Communication in Distributed Systems

Outline

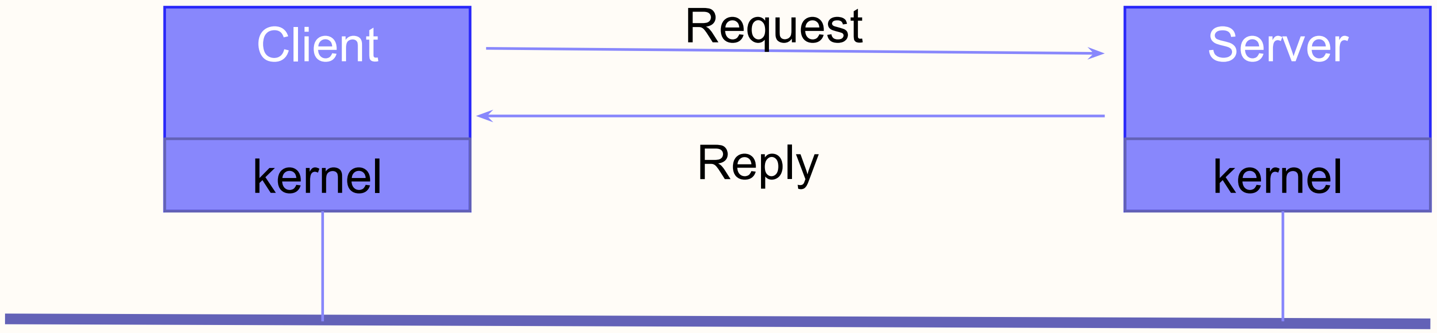
- Interprocess communication
- Client server model
- distributed system, it is completely different from uniporcessor system as there is no shared memory.
-

Interprocess Communication

- In distributed system, it is completely different from uniporcessor system as there is no shared memory.
- Certain rules need to be followed for interprocess communication called Protocols.
- For wide area distributed systems, these protocols take the form of multiple layers such as OSI , ATM.
- OSI model addresses only a small aspect of the communication - sending bits from the sender to the receiver, with much overheads.

Client - Server model

- It is based on simple connectionless request / reply protocol.
- Client sends a request message to the server and the server returns the data requested or an error code indicating the reason of failure.
- It is simple. No connection to be established before use and no connection to be closed after use.
- Simplicity leads to efficiency. Only three levels of protocol are needed.



Client - Server model

- Physical and datalink protocol take care of getting the packets from client to server and back.
- No routing and no connections - layers 3 & 4 not needed.
- Layer 5 is the request/ reply protocol. No sessions required.
- Communication provided by the micro-kernels using two system calls -
    - *send (dest, &mptr)*
    - *receive(addr, &mptr)*

mptr - message pointer

dest - destination process

addr - source address

7	
6	
5	Request / reply
4	
3	
2	Datalink
1	Physical

Example - client and server

```
/* Definitions needed by clients and servers.                    */
#define TRUE             1
#define MAX_PATH         255   /* maximum length of file name          */
#define BUF_SIZE         1024  /* how much data to transfer at once    */
#define FILE_SERVER      243   /* file server's network address        */

/* Definitions of the allowed operations */
#define CREATE           1     /* create a new file                    */
#define READ             2     /* read data from a file and return it  */
#define WRITE            3     /* write data to a file                 */
#define DELETE           4     /* delete an existing file              */

/* Error codes. */
#define OK               0     /* operation performed correctly        */
#define E_BAD_OPCODE     -1    /* unknown operation requested          */
#define E_BAD_PARAM      -2    /* error in a parameter                 */
#define E_IO             -3    /* disk error or other I/O error        */

/* Definition of the message format. */
struct message {
      long source;               /* sender's identity                  */
      long dest;                 /* receiver's identity                */
      long opcode;               /* requested operation                */
      long count;                /* number of bytes to transfer        */
      long offset;               /* position in file to start I/O      */
      long result;               /* result of the operation            */
      char name[MAX_PATH];       /* name of file being operated on     */
      char data[BUF_SIZE];       /* data to be read or written         */
};
```

header.h

Example - client and server

```
#include <header.h>
void main(void) {
    struct message ml, m2;              /* incoming and outgoing messages     */
    int r;                              /* result code                        */

    while(TRUE) {                       /* server runs forever                */
        receive(FILE_SERVER, &ml);      /* block waiting for a message        */
        switch(ml.opcode) {             /* dispatch on type of request        */
            case CREATE:    r = do_create(&ml, &m2); break;
            case READ:      r = do_read(&ml, &m2); break;
            case WRITE:     r = do_write(&ml, &m2); break;
            case DELETE:    r = do_delete(&ml, &m2); break;
            default:        r = E_BAD_OPCODE;
        }
        m2.result = r;                  /* return result to client            */
        send(ml.source, &m2);           /* send reply                         */
    }
}
```

A sample server.

Example - client and server

(a)

```
#include <header.h>
int copy(char *src, char *dst){            /* procedure to copy file using the server    */
    struct message ml;                     /* message buffer                             */
    long position;                         /* current file position                      */
    long client = 110;                     /* client's address                           */

    initialize( );                         /* prepare for execution                      */
    position = 0;
    do {
        ml.opcode = READ;                  /* operation is a read                        */
        ml.offset = position;              /* current position in the file               */
        ml.count  = BUF_SIZE;                                                            /* how many bytes to read*/
        strcpy(&ml.name, src);             /* copy name of file to be read to message    */
        send(FILESERVER, &ml);             /* send the message to the file server        */
        receive(client, &ml);              /* block waiting for the reply                */

        /* Write the data just received to the destination file.                         */
        ml.opcode = WRITE;                 /* operation is a write                       */
        ml.offset = position;              /* current position in the file               */
        ml.count  = ml.result;             /* how many bytes to write                    */
        strcpy(&ml.name, dst);             /* copy name of file to be written to buf     */
        send(FILE_SERVER, &ml);            /* send the message to the file server        */
        receive(client, &ml);              /* block waiting for the reply                */
        position += ml.result;             /* ml.result is number of bytes written       */
    } while( ml.result > 0 );              /* iterate until done                         */
    return(ml.result >= 0 ? OK : ml result); /* return OK or error code                  */
}
```

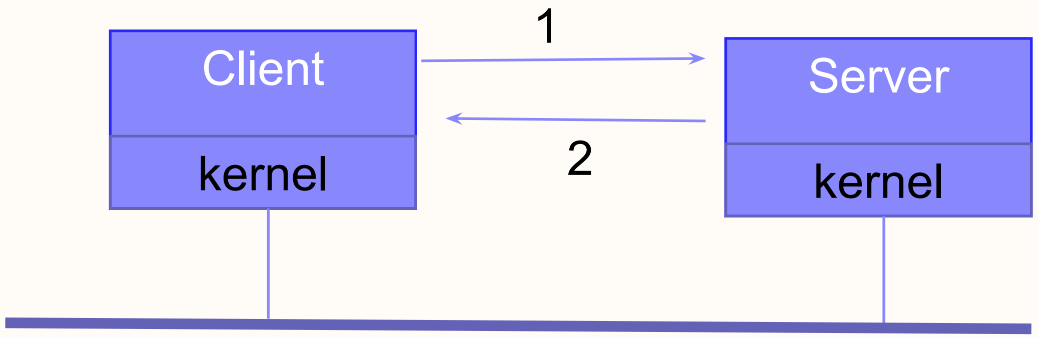
A client using the server to copy a file.

Addressing

- One way of mentioning server address is to mention it in ***header.h*** as a constant.
- Sending kernel can extract it (ex - 243 refering to machine) from message structure and use it for sending packets to server.
- Ambiguity arises if multiple processes are running on the same server.

Alternative 1 -

- Send messages to processes , not machines.
- Process identification - two part names - machine + process no.
  Ex - 243.4 or 4@243
- Each machine can number its processes starting from 0. So there is no confusion between process ‘n’ of different machines.
- No global coordination is required.



1. Request to 243.0
2. Reply to 199.0

Addressing

Alternative 2 -

- use *machine.local-id* instead of *machine.process*
- Each process is assigned a *local-id* and informs kernel that it listens to *local-id*
- Problem - user is aware of the location of the machine (243). If the machine is down, compiled programs with *header.h* will not work, although another machine (365) is available. No transparency.

Alternative 3-

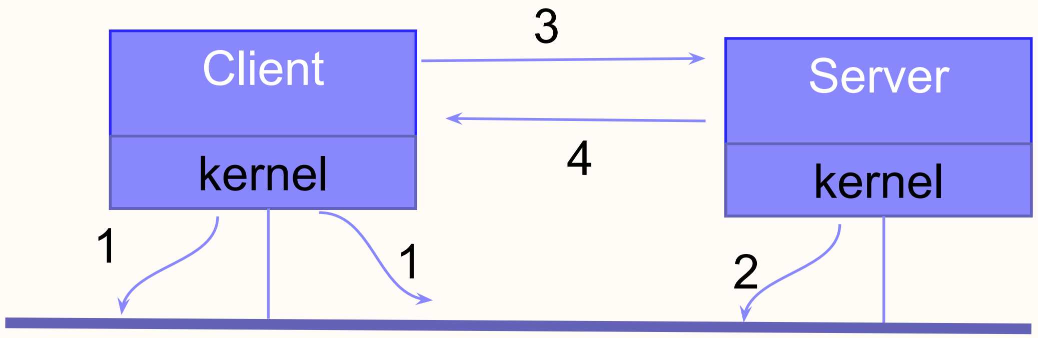
- Each process has a unique address that doesn't contain machine number.
- A centralized **process address allocator** maintains a counter. Upon receiving a request, it returns the current value of the counter.
- Problem - Such centralized components do not scale to large systems.

Addressing

Alternative 4 -

- Each process picks its own identifier from a large address space (space of 64 bit binary integer).
- It is scalable.
- Identification of machine -Sender broadcasts a special **Locate packet** containing the address of the destination process. It will be received by all machines on the network. The matched kernel respondes with a message “here i am” alongwith the machine number. So the sending kernel uses this machine number for further communication.
- Problem - Broadcasting is an overload to the system.

1. Broadcast
2. Here I am
3. Request
4. Reply



Addressing

Alternative 5 -

- Overload can be avoided by providing an extra machine to map high-level (ASCII) service names to machine address.
- These names are embeded in the programs, not binary machine numbers.
- For the first time client sends a query to the Name server, asking the machine number where the server is currently located. Then the request can be sent directly to the machine address.
- Problem - If the name server is replicated, consistency problem may arise.

1.Lookup
2. NS reply
3. Request
4. Reply