TCP ECHO

Presented by: Dr. Islam Alkabbany

Echo server/Client

- A client/server that echoes input lines is a valid, yet simple, example of a network application.
- All the basic steps required to implement any client/server are illustrated by this example.

TCP Echo Server

- A TCP Echo Server is a server that sends back (echoes) whatever data it receives from a client.
- Use Cases: Testing and debugging network applications.
- Server listens for connections, accepts a client, and echoes back received data.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[BUFFER_SIZE] = {0};
    char inputbuf[BUFFER_SIZE] ;

    // Creating socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
```

```
 // Bind the socket to the network address and port
    if (bind(server_fd, (struct sockaddr *)&address,
sizeof(address)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    // Listen for incoming connections
    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    while (1)
    {
      // Accept a connection
      if ((new_socket = accept(server_fd, (struct sockaddr
*)&address, (socklen_t*)&addrlen)) < 0) {
            perror("accept");
            exit(EXIT_FAILURE);
            }
      // Echo back received data
      int read_size;
       while ((read_size = recv(new_socket, buffer, BUFFER_SIZE,
0)) > 0) {
            send(new_socket, buffer, read_size, 0);
      close(new_socket);
    }
    close(server_fd);
    return 0;
}
```

TCP Echo Client

- TCP Echo Client connects to an echo server, sends data, and receives the echoed data back.
- Use Cases: Testing client-server communication, network troubleshooting.
- Client connects to the server, sends data, and receives the same data back.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main(int argc, char **argv) {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};

    // Creating socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
```

```
 // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, argv[1], &serv_addr.sin_addr) <= 0) {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }

    // Connect to the server
    if (connect(sock, (struct sockaddr *)&serv_addr,
sizeof(serv_addr)) < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }

    // Send data
    while (1){
     scanf("%s", &inputbuf)
     if ( strlen(inputbuf)==0) break;

    send(sock, inputbuf, strlen(inputbuf), 0);
    printf("Hello message sent\n");

    // Receive echoed data
    int read_size = read(sock, buffer, BUFFER_SIZE);
    printf("Echoed message: %s\n", buffer);
     }
    close(sock);
    return 0;
}
```

Our Wrapping function

- We will define our wrapping function to make the program shorter
- It could be add as include file in each program
- Also it could be compiled and the resulting object file could be linked with main program,

```
int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr)
{     int          n;
again:
      if ( (n = accept(fd, sa, salenptr)) < 0) {
#ifdef      EPROTO
            if (errno == EPROTO || errno == ECONNABORTED)
#else
            if (errno == ECONNABORTED)
#endif
                  goto again;
            else
                  err_sys("accept error");
      }
      return(n);
}

void Bind(int fd, const struct sockaddr *sa, socklen_t salen)
{
      if (bind(fd, sa, salen) < 0)
            err_sys("bind error");
}

      err_quit("inet_pton error for %s", strptr);     /* errno not set
*/
}
```

```
void Connect(int fd, const struct sockaddr *sa, socklen_t salen)
{
      if (connect(fd, sa, salen) < 0)
            err_sys("connect error");
}


void Close(int fd)
{
      if (close(fd) == -1)
            err_sys("close error");
}

void Inet_pton(int family, const char *strptr, void *addrptr)
{
      int n;
      if ( (n = inet_pton(family, strptr, addrptr)) < 0)
            err_sys("inet_pton error for %s", strptr);      /* errno
set */
      else if (n == 0)
            err_quit("inet_pton error for %s", strptr);     /* errno
not set */
}
```

Echo Server

```
#define PORT 8080
#define BUFFER_SIZE 1024
#include "wraping.c"
int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[BUFFER_SIZE] = {0};
    server_fd = Socket(AF_INET, SOCK_STREAM, 0)
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    Bind(server_fd, (struct sockaddr *)&address, sizeof(address))
    Listen(server_fd, 3)
    while (1)
   {
        new_socket = Accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen))
        int read_size;
        while ((read_size = recv(new_socket, buffer, BUFFER_SIZE, 0)) > 0) {
                send(new_socket, buffer, read_size, 0);
        close(new_socket);
    }
    Close(server_fd);
    return 0;
}
```

Echo Client

```
#define PORT 8080
#define BUFFER_SIZE 1024
#include "wraping.c"
int main(int argc, char **argv) {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};
    char inputbuf[BUFFER_SIZE] ;
    sock = Socket(AF_INET, SOCK_STREAM, 0))
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    Inet_pton(AF_INET, argv[1], &serv_addr.sin_addr)
    Connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr))

    while (1){
     scanf("%s", &inputbuf)
     if ( strlen(inputbuf)==0) break;

    send(sock, inputbuf, strlen(inputbuf), 0);
    printf("Hello message sent\n");
    int read_size = read(sock, buffer, BUFFER_SIZE);
    printf("Echoed message: %s\n", buffer);
    }
    Close(sock);
    return 0;
}
```

Long message Handling

- Stream sockets (e.g., TCP sockets) exhibit a behavior with the read and write functions that differs from normal file I/O. A read or write on a stream socket might input or output fewer bytes than requested, but this is not an error condition.
- The reason is that buffer limits might be reached for the socket in the kernel.
- All that is required to input or output the remaining bytes is for the caller to invoke the read or write function again
- Some versions of Unix also exhibit this behavior when writing more than 4,096 bytes to a pipe. This scenario is always a possibility on a stream socket with read, but is normally seen with write

Our Writen

```
 ssize_t   writen(int fd, const void *vptr, size_t n)
{
        size_t        nleft;
        ssize_t               nwritten;
        const char    *ptr;
        ptr = vptr;
        nleft = n;
        while (nleft > 0) {
                if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
                        if (nwritten < 0 && errno == EINTR)
                                nwritten = 0;
                        else
                                return(-1);
                }
                nleft -= nwritten;
                ptr   += nwritten;
        }
        return(n);
}

Void Writen(int fd, void *ptr, size_t nbytes)
{
        if (writen(fd, ptr, nbytes) != nbytes)
                err_sys("writen error");
}
```

Our Readn

```
 ssize_t readn(int fd, void *vptr, size_t n)
{
        size_t  nleft;
        ssize_t nread;
        char    *ptr;
        ptr = vptr;
        nleft = n;
        while (nleft > 0) {
                if ( (nread = read(fd, ptr, nleft)) < 0) {
                        if (errno == EINTR)
                                nread = 0;
                        else
                                return(-1);
                } else if (nread == 0)
                        break;

                nleft -= nread;
                ptr   += nread;
        }
        return(n - nleft);
}
ssize_t Readn(int fd, void *ptr, size_t nbytes)
{
        ssize_t         n;

        if ( (n = readn(fd, ptr, nbytes)) < 0)
                err_sys("readn error");
        return(n);
}
```

Readline

```
ssize_t read_line(int sockfd, char *buffer, size_t max_len) {
    static char temp_buf[BUFFER_SIZE];     static size_t
temp_len = 0;
    ssize_t total_read = 0;     char *buf_ptr = buffer;
    while (1) {
    if (temp_len == 0) {
            ssize_t rc = read(sockfd, temp_buf, BUFFER_SIZE -
1);
            if (rc == -1) {
                if (errno == EINTR) {
                    continue;
                }   return -1; // Other errors
            } else if (rc == 0) {                  // End of
file (connection closed by peer)
                if (total_read == 0) {
                    return 0; // No data read
                } else {
                    break; // Some data was read
                }
            }
            temp_buf[rc] = '\0'; // Null-terminate the buffer
            temp_len = rc;
        }
```

Readline

```
// Check characters in the temporary buffer
        for (size_t i = 0; i < temp_len; i++) {
            if (total_read < max_len - 1) {
                buffer[total_read++] = temp_buf[i];
                if (temp_buf[i] == '\n') {
                    // Move remaining characters in the
buffer to the start
                    memmove(temp_buf, temp_buf + i + 1,
temp_len - i - 1);
                    temp_len -= (i + 1);
                    buffer[total_read] = '\0'; // Null-
terminate the string
                    return total_read;
                }
            }
        }
        // No newline found, reset the temporary buffer
length
        temp_len = 0;
    }
    buffer[total_read] = '\0’; ]
    return total_read;
}
```

Echo Server

```
#define PORT 8080
#define BUFFER_SIZE 1024
#include "wraping.c"
void  str_echo(int sockfd) {
     ssize_t n;
     char    buf[BUFFER_SIZE ];
   again: while ( (n = read(sockfd, buf,
BUFFER_SIZE )) > 0)
          Writen(sockfd, buf, n);
       if (n < 0 && errno == EINTR)
           goto again;
       else if (n < 0)
           err_sys("str_echo: read error");
}
```

Echo Server

```
int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[BUFFER_SIZE] = {0};
    server_fd = Socket(AF_INET, SOCK_STREAM, 0)
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    Bind(server_fd, (struct sockaddr *)&address, sizeof(address))
    Listen(server_fd, 3)
    while (1)
    {
      new_socket = Accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen))
      int read_size;
      str_echo(sockfd)
      close(new_socket);
   }
    Close(server_fd);
    return 0;
}
```

Echo Client

```
#define PORT 8080
#define BUFFER_SIZE 1024
#include "wraping.c“
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);
  }
}
```

Echo Client

```
int main(int argc, char **argv) {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};
    char inputbuf[BUFFER_SIZE] ;

    sock = Socket(AF_INET, SOCK_STREAM, 0))
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    Inet_pton(AF_INET, argv[1], &serv_addr.sin_addr)
    Connect(sock, (struct sockaddr *)&serv_addr,
sizeof(serv_addr))
    str_cli(stdin, sockfd);
    Close(sock);
    return 0;
}
```

Concurrent server

- We can use fork to handle each client in different Process
- This approach allows the server to handle multiple client connections concurrently
- The child process closes the listening socket to avoid unnecessary resource usage and potential conflicts

Echo Concurrent Server

```
Listen(server_fd, 3)
    while (1)
    {
     new_socket = Accept(server_fd, (struct sockaddr
*)&address, (socklen_t*)&addrlen))
    if ( (childpid = Fork()) == 0) {/* child
process */
            Close(server_fd);    /* close listening
socket */
     str_echo(sockfd)
     Close(new_socket);
     exit(0);}
   }
     close(server_fd);
     return 0;
}
```

Normal Startup

- it is essential that we understand how the client and server start, how they end, and most importantly, what happens when something goes wrong:
  - the client host crashes
  - the client process crashes
  - network connectivity is lost, and so on
- Only by understanding these boundary conditions, and their interaction with the TCP/IP protocols, can we write robust clients and servers that can handle these conditions.

Normal Startup

- We first start the server in the background on the host
  tcpserv01 &

- When the server starts, it calls socket, bind, listen, and accept, blocking in the call to accept. (We have not started the client yet.) Before starting the client, we run the netstat program to verify the state of the server's listening socket.
  netstat -a

Normal Startup

- We then start the client on the same host *tcpcli01 127.0.0.1*

- The client calls socket and connect, the latter causing TCP's three-way handshake to take place. When the three-way handshake completes, connect returns in the client and accept returns in the server. The connection is established

Normal Startup

- The following steps then take place:
  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 Startup

- We have three processes, and all three are asleep (blocked): client, server parent, and server child.
- We purposely run the client and server on the same host because this is the easiest way to experiment with client/server applications. Since we are running the client and server on the same host, netstat now shows two additional lines of output, corresponding to the TCP connection

Normal Termination

- At this point, the connection is established and whatever we type to the client is echoed back
- We can follow through the steps involved in the normal termination of our client and server:
  1. When we type our EOF character, fgets returns a null pointer and the function str_cli
  2. When str_cli returns to the client main function it Close the socket and end client program
  3. Even without Close part of process termination is the closing of all open descriptors, so the client socket is closed by the kernel. This sends a FIN to the server, to which the server TCP responds with an ACK. This is the first half of the TCP connection termination sequence. At this point, the server socket is in the CLOSE_WAIT state and the client socket is in the FIN_WAIT_2 state

Normal Termination

4. When the server TCP receives the FIN, the server child is blocked in a call to readline and readline then returns 0. This causes the str_echo function to return to the server child main
5. The server child Close socket and terminates by calling exit
6. Even if no Close called All open descriptors in the server child are closed. The closing of the connected socket by the child causes the final two segments of the TCP connection termination to take place: a FIN from the server to the client, and an ACK from the client . At this point, the connection is completely terminated. The client socket enters the TIME_WAIT state.
7. If no closed called, the SIGCHLD signal is sent to the parent when the server child terminates. This occurs in this example, but we do not catch the signal in our code, and the default action of the

POSIX signals

- Definition: Signals are software interrupts sent to a process by the operating system.
- Purpose: Used to notify a process that a specific event has occurred.
- Common Signals: SIGINT (Interrupt), SIGTERM (Terminate), SIGHUP (Hang Up), SIGKILL (Kill), SIGSEGV (Segmentation Violation).
- Signals usually occur asynchronously. By this we mean that a process doesn't know ahead of time exactly when a signal will occur

POSIX signals

- Signals can be sent
  - By one process to another process (or to itself)
  - By the kernel to a process
- The SIGCHLD signal that we described at the end of the previous section is one that is sent by the kernel whenever a process terminates, to the parent of the terminating process.

POSIX signals

- Every signal has a disposition, which is also called the action associated with the signal. We set the disposition of a signal by calling the sigaction function (described shortly) and we have three choices for the disposition

1. We can provide a function that is called whenever a specific signal occurs. This function is called a signal handler and this action is called catching a signal. The two signals SIGKILL and SIGSTOP cannot be caught.

   Our function is called with a single integer argument that is the signal number and the function returns nothing. Its function prototype is therefore
   void handler (int signo);

   For most signals, calling sigaction and specifying a function to be called when the signal occurs is all that is required to catch a signal. But we will see later that a few signals, SIGIO, SIGPOLL, and SIGURG, all require additional actions on the part of the process to catch the signal.

POSIX signals

2. We can ignore a signal by setting its disposition to SIG_IGN. The two signals SIGKILL and SIGSTOP cannot be ignored.
3. We can set the default disposition for a signal by setting its disposition to SIG_DFL. The default is normally to terminate a process on receipt of a signal, with certain signals also generating a core image of the process in its current working directory. There are a few signals whose default disposition is to be ignored: SIGCHLD and SIGURG

```
Sigfunc *signal(int signo, Sigfunc *func)
{
        struct sigaction    act, oact;

        act.sa_handler = func;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;
        if (signo == SIGALRM) {
#ifdef SA_INTERRUPT
                act.sa_flags |= SA_INTERRUPT;     /* SunOS 4.x */
#endif
        } else {
#ifdef SA_RESTART
                act.sa_flags |= SA_RESTART;             /* SVR4, 44BSD */
#endif
        }
        if (sigaction(signo, &act, &oact) < 0)
                return(SIG_ERR);
        return(oact.sa_handler);
}

Sigfunc *Signal(int signo, Sigfunc *func)      /* for our signal() function
*/
{
        Sigfunc      *sigfunc;

        if ( (sigfunc = signal(signo, func)) == SIG_ERR)
                err_sys("signal error");
        return(sigfunc);
}
```

Handling SIGCHLD Signals

- The purpose of the zombie state is to maintain information about the child for the parent to fetch at some later time.
- This information includes the process ID of the child, its termination status, and information on the resource utilization of the child (CPU time, memory, etc.).
- If a process terminates, and that process has children in the zombie state, the parent process ID of all the zombie children is set to 1 (the init process), which will inherit the children and clean them up (i.e., init will

Handling Zombies

- Obviously we do not want to leave zombies around. They take up space in the kernel and eventually we can run out of processes.
- Whenever we fork children, we must wait for them to prevent them from becoming zombies.
- To do this, we establish a signal handler to catch SIGCHLD, and within the handler, we call wait

Handling Zombies

- We establish the signal handler by adding the function call
  Signal (SIGCHLD, sig_chld);

- It should called after the call to listen. (It must be done sometime before we fork the first child and needs to be done only once.) We then define the signal handler, the function sig_chld

Handling Zombies

```
void sig_chld(int signo)
{
    pid_t    pid;
    int      stat;
    pid = wait(&stat);
    printf("child %d terminated\n", pid);
    return;
}
```

Handling Zombies

- The sequence of steps is as follows:

1. We terminate the client by typing our EOF character. The client TCP sends a FIN to the server and the server responds with an ACK.
2. The receipt of the FIN delivers an EOF to the child's pending readline. The child terminates.
3. The parent is blocked in its call to accept when the SIGCHLD signal is delivered. The sig_chld function executes (our signal handler), wait fetches the child's PID and termination status, and printf is called from the signal handler. The signal handler returns.
4. Since the signal was caught by the parent while the parent was blocked in a slow system call (accept), the kernel causes the accept to return an error of EINTR (interrupted system call). The

Handling Interrupted System Calls

```
for ( ; ; )
{
clilen = sizeof (cliaddr);
 if ( (connfd = accept (listenfd, (SA *) &cliaddr,
&clilen)) < 0)
 {
     if (errno == EINTR)
          continue; /* back to for () */
     else
          err_sys ("accept error");
}
```

wait and waitpid Functions

```
pid_t wait (int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options)
```

- Both return: process ID if OK, 0 or 1 on error
- Both return two values: the return value of the function is the process ID of the terminated child, and the termination status of the child (an integer) is returned through the statloc pointer
- If there are no terminated children for the process calling wait, but the process has one or more children that are still executing, then

wait and waitpid Functions

```
pid_t wait (int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options)
```

- waitpid gives us more control over which process to wait for and whether or not to block.
- First, the pid argument lets us specify the process ID that we want to wait for. A value of -1 says to wait for the first of our children to terminate. (There are other options, dealing with process group IDs, but we do not need them in this text.)
- The options argument lets us specify additional options. The most common option is WNOHANG. This option tells the kernel not to block if there are no terminated children.

```
int main(int argc, char **argv)
{
        int    listenfd, connfd;
        pid_t  childpid;
        socklen_t     clilen;
        struct sockaddr_in   cliaddr, servaddr;
        void     sig_chld(int);
        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);
        Signal(SIGCHLD, sig_chld);  /* must call waitpid() */
        for ( ; ; ) {
                clilen = sizeof(cliaddr);
                if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
                        if (errno == EINTR)
                                continue;            /* back to for() */
                        else
                                err_sys("accept error");
                }
                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 */
        }
}
```

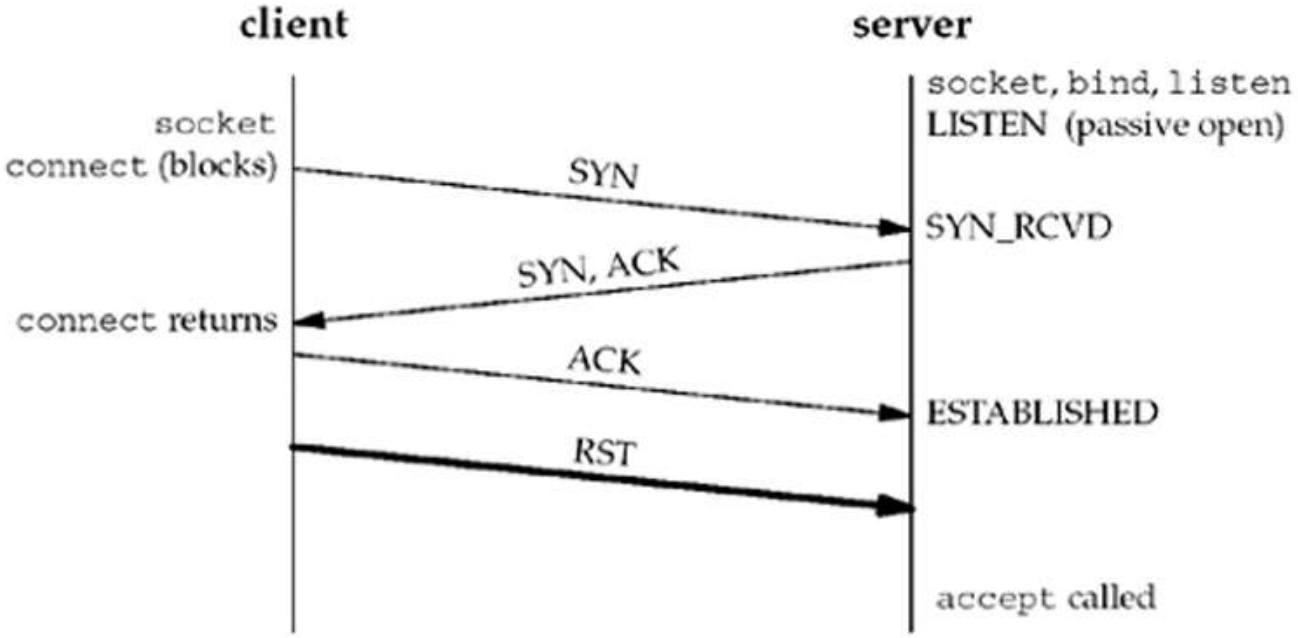
sig_child with waitpid

```
#include  "unp.h"
void
sig_chld(int signo)
{
     pid_tpid;
     int       stat;

     while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
          printf("child %d terminated\n", pid);
     return;
}
```

Connection Abort before accept Returns

- There is another condition similar to the interrupted system call example in the previous section that can cause accept to return a nonfatal error, in which case we should just call accept again.



Connection Abort before accept Returns

- Unfortunately, what happens to the aborted connection is implementation-dependent.
- Berkeley-derived implementations handle the aborted connection completely within the kernel, and the server process never sees it, however, return an error to the process as the return from accept, and the error depends on the implementation errno of EPROTO ("protocol error").
- But POSIX specifies that the return must be ECONNABORTED ("software caused connection abort") instead

Termination of Server Process

- We will now start our client/server and then kill the server child process. This simulates the crashing of the server process, so we can see what happens to the client

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. This causes a FIN to be sent to the client, and the client TCP responds with an ACK. This is the first half of the TCP connection termination.
3. The SIGCHLD signal is sent to the server parent and handled correctly (Figure 5.12).
4. 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.
5. Running netstat at this point shows the state of the sockets.

```
linux % netstat -a | grep 9877
tcp 0 0 *:9877 *:* LISTEN
```

6. When we type "another line," str_cli calls writen and the client TCP sends the data to the server. This is allowed by TCP because the receipt of the FIN by the client TCP only indicates that the server process has closed its end of the connection and will not be sending any more data. The receipt of the FIN does not tell the client TCP that the server process has terminated (which in this case, it has)
7. When the server TCP receives the data from the client, it responds with an RST since the process that had that socket open has terminated.
8. The client process will not see the RST because it calls readline immediately after the call to writen and readline returns 0 (EOF) immediately because of the FIN that was

- What we have described also depends on the timing of the example.
- The client's call to readline may happen before the server's RST is received by the client, or it may happen after.
- If the readline happens before the RST is received, as we've shown in our example, the result is an unexpected EOF in the client.
- But if the RST arrives first, the result is an ECONNRESET ("Connection reset by peer") error return from readline
- The problem in this example is that the client is blocked in the call to fgets when the FIN arrives on the socket. The client is really working with two descriptors the socket and the user input and

SIGPIPE Signal

- What happens if the client ignores the error return from readline and writes more data to the server? This can happen, for example, if the client needs to perform two writes to the server before reading anything back, with the first write eliciting the RST
- The rule that applies is: When a process writes to a socket that has received an RST, the SIGPIPE signal is sent to the process. The default action of this signal is to terminate the process, so the process must catch the signal to avoid being involuntarily terminated.
- If the process either catches the signal and returns from the signal handler, or ignores the

SIGPIPE Signal

```
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);
      }
}
```

Crashing of Server Host

- This scenario will test to see what happens when the server host crashes.
- To simulate this, we must run the client and server on different hosts. We then start the server, start the client, type in a line to the client to verify that the connection is up, disconnect the server host from the network, and type in another line at the client
- This also covers the scenario of the server host being unreachable when the client sends data

Crashing of Server Host

1. When the server host crashes, nothing is sent out on the existing network connections. That is, we are assuming the host crashes and is not shut down by an operator (which we will cover in Section 5.16).
2. We type a line of input to the client, it is written by writen (Figure 5.5), and is sent by the client TCP as a data segment. The client then blocks in the call to readline, waiting for the echoed reply.
3. If we watch the network with wireshark , we will see the client TCP continually retransmitting the data segment, trying to receive an ACK from the server. Section 25.11 of TCPv2 shows a typical pattern for TCP retransmissions: Berkeley-derived implementations retransmit the data segment 12 times, waiting for around 9 minutes before giving up.
4. When the client TCP finally gives up (assuming the server host has not been rebooted during this time, or if the server host has not crashed but was unreachable on the network, assuming the host was still unreachable), an error is returned to the client process. Since the client is blocked in the call to readline, it returns an error. Assuming the server host crashed and there were no responses at all to the client's data segments, the error is ETIMEDOUT.
5. But if some intermediate router determined that the server host was unreachable and responded with an ICMP "destination unreachable' message, the error is either EHOSTUNREACH or ENETUNREACH

Crashing and Rebooting of Server Host

- In this scenario, we will establish a connection between the client and server and then assume the server host crashes and reboots. In the previous section, the server host was still down when we sent it data. Here, we will let the server host reboot before sending it data. The easiest way to simulate this is to establish the connection, disconnect the server from the network, shut down the server host and then reboot it, and then reconnect the server host to the network. We do not want the client to see the server host shut down

Crashing and Rebooting of Server Host

1. We start the server and then the client. We type a line to verify that the connection is established.
2. The server host crashes and reboots.
3. We type a line of input to the client, which is sent as a TCP data segment to the server host.
4. When the server host reboots after crashing, its TCP loses all information about connections that existed before the crash. Therefore, the server TCP responds to the received data segment from the client with an RST.
5. Our client is blocked in the call to readline when the RST is received, causing readline to return the error ECONNRESET.

5.16 Shutdown of Server Host

1. The previous two sections discussed the crashing of the server host, or the server host being unreachable across the network. We now consider what happens if the server host is shut down by an operator while our server process is running on that host.
2. When a Unix system is shut down, the init process normally sends the SIGTERM signal to all processes (we can catch this signal), waits some fixed amount of time (often between 5 and 20 seconds), and then sends the SIGKILL signal (which we cannot catch) to any processes still running. This gives all running processes a short amount of time to clean up and terminate. If we do not catch SIGTERM and terminate, our server will be terminated by the SIGKILL signal.
3. When the process terminates, all open descriptors are closed, and we then follow the same sequence of steps discussed in Section 5.12 . As stated there, we must use the select or poll function in our client to