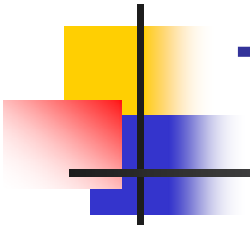
Computer Network Programming

TCP Client/Server Example

Dr. Sam Hsu
Computer Science & Engineering
Florida Atlantic University

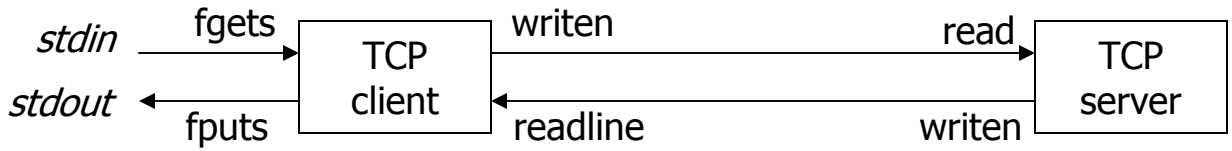


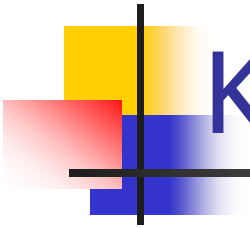
TCP Client/Server Example

- A Simple Echo Client/Server Setting
- Server Functions and Algorithms
- Client Functions and Algorithms
- Normal Operations
- Signals
- Zombies
- Restart Interrupted Slow System Calls
- Abnormal operations

Simple Echo Client/Server

- A simple client/server example that performs the following:
  - The client reads a line of text from its standard input and writes the line to the server.
  - The server reads the line from its network input and echoes the line back to the client.
  - The client reads the echoed line and displays it on its standard output.





Key Points in This Example

- Basic concepts about implementing a client/server system.
  - One may just change what the server does with the client input to expand this example to other applications.
- To consider normal as well as boundary conditions:
  - Normal and abnormal terminations.
  - Signals, interrupted system calls, server crash, etc.

TCP Echo Server main() Algorithm

- A typical fork()-based concurrent server.
- Algorithm outline:
  - Create socket.
  - Bind it to a designated port (supposedly to be a well-known port).
  - Allow incoming traffic for any local network interface (wildcard address: INADDR_ANY).
  - Convert it to a listening socket.
    - Set up a listening queue.
  - Loop around forever:
    - Block in call to accept(), wait for a client to connect.
    - Spawn a child to handle each client upon successful connection.
      - Close listening socket.
      - Execute str_echo()
    - Close connected socket for child. ⟵ parent no wait

TCP Echo Server – main() (1/2)

- *tcpcliserv/tcpserv01.c* (frist version)

```
 1  #include  "unp.h"
 2  int
 3  main(int argc, char **argv)
 4  {
 5    int          listenfd, connfd;
 6    pid_t        childpid;
 7    socklen_t      clilen;
 8    struct sockaddr_in  cliaddr, servaddr;

 9    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

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

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

15    Listen(listenfd, LISTENQ);
```

TCP Echo Server – main() (2/2)

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

19        if  ( (childpid = Fork()) == 0) {  /* child process */
20            Close(listenfd);  /* close listening socket */
21            str_echo(connfd);  /* process the request */
22            exit(0);
23        }
24        Close(connfd);      /* parent closes connected socket */
25    }
26  }
```

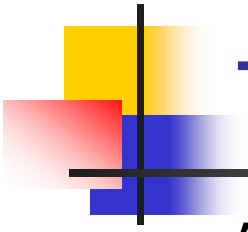
str_echo() Algorithm

- It provides very simple service for each client.
  - It reads data from a client and echoes it back to the client.
- Algorithm outline:
  - Read a buffer from the connected socket.
    - If n (number of characters read) > 0,
      - Echo back to the client (writen(): p. 89), read again.
    - Else if n < 0 & EINTR (got interrupt), read again.
    - Else just n < 0 (error occurred), display error message (and terminate child process in err_sys()).
    - Else if n = 0 (receipt of FIN from client, the normal scenario), return.

TCP Echo Server – str_echo()

- *lib/str_echo.c*

```
 1  #include    "unp.h"
 2  void
 3  str_echo(int sockfd)
 4  {
 5      ssize_t       n;
 6      char          buf[MAXLINE];
 7    again:
 8      while ( (n = read(sockfd,  buf,  MAXLINE))  >  0)
 9          Writen(sockfd,  buf,  n)
10      if ( (n  < 0  &&  errno  ==  EINTR)
11          goto  again;
12      else  if  (n  <  0)
13          err_sys("str_echo:  read  error");
14  }
```



TCP Echo Client main() Algorithm

- Algorithm outline:
  - Check number of commandline arguments.
    - It must be 2 (program name and server address).
    - Quit if not 2 (call to sys_quit()).
  - Open socket.
  - Fill in internet socket address structure.
  - Connect to server.
  - Call str_cli() to handle the rest of the client processing.
  - Exit when no more user input.
- Note: All errors end up in termination of the client in this function. Real applications may need to recover differently.

TCP Echo Client – main()

tcpcliserv/tcpcli01.c (frist version)

```
1  #include  "unp.h"
2  int
3  main(int argc, char **argv)
4  {
5     int         sockfd;
6     struct sockaddr_in  servaddr;
7     if (argc != 2)
8        err_quit("usage: tcpcli <IPaddress>");
9     sockfd = Socket(AF_INET, SOCK_STREAM, 0);
10    bzero(&servaddr, sizeof(servaddr));
11    servaddr.sin_family = AF_INET;
12    servaddr.sin_port = htons(SERV_PORT);
13    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
14    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
15    str_cli(stdin, sockfd);    /* do it all */
16    exit(0);
17 }
```

TCP Echo Client – str_cli()

- *lib/str_cli.c*

```
 1  #include  "unp.h"
 2  void
 3  str_cli(FILE *fp, int sockfd)
 4  {
 5     char  sendline[MAXLINE], recvline[MAXLINE];
 6     while (Fgets(sendline, MAXLINE, fp) != NULL) {
 7          Writen(sockfd, sendline, strlen(sendline));
 8          if (Readline(sockfd, recvline, MAXLINE) == 0)
 9               err_quit("str_cli: server terminated prematurely");
10          Fputs(recvline, stdout);
11     }
12  }
```

Normal Startup (1/3)

- To watch the sequence of client/server.
- To start the server in background:

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

- To check the status of all sockets on a system (-a) before the client starts:

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

  - Note: The output above shows only partial results, and the output format may be different from system to system.

Normal Startup (2/3)

- To start the client on the same machine (using the loopback address):

  linux% **tcpcli01 127.0.0.1**

- Then, check the status of all sockets again:

  linux% **netstat –a**

  Active Internet connections (servers and established)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	localhost:9877	localhost:42758	ESTABLISHED
tcp	0	0	localhost:42758	localhost:9877	ESTABLISHED
tcp	0	0	*:9877	*:*	LISTEN

  - Note: The first tcp connection is for the server child, and the second is for the client, and the third is the server parent.

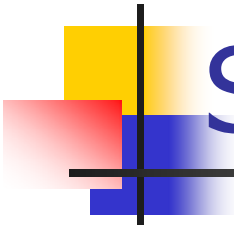
Normal Termination

- To check the socket status right after the client terminates:

```
linux% netstat –a | grep 9877
tcp        0      0 *:9877                  *:*                     LISTEN
tcp        0      0 localhost:42758         localhost:9877          TIME_WAIT
```

- To check again the process status:

```
linux% ps –t pts/6 -o pid,ppid,tty,stat,args,wchan
  PID  PPID TT       STAT COMMAND         WCHAN
22038 22036 pts/6    S    -bash           read_chan
17870 22038 pts/6    S    ./tcpserv01     wait_for_connect
19315 17870 pts/6    S    [tcpserv01 <defu do_exit
```

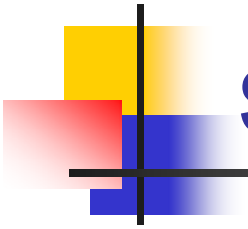


Signals (1/2)

- A signal is a notification from the kernel to a process that some event has happened.
  - It is a software interrupt.
  - Signals usually occur asynchronously.
    - A process does not know ahead of time exactly when a signal will occur.
  - Signals can be sent
    - By one process to another (or to itself) of the same UID.
    - By the kernel to any process.
- Signals are usually identified by a symbolic constant.
  - For example, SIGINT, SIGKILL, SIGCHLD, etc.
  - A complete list can be found in signal.h.

Signals (2/2)

- Each signal has a *disposition*.
  - Action associated with the signal.
- Three different dispositions for signals:
  - Default: system defined, process gets terminated in general.
  - Ignore: Signal is received, but ignored.
  - User-defined: Users may define their own signal handlers to catch and process signals.
  - Syntax: void *UserSignalHanlder*(int *signo*);
- The following signals can never be caught or ignored:
  - SIGKILL, SIGSTOP.



signal() System Calls

- Standard (historical) signal() definition

  void (*signal(int *signo*, void (**func*)(int))) (int);

- New POSIX sigaction() definition

  int sigaction(int *signo*, const struct sigaction **act,*
  struct sigaction **oact);*

- Simplified syntax of signal() by Stevens for readability.

  typedef void Sigfunc(int);

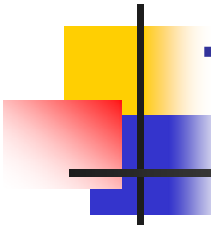
  Sigfunc *signal(int *signo*, Sigfunc **func*);

sigaction()–based signal()

- *lib/signal.c* (defined by Stevens for backward compatibility)

```
 1  #include  "unp.h"
 2  Sigfunc *
 3  signal(int signo, Sigfunc *func)
 4  {
 5      struct sigaction  act,  oact;
 6      act.sa_handler  = func;
 7      sigemptyset(&act.sa_mask);
 8      act.sa_flags  = 0;
 9      if (signo  ==  SIGALRM) {
10  #ifdef  SA_INTERRUPT
11          act.sa_flags  |= SA_INTERRUPT;   /* SunOS 4.x */
12  #endif
13      } else {
14  #ifdef  SA_RESTART
15          act.sa_flags  |=  SA_RESTART;     /* SVR4, 44BSD */
16  #endif
17      }
18      if (sigaction(signo,  &act, &oact) < 0)
19          return(SIG_ERR);
20      return(oact.sa_handler);
21  }
```

The SIGCHLD Signal



- Whenever a process finishes execution, its parent will be notified by the kernel via the SIGCHLD signal.
  - It is generated automatically.
  - The parent process will be interrupted.
  - The parent may choose to either ignore, go by system default, or catch and handle the signal.
- The terminated child process may result in a *zombie* state if its parent does not handle the SIGCHLD signal properly.
  - Information kept in a zombie state include PID, termination status, resource utilization (CPU time, memory use, etc.) of the child.
- A zombie takes up space in the kernel.
  - One may run out of space if zombies are not handled in time.

wait()/waitpid() Functions

- Are used by the parent to wait for a child (or a specific child) process to terminate.
  - One way (better way) to avoid the child become a zombie.

```
#include <sys/wait.h>

pid_t wait(int pid,  int *statloc);

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

Both return: process ID if OK,0 or -1 on error

wait()–based SIGCHLD Signal Handler

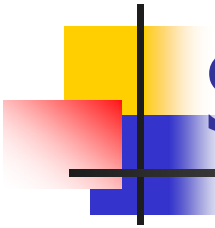
- *tcpcliserv/sigchldwait.c*

```
 1  #include  "unp.h"
 2  void
 3  sig_chld(int signo)
 4  {
 5      pid_t  pid;
 6      int    stat;
 7      pid = wait(&stat);
 8      printf("child %d terminated\n", pid);
 9      return;
10  }
```

waitpid()–based SIGCHLD Signal Handler

- *tcpcliserv/sigchldwaitpid.c*

```
 1  #include  "unp.h"
 2  void
 3  sig_chld(int signo)
 4  {
 5      pid_t  pid;
 6      int    stat;
 7      while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
 8          printf("child %d terminated\n", pid);
 9      return;
10  }
```



Slow System Calls

- System calls are programming interface to kernel service.
  - They are function calls.
- A slow system call is any system call that can block for an undetermined period of time.
  - It may never return.
    - For example, accept() may never return, if no client requests for connection.
  - Most networking functions fall into this category.
- If a process catches a signal, while it is being blocked in a slow system call, and the signal handler returns, the interrupted system may return *–1* with errno set to EINTR.
  - This may cause problems if not handled properly.
    - For example, returning -1 from accept() is considered an error.
  - In need of restarting interrupted slow system calls.

Restart Interrupted System Calls

- Interrupted system calls can be restarted, in general, by setting up a restart flag in a signal handler.
  - SA_INTERRUPT (SunOS 4.x) or SA_RESTART (SVR4, 4.4BSD).
- However, for slow system calls, one may need to do something more, since some kernel implementations may not restart them automatically.
  - A simple solution is to place the slow system call in a loop and ignore its error return (-1) if EINTR is set at the same time.
  - This mechanism works fine for a lot of slow system calls such as accept(), read(), write(), select(), open(), etc.
  - However, connect() can't be handled this way.
    - Need to use select() to help.
- This issue needs attention since a server may be executing a slow system call when a child finishes.

Multiple Connections from A Client

- An example showing one server with five connections from the same client.
  - Server source code: tcpserv03.c
  - Client source code: tcpcli04.c

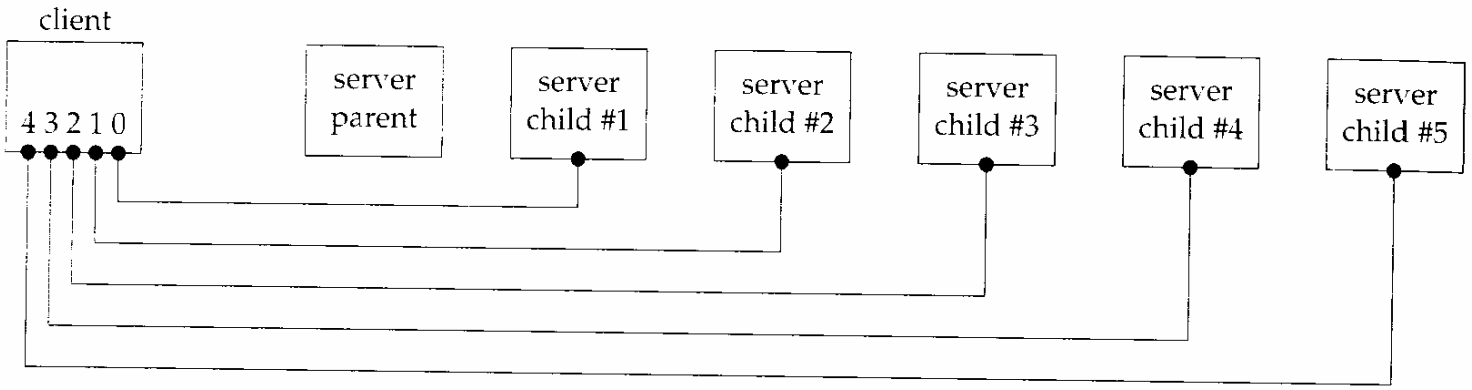


Figure 5.8 Client with five established connections to same concurrent server.

TCP Echo Client – main()

- *tcpcliserv/tcpcli04.c*

```
 1  #include  "unp.h"
 2  int
 3  main(int argc, char **argv)
 4  {
 5      int         i, sockfd[5];
 6      struct sockaddr_in  servaddr;
 7      if (argc != 2)
 8          err_quit("usage: tcpcli <IPaddress>");
 9      for (i = 0; i < 5; i++) {
10          sockfd[i] = Socket(AF_INET, SOCK_STREAM, 0);
11          bzero(&servaddr, sizeof(servaddr));
12          servaddr.sin_family = AF_INET;
13          servaddr.sin_port = htons(SERV_PORT);
14          Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
15          Connect(sockfd[i], (SA *) &servaddr, sizeof(servaddr));
16      }
17      str_cli(stdin, sockfd[0]);    /* do it all */
18      exit(0);
19  }
```

Sample Run

- To run the server in the background, and then start the client.

```
linux%  tcpserv03
[1] 20419
linux%  tcpcli04 127.0.0.1
Hello                          # user input in bold
Hello                          # echoed back from server
^D                             # Type Ctrl-D to terminate input
Child 20426 terminated         # output by server
```

- Then, type **ps** to check the process status:

```
PID     TTY      TIME      CMD
20419   pts/6    00:00:00  tcpserv03
20421   pts/6    00:00:00  tcpserv03  <defunct>
20422   pts/6    00:00:00  tcpserv03  <defunct>
20423   pts/6    00:00:00  tcpserv03  <defunct>
```

  - One may notice that there are several zombies (defunct) there.

Client Termination

- When the client terminates, all 5 connections are terminated at about the same time.
  - Just an example for illustration purpose, may not be practical.

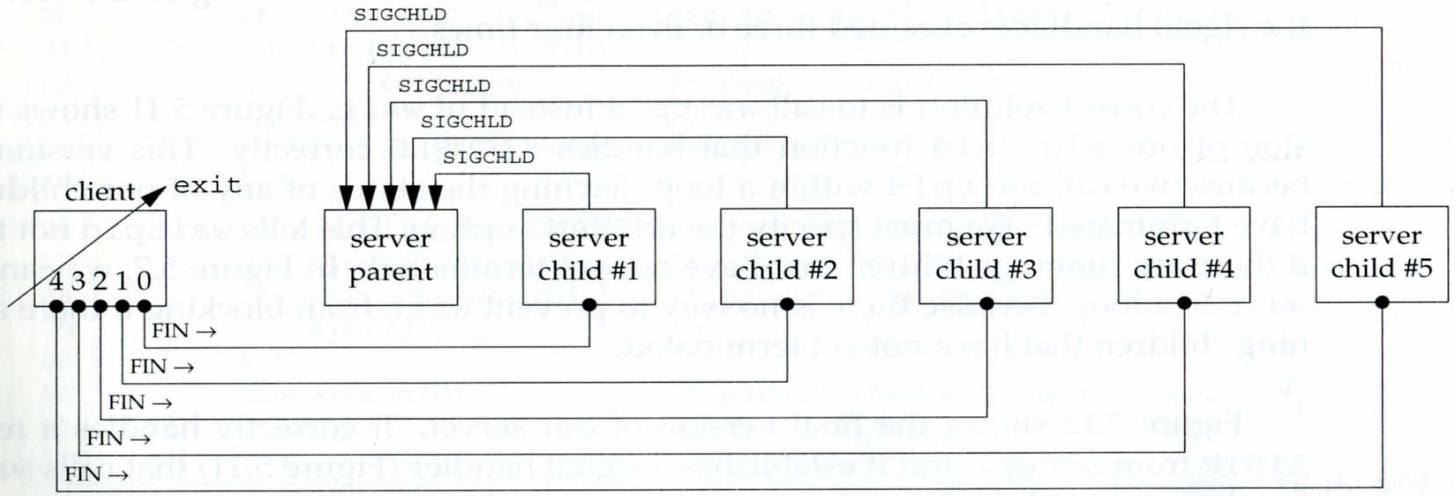


Figure 5.10 Client terminates, closing all five connections, terminating all five children.

Final TCP Server – main() (1/2)

- *tcpcliserv/tcpserv04.c* (final version)

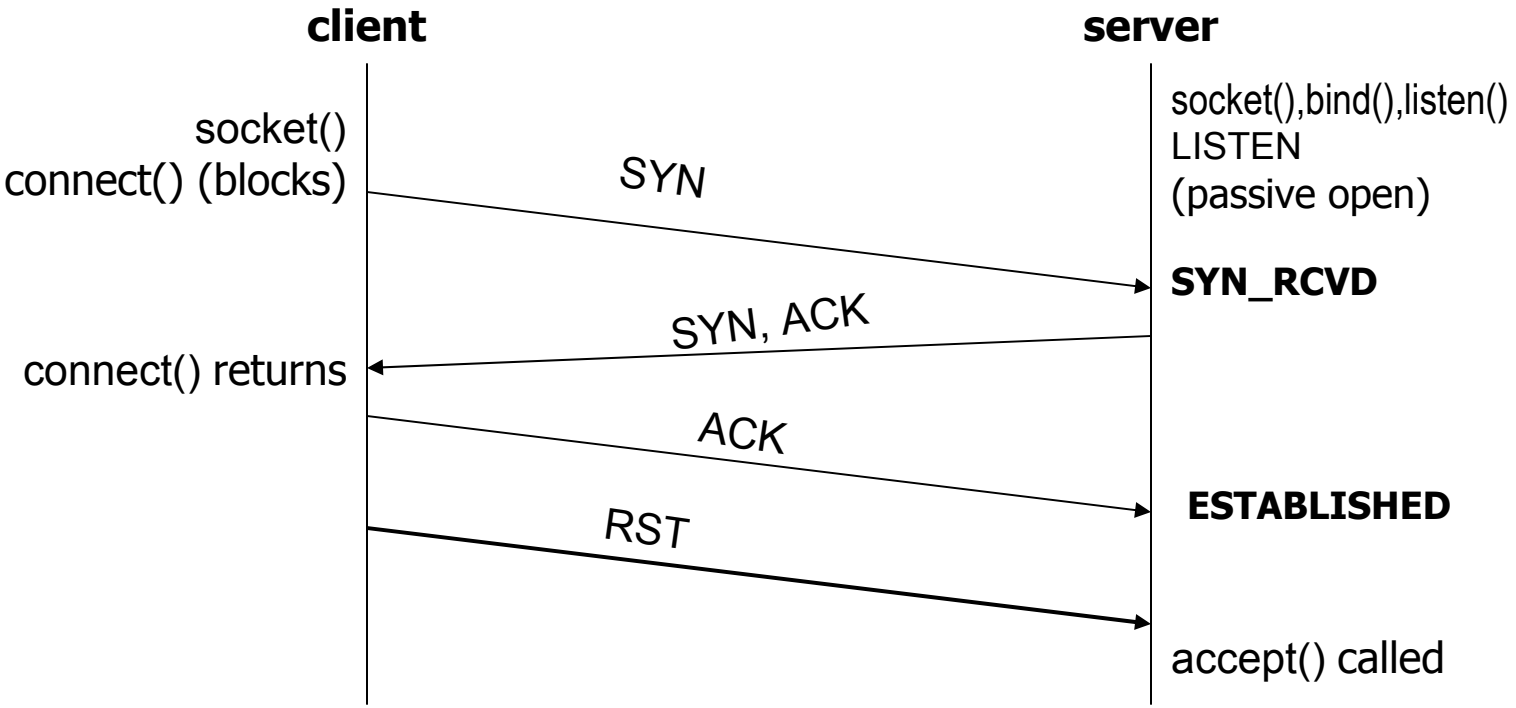
```
 1  #include  "unp.h"
 2  int
 3  main(int argc, char **argv)
 4  {
 5      int          listenfd, connfd;
 6      pid_t        childpid;
 7      socklen_t      clilen;
 8      struct sockaddr_in  cliaddr, servaddr;
 9      void         sig_chld(int);
10      listenfd = Socket(AF_INET, SOCK_STREAM, 0);
11      bzero(&servaddr, sizeof(servaddr));
12      servaddr.sin_family      = AF_INET;
13      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14      servaddr.sin_port        = htons(SERV_PORT);
15      Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
```

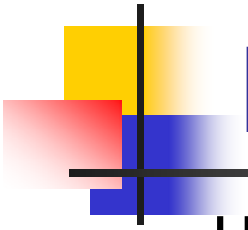
Final TCP Server – main() (2/2)

```
16      Listen(listenfd, LISTENQ);
17      Signal(SIGCHLD, sig_chld);  /* must call waitpid() */
18      for ( ; ; ) {
19          clilen = sizeof(cliaddr);
20          if ( (connfd = accept(listenfd, (SA *) &cliaddr, &clilen)) < 0) {
21              if (errno == EINTR)
22                  continue;    /* back to for */
23              else
24                  err_sys("accept error");
25          }
26          if ( (childpid = Fork()) == 0) {  /* child process */
27              Close(listenfd);  /* close listening socket */
28              str_echo(connfd);  /* process the request */
29              exit(0);
30          }
31          Close(connfd);      /* parent closes connected socket */
32      }
33  }
```

Abort Before accept() Returns

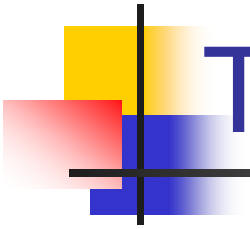
- Receiving an RST from client for an ESTABLISHED connection before accept() is called.





Handling Aborted Connections

- Handling of the aborted connection described above is implementation-dependent.
  - Berkeley-derived implementations handle the aborted connections within the server, and the server process may never see it.
    - accept() does not return.
  - SVR4 implementations return an error to the process when accept() returns. However, depending on implementations, either of the following may happen:
    - accept() returns an errno of EPRTO (protocol error).
    - accept() returns an errno of ECONNABORTED (POSIX).
  - POSIX specifies the return to be ECONNABORTED.
    - *Software caused connection abort*.
    - The server can ignore the error and call accept() again.



Termination of TCP Server

- There are two scenarios.
  - Crashing of the TCP server process.
    - What if the client continues to write to a socket which is closed due to the termination of the server process?
  - Crashing of the TCP server host.
    - Server host crashes, and is unreachable.
    - Server host crashes, but gets rebooted.
    - Server host is shut down by sysadm.

Crashing of Server Process

- Is the client aware of it?
- Procedure:
  - Terminating the server child causes the server TCP to send a FIN to the client.
    - The client TCP responds with an ACK.
    - (The client process is blocked in fgets() waiting for user input).
    - TCP is then half-close.
  - SIGCHLD is sent to the server parent and handled correctly (due to Signal(SIGCHLD, sig_chld)).
  - The client process calls Writen() to send data to the server, and calls Readline() immediately.
  - The server TCP responds with an RST in response to the write.
  - The client process returns from Readline():
    - With an unexpected EOF (because of FIN), if RST is not received yet.
    - With ECONNRESET (connection reset by peer) if RST is already received.
  - The client process then terminates.

SIGPIPE Signal

- What if the client process ignores the error returned from Readline() and proceeds to write more data to its socket? (See *lib/str_cli.c)*
  - The SIGPIPE signal will be sent to the client process by the client kernel after it has received an RST.
  - If SIGPIPE is not caught, the client process will terminate by default with no output.
  - If the process catches SIGPIPE, but returns from the signal handler, or ignores the signal, and proceeds again, the next write operation returns EPIPE.

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.

str_cli() – Calling writen() Twice

- *tcpcliserv/str_cli11.c*

```
 1  #include  "unp.h"
 2  void
 3  str_cli(FILE *fp, int sockfd)
 4  {
 5      char  sendline[MAXLINE], recvline[MAXLINE];
 6      while (Fgets(sendline, MAXLINE, fp) != NULL) {
 7          Writen(sockfd, sendline, 1);
 8          sleep(1);
 9          Writen(sockfd, sendline+1, strlen(sendline)-1);
10          if (Readline(sockfd, recvline, MAXLINE) == 0)
11              err_quit("str_cli: server terminated prematurely");
12          Fputs(recvline, stdout);
13      }
14  }
```

Crashing of Server Host

- What if the client is blocked in Readline() , but the server host has crashed, or unreachable due to some network problems?
  - The client TCP will continuously retransmit the data segment for 12 times, waiting for around 9 minutes before giving up (BSD implementations).
  - 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.

Shutdown of Server Host

- What happens if the TCP server host is shut down by its sysadm personnel?
  - The init process on the server host will first send SIGTERM to all processes on the system, including the TCP server process.
    - This signal can be caught.
  - After waiting for about 5-20 seconds, init will then send SIGKILL to all processes.
    - This signal can not be caught.
  - The server process will close all open descriptors before the system shuts down.
    - A FIN will thus be sent to the client process.
  - The client process will then return from Readline() with EOF.

Rebooting of the Server Host

- What if the client is blocked in Readline() , but the server host has rebooted from the previous crash?
  - Unaware of the server situation, the TCP client will continue to send the same data segment again.
  - Upon receiving a data segment from the client, the server TCP will respond with an RST.
  - The client process will then return from Readline() with the error ECONNRESET.

TCP Client/Server – Client's Perspective

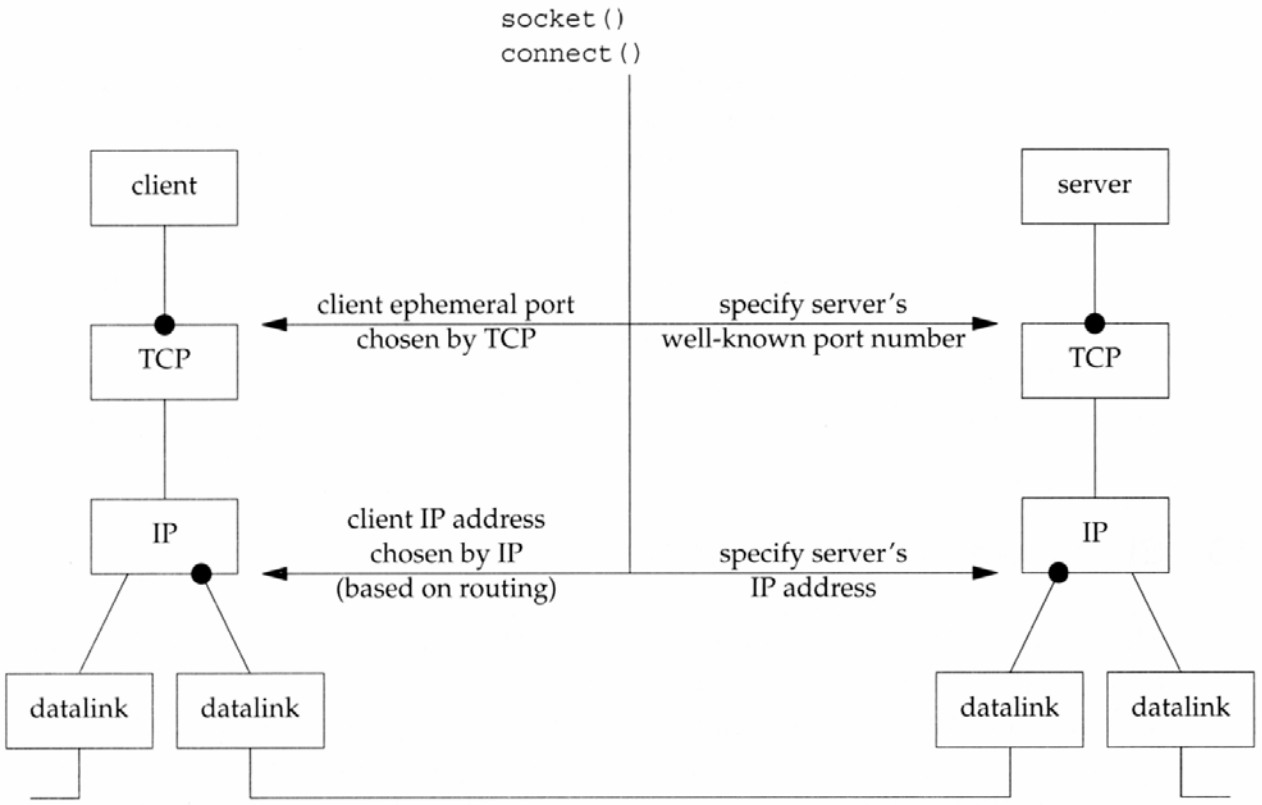


Figure 5.15 Summary of TCP client/server from client's perspective.

TCP Client/Server – Server's Perspective

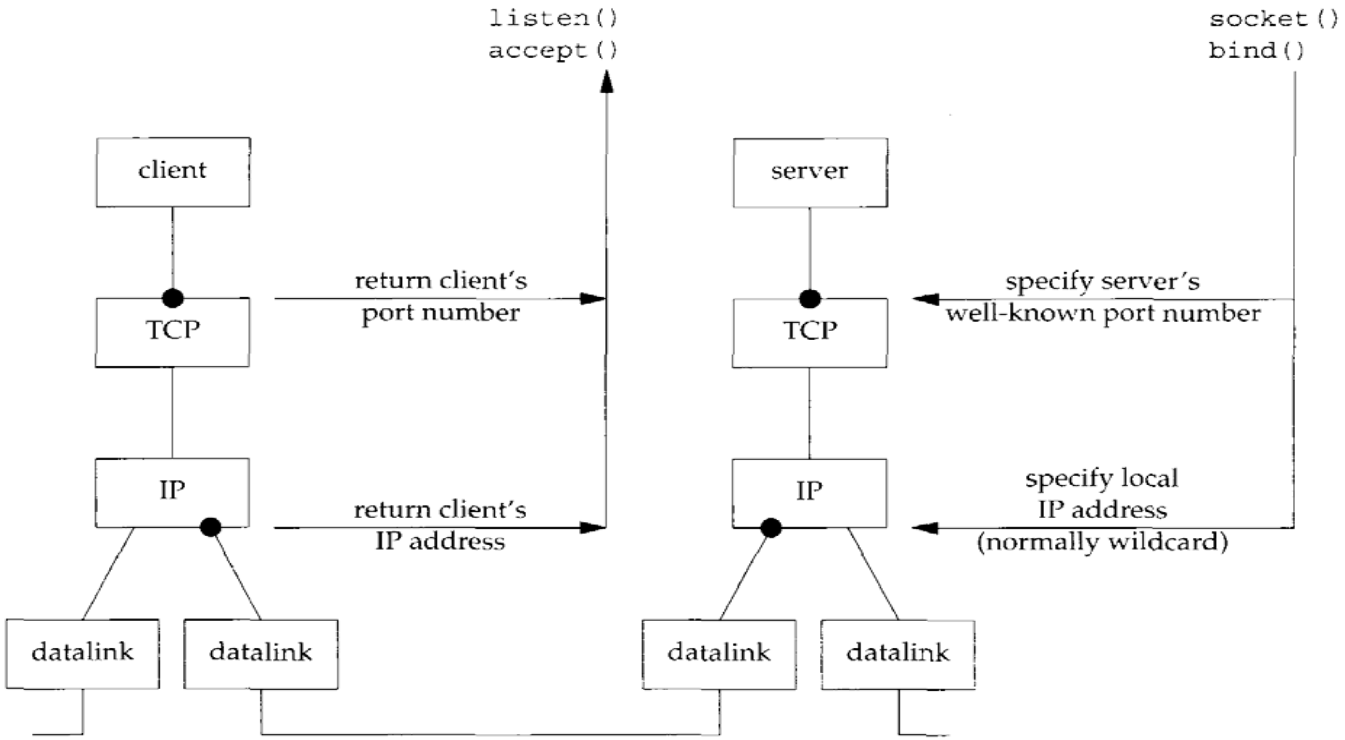


Figure 5.16 Summary of TCP client/server from server's perspective.

str_echo() – Adding 2 Numbers

- *tcpcliserv/str_echo08.c*

```
 1  #include  "unp.h"
 2  void
 3  str_echo(int sockfd)
 4  {
 5      long    arg1, arg2;
 6      ssize_t    n;
 7      char    line[MAXLINE];
 8      for ( ; ; ) {
 9          if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
10              return;    /* connection closed by other end */
11          if (sscanf(line, "%ld%ld", &arg1, &arg2) == 2)
12              snprintf(line, sizeof(line), "%ld\n", arg1 + arg2);
13          else
14              snprintf(line, sizeof(line), "input error\n");
15          n = strlen(line);
16          Writen(sockfd, line, n);
17      }
18  }
```

str_cli() – Sending 2 Binary Int's

- *tcpcliserv/str_cli09.c*

```
 1  #include  "unp.h"
 2  #include  "sum.h"
 3  void
 4  str_cli(FILE *fp, int sockfd)
 5  {
 6      char      sendline[MAXLINE];
 7      struct args    args;
 8      struct result  result;
 9      while (Fgets(sendline, MAXLINE, fp) != NULL) {
10          if (sscanf(sendline, "%ld%ld", &args.arg1, &args.arg2) != 2) {
11              printf("invalid input: %s", sendline);
12              continue;
13          }
14          Writen(sockfd, &args, sizeof(args));
15          if (Readn(sockfd, &result, sizeof(result)) == 0)
16              err_quit("str_cli: server terminated prematurely");
17          printf("%ld\n", result.sum);
18      }
19  }
```

str_echo() – Adding 2 Binary Int's

- *tcpcliserv/str_echo09.c*

```
 1  #include  "unp.h"
 2  #include  "sum.h"
 3  void
 4  str_echo(int sockfd)
 5  {
 6      ssize_t      n;
 7      struct args    args;
 8      struct result  result;
 9      for ( ; ; ) {
10          if ( (n = Readn(sockfd, &args, sizeof(args))) == 0)
11              return;    /* connection closed by other end */
12          result.sum = args.arg1 + args.arg2;
13          Writen(sockfd, &result, sizeof(result));
14      }
15  }
```

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:

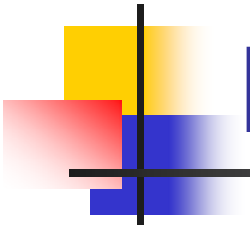
```
solaris%  tcpcli09  12.106.32.254
11  12                          # user input in bold
33                              # result back from server

-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!
```



Reading Assignment

- Read Chapter 5.