiple connected clients at the same time.
if (argc != 2) /* Test for correct number of arguments */
{
    fprintf(stderr, "Usage: %s <Server Port>\n", argv[0]);
    exit(1);
}

echoServPort = atoi(argv[1]); /* First arg: local port */
Bind and Listen

/* Bind to the local address */
if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("bind() failed");

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

for (;;) /* Run forever */
{
    /* Set the size of the in-out parameter */
    clntLen = sizeof(echoClntAddr);
    /* Wait for a client to connect */
    if ((clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr, &clntLen)) < 0)
        DieWithError("accept() failed");
    /* clntSock is connected to a client! */
    printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));
    HandleTCPClient(clntSock);
}
/* NOT REACHED */
Handle one client connection

```c
#include <stdio.h>  /* for printf() and fprintf() */
#include <sys/socket.h>  /* for recv() and send() */
#include <unistd.h>  /* for close() */
#define RCVBUFSIZE 32 /* Size of receive buffer */

void DieWithError(char *errorMessage); /* Error handling function */

void HandleTCPClient(int clntSocket)
{
  local declarations … repeatedly (receive request, send reply) until no more requests …
}

close(clntSocket); /* Close client socket */
```

Pattern for processing a single client: after accepting a connection, (receive request, send reply)* until the receive returns 0.
Repeatedly: receive request and send reply

```c
/* 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 */
{
    /* Echo message back to client */
    if (send(clntSocket, echoBuffer, recvMsgSize, 0) != recvMsgSize)
        DieWithError("send() failed");
    /* See if there is more data to receive */
    if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
        DieWithError("recv() failed");
}
```

Note nested loop: the outer loop is in the calling procedure and contains repeated calls to accept(). It handles multiple client connections. The inner loop, here, contains repeated calls to recv() and send(). It handles multiple sequential interactions for a single client connection.

There is slightly more to the 1st project than meets the eye

- What is all this about sending the time from the server in "network byte order?"
- How does the server know when it has received a full line’s worth of data?
- Are the buffers big enough to contain lines typed at the terminal?