#program to demonstrate concurrent tcp server?

```
   #include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
main()
{
int sockfd, newsockfd ; /* Socket descriptors */
int clilen;
struct sockaddr_in cli_addr, serv_addr;
int i;
char buf[100];
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
printf("Cannot create socket\n");
exit(0);
}
serv_addr.sin_family
= AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port
= htons(6000);
if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
printf("Unable to bind local address\n");
exit(0);
}
listen(sockfd, 5);
while (1) {
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen) ;
if (newsockfd < 0) {
printf("Accept error\n");
exit(0);
}
if (fork() == 0) {
/* This child process will now communicate with the client through the send() and
recv() system calls.
*/
close(sockfd); /* Close the old socket since all communications will be through the
new socket. */
strcpy(buf,"Message from server");
send(newsockfd, buf, strlen(buf) + 1, 0);
/* We again initialize the buffer, and receive a message from the client.
for(i=0; i < 100; i++) buf[i] = '\0';
recv(newsockfd, buf, 100, 0);
printf("%s\n", buf);
}
}
```

#for understanding

Certainly! Let's go through the workflow of the provided code of tcp server:

1. The program starts by including the necessary header files.
2. The `main()` function begins.

3. Socket descriptors `sockfd` and `newsockfd` are declared to represent the server and client sockets, respectively.
4. The `clilen` variable will store the size of the client address structure.
5. Structures `cli_addr` and `serv_addr` are declared to hold client and server socket addresses.
6. An integer variable `i` is declared for use in a loop.
7. A character array `buf` is declared to store the message.

8. The `socket()` function is called to create a socket. If it fails, an error message is printed, and the program exits.
9. The server address structure `serv_addr` is initialized with the server details: address family, IP address, and port number.
10. The `bind()` function is called to bind the socket to the local address. If it fails, an error message is printed, and the program exits.
11. The `listen()` function is called to listen for incoming connections on the socket. The argument `5` specifies the maximum number of pending connections.

12. The server enters an infinite loop to wait for incoming client connections.

13. The `accept()` function is called to accept a new client connection. If it fails, an error message is printed, and the program exits.
14. A child process is created using `fork()`. This allows the server to handle multiple clients concurrently.
15. In the child process, the `sockfd` is closed since communication will now happen through the `newsockfd`.
16. The server sends a message "Message from server" to the client using the `send()` system call.
17. The child process exits, effectively returning to the beginning of the loop to accept the next client connection.

18. The loop continues to accept new client connections.

19. After the loop ends, the `newsockfd` is closed.
20. The program exits.

The workflow of this code is as follows:
1. Create a socket using `socket()` function
2. Initialize the server address structure.
3. Bind the socket to the local address using `bind()` function.
4. Listen for incoming connections using `listen()` function.
5. Enter an infinite loop to accept new client connections.
6. Accept a new client connection using `accept()` function.
7. Fork a child process to handle communication with the client.
8. In the child process, send a message to the client using `send()` function.
9. Close the old socket descriptor in the child process.
10. The child process exits, returning to the beginning of the loop to accept the next client connection.
11. The loop continues to accept new client connections.
12. After the loop ends, close the `newsockfd`.
13. The program exits.

This workflow allows the server to handle multiple client connections concurrently by forking a child process for each connection.

#for understanding

"if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)" explain this?

The line you provided is a statement in the C programming language, typically used for socket programming. Let's break it down step by step:

1. `sockfd`: This is a variable of type `int` that will store the socket descriptor, which is a unique identifier for the socket being created. It will be used to refer to the socket in subsequent operations.

2. `socket()`: This is a function used to create a socket. It takes three arguments:
   - `AF_INET`: This specifies the address domain of the socket, in this case, IPv4.
   - `SOCK_STREAM`: This specifies the type of socket, in this case, a TCP socket. SOCK_STREAM indicates a reliable, connection-oriented socket.
   - `0`: This parameter is used to specify the protocol. When set to 0, the operating system chooses the appropriate protocol based on the given address domain and socket type.

3. `< 0`: This is a comparison condition that checks if the return value of the `socket()` function is less than 0. If it is, it means an error occurred during the socket creation process.

4. `=`: This is the assignment operator. It assigns the return value of the `socket()` function to the `sockfd` variable.

So, when this line of code is executed, it attempts to create a TCP socket using the IPv4 address domain. If the socket creation is successful, the socket descriptor will be stored in the `sockfd` variable. If an error occurs, the value of `sockfd` will be less than 0, indicating the failure of the socket creation.

#for understanding

serv_addr.sin_family = AF_INET; serv_addr.sin_addr.s_addr = INADDR_ANY; serv_addr.sin_port = htons(6000);"explain this?

The code you provided is typically used in socket programming in C to set up the address and port information for a server-side socket. Let's go through each line:

1. `serv_addr.sin_family = AF_INET;`
- `serv_addr` is a structure of type `struct sockaddr_in` that represents the server's address information.
- `sin_family` is a field within the `sockaddr_in` structure that specifies the address family of the socket.
- `AF_INET` is a constant that represents the IPv4 address family.
- By setting `serv_addr.sin_family` to `AF_INET`, you are indicating that the socket will use IPv4 addresses.

2. `serv_addr.sin_addr.s_addr = INADDR_ANY;`
- `sin_addr` is a field within the `sockaddr_in` structure that holds the IP address of the server.
- `s_addr` is a field within the `sin_addr` structure that represents the IP address as an integer.
- `INADDR_ANY` is a constant that indicates that the socket can accept connections from any available network interface on the server.
- By assigning `INADDR_ANY` to `serv_addr.sin_addr.s_addr`, you are specifying that the server will listen on all available network interfaces.

3. `serv_addr.sin_port = htons(6000);`
   - `sin_port` is a field within the `sockaddr_in` structure that holds the port number for the server.
   - `htons()` is a function used to convert the port number from host byte order to network byte order.
   - `6000` is the port number you want to assign to the server.
   - By calling `htons(6000)` and assigning the result to `serv_addr.sin_port`, you are converting the port number to network byte order and setting it as the server's port.

Overall, these lines of code configure the server's address structure (`serv_addr`) by setting the address family to IPv4, allowing connections from any network interface, and specifying the port number to be used by the server.