Writing Client/Server Programs in C Using Sockets (A Tutorial) - Part II

Session 5955

Gary T. Ciampa

sasgyc@unx.sas.com

SAS/C & C++ Support

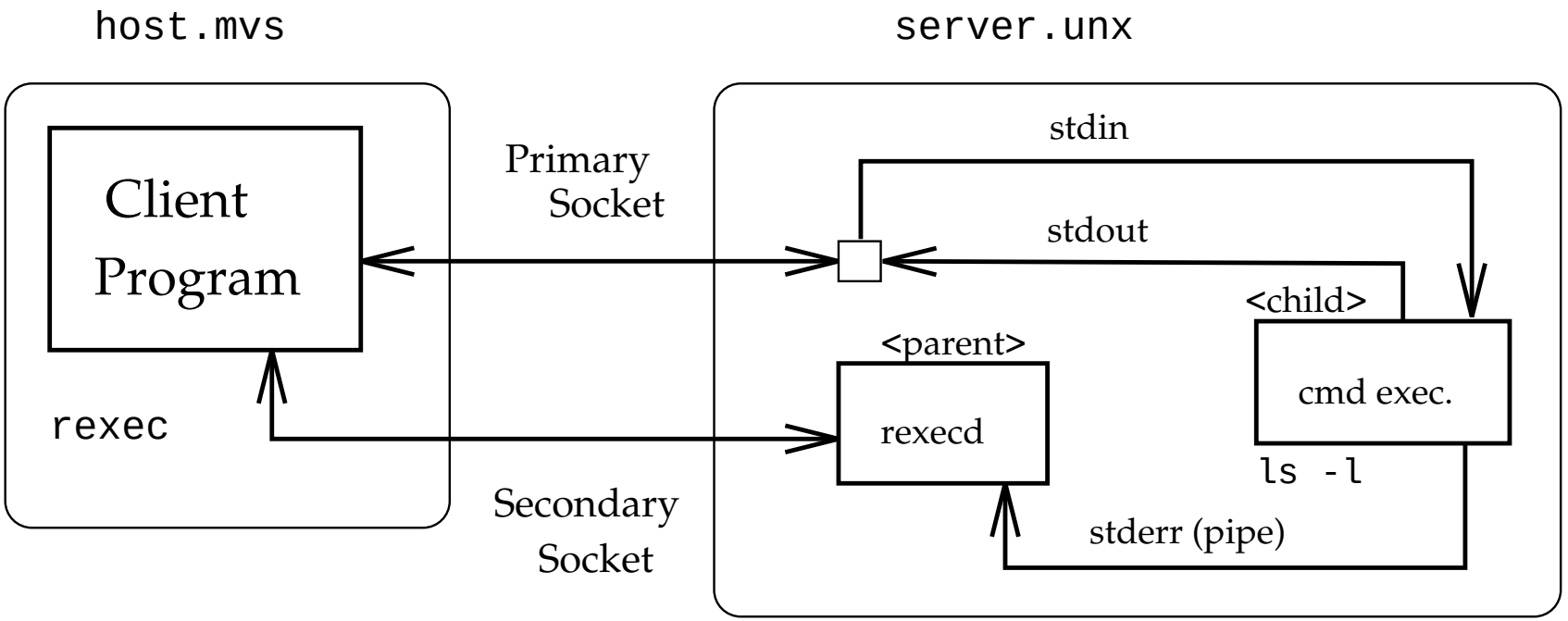
SAS Institute

Cary, NC

❍ ❍ Remote Execution Sample Program ❍ ❍

- Remote Execution Connection Diagram
- getservbyname() coding
- rexec() function declaration
- gethostbyname() and struct hostent
- rexec() caller source code examination
- System call return conventions
- Socket address structures and conventions
- read() and write() conventions
- File descriptor sets and select()

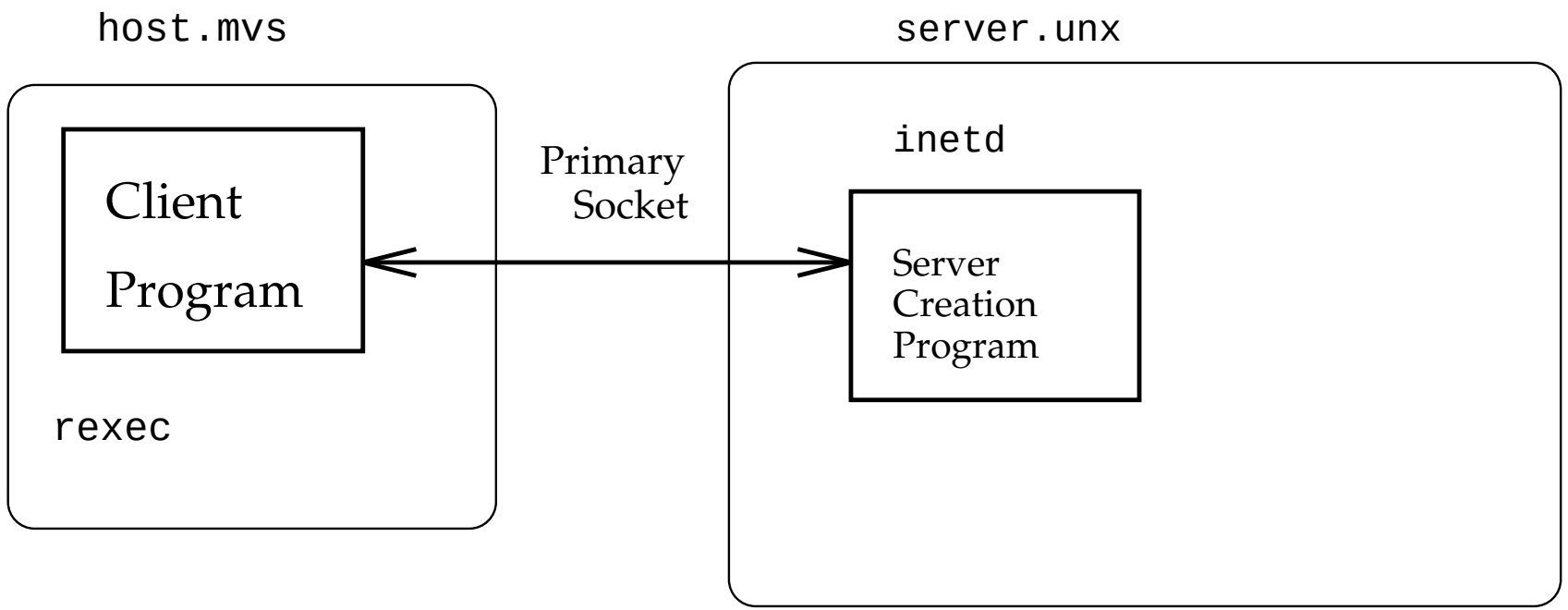
Remote Execution Connection Diagram



Client issues:

```
rexec server.unx mark mypass ls -l
```

REXEC initial connection



Client issues:

```
rexeccmd .c - rexec() caller:
  line 54: getservbyname()
rexec.c - rexec() function:
  line 62: gethostbyname()
  line 71: socket()
  line 81: connect()
```

Server creation program issues:

```
socket()
bind()
listen()
select()
accept()
```

getservbyname() coding

getservbyname() returns NULL for failure or a pointer

to this structure for success:

```
struct servent {
  char *s_name;      /* Official name          */
  char **s_aliases; /* array of aliases       */
  int s_port;        /* well-known port        */
  char *s_proto;     /* Protocol udp/tcp       */
};
```

On Unix, the data comes from the "/etc/services" file with

a format like the following:

```
exec     512/tcp          # remote execution
login    513/tcp          # remote login
who      513/udp  whod # remote who and uptime
```

rexec() function declaration

```
int rexec(              /* Returns primary socket */
                        /*  or -1 for failure */
  char **host,          /* Pointer to hostname by */
                        /*  reference */

  int port,             /* Port for REXEC server */
  char *username,       /* Username on remote host */
  char *passwd,         /* Password on remote host */
  char *cmd,            /* Command to execute */
  int*ptr_secondary     /* Place to store secondary*/
                        /*  descriptor (or NULL) */
);
```

gethostbyname() coding

gethostbyname() returns NULL for failure or a pointer

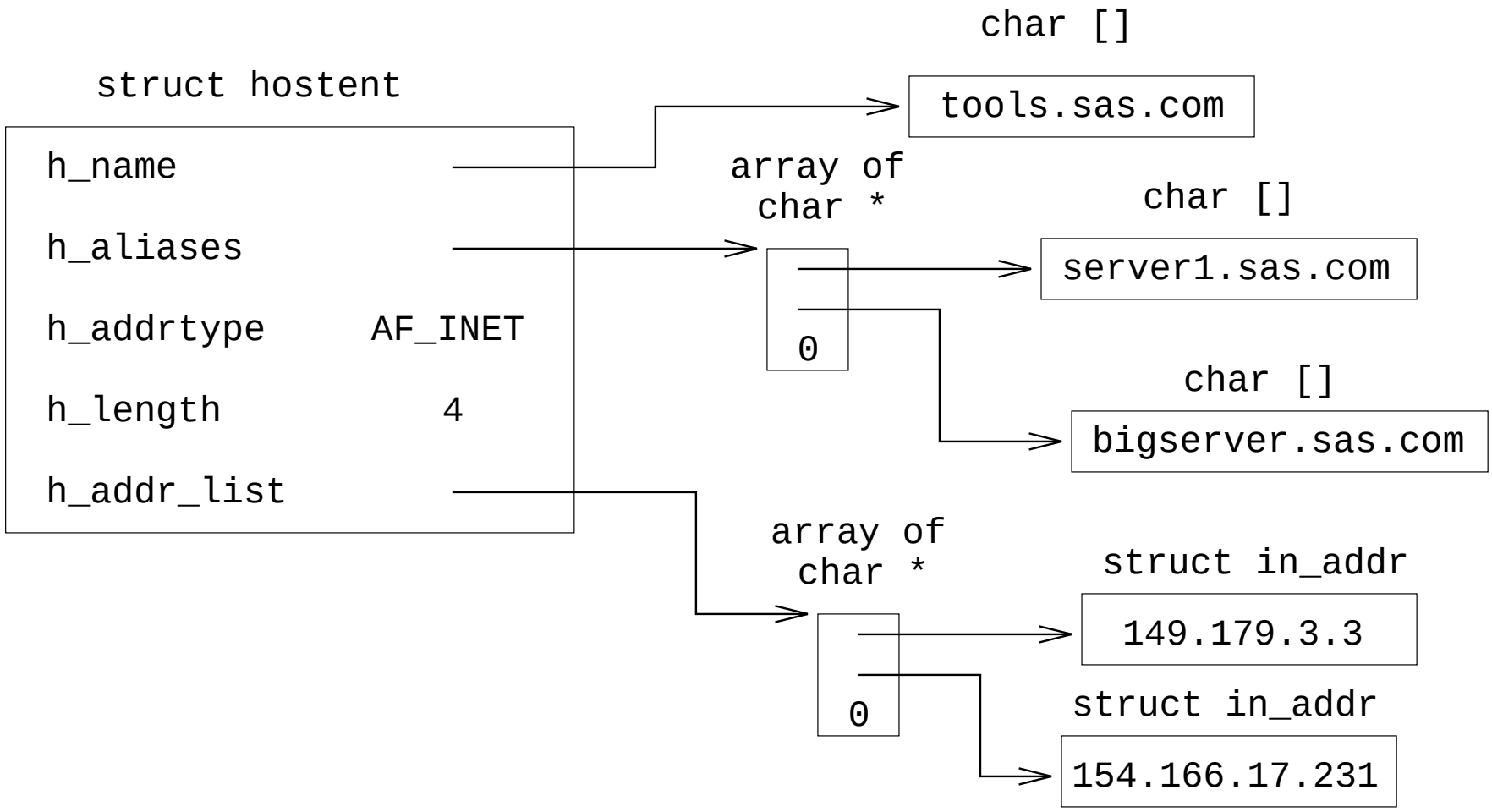
to this structure for success:

```
struct hostent {
  char *h_name;     /* Official name */
  char **h_aliases; /* array of aliases */
  int h_addrtype;   /* AF_INET for TCP/IP */
  int h_length;     /* sizeof(struct in_addr) for TCP/IP */
  char **h_addr_list; /*points to array of struct in_addr */
#define h_addr h_addr_list[0]  /* Primary IP address */
};
```

Here is an example of printing the IP address:

```
struct hostent *hp;
struct in_addr *iap = (struct in_addr *)hp->h_addr;
printf("IP address: %s.\n", inet_ntoa(*iap));
```

"struct hostent" diagram



System call return conventions

- Most system calls have the same return conventions.
- An int value >= 0 indicates success; -1 indicates failure.
- Failed calls set "errno" to indicate the cause of failure;

  perror() will print a message based on the "errno" value.
- Here is the typical coding logic:

```
int s = socket(AF_INET, SOCK_STREAM, 0);
if (s<0)
{
   perror("socket() failed");
   exit(EXIT_FAILURE);
}
```

Socket address structures

Socket address formats vary. Each format is called an

"addressing family". The generic format is:

```
struct sockaddr {
   u_short sa_family;   /* Addressing family */
   char sa_data[14];    /* varies            */
};
```

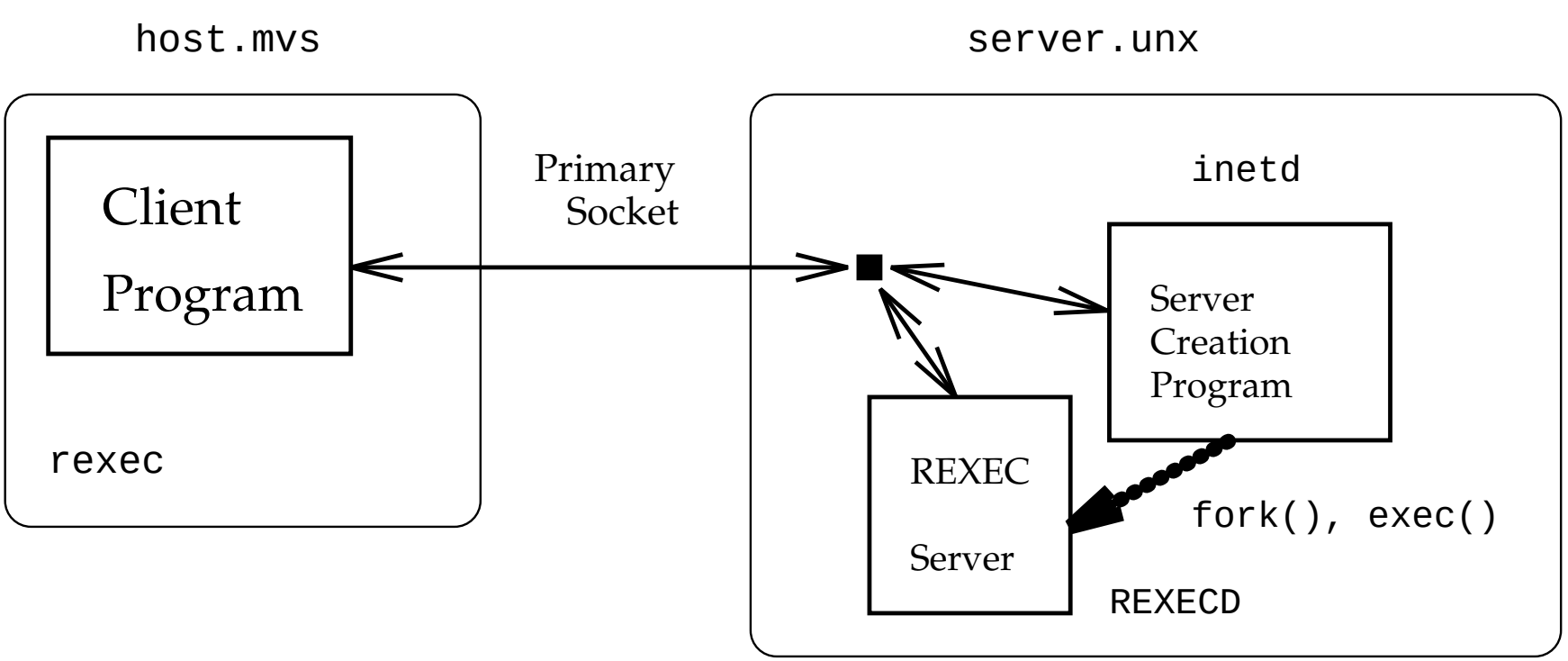
"AF_INET" is the addressing family for TCP/IP. Its format is:

```
struct in_addr {
   u_long s_addr;           /* net byte order */
};
struct sockaddr_in {
   short sin_family;        /* AF_INET        */
   u_short sin_port;        /* TCP or UDP port*/
   struct in_addr sin_addr;/* IP address.     */
   char sin_zero[8];        /* filler         */
};
```

Socket address conventions

- For connect(), the port and IP address must be specified.
- For bind(), either may be set to zero.
- Specifying "sin_port" as zero requests that the system assign a transient port. Use getsockname() to find out what port number was chosen.
- Specifying "sin_addr" as zero binds to all IP addresses on the host (some hosts have more than one).

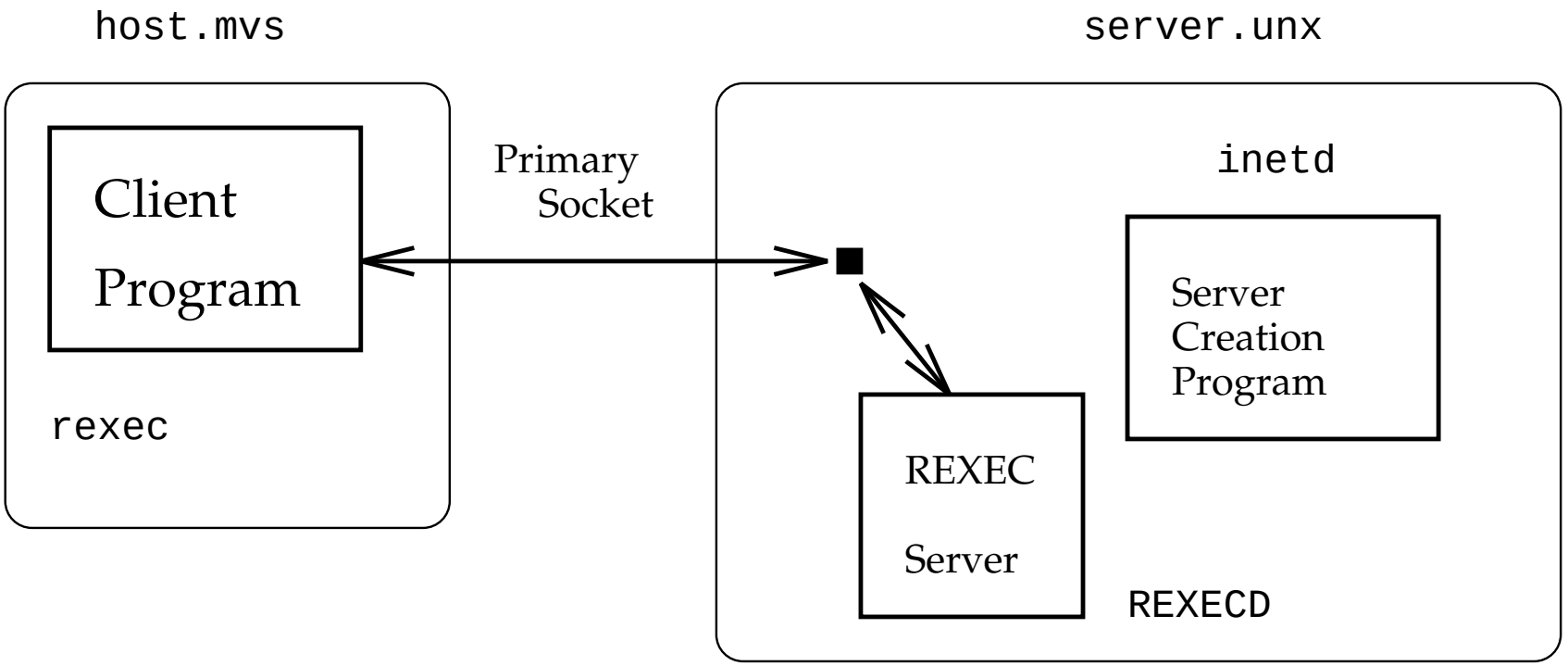
Inetd creates server program



Server creation program issues:

`fork(), exec()`

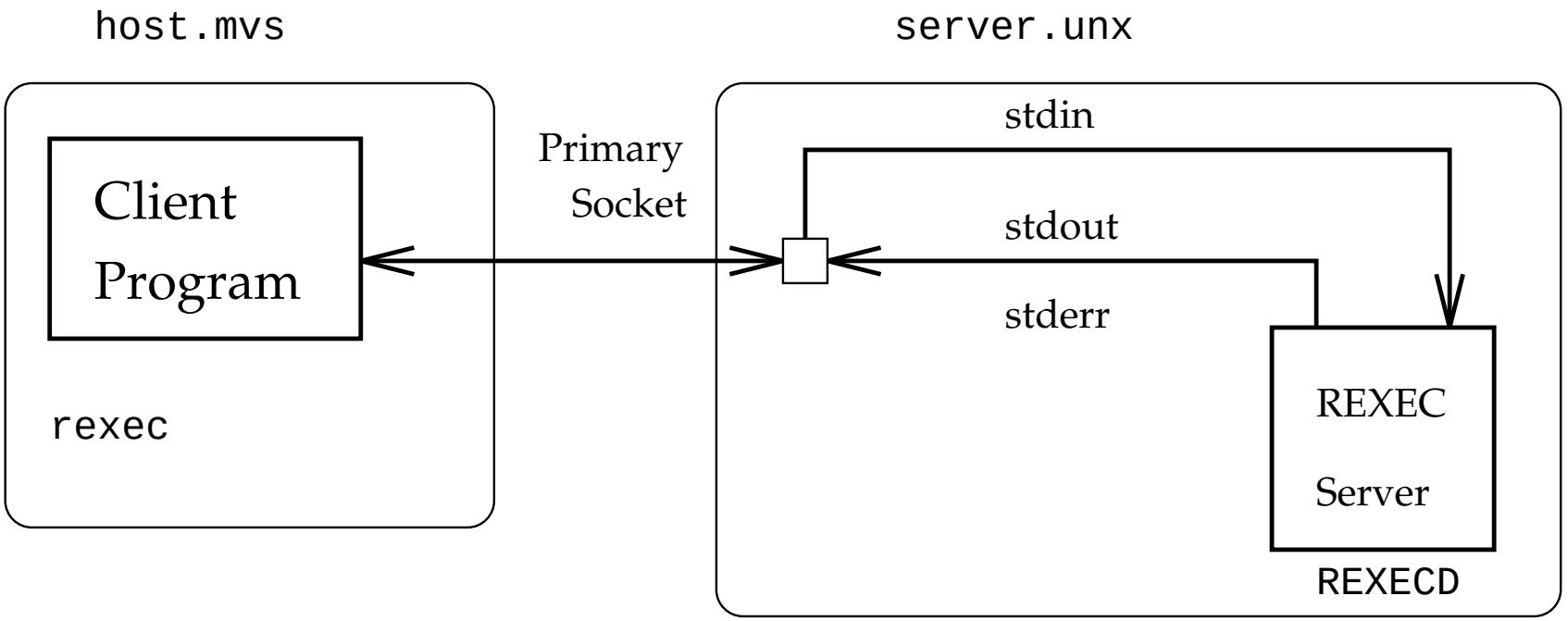
Server creation socket close



Server creation program issues:
`close()`

Server creation program waits
for a new client

Client establishes stderr port



Client issues:

```
line 98: secondary_model = socket()
line 112: bind()
line 114: listen()
line 120: getsockname()
line 130: write(primary, portstr, ...)
```

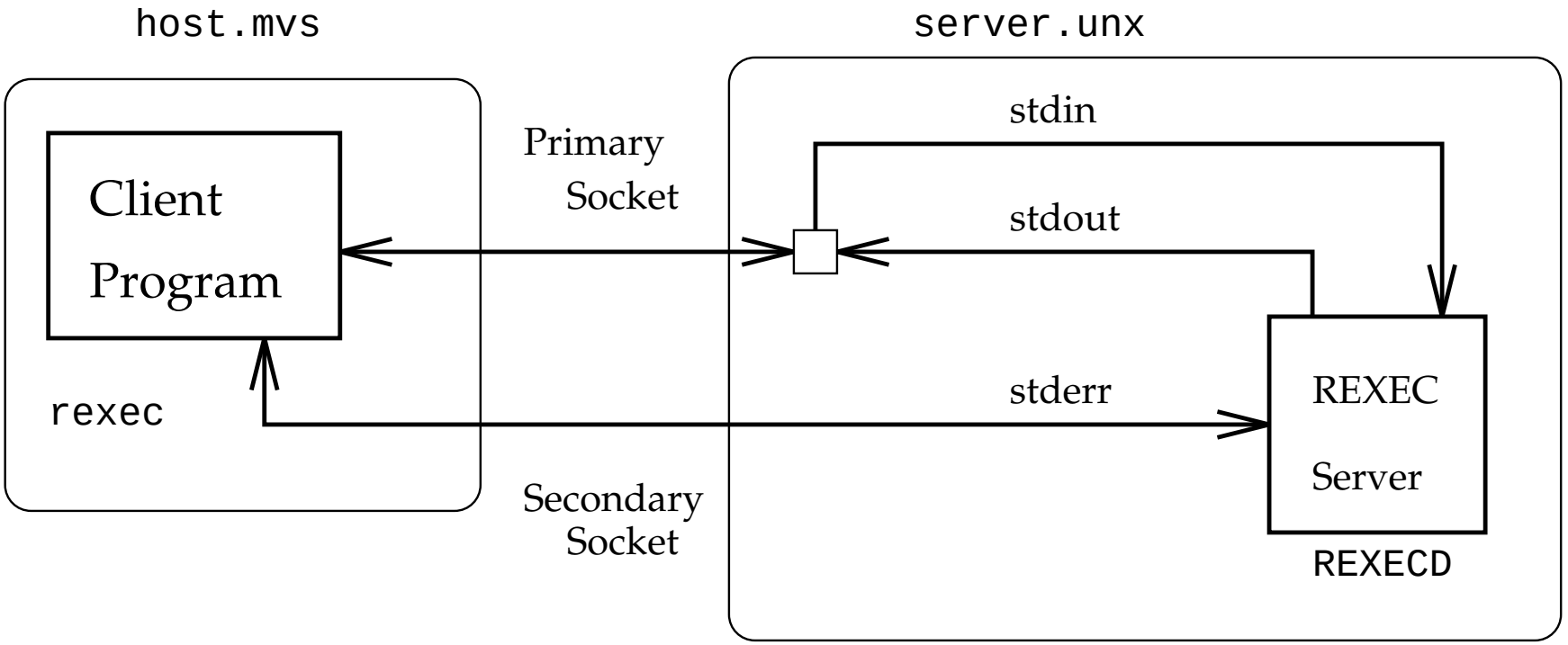
Server issues:

```
dup2(primary, 0)
dup2(primary, 1)
dup2(primary, 2)
read(primary, portstr, ...)
```

read() and write() conventions

- read() returns 0 for EOF; >0 for number of bytes read
- read() often returns fewer bytes than requested;
  call read() in a loop until you have gotten them all
- For TCP, write() does not create record boundaries;
  read() may get the result of more or less than one write()
- read()/write() behavior is a common programming error
  with TCP/IP and socket programming

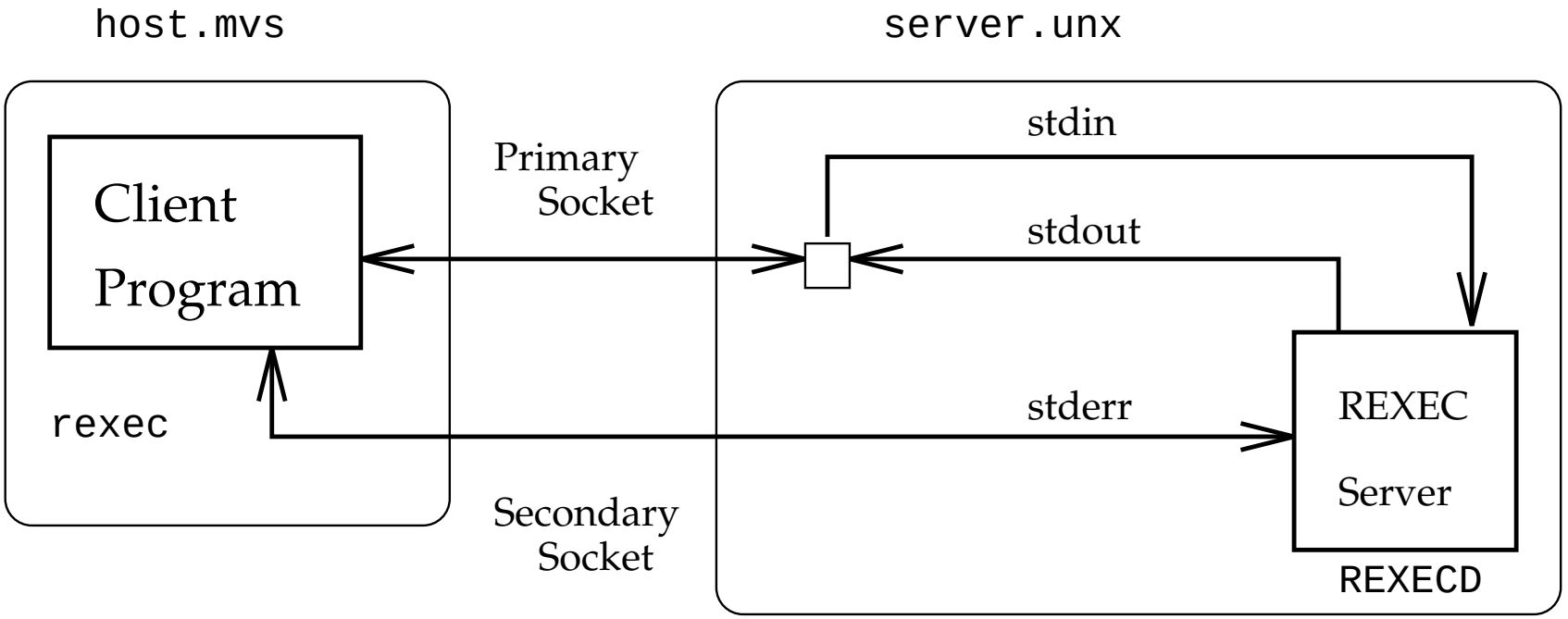
Rexecd connect() stderr port



Client issues:
`line 137: accept()`

Server issues:
`connect()`

Send user, password and command



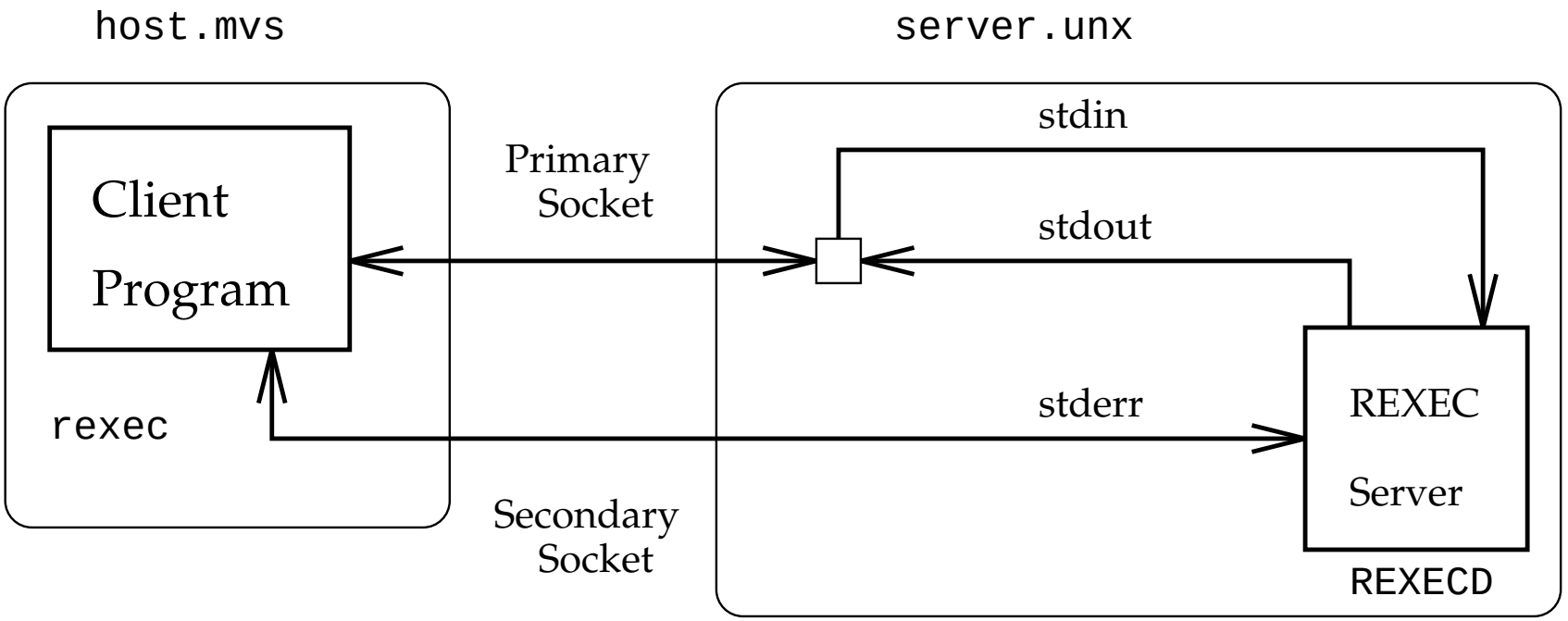
Client issues:

```
line 147: write(primary, name, ...)
line 150: write(primary, passwd, ...)
line 153: write(primary, cmd, ...)
```

Server issues:

```
read()
read()
...
```

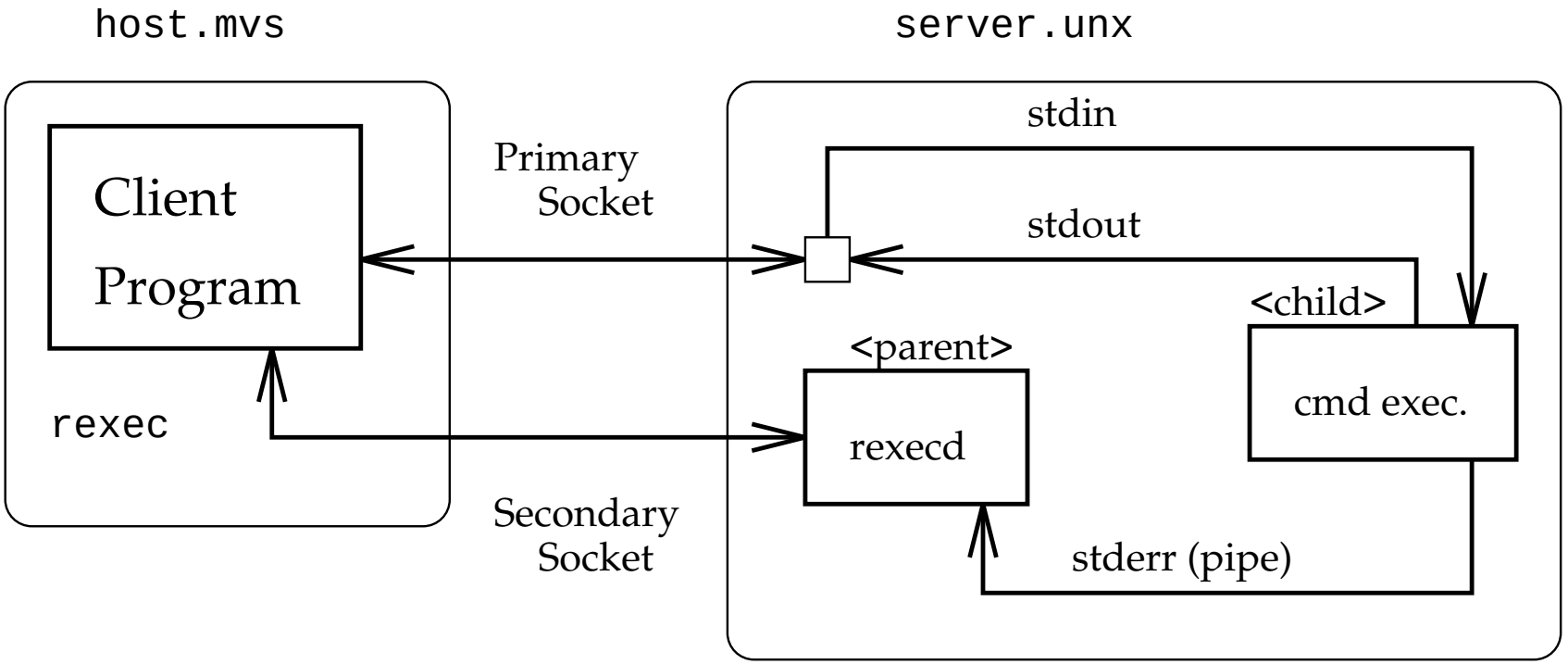
Server confirms



Client issues:
`line 162: read(primary, &c, 1)`

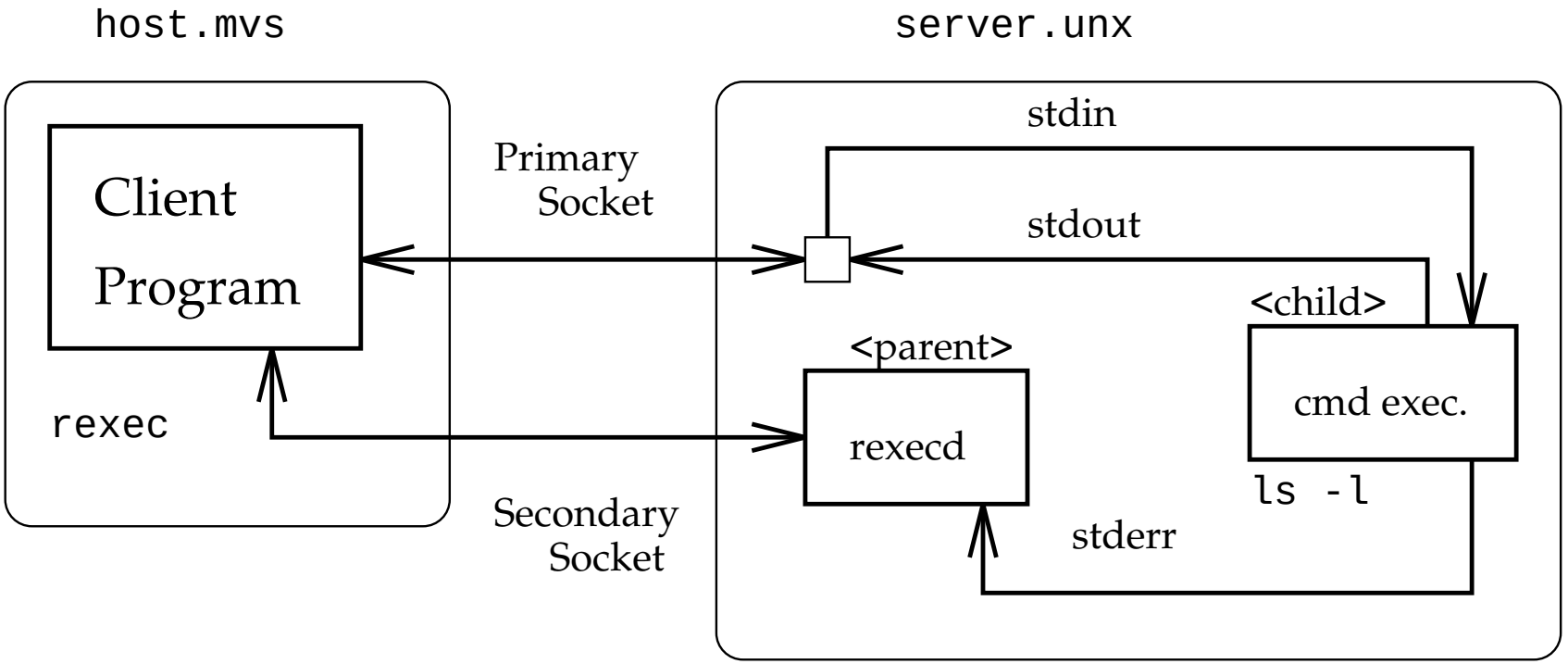
Server issues:
write of a single byte 0 or 1

Server creates command executor



Server issues:
`pipe()`
`fork()`

Child process executes command



Server issues:
`exec()`

File descriptor sets

- A file descriptor is an integer (returned by open() or or socket()) which identifies a file or socket
- Descriptors are numbered 0 through FD_SETSIZE-1
- "fd_set" is a type which contains a bit for each descriptor
- The following macros manipulate file descriptor sets

```
#include <sys/types.h>
int desc; fd_set fds;

FD_ZERO(&fds);     /* zero entire set */
FD_CLR(desc, &fds); /* clear desc bit */
FD_SET(desc, &fds); /* set desc bit */
FD_ISSET(desc, &fds); /* test desc bit */
```

The select() call

- The select() system call awaits activity on descriptor sets

```
int select(     /* returns number active */
   int nfds,    /* Highest desc used + 1 */
   fd_set *read,  /* Await read on these */
   fd_set *write, /* Await write on these*/
   fd_set *except,/* Seldom-used */
   struct timeval *timeout /* timeout if */
);                            /*  no activity */
```

- A"timeout" parameter of NULL specifies indefinite waiting
- A zero time value sets select() non-blocking
- A non-zero time value, sets a maximum wait time

Summary

- Reviewed various socket API function calls
- REXEC / REXECD dialog
- Useable TCPIP Socket Application
- Questions - 5954 and 5955

Bibliography

- Internetworking with TCP/IP: Vol. I and Vol II,
  Douglas Comer, Prentice Hall, 1991
- The Whole Internet User's Guide & Catalog by Ed Kroll;
  O'Reilly and Associates.
- UNIX Network Programming by W. Richard Stevens;
  Prentice Hall, 1990.
- Socket API Programmer's Reference
- Unix "MAN" pages

```
 1 /* -------------------- SAS/C Sample Program --------------------
 2  *
 3  * MODULE: rexeccmd.c - Main routine for sample rexec command
 4  * SYNTAX: rexec host username password cmd-words
 5  *
 6  * NOTE:   This command does not support sending input to the remote
 7  *         program. Non-UNIX implementations often restrict
 8  *         select() to sockets. Thus, there is no portable
 9  *         way to wait for sockets and "stdin" simultaneously.
10  */
11 #include <sys/types.h>
12 #include <sys/socket.h>
13 #include <netdb.h>
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <string.h>

17 #ifdef __SASC__
18 static void to_ebcdic(char *buf, size_t n);
19 #endif

20 #define EXEC_PORT 512  /* Well-know port for exec server. */

21 main(int argc, char *argv[])
22 {
23    int primary;     /* Socket for stdin/stdout. */
24    int secondary;   /* Socket for stderr. */
25    fd_set read_set; /* Set of sockets we will wait on. */
26    char buf[256];   /* Buffer for incoming data. */

27    char *host;      /* Pointer to hostname. */
28    int port;        /* Port for exec server */
29    char cmd[512];   /* Command string. */
30    struct servent *serv; /* exec server information. */
31    int i, n, rc;

32    /*
33     * Handle command line arguments.
34     *    argv[1] = hostname
35     *    argv[2] = username
36     *    argv[3] = password
37     *    argv[4] = first command word
38     *    ...
39     *    argv[argc-1] = last command word
40     */
41    if (argc <5 || *argv[1]=='?' || strcmp(argv[1],"-?")==0) {
42       printf("Usage: %s hostname username password cmd-words\n",
43                  argv[0]);
44       exit(EXIT_FAILURE);
45    }
46    host = argv[1];  /* Pointer can be modified by rexec() call. */

47    /* Build up command string from remaining arguments. */
48    strcpy(cmd,argv[4]);
49    for (i=5; i<argc; i++) {
50       strcat(cmd," ");
51       strcat(cmd,argv[i]);
52    }

53    /* Lookup standard REXEC server port. */
54    serv = getservbyname("exec","tcp");
55    if (serv==NULL) {
56       perror("getservbyname failed");
57       port = EXEC_PORT;  /* Assume the usual. */
58    }
59    else
60       port = serv->s_port;

61    primary = rexec(&host, port, argv[2], argv[3], cmd, &secondary);
62    if (primary<0) return EXIT_FAILURE;
```

```
 63     FD_ZERO(&read_set);
 64     while (primary>=0 || secondary>=0) {
 65        /*
 66         * Wait for incoming data on both sockets.
 67         *   select() clears bits from the mask, so we must reset
 68         *   the bits each time through the loop.
 69         */
 70        if (primary>=0) FD_SET(primary, &read_set);
 71        if (secondary>=0) FD_SET(secondary, &read_set);
 72
 73        rc = select(FD_SETSIZE, &read_set, NULL, NULL, NULL);
 74        if (rc<0) {
 75           perror("select failed");
 76           exit(EXIT_FAILURE);
 77        }

 78        /*
 79         * If incoming data available on primary...
 80         */
 81        if (primary >= 0 /* Not end of file yet. */
 82        &&  FD_ISSET(primary, &read_set)) {
 83           /* read it (or at least some of it). */
 84           n = read(primary, buf, sizeof(buf));
 85           if (n==0)
 86              primary = -1;  /* Flag end-of-file on primary. */
 87           else if (n<0) {
 88              perror("read failed on remote program's standard output");
 89              exit(EXIT_FAILURE);
 90           }
 91           else {
 92              #ifdef __SASC__
 93              to_ebcdic(buf, n);
 94              #endif
 95              fwrite(buf, n, 1, stdout); /* Write to local stdout */
 96           }
 97        }
 98        /*
 99         * If incoming data available on secondary...
100         */
101        if (secondary >= 0  /* Not end of file yet. */
102        &&  FD_ISSET(secondary, &read_set)) {
103           n = read(secondary, buf, sizeof(buf));
104           if (n==0)
105              secondary = -1;  /* Flag end-of-file on secondary. */
106           else if (n<0) {
107              perror("read failed on remote program's standard error");
108              exit(EXIT_FAILURE);
109           }
110           else {
111              #ifdef __SASC__
112              to_ebcdic(buf, n);
113              #endif
114              fwrite(buf, n, 1, stderr); /* Write to local stderr */
115           }
116        }
117     }  /* End of while loop. */

118     return EXIT_SUCCESS;
119  }

120  #ifdef __SASC__   /* Need conversion to EBCDIC */
121  #include <netinet/in.h> /* prototype for htoncs(). */
122  static void to_ebcdic(char *buf, size_t n)
123  {
124     int i;
125     /*
126      * htoncs() is a SAS/C specific function which converts
127      *   a character from ASCII to EBCDIC.
128      */
129     for (i=0; i<n; i++) buf[i] = ntohcs(buf[i]);
130     return;
131  }
132  #endif
```

```
 1 /* -------------------- SAS/C Sample Program --------------------
 2  *
 3  * MODULE: rexec.c - sample rexec() function implementation
 4  *
 5  */

 6 #include <sys/types.h>
 7 #include <sys/socket.h>
 8 #include <netdb.h>
 9 #include <netinet/in.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>

13 #ifdef __SASC__
14 static char *strcpy_to_ascii(char *ascii, char *ebcdic);
15 #endif
16 static int fail(const char *msg, int primary, int secondary);

17 /*
18  * NAME:   rexec()
19  * PARMS: -Name----------type-----Description----------------------
20  *         host          char **  Host name string pointer - passed
21  *                                  by reference
22  *         port          int      port for remote execution
23  *         name          char *   Username on remote host
24  *         passwd        char *   Password on remote host
25  *         cmd           char *   Command to execute
26  *         ptr_secondary int *    stderr socket descriptor area ptr
27  *                                  or NULL
28  * RETURN:-Name----------type-----Description----------------------
29  *                       int      primary socket or -1 for failure
30  */
31 int rexec(char **host, int port, char *name, char *passwd,
32           char *cmd, int *ptr_secondary)
33 {
34    /* Internet address format of socket address structure. */
35    struct sockaddr_in sa, sa2, server_sa;
36    struct hostent *hp;  /* Results of host lookup. */
37    int primary;         /* Primary connection socket. */
38    int secondary;       /* Secondary connection socket. */
39    int secondary_model; /* Model socket used for bind/accept. */
40    char portstr[6];     /* string for decimal port number */
41    int len, rc, n;
42    char c;

43    #ifdef __SASC__
44    /*
45     * The protocol expects the username, password and
46     *   command string in ASCII. For callers on EBCDIC hosts,
47     *   this routine translates the parameters to ASCII.
48     */
49    #define CMDMAX 1024
50    #define NAMEMAX 80
51    #define PASSMAX 80

52    char name_ascii[NAMEMAX];
53    char passwd_ascii[PASSMAX];
54    char cmd_ascii[CMDMAX];

55    strcpy_to_ascii(name_ascii, name); name = name_ascii;
56    strcpy_to_ascii(passwd_ascii, passwd); passwd = passwd_ascii;
57    strcpy_to_ascii(cmd_ascii, cmd); cmd = cmd_ascii;
58    #endif

59    /*
60     * Lookup host and extract address.
61     */
62    hp = gethostbyname(*host);
63    if (hp==NULL) {
64       herror(*host);  /* Print msg describing error. */
65       return -1;
66    }
67    *host = hp->h_name;
```

```
 68     /*
 69      * Create primary connection socket using TCP.
 70      */
 71     primary = socket(AF_INET, SOCK_STREAM, 0);
 72     if (primary < 0)
 73        return fail("socket() failed", -1, -1);

 74     /*
 75      * Fill in address of server and connect.
 76      */
 77     memset(&sa, '\0', sizeof(sa));
 78     sa.sin_family = AF_INET;             /* Internet-type address. */
 79     memcpy(&sa.sin_addr, hp->h_addr, hp->h_length); /* IP address. */
 80     sa.sin_port = port;                                /* TCP port. */
 81     if (connect(primary, &sa, sizeof(sa))<0)
 82        /* Likely to be "connection refused".Show host name in */
 83        /*   message. */
 84        return fail(*host, primary, -1);

 85     /*
 86      * Create secondary connection (if requested)
 87      */
 88     if (ptr_secondary == NULL) {
 89        /* Write one byte of zero to indicate no additional */
 90        /*   connection desired.  */
 91        write(primary, "", 1);
 92        secondary = -1;
 93     }
 94     else {
 95        /*
 96         * Create model socket for secondary connection.
 97         */
 98        secondary_model = socket(AF_INET, SOCK_STREAM, 0);
 99        if (secondary_model <0)
100           return fail("socket() failed", primary, -1);
101
102        /*
103         * Bind it to a transient port.
104         *   Leaving local address as zero allows all IP
105         *     addresses for the machine to be used
106         *   Leaving the port zero allows TCP to select a
107         *     transient port.
108         * Set up inconming connection queue (using listen()).
109         */
110        memset(&sa2, '\0', sizeof(sa2));
111        sa2.sin_family = AF_INET;
112        if (bind(secondary_model, &sa2, sizeof(sa2))<0)
113           return fail("bind() failed", primary, secondary_model);
114        if (listen(secondary_model, 1)<0)
115           return fail("listen() failed", primary, secondary_model);

116        /*
117         * Get port number and send it to the rexec server.
118         */
119        len = sizeof(sa2);
120        rc = getsockname(secondary_model, &sa2, &len);
121        if (rc < 0  || len < sizeof(sa2))
122           return fail("getsockname() failed", primary, secondary_model);
123        /* The port is in network byte order within the */
124        /*   "struct sockaddr_in". We need it in local  */
125        /*   byte order for sprintf to interpret.       */
126        sprintf(portstr, "%u", ntohs(sa2.sin_port));

127        #ifdef __SASC__
128        strcpy_to_ascii(portstr, portstr); /* Server needs ASCII */
129        #endif

130        if (write(primary, portstr, strlen(portstr)+1)<0)
131           return fail("write() failed",primary,secondary_model);

132        /*
133         * Await an incoming connection request. accept() will
134         *   return the actual socket to use for stderr.
```

```
135         */
136        len = sizeof(server_sa);
137        secondary = accept(secondary_model, &server_sa, &len);
138        close(secondary_model);
139        if (secondary < 0 || len < sizeof(server_sa))
140           return fail("accept() failed",primary,-1);
141        *ptr_secondary = secondary; /* Provide to caller. */
142     }

143     /*
144      * Write username, password and command to the server. In
145      *   each case, the terminating '\0' is sent as a delimiter.
146      */
147      rc = write(primary, name, strlen(name)+1);
148      if (rc < 0)
149         return fail("write() failed",primary,secondary);
150      rc = write(primary, passwd, strlen(passwd)+1);
151      if (rc < 0)
152         return fail("write() failed",primary,secondary);
153      rc = write(primary, cmd, strlen(cmd)+1);
154      if (rc < 0)
155         return fail("write() failed",primary,secondary);

156     /*
157      * Server should send a single '\0' to indicate success.
158      *   Otherwise it should send a single byte of binary 0x1
159      *   to indicate failure, followed by a newline-terminated
160      *   ASCII error message.
161      */
162     if (read(primary, &c, 1) != 1)
163        return fail("read() failed",primary, secondary);
164     if (c!='\0')   /* Server error message. */ {
165        do {
166           n = read(primary, &c, 1);
167           if (n==0)
168             break; /* Really expected '\n', but EOF is good enough. */
169           if (n<0)
170             return fail("read() failed", primary, secondary);
171           #ifdef __SASC__
172           c = ntohcs(c); /* Msg needs to print in EBCDIC. */
173           #endif
174           fputc(c, stderr);
175        } while (c!='\n');
176        return -1;
177     }

178     /*
179      * The caller is now responsible for handling traffic over
180      *   the primary and (if requested) secondary connections.
181      */
182     return primary;
183  }

184  /*
185   * Clean-up logic.
186   */
187  static int fail(const char *msg, int primary, int secondary)
188  {
189     perror(msg);
190     if (primary>=0)
191        close(primary);
192     if (secondary>=0)
193        close(secondary);
194     return -1;
195  }

196  #ifdef __SASC__  /* Need conversion to ASCII */

197  static char *strcpy_to_ascii(char *ascii, char *ebcdic)
198  {
199     int i;
200     /*
201      * htoncs() is a SAS/C specific function which converts
202      *   a character from ASCII to EBCDIC.
```

```
203      */
204     for (i=0; ebcdic[i]!='\0'; i++) ascii[i] = htoncs(ebcdic[i]);
205     ascii[i] = '\0';
206     return ascii;
207  }

208  #endif
```