Distributed Systems
分布式系统

Socket Communication

套接字通信（直接通信)

Clients and Servers

- A distributed system can be generally modeled by clients and servers (processes).
- Clients and servers are usually on different machines and they interact with each other via message passing.
- To offer a service, a server must get a **transport address** for a particular service
  - well-defined location

Transport Address

- A server (process) is associated with a transport address so that clients can communicate with it.
  - IP addresses identify machines
    - Not able to identify sending or receiving process
  - Transport layer uses **port number** to identify process

    machine address (IP address) → building address

    transport address (port number) → apartment number

- A client obtains the transport address via either:
  - hard coded
  - database (/etc/services, directory server)

Transport Layer Protocols

- Transport Layer supports communications between processes.

- Two categories of protocols:
  - **connection-oriented protocols**
  - **connectionless protocols**

Connection-oriented Protocols

1. establish connection
2. [negotiate protocol]
3. exchange data
4. terminate connection

analogous to phone call
dial phone number
[decide on a language]
speak
hang up

virtual circuit (stream) service

- provides illusion of having a dedicated circuit
- messages guaranteed to arrive in-order
- applications use connection ID, instead of address in each message

e.g., TCP – Transport Control Protocol

Connectionless Protocols

analogous to mailbox

- no call setup
- send/receive data
  (each packet addressed)
- no termination

drop letter in mailbox
(each letter addressed)

Datagram service

- client is not sure if messages are received by destination
- no state has to be maintained at client or server
- cheaper but less reliable than virtual circuit service

e.g., UDP – User Datagram Protocol

Sockets: Transport Layer Communication

- A popular abstraction for transport layer communication
  - Developed at Berkeley in 1982
- Goals:
  - Communication between processes should not depend on whether they are on the same machine
    - Uniform all data exchanges (accesses) as file accesses
  - Application can select particular style of communication
    - Virtual circuit, datagram, message-based, in-order delivery
  - Support different protocols and naming conventions (not just for TCP/IP or UDP/IP)

Programming operations

client	server
1. Create a socket 2. Construct server's address	1. Create a socket 2. Bind socket to an address
3c. Connect to server's socket	3a. Set the socket for listening 3b. Accept a connection
4. Communicate 5. Close the socket	4. Communicate 5. Close the socket

Steps 3a-3c are not needed for connectionless communication

Socket Operations

List of socket operations:

1. socket
2. bind
3. listen, accept, connect
4. read/write, send/recv, sendto/recvfrom, sendmsg/recvmsg
5. close/shutdown

Create a socket

socket system call:

int *s* = **socket**(*domain*, *type*, *protocol*);

parameters:

domain: identifies address family
AF_INET: IPC on the Internet,
AF_UNIX: IPC within a computer
AF_NS: IPC on Xeroxs Network Systems

type: type of service required by application
SOCK_STREAM: virtual circuit
SOCK_DGRAM: datagram
SOCK_RAW: raw IP access

protocol: specify a protocol. To support user self defined protocols.
Default value is 0, i.e., system defined protocol.

return: an integer as the socket number (file descriptor)

Bind socket to an address

bind system call:

int *error* = **bind**(*s*, *addr*, *addrlen*);

parameters:

s:	socket descriptor returned by *socket()*
addr:	address structure (struct sockaddr *)
addrlen:	length of address structure

return: error code

Binding a file name to a UNIX socket (intra-machine communication)

/usr/include/sys/un.h

```
struct  sockaddr_un {
        sa_family_t    sun_family;          /* AF_UNIX */
        char           sun_path[108];       /* path name */
};
```

```
struct sockaddr_un addr;
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind (s, &addr, strlen(addr.sun_path) + sizeof addr.sun_family);
```

Demo of UNIX Stream Sockets

www/C/socket/unix/receiver.c

```
main(argc, argv) int argc; char *argv[]; {
   struct  sockaddr_un myname;
   int s, new_s;
   char buf[256], rdata[256];
   s = socket(AF_UNIX, SOCK_STREAM, 0);
   myname.sun_family = AF_UNIX;
   strcpy(myname.sun_path, "/tmp/123");
   bind(s, &myname, strlen(myname.sun_path) +
      sizeof(myname.sun_family))
   listen(s, 5);
   new_s = accept(s, NULL, NULL);
   while (1) {
      read(new_s, buf, sizeof(buf));
      strcpy(rdata, “Echoed msg: ");
      strcat(rdata, buf);
      write(new_s, rdata, strlen(rdata))
   }
}
```

www/C/socket/unix/sender.c

```
main(argc, argv) int argc; char *argv[]; {
    struct  sockaddr_un hisname;
    int s;
    char buf[256], data[80];
    s = socket(AF_UNIX, SOCK_STREAM, 0);
    hisname.sun_family = AF_UNIX;
    strcpy(hisname.sun_path, "/tmp/123");

    connect(s, &hisname, trlen(hisname.sun_path)+
                        sizeof (hisname.sun_family));
    scanf("%s", data);
    while (data[0] != '.') {
        write(s, data, strlen(data));
        read(s, buf, sizeof(buf))
        printf ("Received reply: %s\n", buf);
        scanf("%s", data);
    }
}
```

Binding an Internet address to a socket (Internet communication)

/usr/include/netinet/in.h

```
struct sockaddr_in    {
        short             sin_family;
        u_short           sin_port;
        struct in_addr    sin_addr;
        char              sin_zero[8];
}
```

```
struct sockaddr_in      addr;
unsigned char ip_addr[] = {144, 214, 120, 114};
addr.sin_family = AF_INET;
addr.sin_port   = htons(PORT);
bcopy(ip_addr, & addr.sin_addr, 4);
bind(s, & addr, sizeof(addr));
```

Server: set socket for listening

listen system call:

int *error* = **listen**(*s*, *backlog*);

parameters:

s: socket descriptor returned by *socket()*

backlog: queue length for pending connections

return: error code

Server: accept connections

accept system call:

int *snew* = **accept**(*s*, *clntaddr*, *addrlen*);

parameters:

s: socket descriptor returned by *socket()*

clntaddr: struct sockaddr * contain returned client addr

addrlen: int * contain length of client addr

return: a new socket to be used for this communication session

Client: connect

connect system call:

int *error* = **connect**(*s*, *svraddr*, *addrlen*);

parameters:

s:	socket descriptor returned by *socket()*
svraddr:	struct sockaddr * contains server address
addrlen:	length of server address

return: error code

Exchange data and close sockets

Exchanging data via sockets is the same as file access:

read(s, buf, length);

write(s, buf, length);

Close a socket by:

close(s); or

shutdown(int s, int how);

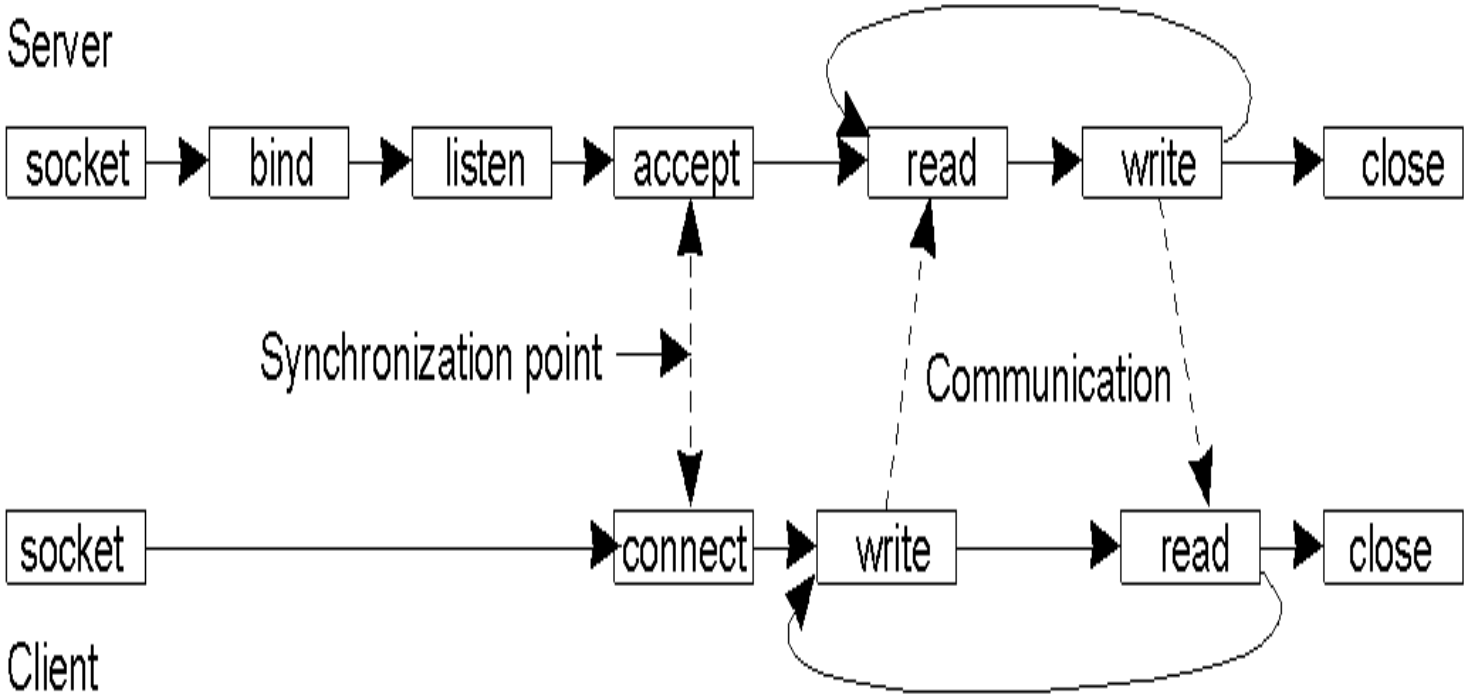
s: socket

how: 0: further receives disallowed

1: further sends disallowed

2: further sends and receives disallowed

Client-Server Synchronization



Demo of Internet Sockets

www/C/socket/inet/receiver.c

```
main (argc, argv) {
      int s, new_s;
      struct sockaddr_in server;

      s = socket (AF_INET, SOCK_STREAM, 0);
      server.sin_family = AF_INET;
      server.sin_addr.s_addr = INADDR_ANY;
      server.sin_port = 10000;
      bind (s, &server, sizeof (server));
      listen( s, 5)
      while (1) {
          tmp_len = sizeof(remote_addr);
          new_s = accept(s, &remote_addr, &tmp_len);
          while (read(new_s, &ch, 1) > 0) {
                write(new_s, &ch, 1);
                putchar(ch);
     }
}
```

www/C/socket/inet/sender.c

```
main (argc, argv) {
      int s;
      struct sockaddr_in remote_addr;
      struct hostent *remote_ent; // def in netdb.h
      s = socket(AF_INET, SOCK_STREAM, 0);
      remote_addr.sin_family = AF_INET;
      remote_addr.sin_port = htons(PORT);
      remote_ent = gethostbyname(argv[1]);
      bcopy(remote_ent->h_addr_list[0],
              &remote_addr.sin_addr,
              remote_ent->h_length);
      connect(s, &remote_addr, sizeof(remote_addr));
      while ((ch=getchar()) != '.') {
            write(s, &ch, 1);
            read(s, &ch, 1);
            putchar(ch);
     }
}
```

Data Structure for Socket Operations

- Client only sends data to {machine, port}
- How to keep track of simultaneous sessions to the same server process (e.g., HTTP server)?
- OS maintains a structure called the **Protocol Control Block** (PCB) 协议控制模块

Protocol Control Block

Each entry of PCB contains:

- Local address
- Local port
- Foreign address
- Foreign port
- Is the socket used for listening?
- Reference to the socket (file descriptor)

socket(): Allocate a new empty entry in PCB table

client:

	local addr	local port	foreign addr	foreign port	listen?
s →	0	0	0	0	

server:

	local addr	local port	foreign addr	foreign port	listen?
s →	0	0	0	0	

bind(): Assign local address, port

client:

	local addr	local port	foreign addr	foreign port	listen?
s →	0.0.0.0	7801	0	0	

server:

	local addr	local port	foreign addr	foreign port	listen?
s →	0.0.0.0	1234	0	0	

listen(): Set socket for receiving connections

client:

	local addr	local port	foreign addr	foreign port	listen?
s →	0.0.0.0	7801	0	0	

server:

	local addr	local port	foreign addr	foreign port	listen?
s →	0.0.0.0	1234	0	0	Yes

connect(): Send a *connect* request to server

request from [135.250.68.3:7801] to [192.11.35.15:1234]

client:

	local addr	local port	foreign addr	foreign port	listen?
s →	0.0.0.0	7801	192.11.35.15	1234	

server:

	local addr	local port	foreign addr	foreign port	listen?
s →	0.0.0.0	1234	0	0	Yes

accept(): Send an acknowledgement to client

ACK. from [192.11.35.15:1234] to [135.250.68.3:7801]

client:

	local addr	local port	foreign addr	foreign port	listen?
s →	0.0.0.0	7801	192.11.35.15	1234	

server:

	local addr	local port	foreign addr	foreign port	listen?
s →	0.0.0.0	1234	0	0	Yes
snew →	192.11.35.15	1234	135.250.68.3	7801	

Message Exchanges via Sockets

- Each message from client is tagged as either *data* or *control mesg* (e.g. *connect*).
- If data – search through table where foreign addr and foreign port match incoming message and *listen* is not set.
- If control – search through table where the local port matches the dest port in the message and *listen* is set.

server:

	local addr	local port	foreign addr	foreign port	listen?
s →	0.0.0.0	1234	0	0	Yes
snew →	192.11.35.15	1234	135.250.68.3	7801	

Datagram Sockets

Create sockets, bind addresses, close sockets are the same as stream sockets, but no listen, accept and connect operations.

Data exchange via:

sendto/recvfrom

int sendto(int s, void *msg, int len, int flags, struct sockaddr *to, int tolen)

int recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, int *fromlen)

flags is usually set to 0. It's for out-of-band data if it's non-zero.

sendmsg/recvmsg

int sendmsg(int s, struct msghdr *msg, int flags)

int recvmsg(int s, struct msghdr *msg, int flags)

Demo of Datagram sockets

www/C/socket/inet/recvfrom.c

```
main (argc, argv) {
  struct sockaddr_in name;
  s = socket (AF_INET, SOCK_DGRAM, 0);
  name.sin_addr.s_addr = INADDR_ANY;
  name.sin_port = 12000;
  bind (s, &name, sizeof name);
  while (1) {
     recvfrom (s, buf, sizeof(buf), 0,
            &from, &len_from);
     strcpy(rdata, "echoed string: ");
     strcat(rdata, buf);
     sendto (s, rdata, sizeof(rdata), 0,
            &from, from_len)
  }
}
```

www/C/socket/inet/sendto.c

```
main (argc, argv) {
      struct sockaddr_in recv;
      s = socket (AF_INET, SOCK_DGRAM, 0);
      hp = gethostbyname(argv [ 1]);
      bcopy (hp->h_addr_list[0],
            &recv.sin_addr, hp->h_length);
      recv.sin_port = htons(12000);
      scanf("%s", data);
      while (data[0] != '.') {
        sendto(s, data, sizeof(data), 0,
            &recv, sizeof(recv));
        recvfrom(s,buf, sizeof (buf), 0,NULL,NULL);
       printf("Received Reply: %s\n", buf);
       scanf("%s", data);
      }
}
```

Distributed System
分布式系统

RPC

远程过程调用

An Example of Local Procedure Call

```
/* save a string to file msg.dat */
savemsg(char *msg) {
        FILE *fp;
fp = fopen("msg.dat", "w+");
 fprintf(fp, "%s\n", msg);
        fclose (fp);
        }

        main() {
char *str = "This is an example of LPC.";
        savemsg(str);
        }
```

Moving the local procedure to a remote machine

server:

```
.....
savemsg_1(char **argp; struct svc_req *rqstp)
{
    FILE *fp;
  fp = fopen("msg.dat", "w+");
  fprintf(fp, "%s\n", *argp);
      fclose (fp);
}
```

client:

```
main() {
   char *str = "This is an example of RPC.";
  char *remote_host = "sus1.cs.cityu.edu.hk";
      CLIENT *clnt;
clnt = clnt_create(remote_host, MSGPROG, MSGVER, "netpath");
    if (clnt == (CLIENT *) NULL) {
     clnt_pcreateerror(host); exit(1);
                }
     savemsg_1(&str, clnt);
}
```

Remote Procedure Call

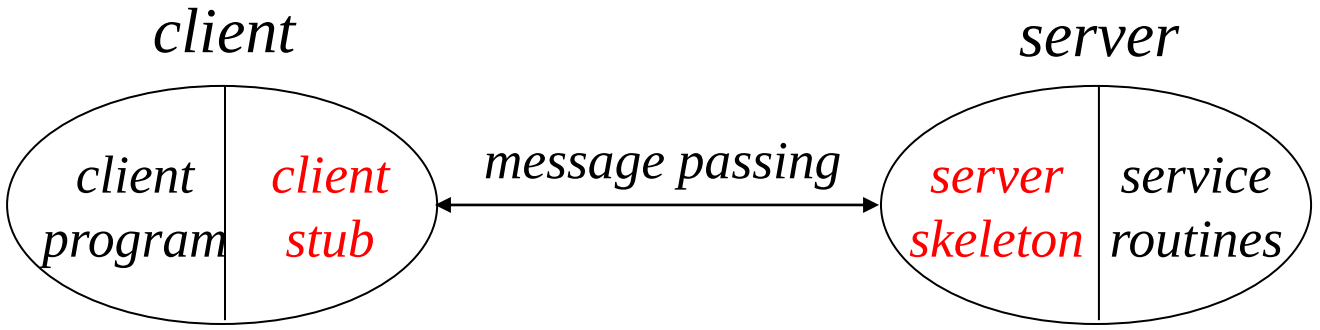
1984: Birrell & Nelson

- Mechanism to call procedures on other machines
- Process on machine *A* can call a procedure on machine *B*
  - *A* is suspended
  - Execution continues on *B*
  - When *B* returns, control passed back to *A*

Goal: Make a remote procedure call looking the same as a local call to programmers.

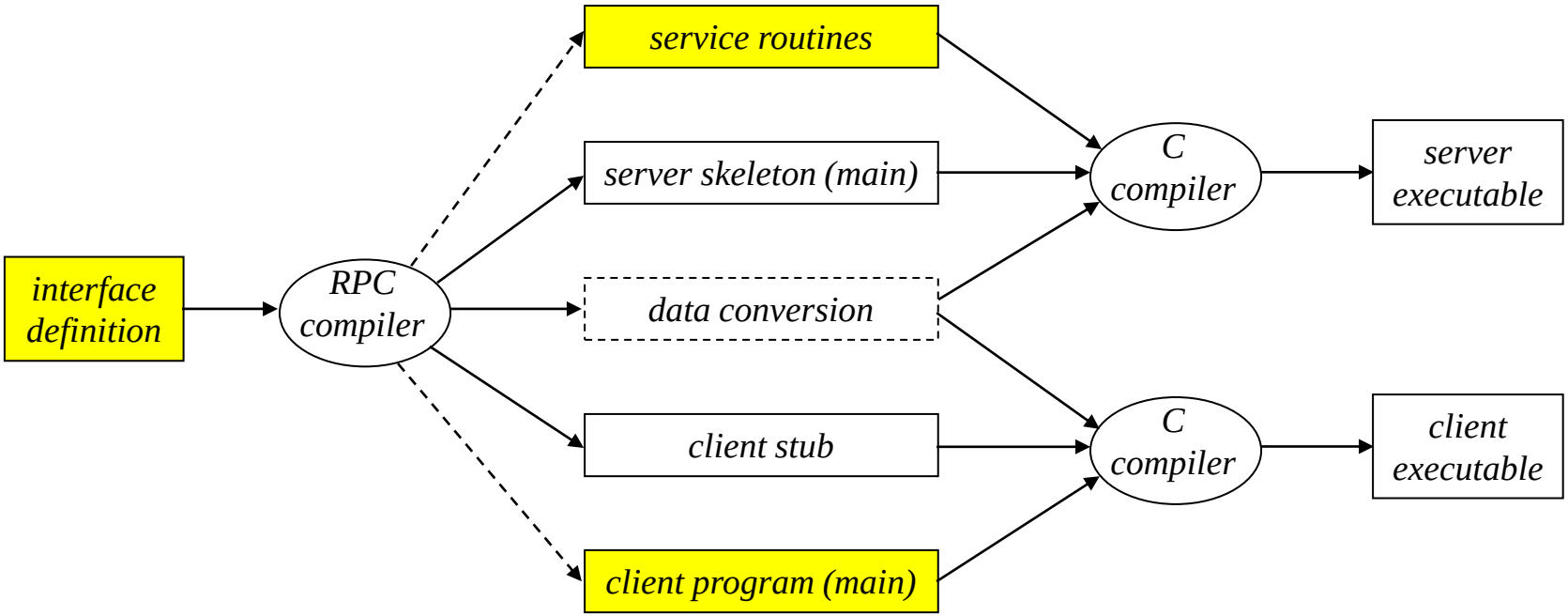
RPC implementation

The RPC compiler auto-generates **stub/skeleton routines** to make an RPC to the user as if it is a local call. What stub routines do:



- marshalling / unmarshalling parameters and return values
- handling different data formats between different machines
- detecting and handling failures of client/server processes and the networks

Compilation in SUN RPC



Demo of RPC

Interface file msg.x:

```
program MSGPROG {
        version MSGVER{
                int SAVEMSG(string)= 1;
                string READMSG(int)= 2;
        } = 2;
} = 345678;
```

Generate stub routines by:

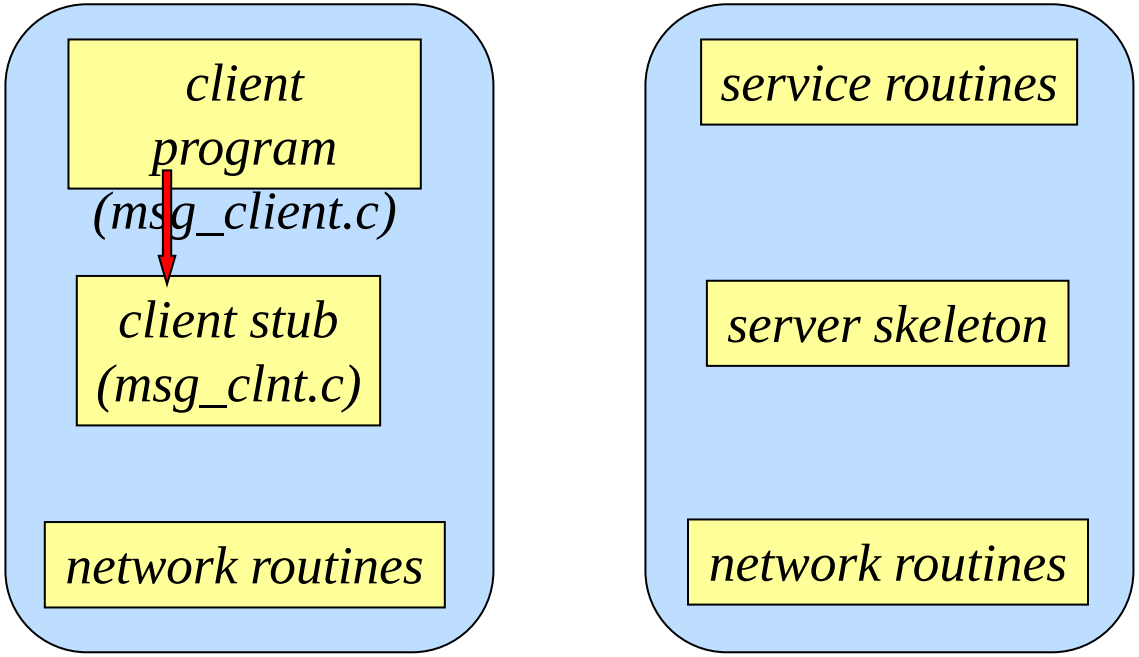
```
rpcgen –a msg.x
```

Compile program by:

```
cc –o object xxx.c
```

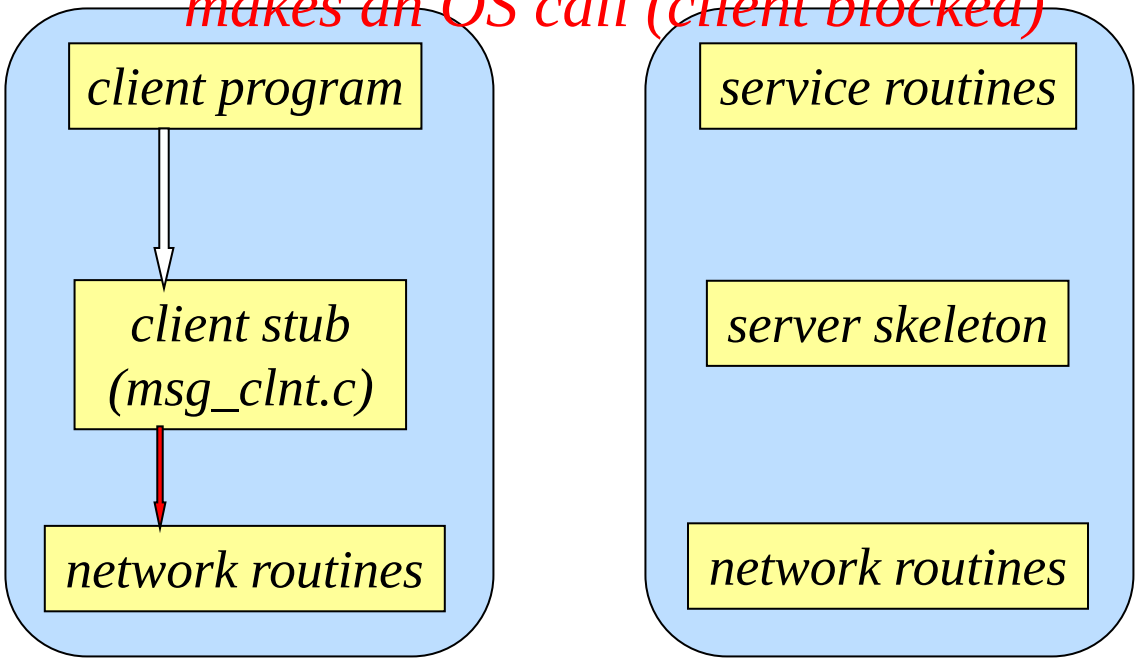
Steps in a RPC

1. Client calls stub (push parameters onto stack)



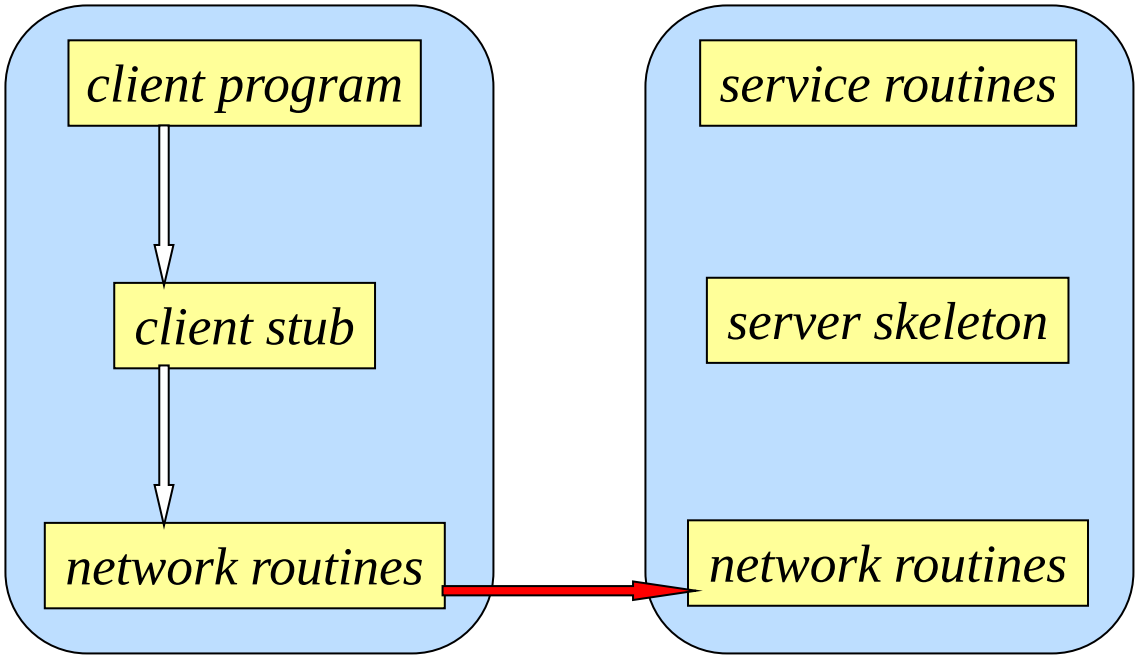
Steps in a RPC

2. Clnt_stub marshals parameters to message
&
makes an OS call (client blocked)



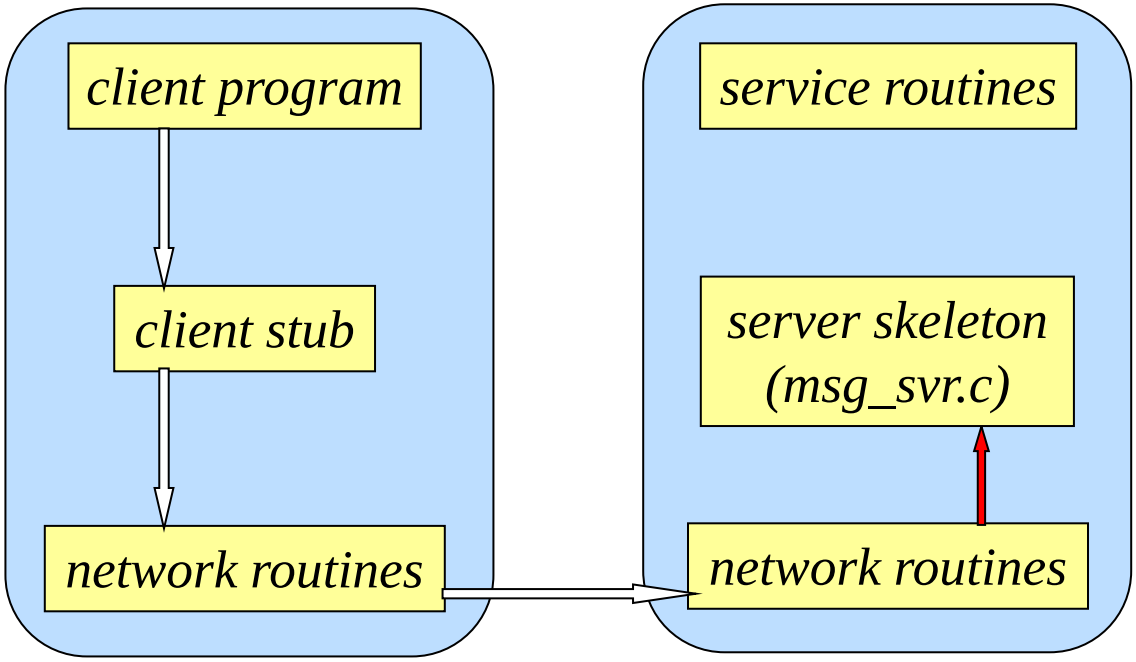
Steps in a RPC

3. Network message sent to server



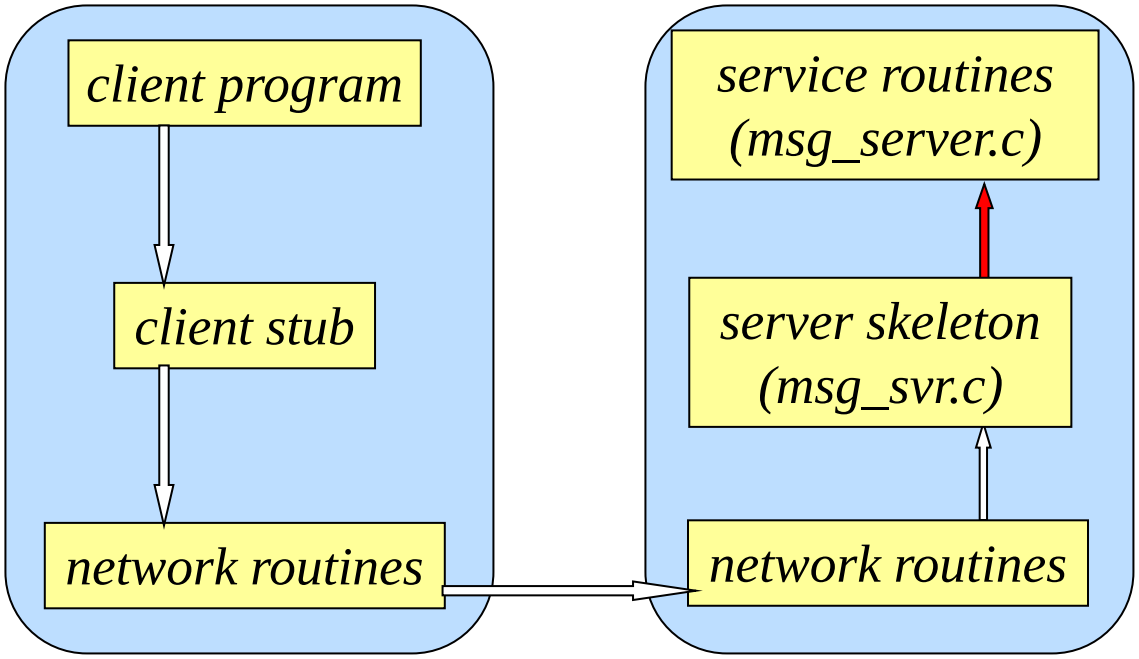
Steps in a RPC

4. Deliver message to server stub & unblock server



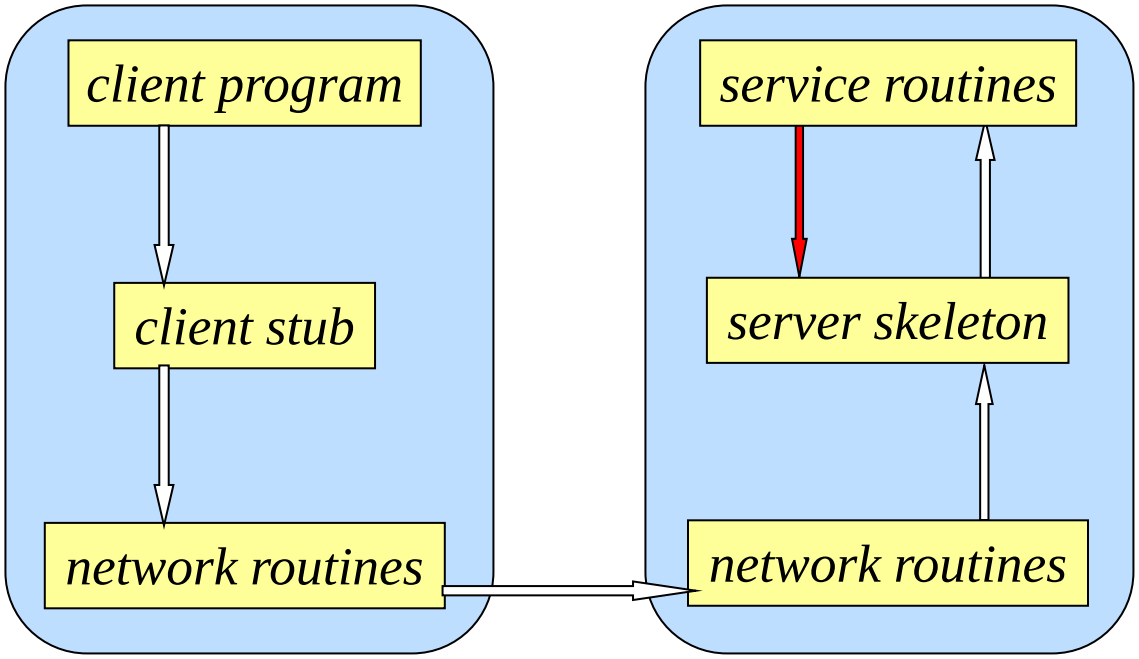
Steps in a RPC

5. Svr-stub unmarshals parameters & calls service routine (local call)



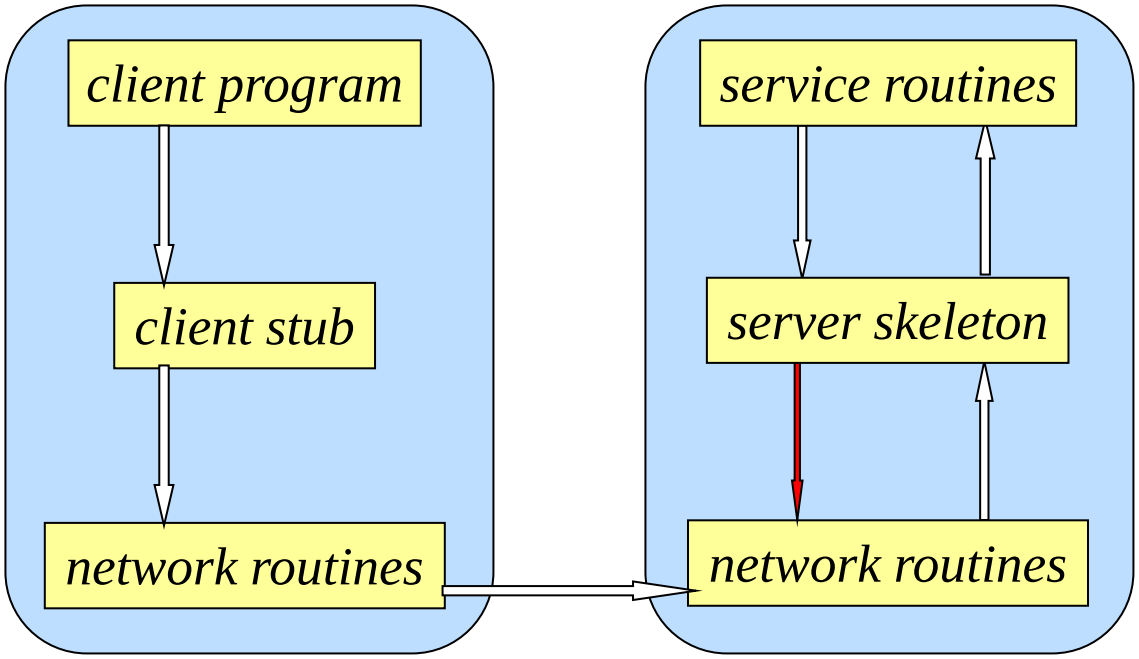
Steps in a RPC

6. Return to the stub from service routine



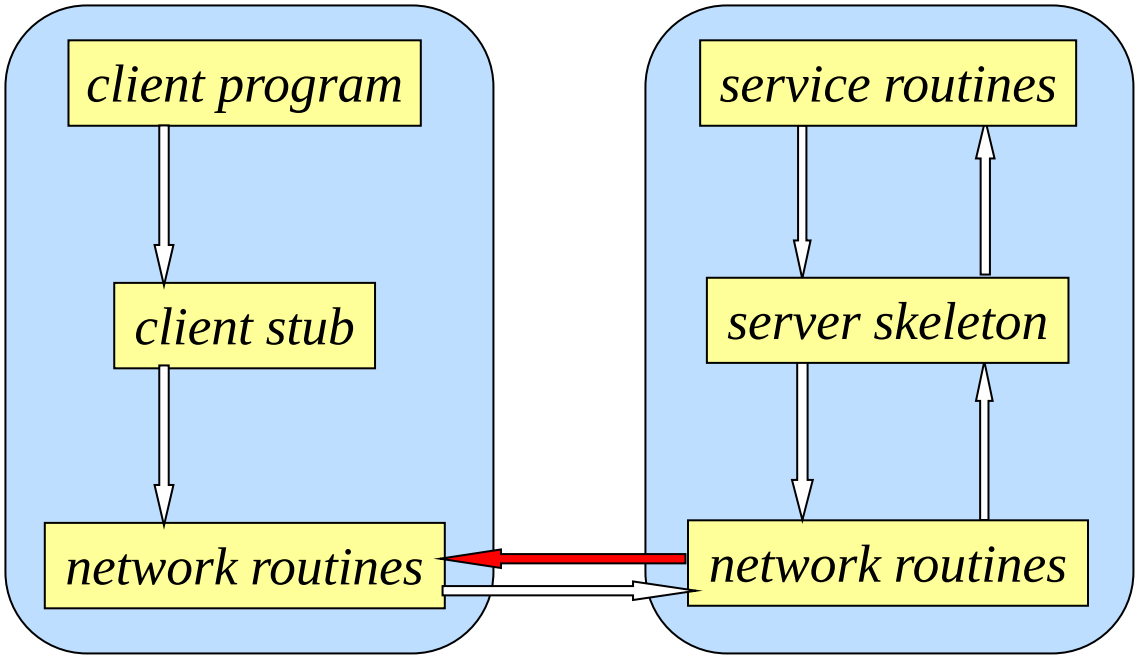
Steps in a RPC

7. Svr_stub marshals return-value & requests OS to send out message



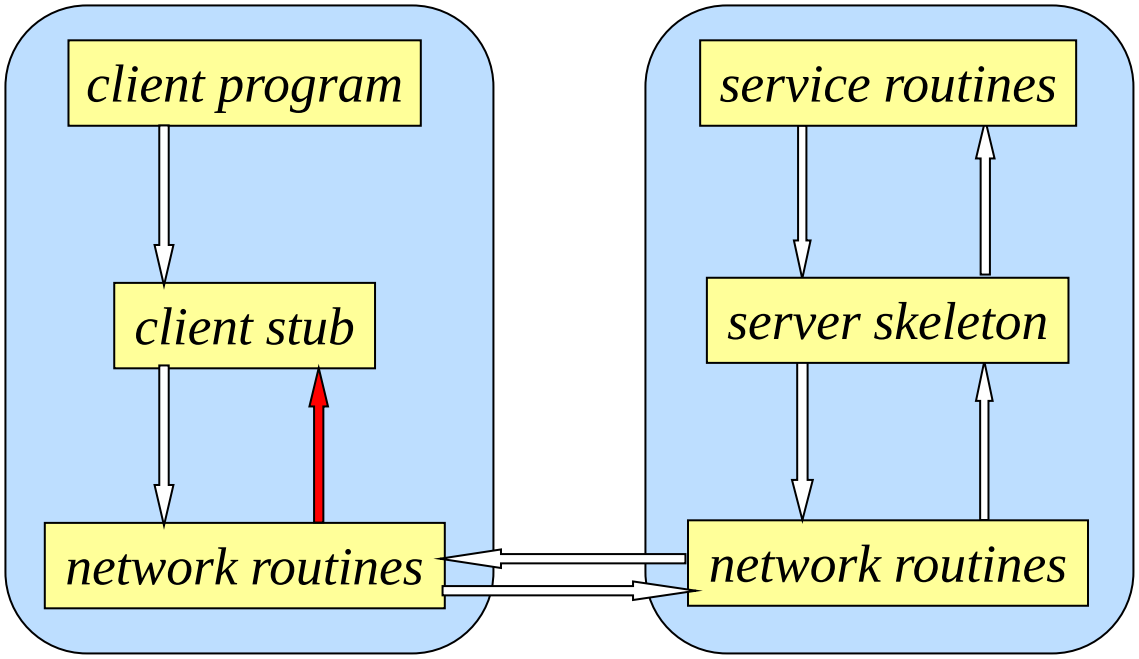
Steps in a RPC

8. Transfer message over network



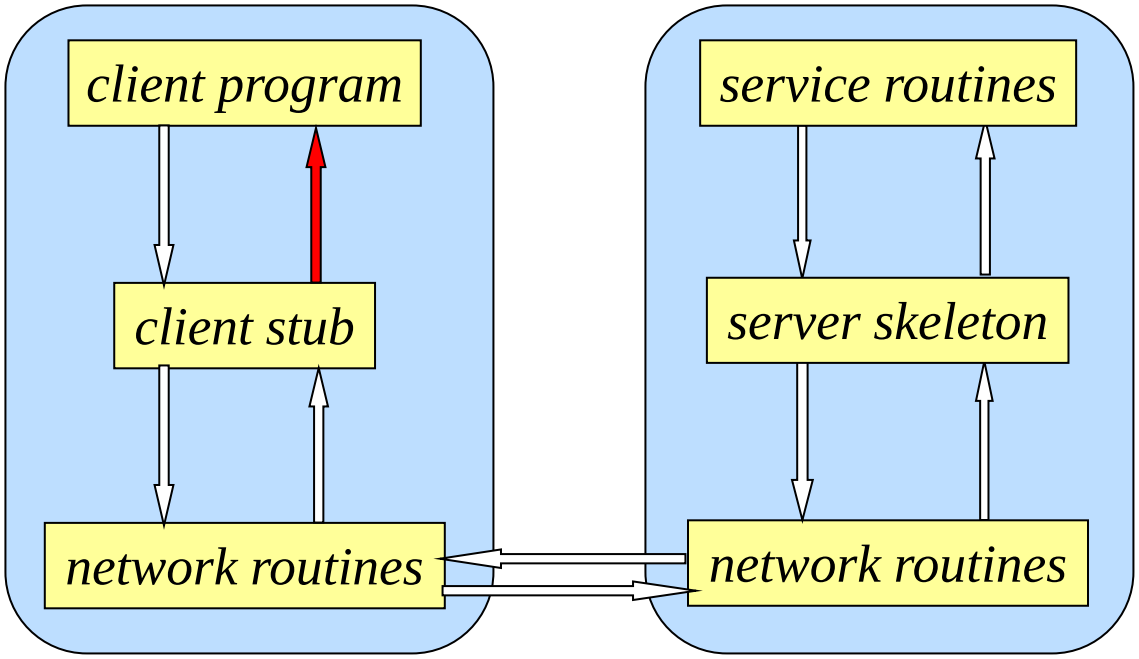
Steps in a RPC

9. Deliver message to client (unblock client)



Steps in a RPC

10. Clnt_stub unmarshals return-value & returns to client program…



Writing the programs

Programmers need to write two pieces of programs:

- Client program
  - Specify server's location
  - Parameters and return value of RPC are pointers

- Service routines
  - Generally the same as local procedures, except the parameters and parameter types