Remote Procedure Call

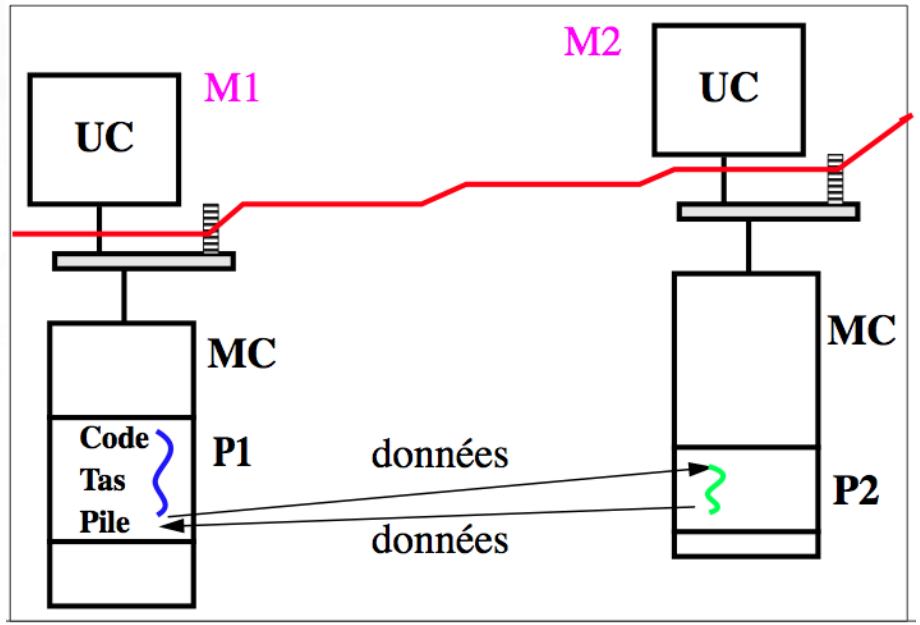
Université de la Réunion

Problème général

- Propriétés souhaitables des langages/modèles
  - Une bonne abstraction des primitives de base des architectures
    - trop concret, bas niveau : obsolète rapidement
    - trop haut niveau : performances difficiles à obtenir
  - Nécessité de compromis, en particulier par rapport à un certain nombre de propriétés
    - Indépendant de l'architecture : exécuter le programme sur des architectures diffé- rentes, sans modifier le source Le modèle, théoriquement, doit s'abstraire de toutes les caractéristiques spécifiques à certaines architec- tures.
    - Abstrait : Cacher certains aspects du parallélisme (Ex : synchronisations)
- Une des abstractions : Remote Procedure Call

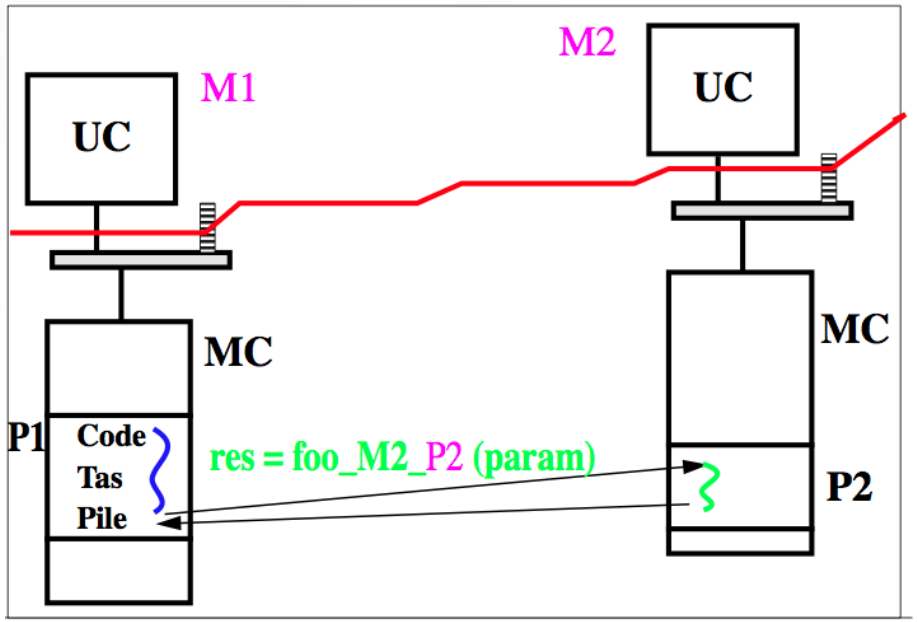
Problème général

- Schéma de principe
  - L'UC de la machine M1 ne peut pas exécuter des opérations sur la MC (mémoire centrale) de M2.
  - Depuis le processus P1 de la machine M1, on a besoin d'exécuter un morceau de code sur la machine M2, evt. dans le cadre du processus P2. On a également besoin de transmettre des informa- tions (données) de P1 vers P2. Eventuellement, P1 souhaite récupérer des données ou un acquittement à la fin.
  - Avant, ou programmation bas niveau : sockets



Problème général

- Communications + Exécution : RPC
  - Depuis un processus (P1), sur une machine (M1),
  - Depuis le processus P1 de la machine M1, on a besoin d'exécuter un morceau de code sur la machine M2, evt. dans le cadre du processus P2. On a également besoin de transmettre des informa- tions (données) de P1 vers P2.
  - On va exécuter la procédure foo, avec des paramètres param, sur une autre machine (M2) (evt. dans un processus (P2))
  - On récupère éventuellement un résultat res



Problems with sockets

- Sockets interface is straightforward

connect

  - read/write

disconnect

- BUT ... it forces read/write mechanism
  - We usually use a procedure call
- To make distributed computing look more like centralized :
  - I/O is not the way to go

Remote Procedure Call

- 1984 : Birrell & Nelson
  - Mechanism to call procedures on other machines
- Builds on message passing.
- Main idea : extend traditional (local) procedure call to perform transfer of control and data across network.
  - It should appear to the programmer that a normal call is taking place
- Easy to use : analogous to local calls.
- But, procedure is executed by a different process, probably on a different machine.
- Fits very well with client-server model.

Regular procedure calls

- Machine instructions for call & return but the compiler really makes the procedure call abstraction work :
  - Parameter passing
  - Local variables
  - Return data

Regular procedure calls

- You write :
  - x = f(a, "test", 5) ;
- The compiler parses this and generates code to :
  1. Push the value 5 on the stack
  2. Push the address of the string "test" on the stack
  3. Push the current value of a on the stack
  4. Generate a call to the function f
- In compiling f, the compiler generates code to :
  1. Push registers that will be clobbered on the stack to save the values b. Adjust the stack to make room for local and temporary variables
  2. Before a return, unadjust the stack, put the return data in a register, and issue a return instruction

Implementing RPC

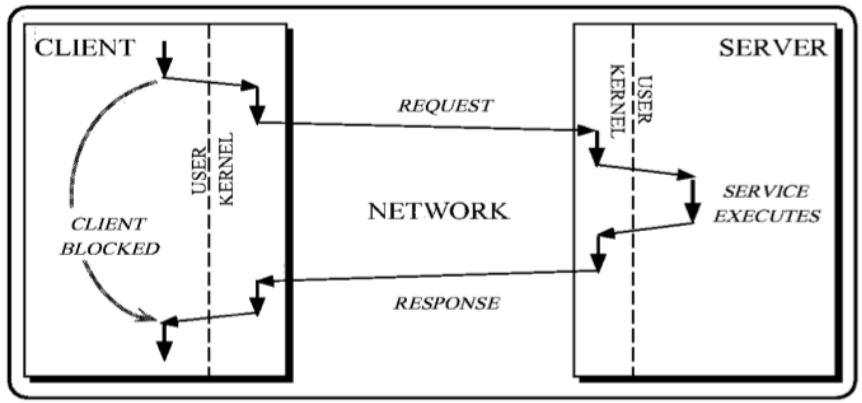
- No architectural support for remote procedure calls
- Simulate it with tools we have (local procedure calls)
  - Simulation makes RPC a language-level construct instead of an operating system construct
- The trick :
  1. Create stub functions to make it appear to the user that the call is local
  2. Stub function contains the function's interface

RPC

- Modèle Client-Serveur
  - Plus structuré que simplement envoyer un message :
    - On communique : paramètres
    - On spécifie le code à exécuter : procédure
    - On communique : résultats
  - Notes :
    - NFS utilise pour son implémentation le RPC
    - On utilise quelques fois le RPC même si il y a une mémoire unique : communications sans partage afin de proté- ger les espaces d'adressage (OS)

RPC

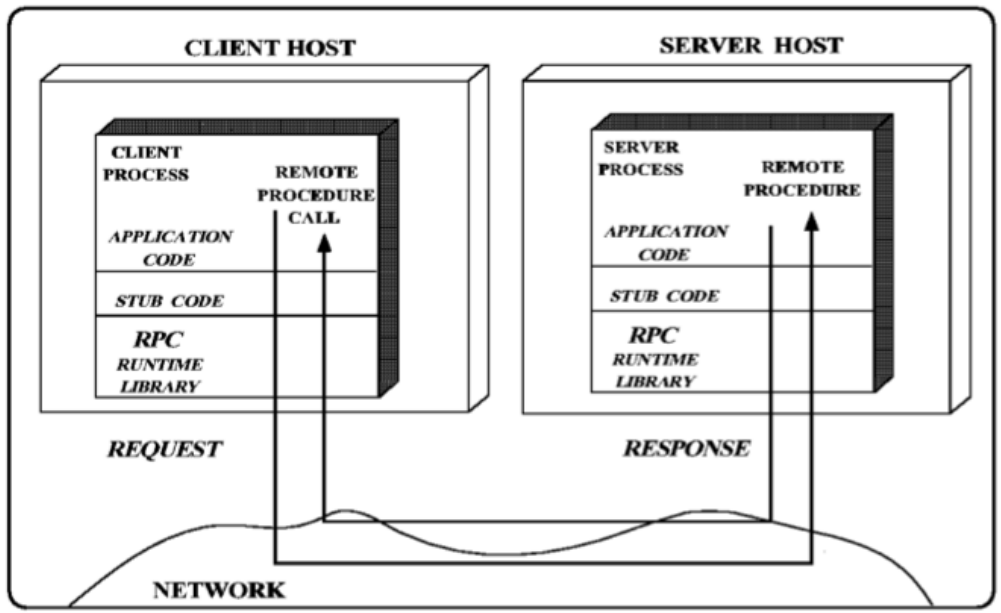
- Vue temporelle d'un RPC



- An RPC protocol contains two sides, the *sender* and the *receiver* (*i.e.*, *client* and *server*)
  - However, a server might also be a client of another server and so on...

RPC

- RPC runtime mechanism responsible for retransmissions, acknowledgments.
- Stubs responsible for data packaging and un-packaging ;
  - AKA marshalling and un-marshalling : putting data in form suitable for transmission. Example : Sun's XDR.
- Vue en couches d'un RPC



  - Stub client = Talon client ou proxy
  - Stub server = Talon serveur ou Skeleton

RPC Mechanism

1. Invoke RPC.
2. Calling process suspends.
3. Parameters passed across network to target machine.
4. Procedure executed remotely.
5. When done, results passed back to caller.
6. Caller resumes execution.

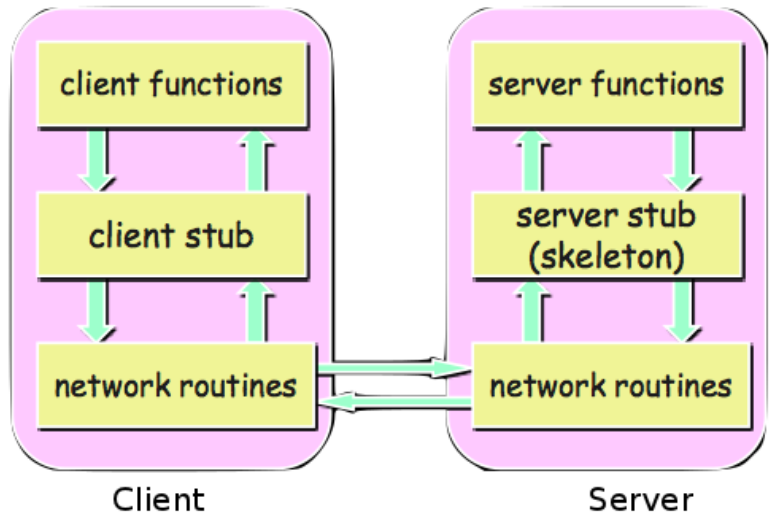
Is this synchronous or asynchronous?

RPC Advantages

- Procedure call interface
- Easy to use
- Writing applications is simplified
  - RPC hides all network code into stub functions
  - Application programmers don't have to worry about details
    - Sockets, port numbers, byte ordering
- Well-known mechanism
- Abstract data type
  - Client-server model
  - Server as collection of exported procedures on some shared resource.
  - Example : file server.
- RPC : presentation layer in OSI model
- Reliable

RPC operations

1. Client calls stub (params on stack)
2. Stub marshals params to net message
3. Network message sent to server
4. Receive message : send to stub
5. Unmarshal parameters, call server func
6. Return from server function
7. Marshal return value and send message
8. Transfer message over network
9. Receive message : direct to stub
10. Unmarshal return, return to client code



Parameter passing

- Pass by value
  - Easy : just copy data to network message
- Pass by reference
  - Makes no sense without shared memory
- Passing by reference with RPC
  - Operations
    1. Copy items referenced to message buffer
    2. Shipthemover
    3. Unmarshal data at server
    4. Pass local pointer to server stub function
    5. Send new values back
  - To support complex structures
    - Copy structure into pointerless representation
    - Transmit
    - Reconstruct structure with local pointers on server

Representing data

- No such thing as incompatibility problems on local system
- Remote machine may have :
  - Different byte ordering
  - Different sizes of integers and other types
  - Different floating point representations
  - Different character sets
  - Alignment requirements

Representing data

- IP (headers) forced all to use big endian byte ordering for 16 and 32 bit values
  - Most significant byte in low memory
    - Sparc, 680x0, MIPS, PowerPC G5
    - Intel I-32 (x86/Pentium) use little endian

```
main() {
unsigned int n;
    char *a = (char *)&n;
    n = 0x11223344;
    printf("%02x, %02x, %02x, %02x\n",
     a[0], a[1], a[2], a[3]);
}
```

  - Output on a Pentium : 44, 33, 22, 11
  - Output on a PowerPC : 11, 22, 33, 44

Representing data

- Need standard encoding to enable communication between heterogeneous systems
  - e.g. Sun's RPC uses XDR (eXternal Data Representation)
  - ASN.1 (ISO Abstract Syntax Notation)
- Implicit typing
  - only values are transmitted, not data types or parameter info
  - e.g., Sun XDR
- Explicit typing
  - Type is transmitted with each value - e.g., ISO's ASN.1, XML

Where to bind ?

- Need to locate host and correct server process
- Solution 1
  - Maintain centralized DB that can locate a host that provides a particular service (Birrell & Nelson's 1984 proposal)
- Solution 2
  - A server on each host maintains a DB of locally provided services
  - Solution 1 is problematic for Sun NFS - identical file servers serve different file systems

The portmapper

- portmapper provides a central registry for RPC services.
- Servers, at startup, tell the portmapper what services they offer and on what port
  - Port is a standard UDP/TCP port
  - Services are identified by well-known numbers
- Clients ask a remote portmapper for the port number corresponding to a particular Program ID.
- The portmapper is itself an RPC server !
- The portmapper is available on a well known port (111).
- How do we know which ports are being used by services ?
  - rpcinfo -p

Transport protocol

- Which one ?
  - Some implementations may offer only one (e.g. TCP)
  - Most support several
    - Allow programmer (or end user) to choose

When things go wrong

- Local procedure calls do not fail
  - If they core dump, entire process dies
- More opportunities for error with RPC :
- Transparency breaks here
  - Applications should be prepared to deal with RPC failure
- A local procedure call is exactly called once
- A remote procedure call may be called :
  - 0 times : server crashed or server process died before executing server code
  - 1 time : everything worked well
  - 1 or more : excess latency or lost reply from server and client retransmission

RPC Semantics

- Most RPC systems will offer either :
  - at least once semantics
  - or at most once semantics
- Understand application :
  - idempotent functions : may be run any number of times without harm
  - non-idempotent functions : side-effects

RPC Semantics

- "Maybe call" :
  - Clients cannot tell for sure whether remote procedure was executed or not due to message loss, server crash, etc.
  - Usually not acceptable.
- "At-least-once" call :
  - Remote procedure executed at least once, but maybe more than once.
  - Retransmissions but no duplicate filtering.
  - Idempotent operations OK ; e.g., reading data that is read-only.
- "At-most-once" call
  - Most appropriate for non-idempotent operations.
  - Remote procedure executed 0 or 1 time, ie, exactly once or not at all.
  - Use of retransmissions and duplicate filtering.
  - Example : Birrel et al. implementation.
    - Use of probes to check if server crashed.

More issues

- Performance
  - RPC is slower ... a lot slower
- Security
  - messages visible over network - Authenticate client
  - Authenticate server

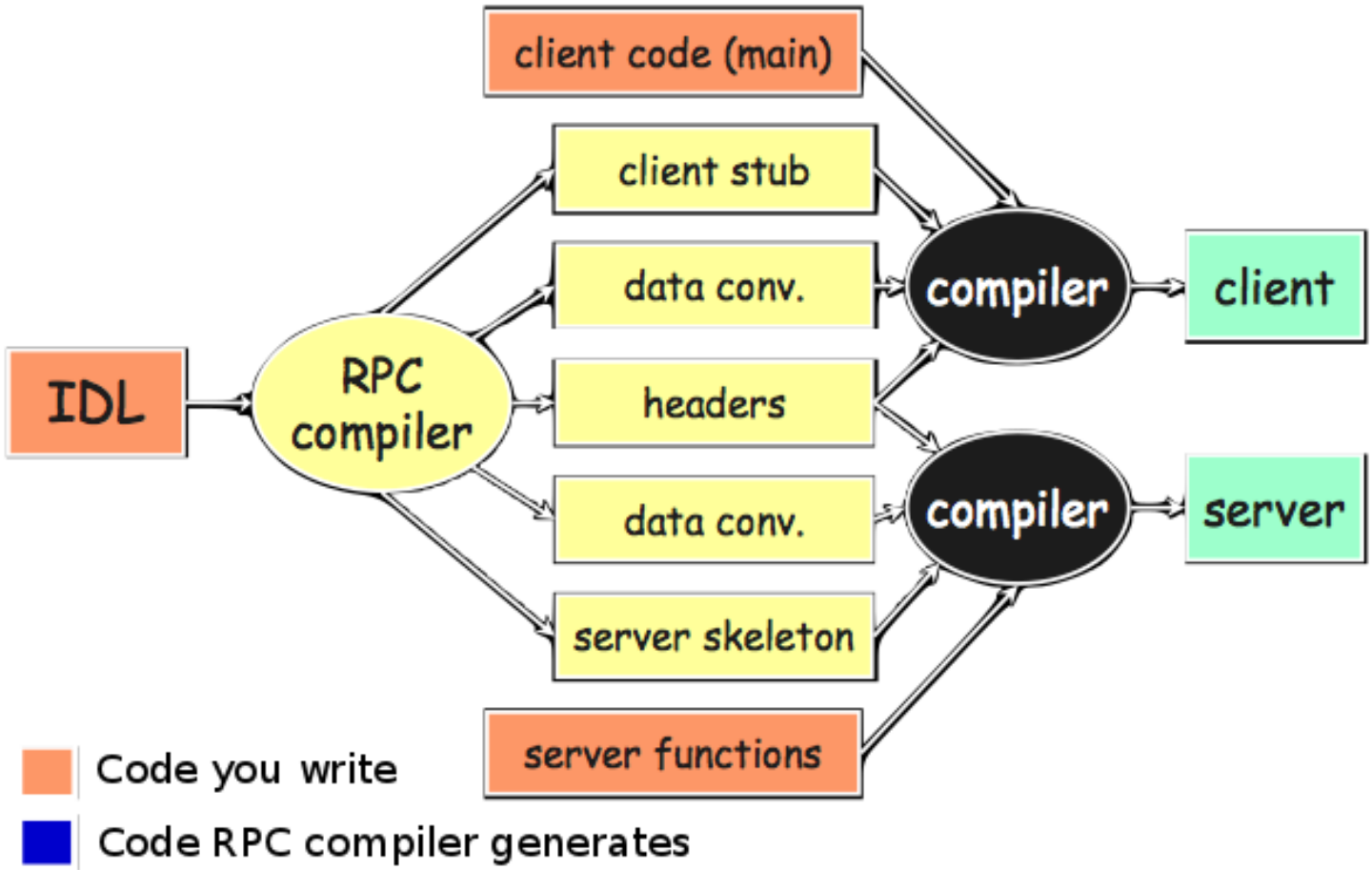
Programming with RPC

- Language support
  - Most programming languages (C, C++, Java, ...) have no concept of remote procedure calls
  - Language compilers will not generate client and server stubs
- Common solution :
  - Use a separate compiler to generate stubs (pre-compiler)

Interface Definition Language

- Allow programmer to specify remote procedure interfaces (names, parameters, return values)
- Pre-compiler can use this to generate client and server stubs :
  - Marshaling code
  - Unmarshaling code
  - Network transport routines
  - Conform to defined interface
- Similar to function prototypes

RPC compiler



Writing the program

- Client code has to be modified
  - Initialize RPC-related options
    - Transport type
  - Locate server/service
  - Handle failure of remote procedure call
- Server functions
  - Generally need little or no modification

RPC API

- What kind of services does an RPC system need ?
  - Name service operations
    - Export/lookup binding information (ports, machines)
    - Support dynamic ports
  - Binding operations
    - Establish client/server communications using appropriate protocol (establish endpoints)
  - Endpoint operations
    - Listen for requests, export endpoint to name server
  - Security operations
    - Authenticate client/server
  - Internationalization operations
  - Marshaling/data conversion operations
  - Stub memory management
    - Dealing with "reference" data, temporary buffers
  - Program ID operations
    - Allow applications to access IDs of RPC interfaces

Exemple

- Créer les fichiers suivants
  - msg_proc.c : implémentation de l'appel distant
  - rprintmsg.c : programme client
  - msg.x : RPC Language (IDL)
- Générer les fichiers avec rpcgen et compiler les programmes client et serveur

```
rpcgen msg.x
gcc -o msg_server msg_svc.c msg_proc.c
gcc -o rprintmsg msg_clnt.c rprintmsg.c
```

- Tester l'application en exécutant les programmes serveurs et client dans l'ordre

```
sudo ./msg_server
./rprintmsg "127.0.0.1" hello
```

msg.x

```
/* * msg.x: Remote message printing protocol */
program MESSAGEPROG {
version MESSAGEVERS {
int PRINTMESSAGE(string) = 1;
} = 1;
} = 99;
```

msg_proc.c

```
/* * msg_proc.c: implementation of the remote procedure "printmessage" */
#include <stdio.h>
#include <rpc/rpc.h>/* always needed */
#include "msg.h" /* need this too: msg.h will be generated by rpcgen */

/* * Remote verson of "printmessage" */
int * printmessage_1_svc(char **msg, struct svc_req *req) {
static int result; /* must be static! */
FILE *f;
f = fopen("/dev/console", "w");
if (f == NULL) {
result = 0; return (&result);
}
fprintf(f, "Hello %s\n", *msg);
fclose(f);
result = 1;
return (&result);
}
```

rprintmsg.c

```
/* rprintmsg.c: remote version of "printmsg.c" */
#include <stdio.h>
#include <rpc/rpc.h>/* always needed */
#include "msg.h" /* need this too: msg.h will be generated by rpcgen */

int main(int argc, char *argv[]) {
CLIENT *cl;
int *result;
char *server;
char *message;

if (argc < 3) {
fprintf(stderr, "usage: %s host message\n", argv[0]);
exit(1);
}

/*
* Save values of command line arguments
*/
server = argv[1];
message = argv[2];

/*
* Create client "handle" used for calling MESSAGEPROG on the
* server designated on the command line. We tell the RPC package
* to use the "tcp" protocol when contacting the server.
*/
cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");

if (cl == NULL) {
/*
```

rprintmsg.c

```
* Couldn't establish connection with server.
* Print error message and die.
*/
clnt_pcreateerror(server);
exit(1);
}
/* * Call the remote procedure "printmessage" on the server */
result = printmessage_1(&message, cl);
if (result == NULL) {
/*
* An error occurred while calling the server.
* Print error message and die.
*/
clnt_perror(cl, server);
exit(1);
}

/*
* Okay, we successfully called the remote procedure.
*/
if (*result == 0) {
/*
* Server was unable to print our message.
* Print error message and die.
*/
fprintf(stderr, "%s: %s couldn't print your message\n", argv[0], server);
exit(1);
}
/* * The message got printed on the server's console */
printf("Message delivered to %s!\n", server);
}
```

Mise en pratique

- Développer une application calculatrice distribuée permettant de réaliser les 4 opérations arithmétiques

References

- Cours de S. Krakowiak
- "Distributed Systems : Principles and Paradigms", A. S. Tanenbaum
- "Distributed Systems", P. Krzyzanowski