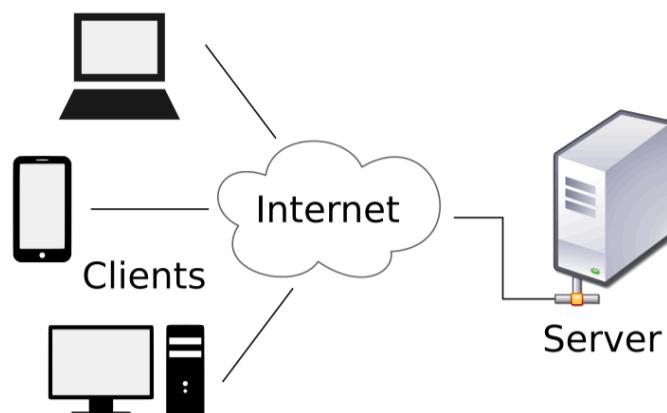


CLIENT/SERVER



C'è da qualche parte un processo P che, dopo aver fatto qualcosa, rimane in attesa (chiamando qualche API) che arrivi una richiesta per svolgere del lavoro, proveniente da un altro processo P' che, se parte, ovviamente chiama anche lui qualche API per far sì che questa richiesta possa essere inviata verso P che aspettava.

Utilizzando la message queue questa cosa è plausibile tanto è vero che le message queues vengono utilizzate in tanti ambiti UNIX per creare applicazioni client/server che lavorino sulla stessa istanza di sistema unix.

Quindi P' lancia un messaggio M a P, P estrae questo messaggio, riprende la sua esecuzione, e P' si mette in attesa di una risposta. E quindi P dovrà fornire un messaggio M' a P'.

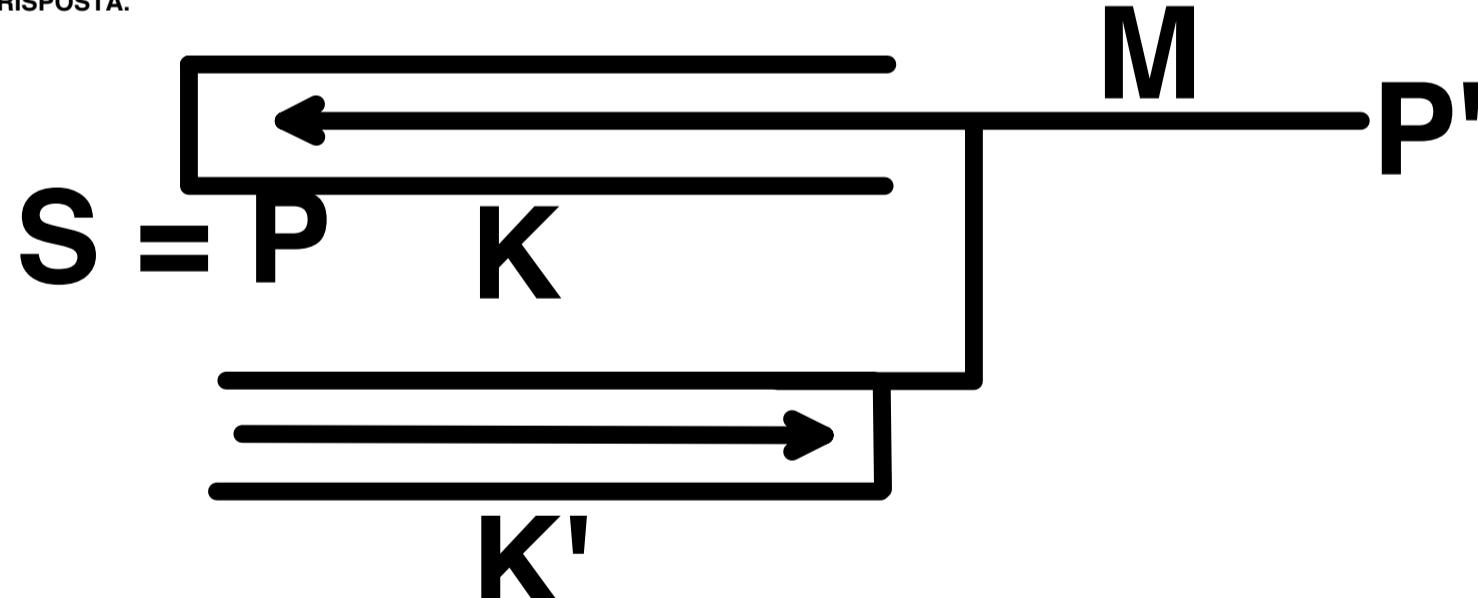
M gira dal client al server e M' che gira dal server al client.

Il server installa una message queue per ricevere richieste che provengono da altri processi P' e P'', per esempio due client. P' può scrivere una richiesta dentro e P'' anche.



P quando va ad inoltrare le risposte NON UTILIZZA PIÙ questa coda di messaggi, perché viene usata solo per le richieste, ma utilizziamo altre message queues, ossia una message queue predisposta che il server utilizza per poi mandare la risposta che deve essere estratta dal processo P', e per l'altro processo abbiamo un'altra message queue. Quindi due code sia per P' che per P''. Queste code le installano i processi stessi.

Quindi ognuno di questi processi, ognuno dei client, fa esattamente questa attività: Conosce qual è la message queue del server, quindi abbiamo un codice K che deve essere noto a P' per poter lavorare e inserire una richiesta, ma P' prima di fare questa operazione installa la sua message queue, e nel messaggio di richiesta M va anche ad indicare quale sia il K' della message queue PER POTER RICEVERE LA RISPOSTA.



Ovviamente il K' viene ad essere utilizzato dal server, perché quando il processo P (server) estrae dalla sua MESSAGE QUEUE il messaggio che rappresenta questa richiesta, in questo messaggio c'è proprio K', e quindi viene a conoscenza di qual è il CANALE message queue per comunicare le risposte verso il processo P'. Se arriva un altro client P'' crea un'altra message queue non con il codice K', ma con K''.

ESEMPIO IN CODICE

```

1  /***** client process *****/
2  /***** client process *****/
3  /***** client process *****/
4  /***** client process *****/
5
6  #include <sys/types.h>
7  #include <sys/ipc.h>
8  #include <sys/msg.h>
9  #include <sys/conf.h>
10 #include <stropts.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <unistd.h>
14 #include "header.h"
15
16 int my_id_coda,id_coda, ret, STATUS;
17 long key;
18
19 request_msg *request_message;
20 response_msg *response_message;
21
22
23
24 int main(int argc, char *argv[]) {

```

```

15  #include "header.h"
16
17
18  int id_coda, ret, STATUS;
19  long key = 58;
20  char *response = "done";
21
22  request_msg *request_message;
23  request_msg *new_message;
24  pthread_t tid;
25
26
27
28
29  void * do_work(void* request){
30
31  response_msg *response_message;
32  request_msg *request_message = (request_msg*)request;
33
34  printf("process %d - asked service of type %d - response channel is %d\n", getpid(), request_message->mtype, response_message->mtext);
35  sleep(5); //this simulates that the server is taking some time to do the job
36
37  response_message.mtype = 1;
38  memcp(response_message.mtext, response, strlen(response)+1);

```

```

25
26
27
28 key = getpid();
29
30 my_id_coda = msgget(key, IPC_CREAT|IPC_EXCL|0666);
31 if( my_id_coda == -1){
32 my_id_coda = msgget(key, IPC_CREAT|0666);
33 my_id_coda = msgget(key, IPC_CREAT|IPC_EXCL|0666);
34 if( my_id_coda == -1 ){
35 | printf("process %d - cannot install client queue, please check with the problem\n",getpid());
36 | exit(EXIT_FAILURE);
37 }
38
39
40 id_coda = msgget(50, IPC_CREAT|0666);
41 if( id_coda == -1){
42 | printf("process %d - cannot open server queue, please check with the problem\n",getpid());
43 | exit(EXIT_FAILURE);
44 }
45
46 request_message.mtype = 1;
47 request_message.req.service_code = 1;
48 request_message.req.response_channel = my_id_coda;
49 printf("process %d - response channel has id %d\n",getpid(),my_id_coda);
50
51 if ( msgsnd(id_coda, &request_message, sizeof(request), FLAG) == -1 ) {
52 | printf("process %d - cannot send request to the server\n",getpid());
53 | exit(EXIT_FAILURE);
54 }
55
56
57 if ( msgrcv(my_id_coda, &response_message, SIZE, 1, FLAG) == -1 ) {
58 | printf("error while receiving the server response, please check with the problem\n");
59 | exit(EXIT_FAILURE);
60 }
61 else {
62 | printf("process %d - server response: %s\n",getpid(),response_message.mtext);
63 | exit(0);
64 }
65
66 return 0;
67
68 /* end main*/

```

Client.c

```

39 if ( msgsnd(request_message->req.response_channel, &response_message, SIZE, FLAG) == -1 ) {
40 | printf("process %d - cannot return response to the client\n",getpid());
41 | exit(EXIT_FAILURE);
42 }
43
44 #ifdef MULTITHREAD
45 free(request_message);
46 pthread_exit((void *)NULL);
47#endif
48
49 /* end do_work */
50
51 int main(int argc, char *argv[]) {
52
53
54 id_coda = msgget(key, IPC_CREAT|IPC_EXCL|0666);
55 if( id_coda == -1){
56 id_coda = msgget(key, IPC_CREAT|0666);
57 ret = msgctl(id_coda,IPC_RMID,0);
58 id_coda = msgget(key, IPC_CREAT|IPC_EXCL|0666);
59 if( id_coda == -1){
60 | printf("process %d - cannot install server queue, please check with the problem\n",getpid());
61 | exit(EXIT_FAILURE);
62 }
63
64
65
66 while(1) {
67
68 if ( msgrcv(id_coda, &request_message, sizeof(request), 1, FLAG) == -1 ) {
69 | printf("process %d - message receive error, please check with the problem\n",getpid());
70 | fflush(stdout);
71 }
72 else {
73
74 #ifdef MULTITHREAD
75 new_message = malloc(sizeof(request_msg));
76 if(!new_message){
77 | printf("process %d - cannot allocate new request message buffer \n",getpid());
78 | fflush(stdout);
79 | exit(EXIT_FAILURE);
80 }
81 memcpy((char*)new_message,(char*)&request_message,sizeof(request_msg));
82
83 if( pthread_create(&tid, NULL, do_work, (void *)new_message) != 0){
84 | printf("process %d - cannot activate new thread\n",getpid());
85 | fflush(stdout);
86 | exit(EXIT_FAILURE);
87 }
88#endif
89
90 #ifndef MULTITHREAD
91 do_work(&request_message);
92#endif
93 }
94 }
95 /* end while */
96
97 return 0;
98 }
99 /* end main*/
100
101
102

```

Server.c

```

1 #ifndef __HEADER__
2 #define __HEADER__
3
4 #define ERRORE -1
5 #define FLAG 0
6 #define SIZE 128
7
8
9 typedef struct {
10     int response_channel;
11     int service_code;
12 } request;
13
14
15 typedef struct {
16     long mtype;
17     request req;
18 } request_msg;
19
20
21 typedef struct {
22     long mtype;
23     char mtext[SIZE];
24 } response_msg;
25
26 #endif
27
28

```

Header.h

La richiesta che farà il client sul server per un messaggio

La risposta che il Server invierà.

C'è scritto il tipo del messaggio e poi la richiesta, ma la richiesta è request. E request è una struttura che ha come primo campo l'ID numerico (il codice da utilizzare per SPEDIRMI LA RISPOSTA su un'altra message queue) e come secondo parametro il tipo di richiesta che ti faccio (di tipo 0, 1, 2 e così via..).

Nel server, abbiamo nel main, sicuramente la creazione della coda di messaggio con una certa chiave, e questa chiave deve essere nota anche al software del client, perché altrimenti il client non riesce a poter aprire quella message queue, farsi ritornare l'id_coda per poter andare a scrivere la richiesta (questo lo farà il client).

Il server crea questa message queue esclusivamente: se c'è un errore però ci riprova. Non demorde.

Cerca di ricrearla, e quindi eventualmente solo di aprirla, in maniera NON ESCLUSIVA. Quindi la eliminiamo per poi e poi ripartiamo per reinstallarla esclusivamente. Se otteniamo un errore a quel punto un'altra volta, abbiamo un errore dal punto di vista del software.

Installazione esclusiva, se non ci riesco apro l'oggetto esistente - lo rimuovo - provo a reinstallare esclusivamente.

In un while(1) il server riceve richieste con msgrcv e ogni volta che riceve il messaggio, e lo scarica all'interno di una specifica area denominata request_message, va a verificare che la ricezione non abbia ritornato -1, quindi nel caso ci fosse stato un errore, e se questo non è vero andiamo sotto, e se siamo MULTITHREAD abbiamo l'allocazione di un'area dove noi vogliamo ospitare il messaggio ricevuto, grande quindi la taglia del messaggio di richiesta, e il messaggio di richiesta è definito all'interno dell'header.h, ossia la variabile request_msg, quindi noi allochiamo un'area che ci permette di ospitare un messaggio di richiesta, verifichiamo che sia tutto apposto, e poi ovviamente effettuiamo una copia di memoria dell'area di memoria in cui avevamo ricevuto il messaggio (request_message) nell'area di memoria in cui il thread che parte per processare questa richiesta potrà lavorare (new_message). Così noi nel main thread potremmo riutilizzare l'area di memoria di request_message, senza andare a danneggiare il messaggio che chiaramente il thread che sta processando questa richiesta dovrà utilizzare.

Ora PARTE il thread e gli passiamo new_message, ossia l'area in cui abbiamo copiato questo messaggio che ci era arrivato tramite la coda di messaggio, e poi riparte il while(1) e torniamo ad aspettare un altro messaggio.

Andiamo a vedere la procedura do_work().

Andiamo innanzitutto a stampare un messaggio che dice che è arrivata una richiesta, in particolare è stato chiesto un servizio con un certo codice numerico, e con getpid() facciamo vedere chi è che emette questo messaggio, e per il codice numerico del servizio abbiamo il pointer all'area di memoria dove c'è la richiesta-> nella zona request->e prelevare il service code. E poi ovviamente indichiamo anche qual è il canale di risposta.

Poi dormiamo 5 secondi.

Poi impacchettiamo una risposta. Assegniamo 1 al message type, e andiamo a copiare una certa stringa response all'interno del campo mtext di quest'area di memoria. E poi con una message send la spediamo. E la spediamo sulla CODA di cui abbiamo ottenuto il codice operativo direttamente tramite la richiesta del nostro client. (request_message->req.response_channel).

Il client con una getpid() va a chiedere qual è il suo codice numerico, e poi cerchiamo di creare una coda di messaggio con il codice numerico che ci siamo fatti dare (key). Questo garantisce l'univocità.

Dopo faccio un ulteriore message get IPC_CREAT senza esclusive, eventualmente per aprire la coda del server che aveva chiave pari a 50, e dopo vado a spedire questa richiesta.

La richiesta ha codice numerico 1, e per avere la risposta dal server, quest'ultimo deve scriverla utilizzando my_id_coda come descrittore di coda.

Il tipo è pari a 1.

E ora spedisco.

Faccio la send sulla coda del server id_coda e ricevo sulla mia coda my_id_coda il messaggio di risposta.

```
client client.c header.h Makefile server server.c
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> ./client &
[1] 12411
process 12411 - asked service of type 1 - response channel is 11114128
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> ./server
[2] 12388
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> ./client &
[3] 12412
process 12412 - asked service of type 1 - response channel is 1146897
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> ./server
[1] 12388
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> ./client &
```

Ho lanciato 3 richieste client. E nemmeno siamo multithread, quindi processiamo una richiesta e dormiamo. 1 dopo l'altra.

```
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> ./server
process 12410 - asked service of type 1 - response channel is 11114128
process 12410 - asked service of type 1 - response channel is 1146897
process 12410 - asked service of type 1 - response channel is 1179666
```

```
es> process 12415 - response channel has id 1179666
process 12411 - server response: done
process 12413 - server response: done
process 12415 - server response: done
```

MULTITHREADING PER LA CONCORRENZA

```
PES-MESSAGES/UNIX/client-server-via-messages> make multithread
gcc client.c -o client
gcc server.c -o server -DMULTITHREAD -lpthread
```

```
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> ./client &
[1] Done ./client
[2] Done ./client
[3] Done ./client
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> ./client &
[1] 12483
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> ./client &
[2] 12486
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> ./client &
[3] 12489
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> ./client &
```

```
process 12489 - response channel has id 1310741
francesco@linux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI/CURRENT/SOFTWARE-EXAMPLES/PI/PES-MESSAGES/UNIX/client-server-via-messages> process 12483 - server response: done
process 12486 - server response: done
process 12489 - server response: done
```

MAKEFILE

```
Makefile
all:
    gcc client.c -o client
    gcc server.c -o server

multithread:
    gcc client.c -o client
    gcc server.c -o server -DMULTITHREAD -lpthread
```

```
PES-MESSAGES/UNIX/baseline-msg-queue-example> gcc example.c -DSYNCHRONIZED
francescodlinux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI
PES-MESSAGES/UNIX/baseline-msg-queue-example> ./a.out
process 12555 - digitare le stringhe da trasferire (quit per terminare):
ciao
a tutti
quit
process 12554 - ciao
process 12554 - a
process 12554 - tutti
process 12554 - quit
francescodlinux-mxb5:~/git-web-site/FrancescoQuaglia.github.io/TEACHING/SISTEMI-OPERATIVI
PES-MESSAGES/UNIX/baseline-msg-queue-example> █
```