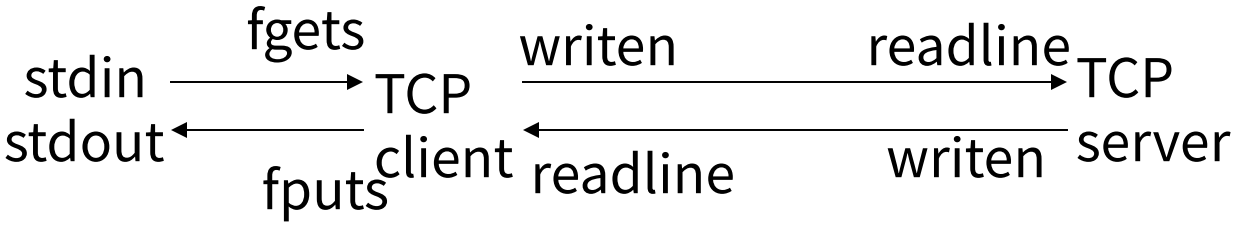
TCP Client-Server Example

- TCP echo server: *main* and *str_echo*
- TCP echo client: *main* and *str_cli*
- Normal startup and termination
- POSIX signal handling
- Handling SIGCHILD, interrupted system calls, and preventing zombies
- Connection abort before *accept* returns
- Crash of server process

- SIGPIPE signal
- Crash, reboot, shutdown of server host
- Summary of TCP example
- Data format: passing string or binary

TCP Echo Server and Client



To expand this example to other applications, just change what the server does with the client input.

Many boundary conditions to handle: signal, interrupted system call, server crash, etc. The first version does not handle them.

TCP Echo Server: main function

tcpcliserv/tcpserv01.c

```
#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 socke
t */
    }
}
```

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);
      }
}
```

TCP Echo Client: main function

tcpcliserv/tcpcli01.c

```
#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);          }
```

TCP Echo Client: str_cli function

lib/str_cli.c

```
#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);
      }
}
```

Normal Startup and Termination

watching the sequences in client-server and TCP internals

Startup: socket, bind, listen, accept, connect, str_cli, fgets, accept, fork, str_ec
Termination: fgets, str_cli, exit, readline, str_echo, exit

To check the status of all sockets on a system: netstat -a
Assume server (background) and client are run on the same host.

After the server starts but before the client starts:

```
Proto  Recv-Q  Send-Q  Local Address    Foreign Address   (state)
tcp    0       0       *.9877           *.*               LISTEN
```

After the client starts:

```
Proto       Local Address      Foreign Address (state)
tcp    0  0  localhost.9877     localhost.1052  ESTABLISHED (server child)
tcp    0  0  localhost.1052     localhost.9877  ESTABLISHED (client)
tcp    0  0  *.9877             *.*             LISTEN (server parent)
```

Normal Startup and Termination (cont.)

To check the process status: ps -l

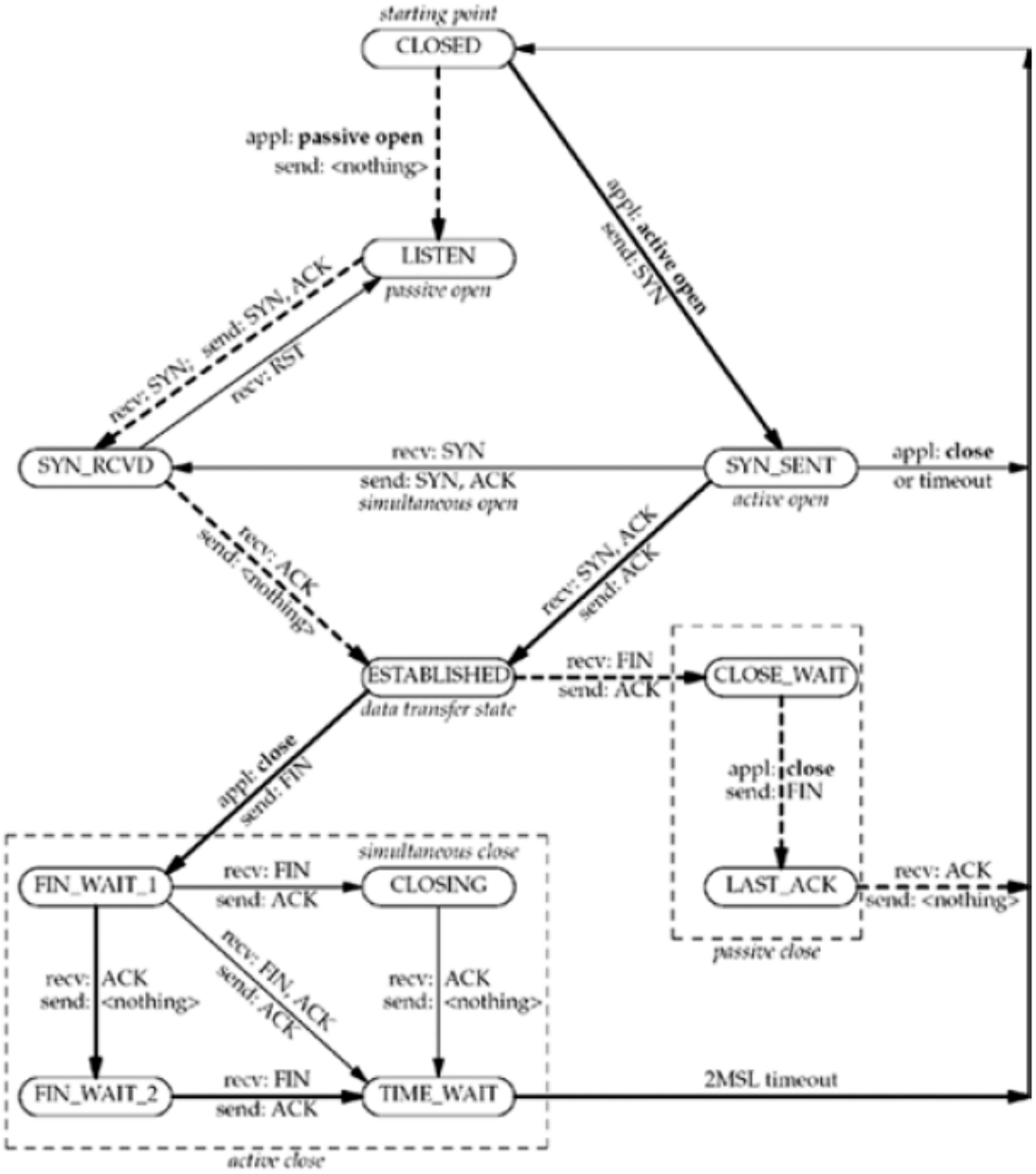
pid	ppid	WCHANSTAT	TT	TIME	COMMAND
19130	19129	wait	Is	p1	0:04.00 -ksh (ksh)
21130	19130	netcon	I	p1	0:00.06 tcpserv (server parent)
21131	19130	ttyin	I+	p1	0:00.09 tcpcli 127.0.0.1
21132	21130	netio	I	p1	0:00.01 tcpserv (server child)
21134	21133	wait	Ss	p2	0:03.50 -ksh (ksh)
21149	21134	-	R+	p2	0:00.05 ps -l

Right after the client terminates:

```
tcp    0    0  localhost.1052      localhost.9877  TIME_WAIT (client)
tcp    0    0  *.9877              *.*             LISTEN (server parent)
```

pid	TT	STAT	TIME	COMMAND
19130	p1	Ss	0:05.08	-ksh (ksh)
21130	p1	I	0:00.06	./tcpserv
21132	p1	Z	0:00.00	(tcpserv) (zombie server child process)
21167	p1	R+	0:00.10	ps

Figure 2.4. TCP state transition diagram.



1. When we type our EOF character, `fgets` returns a null pointer and the function `str_cli` (Figure 5.5) returns.

2. When `str_cli` returns to the client `main` function (Figure 5.4), the latter terminates by calling `exit`.

3. 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 (Figures 2.4 and 2.5).

4. When the server TCP receives the FIN, the server child is blocked in a call to `readline` (Figure 5.3), and `readline` then returns 0. This causes the `str_echo` function to return to the server child `main`.

5. The server child terminates by calling `exit` (Figure 5.2).

6. 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 (Figure 2.5). At this point, the connection is completely terminated. The client socket enters the TIME_WAIT state.

7. Finally, 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,

POSIX Signal Handling

- Signal (software interrupt): sent by one process to another process (or to itself) or by the kernel to a process
- SIGCHLD: by the kernel to the parent
- Disposition of a signal:
  - catch the signal by a specified signal handler
  - SIG_IGN: ignore it
  - SIG_DFL: default: terminate or ignore
- To enable automatic restart of an interrupted system call by the kernel -- write our own *signal* function

Simplify function prototype using `typedef`

2-3 The normal function prototype for `signal` is complicated by the level of nested parentheses.

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

To simplify this, we define the `Sigfunc` type in our `unp.h` header as

```
typedef  void  Sigfunc(int);
```

stating that signal handlers are functions with an integer argument and the function returns nothing (`void`). The function prototype then becomes

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

A pointer to a signal handling function is the second argument to the function, as well as the return value from the function.

signal Function That Enables System Call Restart

lib/signal.c

```
#include        "unp.h"
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);  }
```

signal Function That Enables System Call Restart (cont

```
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);
}
```

POSIX signal semantics:

1. Once a signal handler is installed, it remains installed.
2. While a signal handler is executing, the signal being delivered is blocked.
3. By default, signals are not queued. That is, if a signal is generated one or more times while it is blocked, it is normally delivered only one time after the signal is unblocked.

Handling SIGCHILD, Interrupted System Calls, and Preventing Zombies

- Ignored SIGCHLD --> zombie server child
- To catch SIGCHLD: call wait or waitpid in handler
- Interrupted slow system call (*accept*) in parent: EINTR returned; abort process if not handled
- Some kernels automatically restart some interrupted system calls, while some don't. For portability:

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

Version of SIGCHLD Handler That Calls wait

tcpcliserv/sigchldwait.c

```
#include      "unp.h"

void
sig_chld(int signo)
{
      pid_t   pid;
      int     stat;

      pid = wait(&stat);
      printf("child %d terminated\n", pid);
      return;
}
```

wait and waitpid Functions
cleaning up zombies

```
#include <sys/wait.h>
pid_t wait (int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options);
                    returns: process ID if OK, o, or -1 on error
```

Difference between wait and waitpid:
(Consider a client that establishes five connections with server.)

Because signals are not queued, the signal handler is executed once (if client and server run on the same host), twice or more (depending on the timing of FINs arriving at the server host).

Result: four or less zombies left
Solution: call *waitpid* (non-blocking) within a loop

Client Terminates All Five Connections
catching all SIGCHLD signals in server parent

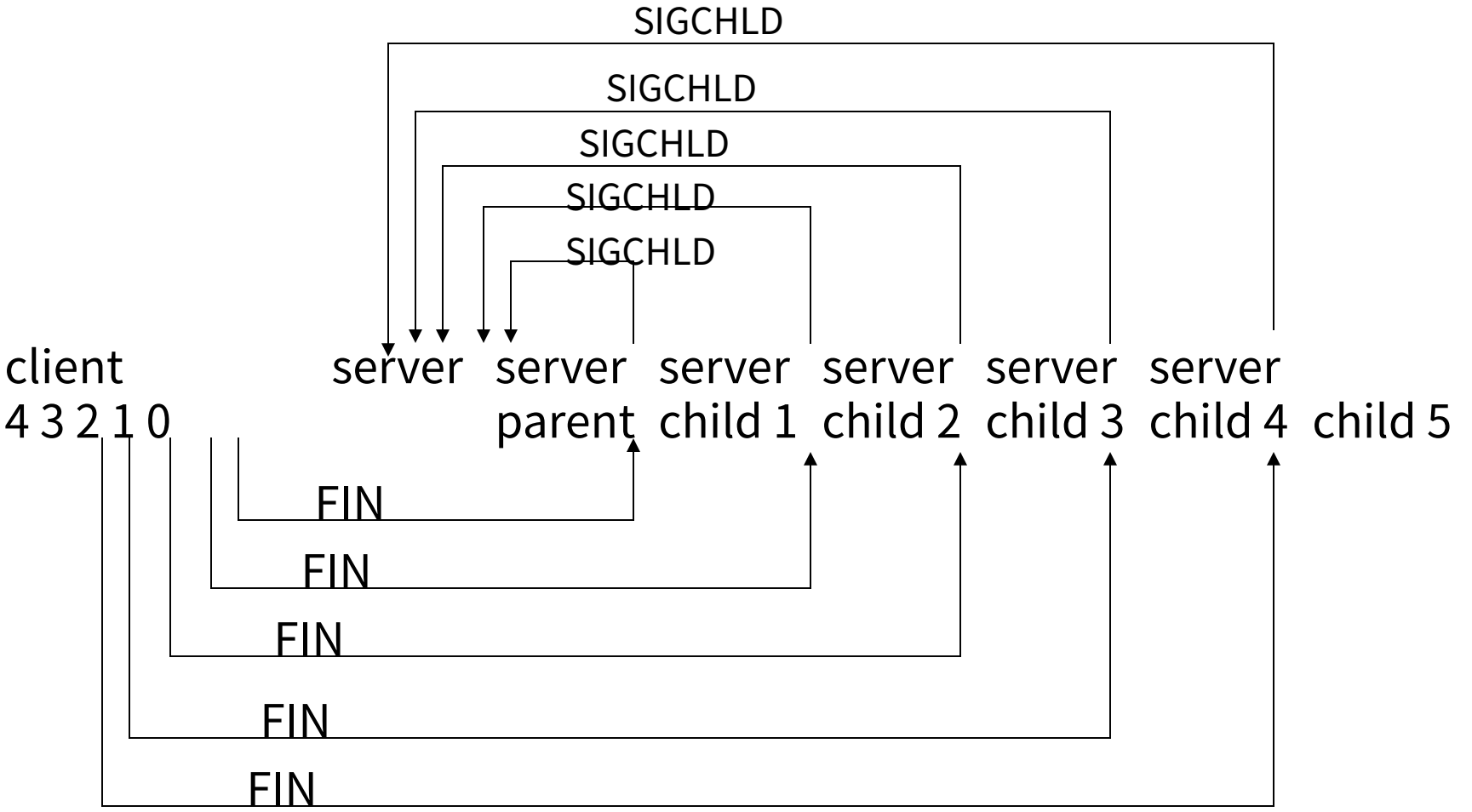


Figure 5.9 TCP client that establishes five connections with server.

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 }
```

Final (correct) Version of TCP Echo Server
handling SIGCHLD, EINTR from accept, zombies

tcpcliserv/tcpserv04.c

```
#include      "unp.h"
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);
```

Final (correct) Version of TCP Echo Server (cont.

tcpcliserv/tcpserv04.c

```
    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 socke
t */
        }
}
```

Final (correct) Version of sig_chld Function That Calls waitpid

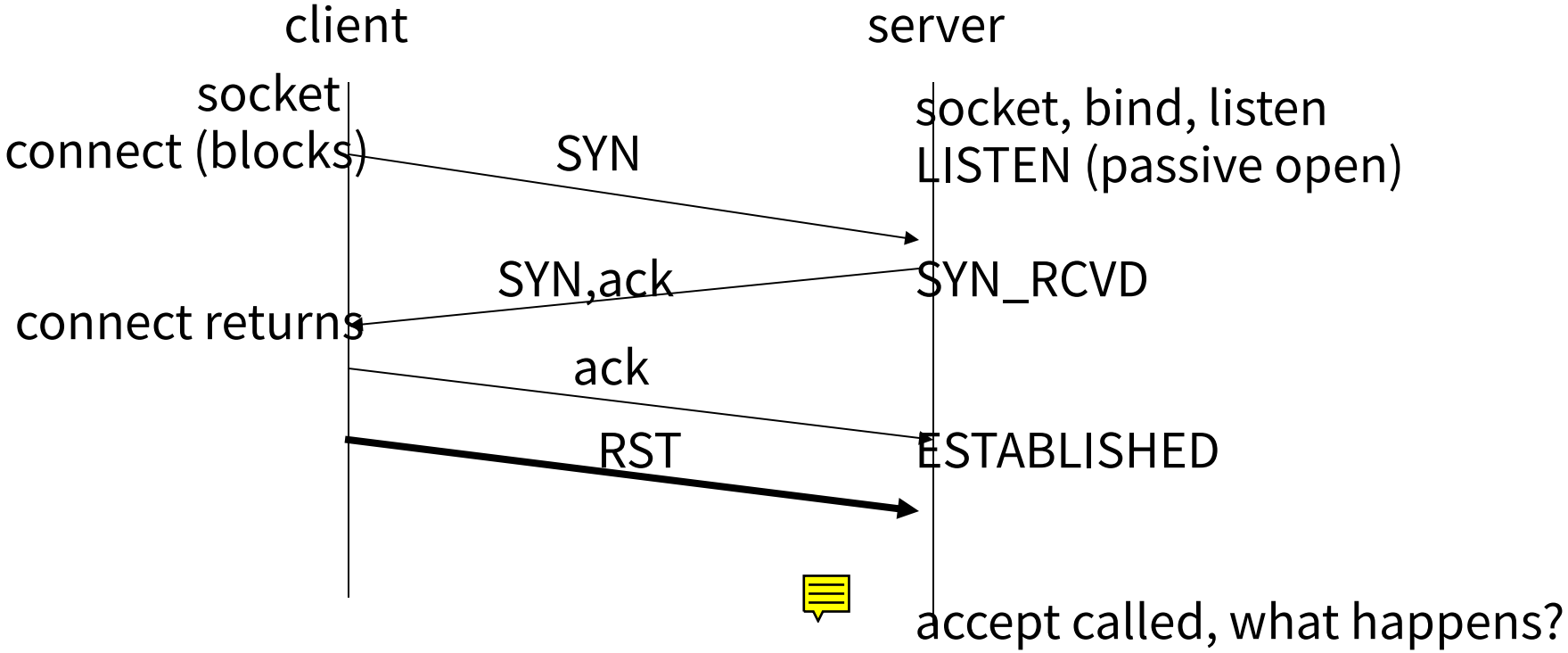
tcpcliserv/sigchldwaitpid.c

```
#include      "unp.h"

void
sig_chld(int signo)
{
      pid_t   pid;
      int           stat;

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

Connection Abort Before *accept* Returns
implementation dependent !



In BSD, kernel handles this. *accept* does not return.
In SVR4, *accept* is returned with EPROTO.
In POSIX.1g, *accept* is returned with ECONNABORTED.

Crashing of Server Process

Is the client aware of ?

Procedure:

1. Server TCP sends FIN to client TCP, which responds with an ACK.
   (TCP half-close)
   (The client process is blocked in *fgets* when client TCP receives FIN.)
2. SIGCHLD signal is sent to the server parent.
3. The client process calls *writen* to send data to server.
4. The server TCP responds with an RST.
5. The client process returns from *readline*, 0, when client TCP receives RST.
6. The client process terminates.

Problem:

The client should be aware of server process crash when FIN is received.

Solution:

Use *select* or *poll* to block on either socket or stdio.

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:

Catch the SIGPIPE signal for further processing. The write operation returns EPIPE.

Figure 5.14 `str_cli` that calls `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 }
```

Crash, Reboot, Shutdown of Server Host

- Crash of server host:
  - client TCP continuously retx data and timeout around 9 min
  - readline returns ETIMEDOUT or EHOSTUNREACH
  - To quickly detect: timeout on readline, SO_KEEPALIVE socket option, heartbeat functions
- Reboot of server host:
  - After reboot, server TCP responds to client data with an RST
  - readline returns ECONNRESET
- Shutdown (by operator) of server host:
  - init process sends SIGTERM to all processes
  - init waits 5-20 sec and sends SIGKILL to all processes

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

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

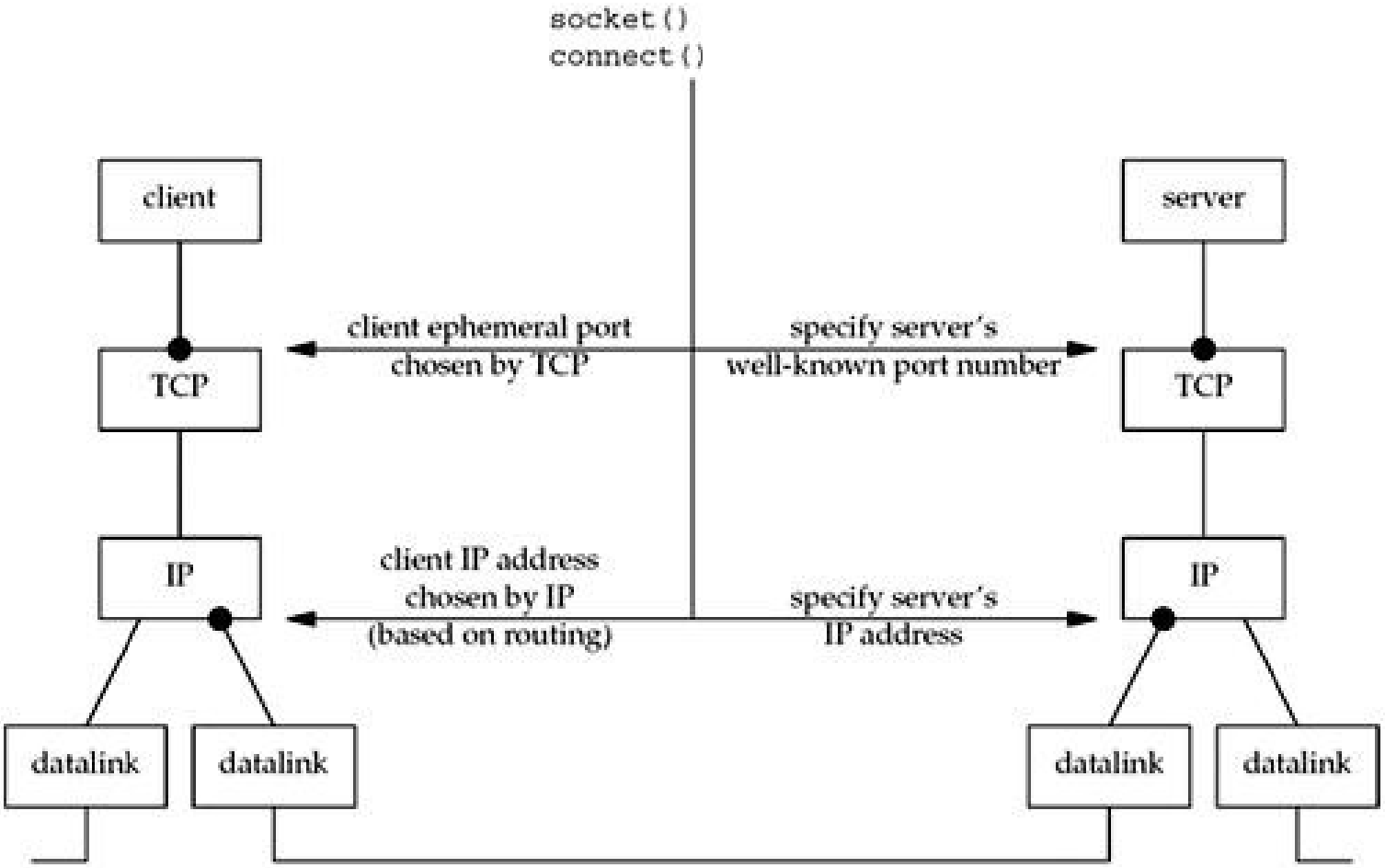
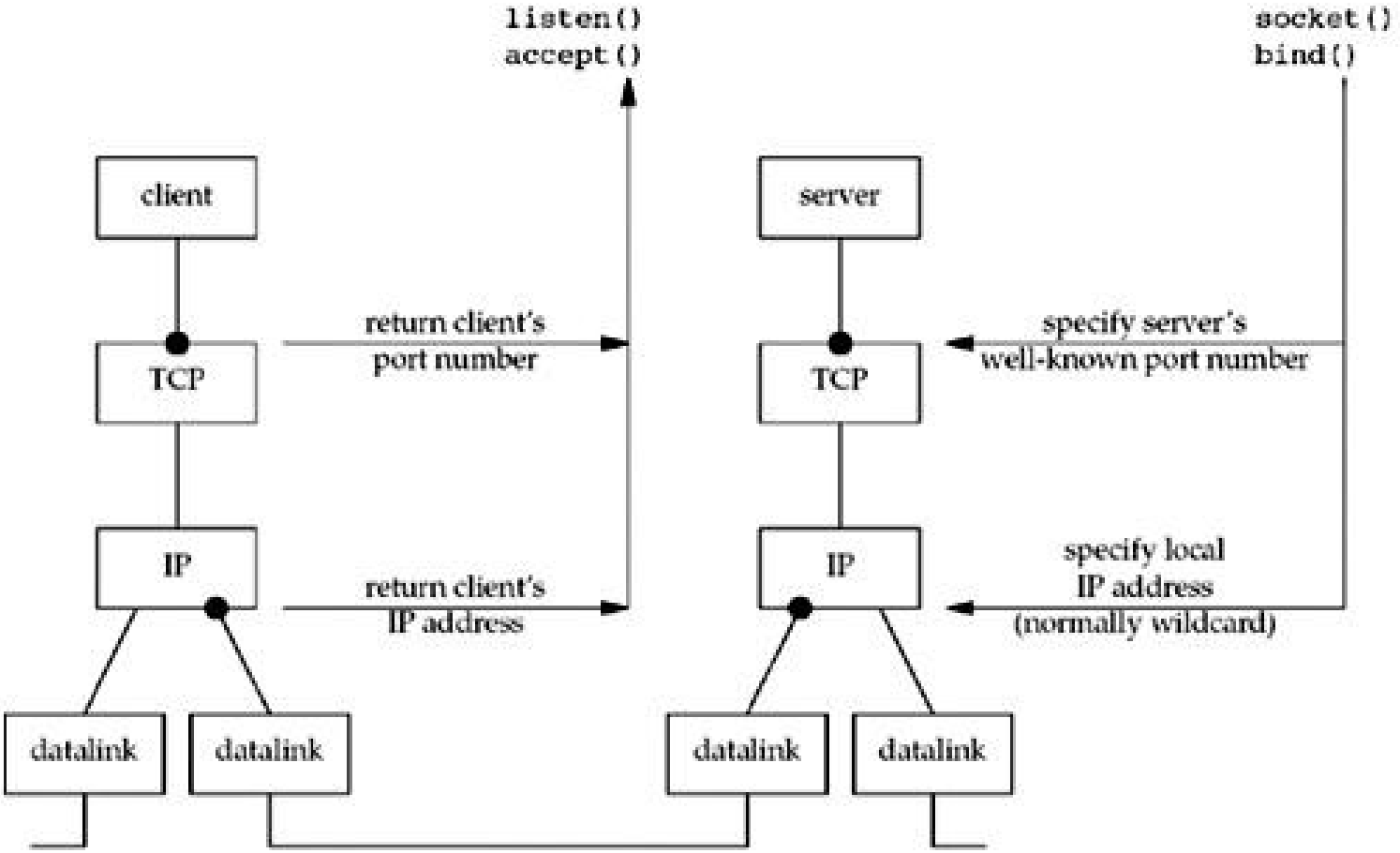


Figure 5.16. Summary of TCP client/server from server's perspective.



Data Format

string or binary between client and server

- In server process, add two numbers from client:
  - In str_echo: sscanf converts string to long integer, snprintf converts long back to string
- Pass 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
- Different implementions store binary, C datatype, structure differently.
- Suggestions:
  - pass string only
  - explicitly define the format of datatypes (e.g. RPC's XDR -- external data representation)

Passing text string

Figure 5.17 `str_echo` function that adds two numbers.

tcpcliserv/str_ech08.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 }
```

Passing binary value

Figure 5.19 `str_cli` function which sends two binary integers to server.

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 }
```

```
1 struct args {
2     long    arg1;
3     long    arg2;
4 };

5 struct result {
6     long    sum;
7 };
```

Figure 5.18 sum.h header.

Figure 5.20 `str_echo` function that adds two binary integers.

tcpcliserv/str_ech09.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 }
```