OPERATING SYSTEMS

Interprocess Communication

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

Contents

Overview of C/S Communications

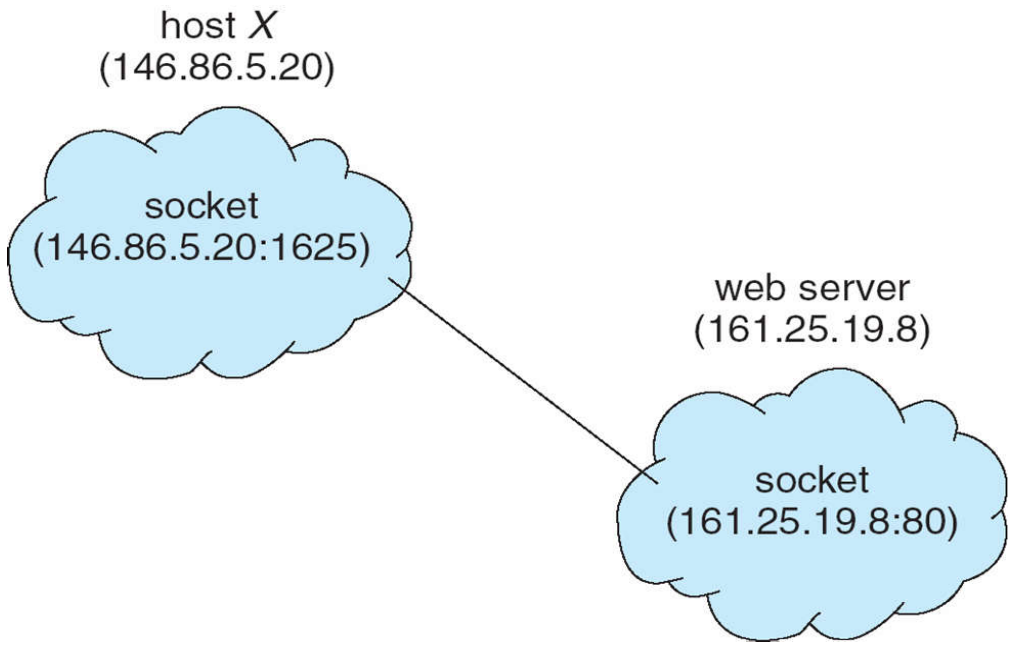
- Shared-memory and Message-passing can be used for communication in Client–Server systems as well.
- Now we explore two other strategies for communication in C/S systems
  - Sockets (Internet)
  - Remote Procedure Calls (RPCs)
    - Not only useful for C/S computing, but also used by Android as a form of IPC between processes running on the same system.

Sockets

- A socket is defined as an *endpoint* for communication.
- A pair of processes communicating over a network employs a pair of sockets—one for each process.
- A socket is a concatenation of IP address and port number.
- In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.
- Servers implementing standard services listen to *well-known* ports below 1024.
  - SSH server – port 22
  - FTP server – port 21
  - HTTP server – port 80
- When a client process initiates a request for a connection, it is assigned a port by its host computer. This port has some arbitrary number greater than 1024.

Sockets

- Example.
  - Suppose that a client process on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8. The client may be randomly assigned a port number 1625 greater than 1024. The connection will consist of a pair of sockets (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server.



Sockets

- Example.
  - If another process also on host X wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.

Linux: Socket Programming

Data Structures

```
#include <netinet/in.h>
struct sockaddr {
    unsigned short sa_family; /* socket address family, AF_xxx */
    char sa_data[14]; /* 14 bytes of protocol address */
};

struct in_addr {
    unsigned long s_addr;
};

struct sockaddr_in {
    short int sin_family; /* AF_INET for ipv4 */
    unsigned short int sin_port; /* Port number */
    struct in_addr sin_addr; /* IP address */
    unsigned char sin_zero[8]; /* padding with 0 to keep the same size as
struct sockaddr (16bytes) */
```

Linux: Socket Programming

Data Structures

```
#include <ifaddrs.h>
struct ifaddrs {
    struct ifaddrs *ifa_next; /* Next item in list */
    char *ifa_name; /* Name of interface */
    unsigned int ifa_flags; /* Flags from SIOCGIFFLAGS */
    struct sockaddr *ifa_addr; /* Address of interface */
    struct sockaddr *ifa_netmask; /* Netmask of interface */
    union {
        struct sockaddr *ifu_broadaddr;
            /* Broadcast address of interface */
        struct sockaddr *ifu_dstaddr;
            /* Point-to-point destination   address */
    } ifa_ifu;
#define ifa_broadaddr ifa_ifu.ifu_broadaddr
#define ifa_dstaddr ifa_ifu.ifu_dstaddr
    void *ifa_data; /* Address-specific data */
};
```

Linux: Socket Programming

- Socket APIs
  - socket() creates an endpoint for communication and returns a file descriptor (sockfd) that refers to that endpoint.

```
#include<sys/socket.h>
int socket(int domain, int type, int protocol);
```

  - bind() binds the `sockfd` to a socket address structure specified by `addr`.

```
#include<sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

  - getsockname() returns the current address to which the socket sockfd is bound, in the buffer pointed to by addr. The `addrlen` argument should be initialized to indicate the amount of space (in bytes) pointed to by `addr`. On return it contains the actual size of the socket address.

```
#include<sys/socket.h>
int getsockname(int sockfd, struct sockaddr *restrict addr, socklen_t
*restrict addrlen);
```

Linux: Socket Programming

- Socket APIs
  - listen() sets a socket to the state of waiting for incoming connection

```
#include<sys/socket.h>
int listen(int sockfd, int backlog);
/* backlog is the number of entries in ESTABLISHED but not ACCEPTED
status. All SYN_RECV clients have to be waiting until backlog queue
has some empty space */
```

  - connect() connects the socket referred to by the file descriptor sockfd to the address specified by addr.

```
#include<sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t
addrlen);
```

Linux: Socket Programming

- Socket APIs
  - accept() is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET). It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket.

```
#include<sys/socket.h>
int accept(int sockfd, struct sockaddr restrict *addr, socklen_t
restrict  *addrlen);
```

  - send(): sends data to a socket.

```
#include<sys/socket.h>
ssize_t send(int sockfd, const void *buf, socklen_t len, int falgs);
```

  - recv(): receives data from a socket.

```
#include<sys/socket.h>
ssize_t recv(int sockfd, void *buf, socklen_t len, int flags);
```

Linux: Socket Programming

- Socket APIs
  - setsockopt(): setting socket options

```
#include <sys/socket.h>
int setsockopt( int sockfd, int level, int optname, const void
*optval, socklen_t optlen);
/* setsockopt() has many options */
```

  - getifaddrs(): creating a linked list of structures describing the network interfaces of the local system, and stores the address of the first item of the list in *ifap.

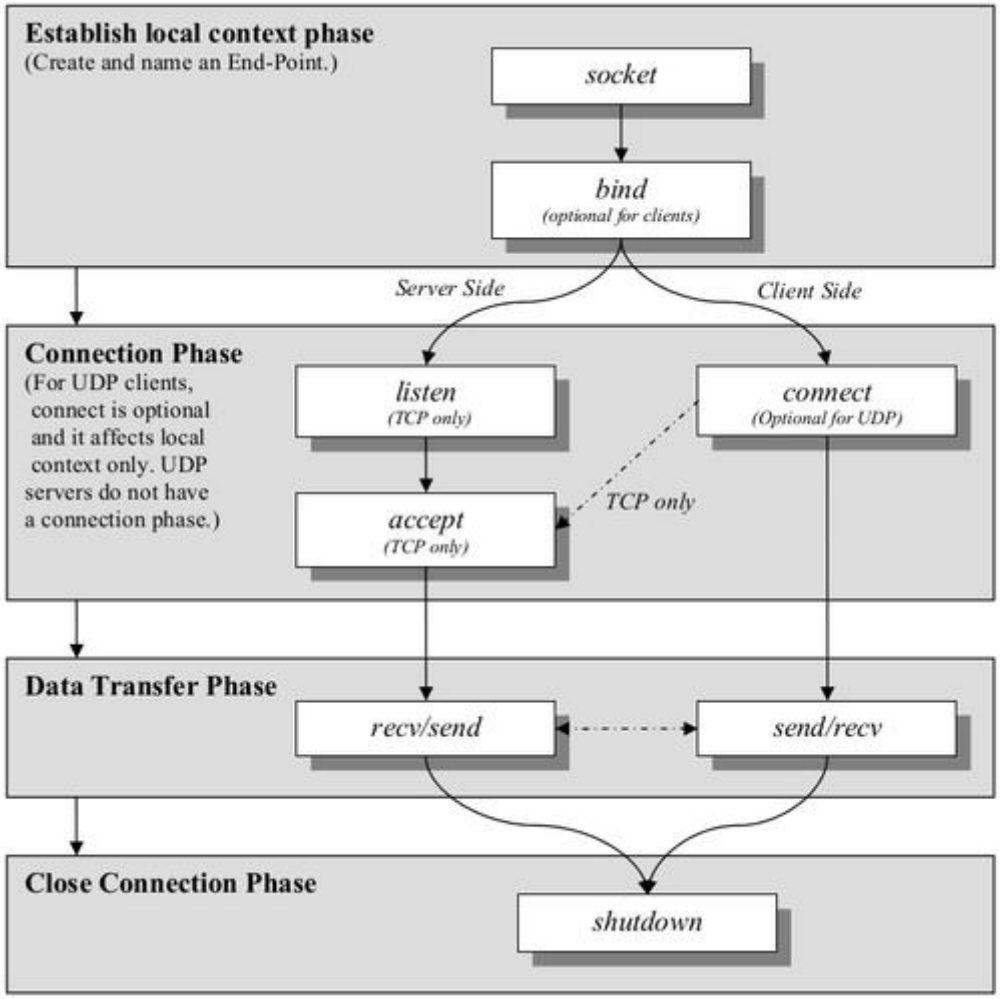
```
#include <sys/types.h>
#include <ifaddrs.h>
int getifaddrs(struct ifaddrs **ifap);
```

  - freeifaddrs(): freeing the data structure returned by getifaddrs() which is dynamically allocated when no longer needed.

```
#include <sys/types.h>
#include <ifaddrs.h>
void freeifaddrs(struct ifaddrs *ifap);
```

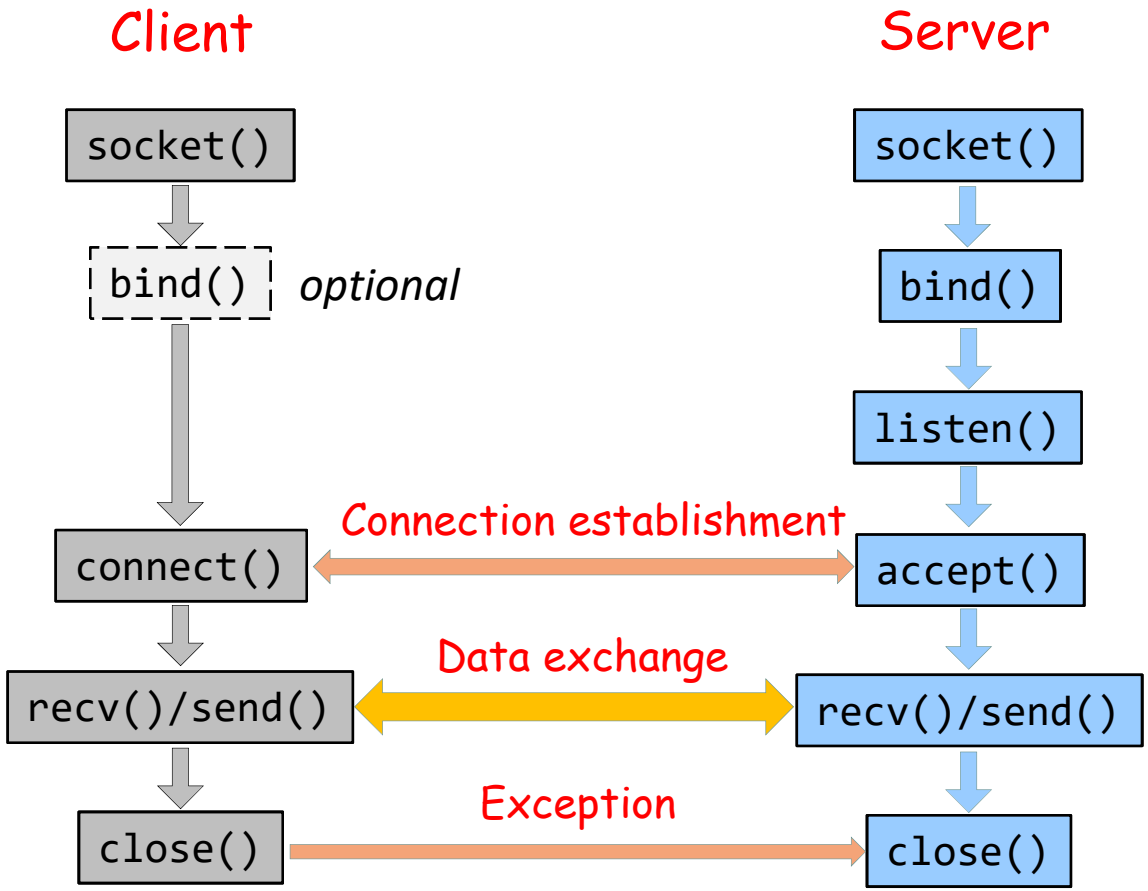
Linux: Socket Programming

- Socket Programming



Linux: Socket Programming

- Socket Programming



Linux: Socket Programming

Algorithm 11-1: get an available port (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>

int main(void)
{
    unsigned short port = 0;
    int sockfd, ret, result = 1;
    struct sockaddr_in myaddr, readdr; /* declared in <netinet/in.h>,
inet_addr in <arpa/inet.h> */
    socklen_t addr_len;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd == -1) {
        perror("socket()");
        return EXIT_FAILURE;
    }
```

TCP IPV4

Linux: Socket Programming

Algorithm 11-1: get an available port (2)

```
    myaddr.sin_family = AF_INET;
    myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    myaddr.sin_port = 0; /* when .sin_port is set to 0, bind() will assign
an available port to it */
    addr_len = sizeof(myaddr);
    ret = bind(sockfd, (struct sockaddr *)&myaddr, addr_len);
    if(ret == 0) {
        addr_len = sizeof(readdr);
        ret = getsockname(sockfd, (struct sockaddr *)&readdr, &addr_len);
        if(ret == 0) {
            port = ntohs(readdr.sin_port);
            printf("Assigned port number = %d\n", port);
        }
        else
            result = 0;
    }
    else
        result = 0;

    if(close(sockfd) != 0) /* close() defined in <unistd.h> */
         result = 0;

    return result;
}
```

Linux: Socket Programming

Algorithm 11-2: socket-server-1.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <ifaddrs.h>
#include <sys/signal.h>

#define BUFFER_SIZE 1024
#define MAX_QUE_CONN_NM 5
#define ERR_EXIT(m) \
    do { \
        perror(m); \
        exit(EXIT_FAILURE); \
    } while(0)
```

```
/* one client, one server, asynchronous send-recv version */
int getipv4addr(char *ip_addr)
{
    struct ifaddrs *ifaddrsptr = NULL;
    struct ifaddrs *ifa = NULL;
    void *tmpptr = NULL;
    int ret;

    ret = getifaddrs(&ifaddrsptr);
    if (ret == -1)
        ERR_EXIT("getifaddrs()");
    for(ifa = ifaddrsptr; ifa != NULL; ifa = ifa->ifa_next) {
        if(!ifa->ifa_addr)
            continue;
        if(ifa->ifa_addr->sa_family == AF_INET) { /* IP4 */
            tmpptr = &((struct sockaddr_in *)ifa->ifa_addr)->sin_addr;
            char addr_buf[INET_ADDRSTRLEN];
            inet_ntop(AF_INET, tmpptr, addr_buf, INET_ADDRSTRLEN);
            printf("%s IPv4 address %s\n", ifa->ifa_name, addr_buf);
            if (strcmp(ifa->ifa_name, "lo") != 0)
                strcpy(ip_addr, addr_buf); /* return the ipv4 address */
        } else if(ifa->ifa_addr->sa_family == AF_INET6) { /* IP6 */
            tmpptr = &((struct sockaddr_in6 *)ifa->ifa_addr)->sin6_addr;
            char addr_buf[INET6_ADDRSTRLEN];
            inet_ntop(AF_INET6, tmpptr, addr_buf, INET6_ADDRSTRLEN);
            printf("%s IPv6 address %s\n", ifa->ifa_name, addr_buf);
        }
    }
```

Linux: Socket Programming

Algorithm 11-2: socket-server-1.c (2)

```
    if (ifaddrsptr != NULL)
        freeifaddrs(ifaddrsptr);
    return EXIT_SUCCESS;
}

int main(void)
{
    int server_fd, connect_fd;
    struct sockaddr_in server_addr, connect_addr;
    socklen_t addr_len;
    int recvbytes, sendbytes, ret;
    char send_buf[BUFFER_SIZE], recv_buf[BUFFER_SIZE];
    char ip_addr[INET_ADDRSTRLEN]; /* ipv4 address */
    char stdin_buf[BUFFER_SIZE];
    uint16_t port_num;
    char clr;
    pid_t childpid;

    server_fd = socket(AF_INET, SOCK_STREAM, 0); /* ipv4 */
    if(server_fd == -1) {
        ERR_EXIT("socket()");
    }
    printf("server_fd = %d\n", server_fd);

    ret = getipv4addr(ip_addr); /* auto server ip address */
    if (ret == EXIT_FAILURE)
        ERR_EXIT("getifaddrs()");
```

Linux: Socket Programming

Algorithm 11-2: socket-server-1.c (3)

```
printf("input server port number: ");
memset(stdin_buf, 0, BUFFER_SIZE);
fgets(stdin_buf, BUFFER_SIZE-1, stdin); /* including '\n' */
port_num = atoi(stdin_buf);
    /* set sockaddr_in */
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(port_num);
//    server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_addr.s_addr = inet_addr(ip_addr);
bzero(&(server_addr.sin_zero), 8); /* padding with 0's */
int opt_val = 1;
setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt_val, sizeof(opt_val));
    /* many options */
addr_len = sizeof(struct sockaddr);
ret = bind(server_fd, (struct sockaddr *)&server_addr, addr_len);
if(ret == -1) {
    close(server_fd);
    ERR_EXIT("bind()");
}
printf("Bind success!\n");

ret = listen(server_fd, MAX_QUE_CONN_NM);
if(ret == -1) {
    close(server_fd);
    ERR_EXIT("listen()");
}
printf("Server ipv4 addr: %s, port: %hu\n", ip_addr, port_num);
printf("Listening ...\n");
```

Linux: Socket Programming

Algorithm 11-2: socket-server-1.c (4)

```
addr_len = sizeof(struct sockaddr);
    /* addr_len should be assigned before each accept() */
connect_fd = accept(server_fd, (struct sockaddr *)&connect_addr, &addr_len);
if(connect_fd == -1) {
    close(server_fd);
    ERR_EXIT("accept()");
}
port_num = ntohs(connect_addr.sin_port);
strcpy(ip_addr, inet_ntoa(connect_addr.sin_addr));
printf("connection accepted: port = %hu, IP addr = %s\n", port_num, ip_addr);
childpid = fork();
if(childpid < 0)
    ERR_EXIT("fork()");
if(childpid > 0) { /* parent pro */
    while(1) { /* sending cycle */
        memset(send_buf, 0, BUFFER_SIZE);
        fgets(send_buf, BUFFER_SIZE-1, stdin); /* including '\n' */
        sendbytes = send(connect_fd, send_buf, strlen(send_buf), 0);
        if(sendbytes <= 0) {
            printf("sendbytes = %d. Connection terminated ...\n", sendbytes);
            break;
        }
        if(strncmp(send_buf, "end", 3) == 0) break;
    }
    close(connect_fd);
    close(server_fd);
    kill(childpid, SIGKILL);
}
```

Linux: Socket Programming

Algorithm 11-2: socket-server-1.c (5)

```
        else { /* child pro */
            while(1) { /* receiving cycle */
                memset(recv_buf, 0, BUFFER_SIZE);
                recvbytes = recv(connect_fd, recv_buf, BUFFER_SIZE-1, 0);
                    /* waiting for client */
                if(recvbytes <= 0) {
                    printf("recvbytes = %d. Connection terminated ...\n", recvbytes);
                    break;
                }
                printf("\t\t\t\t\tserver %s say: %s", ip_addr, recv_buf);
                if(strncmp(recv_buf, "end", 3) == 0)
                    break;
            }
            close(connect_fd);
            close(server_fd);
            kill(getppid(), SIGKILL);
        }
        return EXIT_SUCCESS;
    }
```

Linux: Socket Programming

Algorithm 11-3: socket-input.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

#define TEXT_SIZE 1024
/* msg input terminal should be in the same host for socket sending */
int main(int argc, char *argv[])
{
    char fifoname[80], write_msg[TEXT_SIZE];
    int fdw, ret;

    if(argc < 2) {
        printf("Usage: ./a.out pathname\n");
        return EXIT_FAILURE;
    }
    strcpy(fifoname, argv[1]);
    if(access(fifoname, F_OK) == -1) {
        if(mkfifo(fifoname, 0666) != 0) {
            perror("mkfifo()");
            exit(EXIT_FAILURE);
        }
        else
            printf("new fifo %s created ...\n", fifoname);
    }
```

Linux: Socket Programming

Algorithm 11-3: socket-input.c (2)

```
    fdw = open(fifoname, O_RDWR); /* non-blocking send & receive */

    if(fdw < 0) {
        perror("pipe open()");
        exit(EXIT_FAILURE);
    }
    else {
        while (1) {
            printf("\nEnter some text: ");
            fgets(write_msg, TEXT_SIZE, stdin);
            ret = write(fdw, write_msg, TEXT_SIZE); /* non-blcoking send */
            if (ret <= 0) {
               perror("write()");
                close(fdw);
               exit(EXIT_FAILURE);
                                    }
           }
    }

    close(fdw);
    exit(EXIT_SUCCESS);
}
```

Linux: Socket Programming

Algorithm 11-4: socket-connector-w.c (1)

```
/* asynchronous send-receive version; separating
   input terminal */
int main(int argc, char *argv[])
{
    int connect_fd, sendbytes, recvbytes, ret;
    uint16_t port_num;
    char send_buf[BUFFER_SIZE], recv_buf[BUFFER_SIZE];
    char ip_name_str[INET_ADDRSTRLEN], stdin_buf[BUFFER_SIZE];
    char clr;
    struct hostent *host;
    struct sockaddr_in server_addr, connect_addr;
    socklen_t addr_len;
    pid_t childpid;
    char fifoname[80]; int fdr;

    if(argc < 2) {
        printf("Usage: ./a.out pathname\n");
        return EXIT_FAILURE;
    }
    strcpy(fifoname, argv[1]);
    if(access(fifoname, F_OK) == -1) {
        if (mkfifo(fifoname, 0666) != 0) {
            perror("mkfifo()");
            exit(EXIT_FAILURE);
        }
        else
            printf("new fifo %s named pipe created\n", fifoname);
    }
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/signal.h>
#include <fcntl.h>
#include <sys/stat.h>

#define BUFFER_SIZE 1024
#define ERR_EXIT(m) \
    do { \
        perror(m); \
        exit(EXIT_FAILURE); \
    } while(0)
```

Linux: Socket Programming

Algorithm 11-4: socket-connector-w.c (2)

```
fdr = open(fifoname, O_RDONLY);  /* blocking read */
if (fdr < 0) {
    perror("pipe read open()");
    exit(EXIT_FAILURE);
}

printf("Input server's hostname/ipv4: "); /* www.baidu.com or an ipv4 address */
scanf("%s", stdin_buf);
while((clr = getchar()) != '\n' && clr != EOF); /* clear the stdin buffer */
printf("Input server's port number: ");
scanf("%hu", &port_num);
while((clr = getchar()) != '\n' && clr != EOF);

if((host = gethostbyname(stdin_buf)) == NULL) {
    printf("invalid name or ip-address\n");
    exit(EXIT_FAILURE);
}
printf("server's official name = %s\n", host->h_name);
char** ptr = host->h_addr_list;
for(; *ptr != NULL; ptr++) {
    inet_ntop(host->h_addrtype, *ptr, ip_name_str, sizeof(ip_name_str));
    printf("\tserver address = %s\n", ip_name_str);
}
    /*creat connection socket*/
if((connect_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    ERR_EXIT("socket()");
}
```

Linux: Socket Programming

Algorithm 11-4: socket-connector-w.c (3)

```
        /* set sockaddr_in of server-side */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port_num);
    server_addr.sin_addr = *((struct in_addr *)host->h_addr);
    bzero(&(server_addr.sin_zero), 8);

    addr_len = sizeof(struct sockaddr);
    ret = connect(connect_fd, (struct sockaddr *)&server_addr, addr_len); /* connect to
server */
    if(ret == -1) {
        close(connect_fd);
        ERR_EXIT("connect()");
    }
        /* connect_fd is assigned a port_number after connecting */
    addr_len = sizeof(struct sockaddr);
    ret = getsockname(connect_fd, (struct sockaddr *)&connect_addr, &addr_len);
    if(ret == -1) {
        close(connect_fd);
        ERR_EXIT("getsockname()");
    }
    port_num = ntohs(connect_addr.sin_port);
    strcpy(ip_name_str, inet_ntoa(connect_addr.sin_addr));
    printf("Local port: %hu, IP addr: %s\n", port_num, ip_name_str);

    strcpy(ip_name_str, inet_ntoa(server_addr.sin_addr));
```

Linux: Socket Programming

Algorithm 11-4: socket-connector-w.c (4)

```
childpid = fork();
if(childpid < 0)
    ERR_EXIT("fork()");
if(childpid > 0) { /* parent pro */
    while(1) { /* sending cycle */
        ret = read(fdr, send_buf, BUFFER_SIZE); /* blocking read */
        if (ret <= 0) {
            perror("read()");
            break;
        }
        printf("pipe input: %s", send_buf);
        sendbytes = send(connect_fd, send_buf, strlen(send_buf), 0);
        if(sendbytes <= 0) {
            printf("sendbytes = %d. Connection terminated ...\n", sendbytes);
            break;
        }
        if(strncmp(send_buf, "end", 3) == 0)
            break;
    }
    close(fdr);
    close(connect_fd);
    kill(childpid, SIGKILL);
}
```

Linux: Socket Programming

Algorithm 11-4: socket-connector-w.c (5)

```
        else { /* child pro */
            while(1) { /* receiving cycle */
                memset(recv_buf, 0, BUFFER_SIZE);
                recvbytes = recv(connect_fd, recv_buf, BUFFER_SIZE-1, 0);
                    /* waiting for server */
                if(recvbytes <= 0) {
                    printf("recvbytes = %d. Connection terminated ...\n", recvbytes);
                    break;
                }
                printf("\t\t\t\t\tserver %s say: %s", ip_name_str, recv_buf);
                if(strncmp(recv_buf, "end", 3) == 0)
                    break;
            }
            close(connect_fd);
            kill(getppid(), SIGKILL);
        }
        return EXIT_SUCCESS;
    }
```

Linux: Socket Programming

Algorithm 11-5: socket-server-m.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <ifaddrs.h>
#include <sys/shm.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/wait.h>

#define BUFFER_SIZE 1024
#define MAX_QUE_CONN_NM 5
#define MAX_CONN_NUM 10

#define ERR_EXIT(m) \
    do { \
        perror(m); \
        exit(EXIT_FAILURE); \
    } while(0)
```

```
/* one server, m clients version */
int connect_sn, max_sn; /* from 0 to MAX_CONN_NUM-1 */
int server_fd, connect_fd[MAX_CONN_NUM];
int fd[MAX_CONN_NUM][2];
/* ordinary pipe: pipe_data() gets max_sn from main() */
int fdr;
/* named pipe: pipe_data() gets data from terminal input */
struct sockaddr_in server_addr, connect_addr;

int getipv4addr(char *ip_addr)
{
    struct ifaddrs *ifaddrsptr = NULL;
    struct ifaddrs *ifa = NULL;
    void *tmpptr = NULL;
    int ret;

    ret = getifaddrs(&ifaddrsptr);
    if (ret == -1)
        ERR_EXIT("getifaddrs()");

    for(ifa = ifaddrsptr; ifa != NULL; ifa = ifa->ifa_next) {
        if(!ifa->ifa_addr) {
            continue;
        }
```

Linux: Socket Programming

Algorithm 11-5: socket-server-m.c (2)

```
            if(ifa->ifa_addr->sa_family == AF_INET) { /* IP4 */
                tmpptr = &((struct sockaddr_in *)ifa->ifa_addr)->sin_addr;
                char addr_buf[INET_ADDRSTRLEN];
                inet_ntop(AF_INET, tmpptr, addr_buf, INET_ADDRSTRLEN);
                printf("%s IPv4 address %s\n", ifa->ifa_name, addr_buf);
                if (strcmp(ifa->ifa_name, "lo") != 0)
                    strcpy(ip_addr, addr_buf); /* return the ipv4 address */
            } else if(ifa->ifa_addr->sa_family == AF_INET6) { /* IP6 */
                tmpptr = &((struct sockaddr_in6 *)ifa->ifa_addr)->sin6_addr;
                char addr_buf[INET6_ADDRSTRLEN];
                inet_ntop(AF_INET6, tmpptr, addr_buf, INET6_ADDRSTRLEN);
                printf("%s IPv6 address %s\n", ifa->ifa_name, addr_buf);
            }
        }

        if (ifaddrsptr != NULL) {
            freeifaddrs(ifaddrsptr);
        }

        return EXIT_SUCCESS;
    }
```

Linux: Socket Programming

Algorithm 11-5: socket-server-m.c (3)

```
void pipe_data(void)
{
   /* read terminal input from alg.11-13-socket-input.c
      update max_sn from main()
      select connect_sn by the descritor @**** in start of send_buf */
    char send_buf[BUFFER_SIZE], sub_send_buf[BUFFER_SIZE];
    char stdin_buf[BUFFER_SIZE];
    int flags, sn, ret;

    while(1) {
        ret = read(fdr, send_buf, BUFFER_SIZE); /* blocking read named pipe*/
        if (ret <= 0) {
            perror("read()");
            break;
        }
        printf("pipe input: %s", send_buf);

        flags = fcntl(fd[0][0], F_GETFL, 0);
        fcntl(fd[0][0], F_SETFL, flags | O_NONBLOCK); /* set to non-blocking mode */
        ret = read(fd[0][0], stdin_buf, BUFFER_SIZE); /* non-blocking read ordinary
pipe */
        if (ret > 0) { /* max_sn changed */
            max_sn = atoi(stdin_buf);
            printf("max_sn changed to: %d\n", max_sn);
        }
```

Linux: Socket Programming

Algorithm 11-5: socket-server-m.c (4)

```
        if (send_buf[0] == '@') {
            sscanf(send_buf, "@%d %s", &sn, sub_send_buf);
            if (sn > 0 && sn <= max_sn) {
                ret = write(fd[sn][1], send_buf, BUFFER_SIZE); /* blocking write
ordinary pipe */
                if (ret <= 0) {
                    perror("write()");
                    break;
                }
            }
        }
        else {
            for (sn = 1; sn <= max_sn; sn++) {
                ret = write(fd[sn][1], send_buf, BUFFER_SIZE);
                if (ret <= 0)
                    perror("write()");
            }
        }
    }
    return;
}
```

Linux: Socket Programming

Algorithm 11-5: socket-server-m.c (5)

```
void recv_send_data(int sn)
{
    char recv_buf[BUFFER_SIZE], send_buf[BUFFER_SIZE];
    int recvbytes, sendbytes, ret, flags;

    while(1) { /* receiving cycle */
        flags = fcntl(connect_fd[sn], F_GETFL, 0);
        fcntl(connect_fd[sn], F_SETFL, flags | O_NONBLOCK); // set to non-blocking mode
        memset(recv_buf, 0, BUFFER_SIZE);
        recvbytes = recv(connect_fd[sn], recv_buf, BUFFER_SIZE-1, MSG_DONTWAIT);
            /* non-blocking recv */
        if(recvbytes > 0) {
            printf("\t\t\t\tconnection %d say: %s", sn, recv_buf);
        }
        flags = fcntl(fd[sn][0], F_GETFL, 0);
        fcntl(fd[sn][0], F_SETFL, flags | O_NONBLOCK); /* set to non-blocking mode */
        ret = read(fd[sn][0], send_buf, BUFFER_SIZE);
            /* non-blocking read ordinary pipe */
        if (ret > 0) {
            printf("sn = %d send_buf ready: %s", sn, send_buf);
            sendbytes = send(connect_fd[sn], send_buf, strlen(send_buf), 0);
        }
        sleep(1); /* heart beating */
    }
    return;
}
```

Linux: Socket Programming

Algorithm 11-5: socket-server-m.c (6)

```
int main(int argc, char *argv[])
{
    socklen_t addr_len;
    pid_t pipe_pid, recv_pid, send_pid;
    char stdin_buf[BUFFER_SIZE], ip4_addr[INET_ADDRSTRLEN];
    uint16_t port_num;
    int ret;
    char fifoname[80];

    if(argc < 2) {
        printf("Usage: ./a.out pathname\n");
        return EXIT_FAILURE;
    }
    strcpy(fifoname, argv[1]);
    if(access(fifoname, F_OK) == -1) {
        if (mkfifo(fifoname, 0666) != 0) {
            perror("mkfifo()");
            exit(EXIT_FAILURE);
        }
        else
            printf("new fifo %s named pipe created\n", fifoname);
    }
    fdr = open(fifoname, O_RDONLY);  /* blocking read */
    if (fdr < 0) {
        perror("pipe read open()");
        exit(EXIT_FAILURE);
    }
```

Linux: Socket Programming

Algorithm 11-5: socket-server-m.c (7)

```
for (int i = 0; i < MAX_CONN_NUM; i++) {
    pipe(fd[i]);
}

server_fd = socket(AF_INET, SOCK_STREAM, 0);
if(server_fd == -1) {
    ERR_EXIT("socket()");
}
printf("server_fd = %d\n", server_fd);

getipv4addr(ip4_addr);
printf("input server port number: ");
memset(stdin_buf, 0, BUFFER_SIZE);
fgets(stdin_buf, BUFFER_SIZE-1, stdin);
port_num = atoi(stdin_buf);

    /* set sockaddr_in */
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(port_num);
//    server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_addr.s_addr = inet_addr(ip4_addr);
bzero(&(server_addr.sin_zero), 8); /* padding with 0's */

int opt_val = 1;
setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt_val, sizeof(opt_val));
```

Linux: Socket Programming

Algorithm 11-5: socket-server-m.c (8)

```
addr_len = sizeof(struct sockaddr);
ret = bind(server_fd, (struct sockaddr *)&server_addr, addr_len);
if(ret == -1) {
    close(server_fd);
    ERR_EXIT("bind()");
}
printf("Bind success!\n");

ret = listen(server_fd, MAX_QUE_CONN_NM);
if(ret == -1) {
    close(server_fd);
    ERR_EXIT("listen()");
}
printf("Listening ...\n");

pipe_pid = fork();
if(pipe_pid < 0) {
    close(server_fd);
    ERR_EXIT("fork()");
}
if(pipe_pid == 0) {
    pipe_data();
    exit(EXIT_SUCCESS);
}
```

Linux: Socket Programming

Algorithm 11-5: socket-server-m.c (9)

```
    max_sn = 0;
    connect_sn = 1;
    while(1) {
        if(connect_sn >= MAX_CONN_NUM) {
           printf("connect_sn = %d out of range\n", connect_sn);
           break;
        }
        addr_len = sizeof(struct sockaddr); /* should be assigned each time accept()
called */
        connect_fd[connect_sn] = accept(server_fd, (struct sockaddr *)&connect_addr,
&addr_len);
        if(connect_fd[connect_sn] == -1) {
            perror("accept()");
            break;
        }
        port_num = ntohs(connect_addr.sin_port);
        strcpy(ip4_addr, inet_ntoa(connect_addr.sin_addr));
        printf("New connection sn = %d, fd = %d, IP_addr = %s, port = %hu\n",
connect_sn, connect_fd[connect_sn], ip4_addr, port_num);

        recv_pid = fork();
        if(recv_pid < 0) {
            perror("fork()");
            break;
        }
```

Linux: Socket Programming

Algorithm 11-5: socket-server-m.c (10)

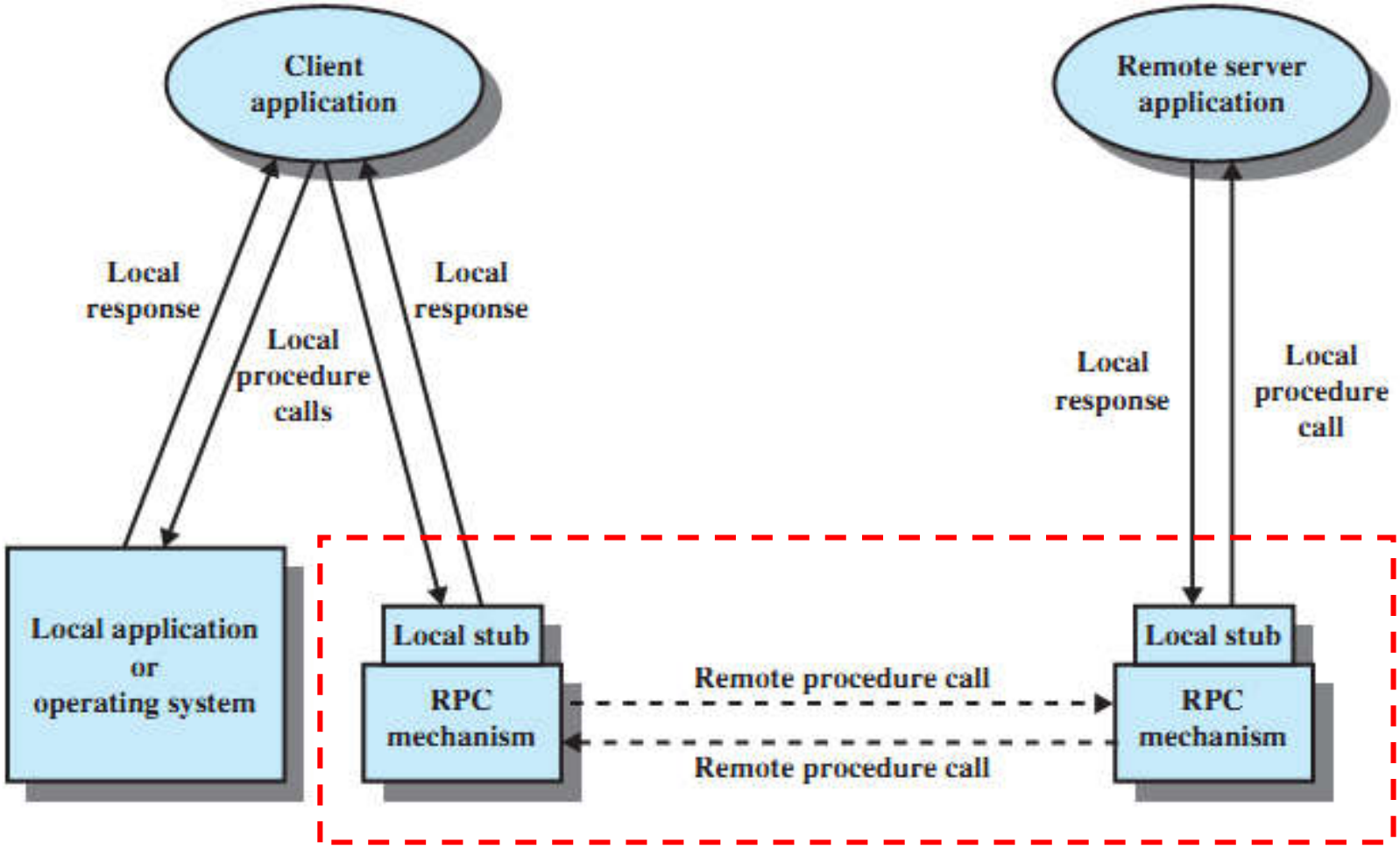
```
            if(recv_pid == 0) {
                recv_send_data(connect_sn);
                exit(EXIT_SUCCESS);
            }

            max_sn = connect_sn;
            sprintf(stdin_buf, "%d", max_sn);
            ret = write(fd[0][1], stdin_buf, BUFFER_SIZE);
            connect_sn++;
            /* parent pro continue to listen to a new client forever */
        }

        wait(0);
        for (int sn = 1; sn <= max_sn; sn++)
            close(connect_fd[sn]);
        close(server_fd);
        exit(EXIT_SUCCESS);
    }
```

Remote Procedure Calls

- Remote Procedure Calls (RPCs) abstracts a Local Procedure Call (LPC) between processes on a networked system.
- The semantics of RPCs allows a client to invoke a procedure on a remote host as it would invoke a procedure locally. The RPC system hides the details that allow communication to take place by providing a *stub* on the client side.
  - Stubs
    - client-side proxy for the actual procedure existing on the server.
- The client-side stub locates the server and *marshals* the parameters (将参数打包).
- The server-side stub/skeleton receives this message, unpacks the marshaled parameters, and performs the procedure on the server.

Remote Procedure Calls

- RPC mechanism.



Remote Procedure Calls

- Execution of a RPC.