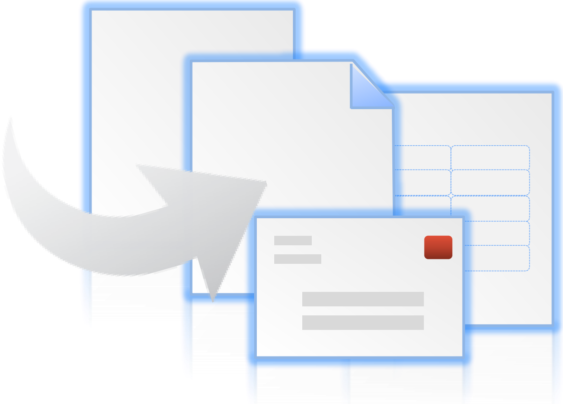
NETWORK PROGRAMMING

TCP CLIENT-SERVER EXAMPLE

BY

MR.PRASAD SAWANT

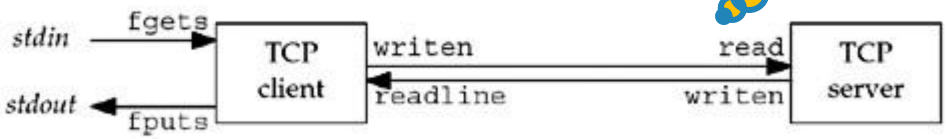


OUT LINE OF UNIT

1. TCP Echo Server: main Function
2. TCP Echo Server: str_echo Function
3. TCP Echo Client: main Function
4. TCP Echo Client: str_cli Function
5. Normal Startup
6. Normal Termination
7. Connection Abort before accept Returns
8. Termination of Server Process,
9. SIGPIPE Signal
10. Crashing of Server Host,
11. Crashing and Rebooting of Server Host
12. Shutdown of Server Host

TCP CLIENT - SERVER COMMUNICATION

Full Duplex Communication

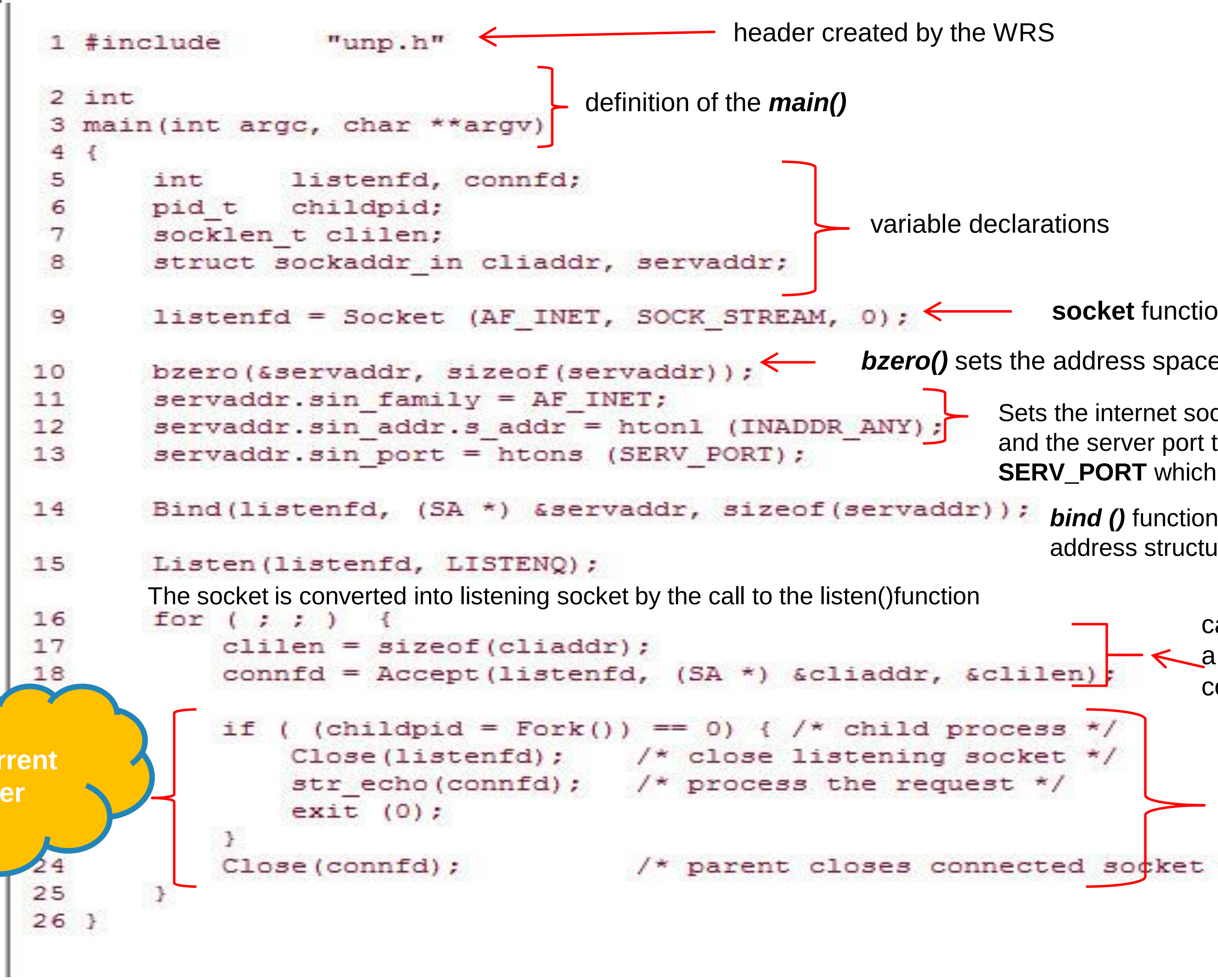


1.The client reads a line of text from its standard input and writes the line to the server.

2.The server reads the line from its network input and echoes the line back to the client.

3.The client reads the echoed line and prints it on its standard output.

tcp echo server: main function



```
#include     "unp.h"

int
main(int argc, char **argv)
{
    int      listenfd, connfd;
    pid_t    childpid;
    socklen_t clilen;
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket (AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
    servaddr.sin_port = htons (SERV_PORT);

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

        if ( (childpid = Fork()) == 0) { /* child process */
            Close(listenfd);    /* close listening socket */
            str_echo(connfd);   /* process the request */
            exit (0);
        }
        Close(connfd);          /* parent closes connected socket
    }
}
```

header created by the WRS

definition of the ***main()***

variable declarations

socket function

bzero() sets the address space to zero.

Sets the internet socket address to wild card addre
and the server port to the number defined in
SERV_PORT which is 9877

bind () function binds the address specified b
address structure to the socket

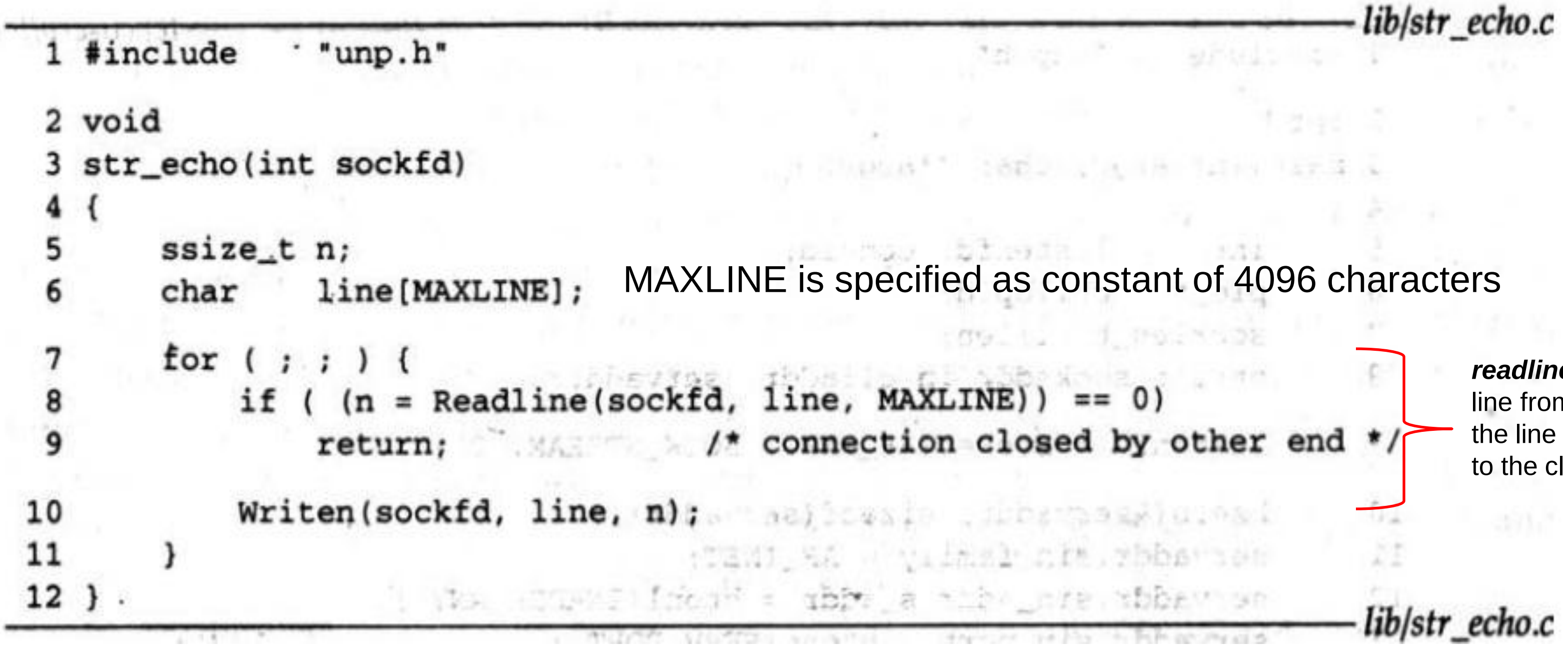
The socket is converted into listening socket by the call to the listen()function

call to accept, waiting for a client connection to complete

Concurrent server

For each client, ***fork()*** spawns a child and the child handles the new client. The child closes the listening socket and the parent closes the connected socket The child then calls ***str_echo ()*** to handle the client

tcp echo server : **str_echo** function



lib/str_echo.c

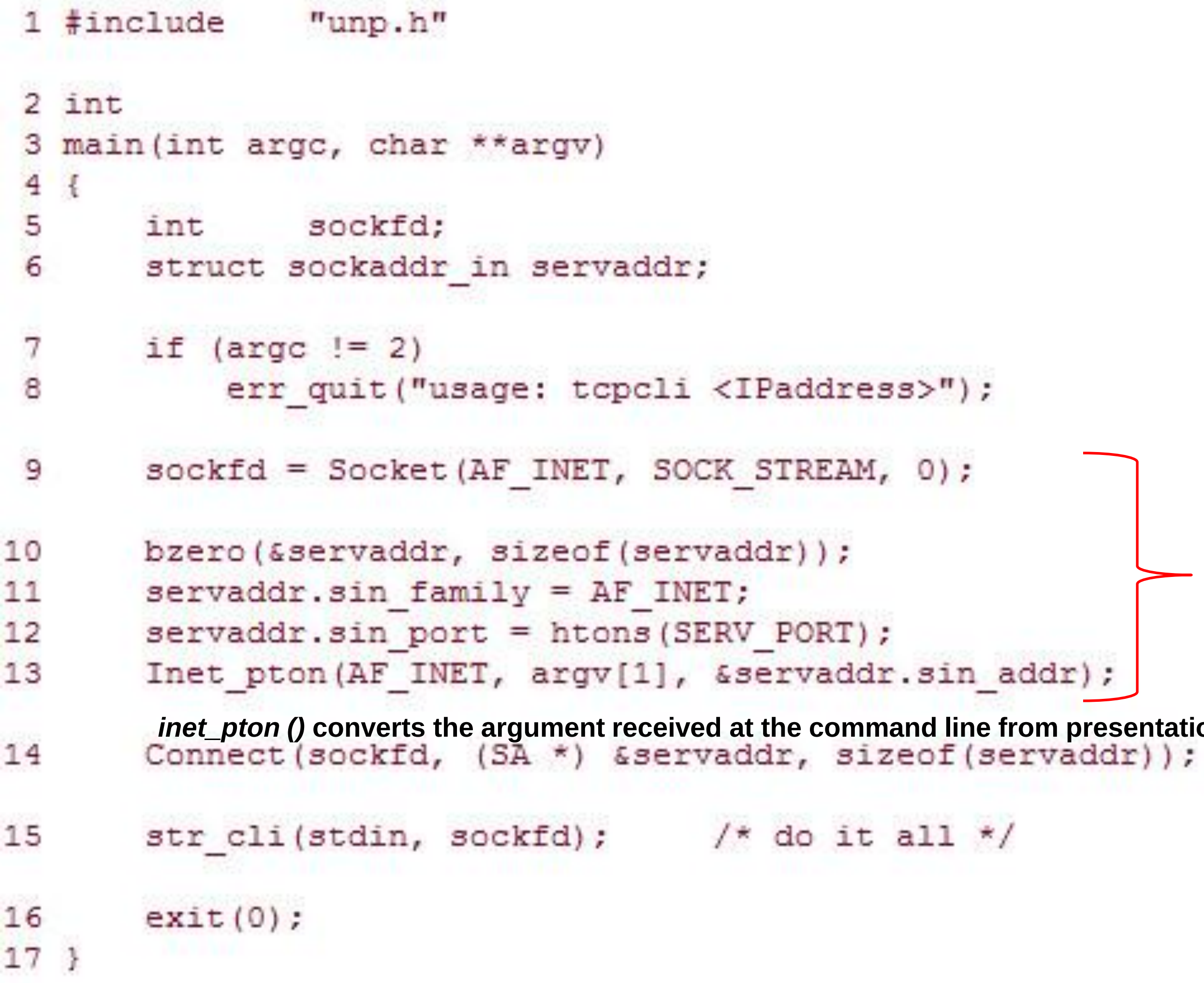
```
#include    "unp.h"

void
str_echo(int sockfd)
{
    ssize_t n;
    char    line[MAXLINE];

    for ( ; ; ) {
        if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
            return;         /* connection closed by other end */

        Writen(sockfd, line, n);
    }
}
```

lib/str_echo.c

MAXLINE is specified as constant of 4096 characters

readline reads the next line from the socket and the line is echoed back to the client by ***writen***

tcp echo client main()



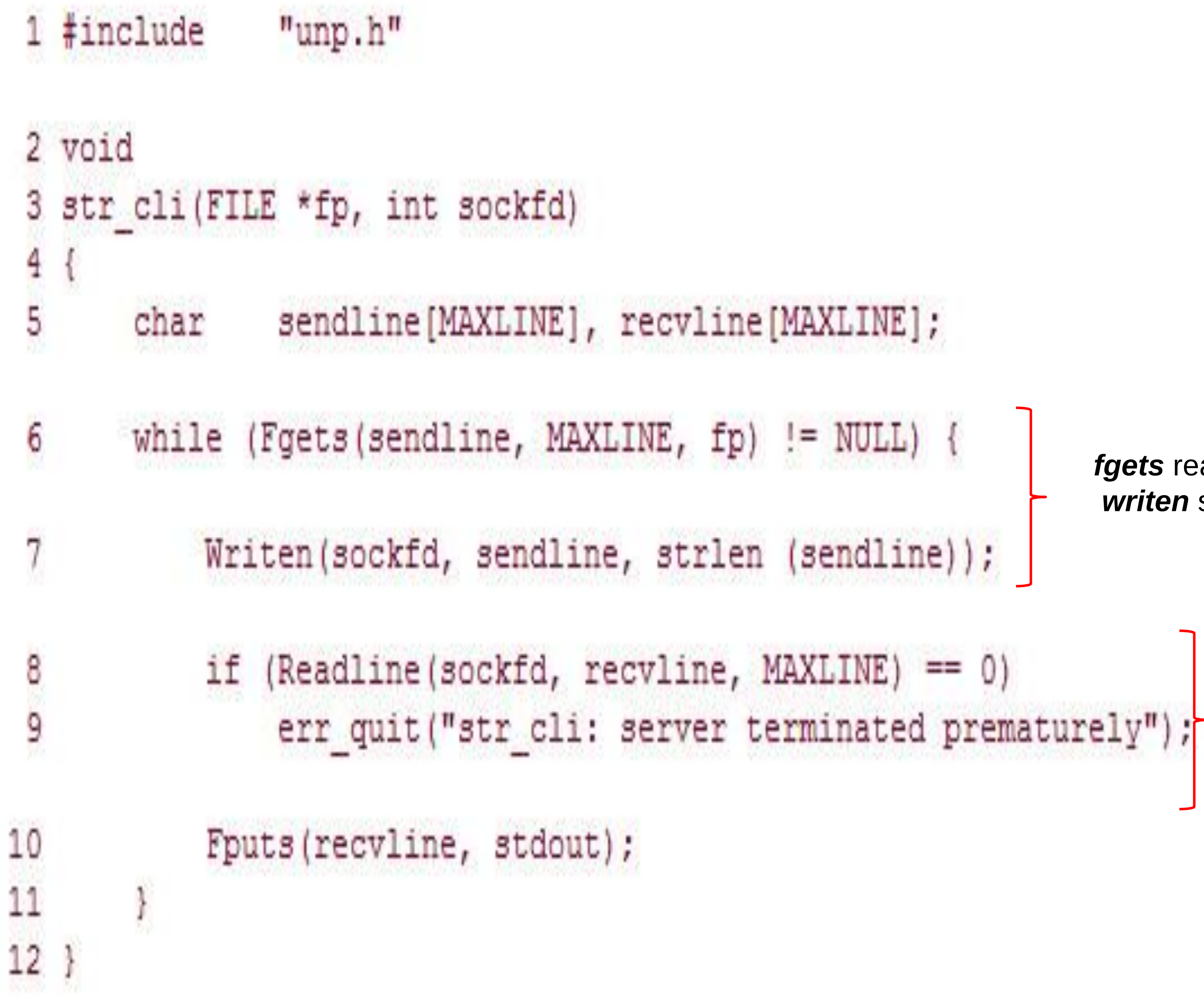
```
#include    "unp.h"

int
main(int argc, char **argv)
{
    int     sockfd;
    struct sockaddr_in servaddr;

    if (argc != 2)
        err_quit("usage: tcpcli <IPaddress>");

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

    str_cli(stdin, sockfd);     /* do it all */

    exit(0);
}
```

A TCP socket is created and an Internal socket address structure is filled in with the server's IP address and port number.

inet_pton () **converts the argument received at the command line from presentation to numeric**

Connection function establishes the connection with the server. The function str_cli () than handles the client processing.

tcp echo client: str_cli function



```
#include    "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    char    sendline[MAXLINE], recvline[MAXLINE];

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Writen(sockfd, sendline, strlen (sendline));

        if (Readline(sockfd, recvline, MAXLINE) == 0)
            err_quit("str_cli: server terminated prematurely");

        Fputs(recvline, stdout);
    }
}
```

fgets reads a line of text and ***writen*** sends the line to the server

readline reads the line echoed back from the server and fputs writes it to the standard output.

NORMAL STARTUP 1

We first start the server in the background on the host linux.

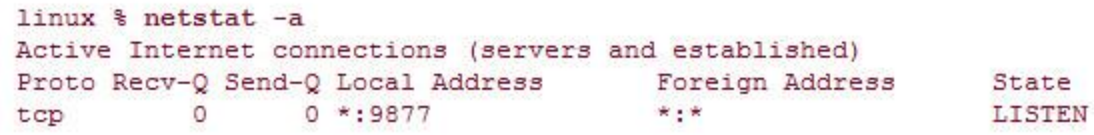
```
linux % tcpserv01 &
[1] 17870
```

When the server starts, it calls socket, bind, listen, and accept, blocking in the call to accept.

Before starting the client, we run the netstat program to verify the state of the server's listening socket.

```
linux % netstat -a
```

NORMAL SETUP 2



```
linux % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:9877                  *:*                     LISTEN
```

netstat: This command shows the status of all sockets on the system, which can be lots of output. We must specify the -a flag to see listening sockets.

We then start the client on the same host, specifying the server's IP address of 127.0.0.1 (the loopback address).

linux % tcpcli01 127.0.0.1

NORMAL SETUP 3

Client calls socket and connect (using 3WH)

1. The client calls str_cli, which will block in the call to fgets, because we have not typed a line of input yet.
2. When accept returns in the server, it calls fork and the child calls str_echo. This function calls readline, which calls read, which blocks while waiting for a line to be sent from the client.
3. The server parent, on the other hand, calls accept again, and blocks while waiting for the next client connection.

NORMAL SETUP

Test method

- tcpserv &
- netstat -a
- tcpcli 127.0.0.1 (local test)
- netstat -a
- ps -l

NORMAL TERMINATION

At this point, the connection is established and whatever we type to the client is echoed back.

```
linux % tcpcli01 127.0.0.1                              we showed this line earlier
hello, world                                            we now type this
hello, world                                            and the line is echoed
good bye
good bye
^D                                                      Control-D is our terminal EOF character
```

NORMAL TERMINATION

- tcpcli 127.0.0.1

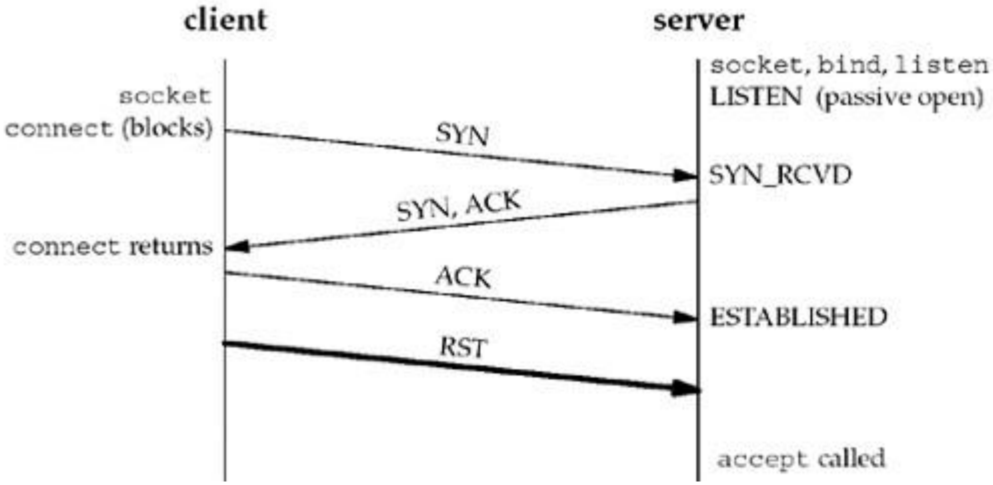
```
hello, world
hello, world
good bye
good bye
^D
```

- netstat -a | grep procID
- ps

```
19130 p1 Ss  -ksh
21130 p1 I  ./tcpserv
21132 p1 Z (tcpserv) (Z:zombie process)
```

CONNECTION
ABORT BEFORE ACCEPT RETURNS

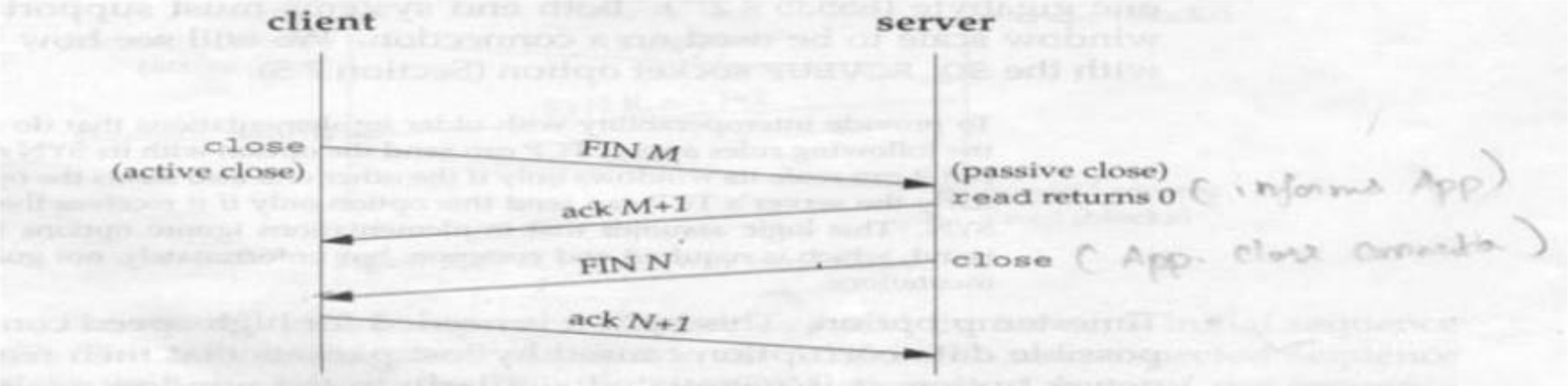
Receiving an RST for an ESTABLISHED connection before accept is called.



Implementation

- BSD : kernel
- SVR4 : return an errno of EPROTO
- Posix.1g : return an errno of ECONNABORTED
    - EPROTO : returned when some fatal protocol-related events occur on the streams subsystem.
- In the case of the ECONNABORTED error, the server can ignore the error and just call accept again.

TCP CONNECTION TERMINATION



TERMINATION OF SERVER PROCESS

1. We start the server and client and type one line to the client to verify that all is okay. That line is echoed normally by the server child.
2. We find the process ID of the server child and kill it. As part of process termination, all open descriptors in the child are closed.
3. Nothing happens at the client. The client TCP receives the FIN from the server TCP and responds with an ACK, but the problem is that the client process is blocked in the call to fgets waiting for a line from the terminal
4. Running netstat at this point shows the state of the sockets

SIGPIPE SIGNAL

when writing to a socket that has received an RST

Procedure:

1. The client writes to a crashed server process. An RST is received at the client TCP and *readline* returns 0 (EOF).
2. If the client ignores the error returned from *readline* and write more, SIGPIPE is sent to the client process.
3. If SIGPIPE is not caught, the client terminates with no output

Problem:

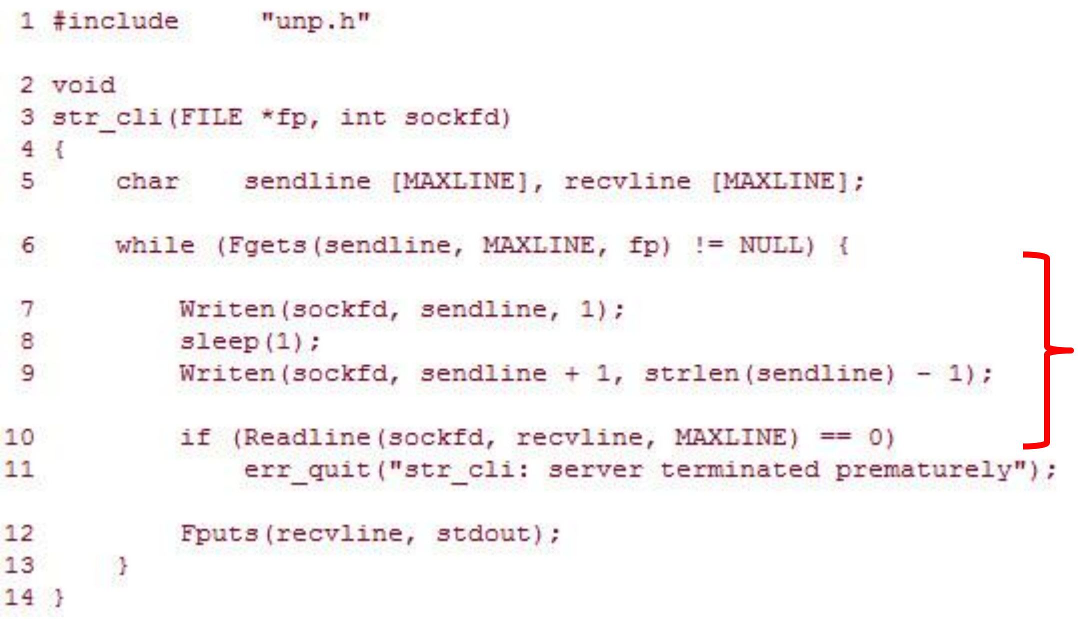
Nothing is output even by the shell to indicate what has happened.
(Have to use “echo $?”to examine the shell’s return value of last command.)

Solution:

1. Setting the signal disposition to SIG_IGN
2. Catch the SIGPIPE signal for further processing. (handle EPIPE error returned from *write)*.

SIG_IGN specifies that the signal should be ignore

SIGPIPE SIGNAL



```
#include    "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    char    sendline [MAXLINE], recvline [MAXLINE];

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        Writen(sockfd, sendline, 1);
        sleep(1);
        Writen(sockfd, sendline + 1, strlen(sendline) - 1);

        if (Readline(sockfd, recvline, MAXLINE) == 0)
            err_quit("str_cli: server terminated prematurely");

        Fputs(recvline, stdout);
    }
}
```

writen two times:
the first time the
first byte of data
is written to the
socket,
followed by a
pause of one
second

An Example to Show SIGPIPE

- To invoke *tcpcli11* which has two write operations to show an example of writing to a closed socket
  - The first write to the closed socket is to solicit RST from the server TCP
  - The second write is to generate SIGPIPE from the local process.
  - An sample run :

```
linux% tcpcli11 127.0.0.1
Hi there                    # user input in bold
Hi there                    # echoed back from server
                            # terminate server child process then
Bye                         # then type this line purposely
Borken pipe                 # output by the shell because of SIGPIPE
```

- Note: To write to a socket which has received a FIN is OK. However, it is an error to write to a socket hat has received an RST

SIGPIPE SIGNAL

```
linux % tcpclill 127.0.0.1
hi there                              we type this line
hi there                              this is echoed by the server
                                      here we kill the server child
bye                                   then we type this line
Broken pipe                           this is printed by the shell
```

CRASH OF SERVER HOST

Scenario

1. client and server run on different hosts
2. make a connection between client and server
3. client types something to transmit data to the server
4. disconnect the server host from the network (destination unreachable)
5. client types something again.

client TCP continuously retx data and timeout around 9 min

The client process will then return with the error ETIMEDOUT.

If some intermediate router determined that the server host was down and responded with an ICMP “destination unreachable” message, the error returned will then be either EHOSTUNREACH or ENETUNREACH

To quickly detect: timeout on *readline*, SO_KEEPALIVE socket option, heartbeat functions

REBOOT OF SERVER HOST

The client does not see the server host shut down

Client sends data to server after the server reboots

server TCP responds to client data with an RST because it loses all connection information

readline returns ECONNRESET

SHUTDOWN OF SERVER HOST (BY OPERATOR)

init process sends SIGTERM to all processes

- We can catch this signal and close all open descriptors by ourselves

init waits 5-20 sec and sends SIGKILL to all processes

- all open descriptors are closed by kernel

ASSIGNMENT # 4

Section A

1. Write and explain TCP Echo Server: main Function
2. Write and explain TCP Echo Server: str_echo Function
3. Write and explain TCP Echo Client: main Function
4. Write and explain TCP Echo Client: str_cli Function
5. Write a note on Connection abort before accept return
6. Write a note on SIGPIPE Signal

Section B

1. Explain Crashing of server Host
2. Explain Crashing and Rebooting of Server Host
3. Explain Shutdown of Server Host

SUMMARY OF TCP EXAMPLE

From client’s perspective:

- **socket** and **connect** specifies server’s port and IP
- client port and IP chosen by TCP and IP respectively

From server’s perspective:

- **socket** and **bind** specifies server’s local port and IP
- **listen** and **accept** return client’s port and IP

TCP Client/Server – Client's Perspective

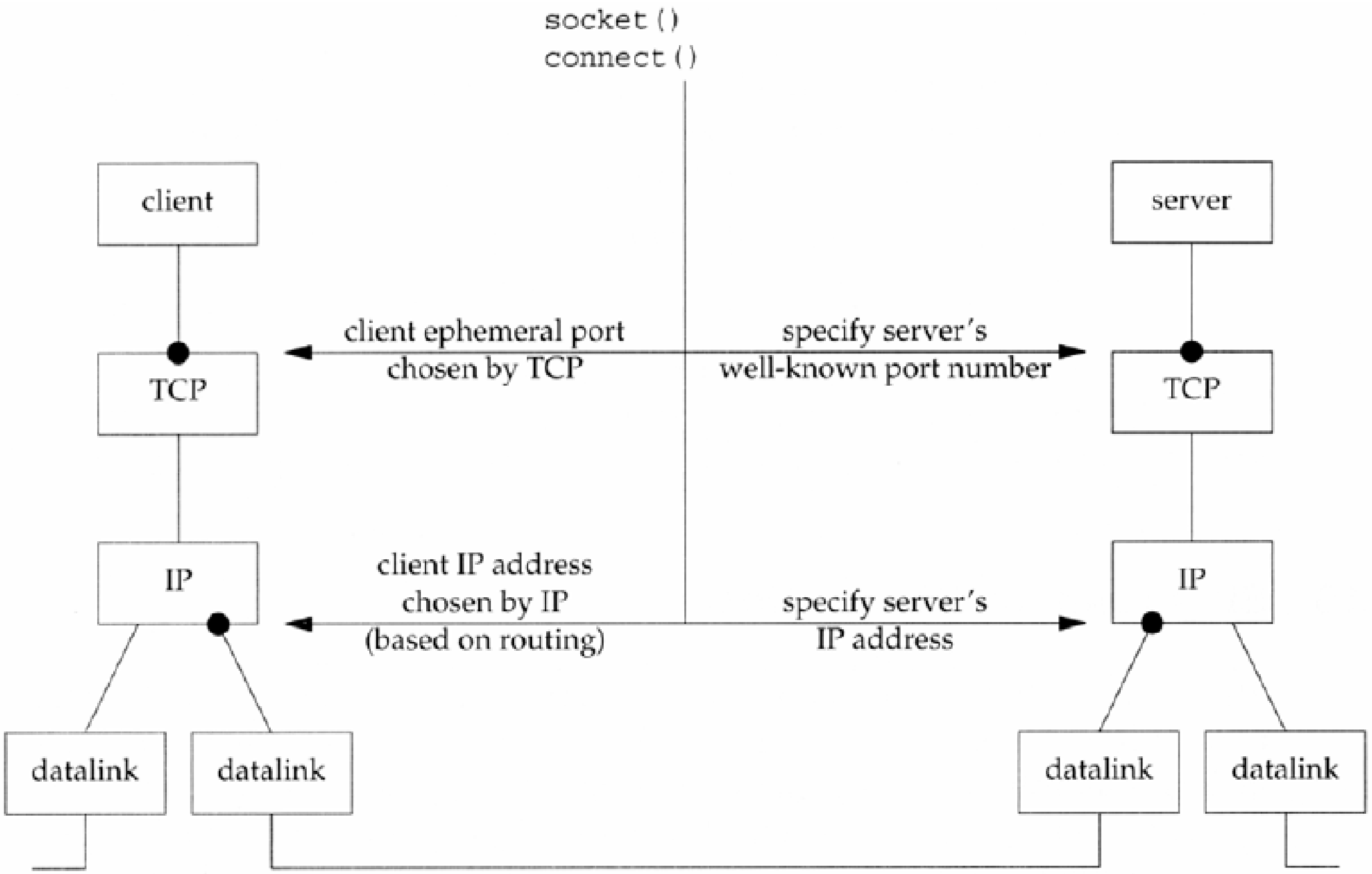

socket()
connect()
client
server
client ephemeral port chosen by TCP
specify server's well-known port number
TCP
TCP
client IP address chosen by IP (based on routing)
specify server's IP address
IP
IP
datalink
datalink
datalink
datalink

Figure 5.15 Summary of TCP client/server from client's perspective.

TCP Client/Server – Client's Perspective

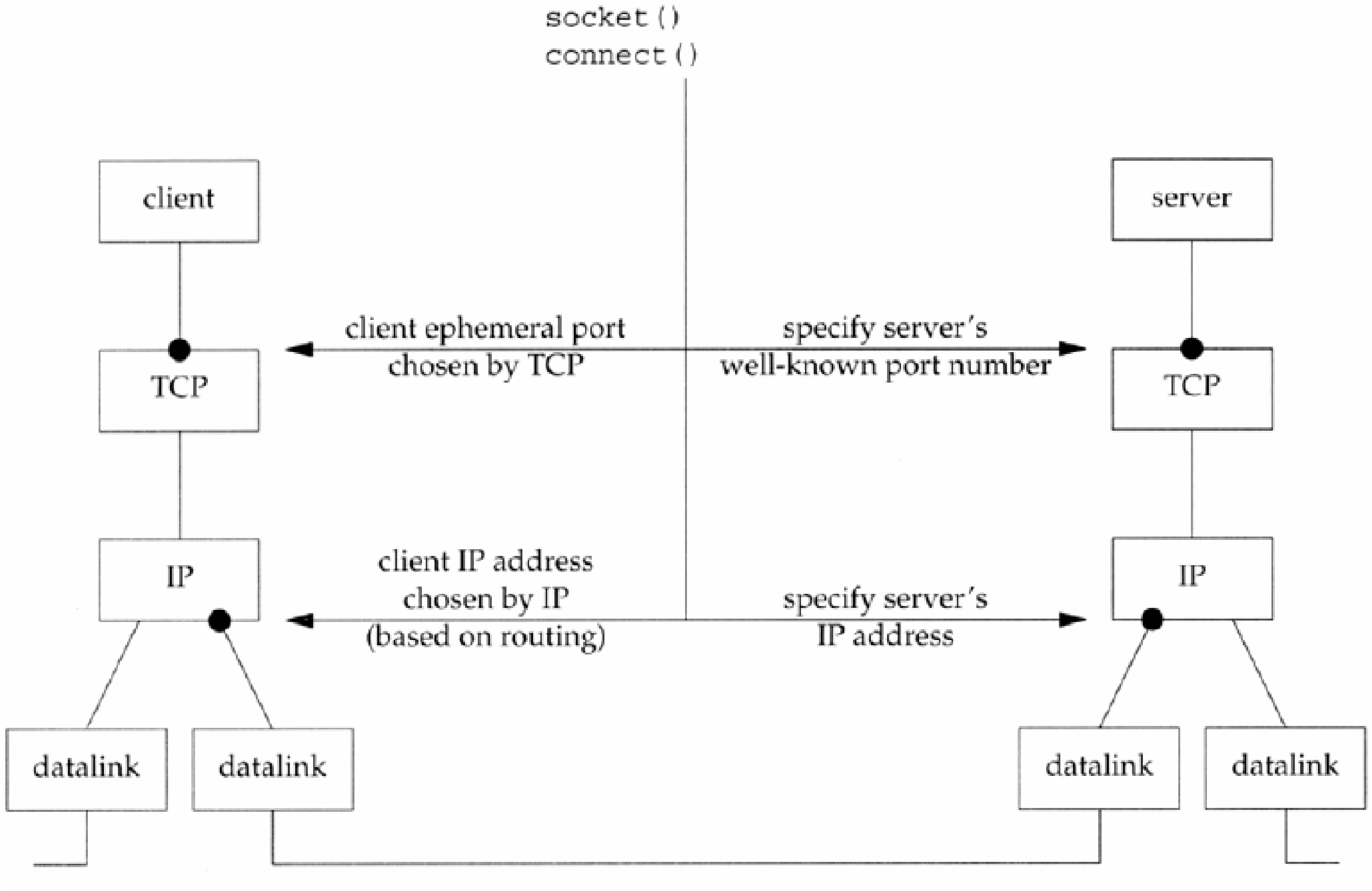

socket()
connect()
client
server
client ephemeral port chosen by TCP
specify server's well-known port number
TCP
TCP
client IP address chosen by IP (based on routing)
specify server's IP address
IP
IP
datalink
datalink
datalink
datalink

Figure 5.15 Summary of TCP client/server from client's perspective.

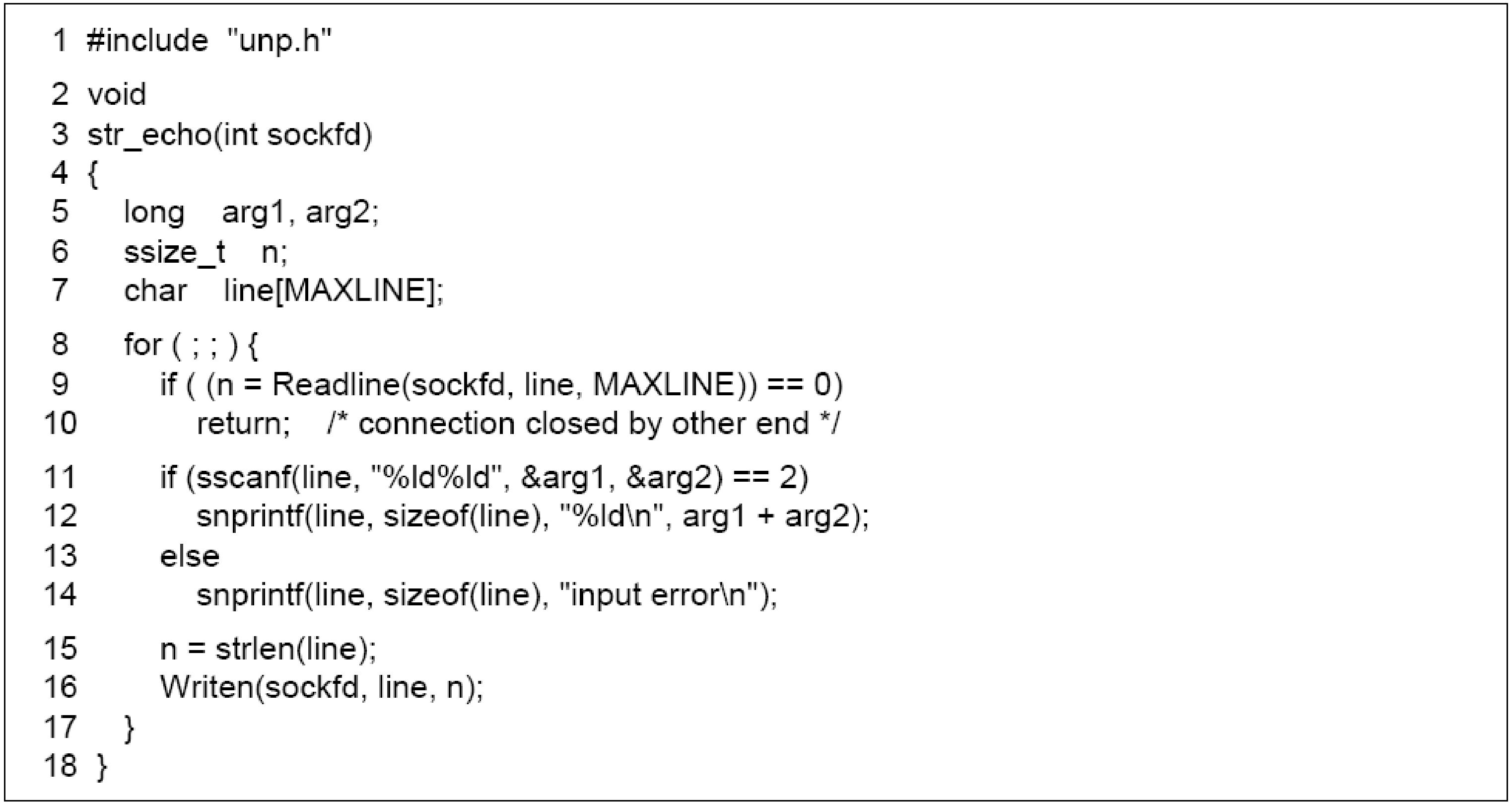
DATA FORMAT: TEXT STRINGS

server process gets two numbers (in a line of text) from client and returns their sum

In str_echo: **sscanf** converts string to long integer, **snprintf** converts long back to string

str_echo() – Adding 2 Numbers

- *tcpcliserv/str_echo08.c*



```c
#include  "unp.h"

void
str_echo(int sockfd)
{
    long    arg1, arg2;
    ssize_t    n;
    char    line[MAXLINE];

    for ( ; ; ) {
        if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
            return;    /* connection closed by other end */

        if (sscanf(line, "%ld%ld", &arg1, &arg2) == 2)
            snprintf(line, sizeof(line), "%ld\n", arg1 + arg2);
        else
            snprintf(line, sizeof(line), "input error\n");

        n = strlen(line);
        Writen(sockfd, line, n);
    }
}
```

DATA FORMAT: BINARY STRUCTURE

Passing binary structure between client and server does not work

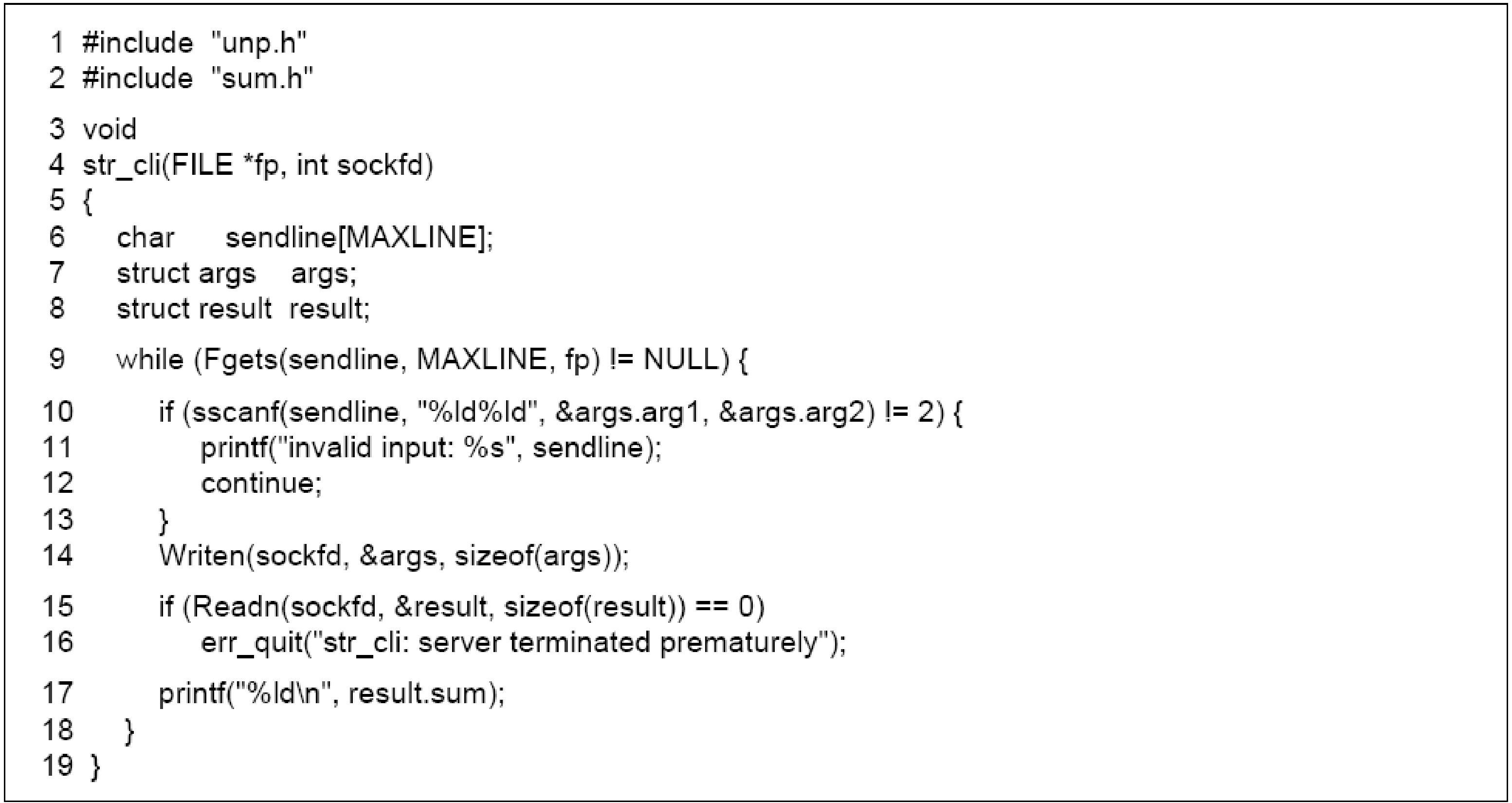
- when the client and server are run on hosts with different byte orders or sizes of long integer
- Since different implementations can store the same C datatype differently.

Suggestions:

- pass in string only
- explicitly define the format of data types (e.g. RPC's XDR -- external data representation)

str_cli() – Sending 2 Binary Int’s

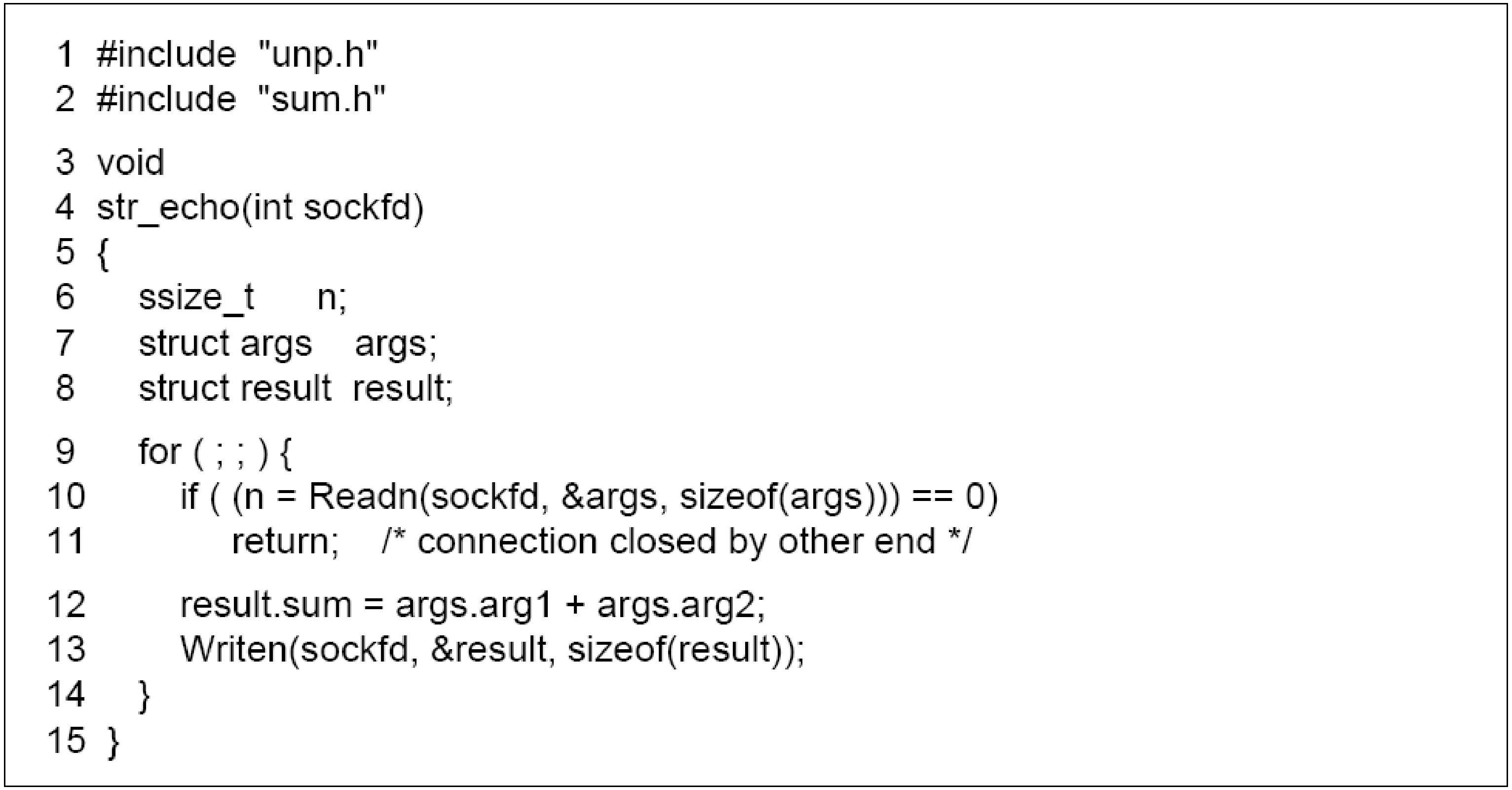
- *tcpcliserv/str_cli09.c*



```
#include  "unp.h"
#include  "sum.h"

void
str_cli(FILE *fp, int sockfd)
{
    char      sendline[MAXLINE];
    struct args    args;
    struct result  result;

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        if (sscanf(sendline, "%ld%ld", &args.arg1, &args.arg2) != 2) {
            printf("invalid input: %s", sendline);
            continue;
        }
        Writen(sockfd, &args, sizeof(args));

        if (Readn(sockfd, &result, sizeof(result)) == 0)
            err_quit("str_cli: server terminated prematurely");

        printf("%ld\n", result.sum);
    }
}
```

str_echo() – Adding 2 Binary Int's

- *tcpcliserv/str_echo09.c*



```
#include  "unp.h"
#include  "sum.h"

void
str_echo(int sockfd)
{
    ssize_t      n;
    struct args    args;
    struct result  result;

    for ( ; ; ) {
        if ( (n = Readn(sockfd, &args, sizeof(args))) == 0)
            return;    /* connection closed by other end */

        result.sum = args.arg1 + args.arg2;
        Writen(sockfd, &result, sizeof(result));
    }
}
```

Beware of Different Byte Orders

- Due to the big-endian and little-endian implementations, sending binary numbers between different machine architectures may end up with different results
  - An example of two big-endian SPARC machines :
    - solaris% **tcpcli09 12.106.32.254**
    - **11 12**   # user input in bold
    - 33   # result back from server
    - **-11 -14**
    - -55
  - An example of big-endian SPARC and little-endian Intel machines :
    - linus% **tcpcli09 206.168.112.96**
    - **1 2**   # user input in bold
    - 3   # It seems to work
    - **-22 -77**
    - -16777314   # oops! It does not work!