- Now that you know how to make computers communicate, you can write any distributed application
  1. Decide which machine does what
  2. Identify when computers need to communicate and what they should tell each other
  3. Program it!

THE END

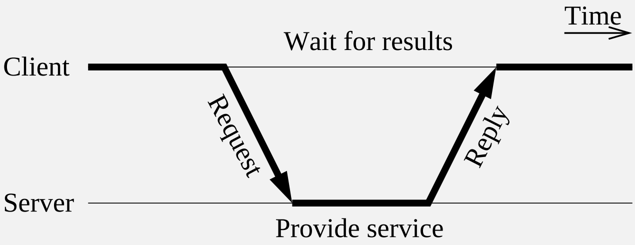
(well, almost. . .)

Except that...

- You will have to deal with *many* issues:
  - How to structure your program
  - Define an application protocol
    - That is powerful enough for all your needs
    - Yet efficiently implemented
  - Deal with machines of different architectures
    - They may represent data differently
  - Locate machines
    - "Which machine implements this task?"
  - etc...
- We are in need of a middleware
- A **middleware** is a piece of software in charge of these issues
  - You program the *application code*
  - The middleware takes care of distribution issues
    - At least some of them...

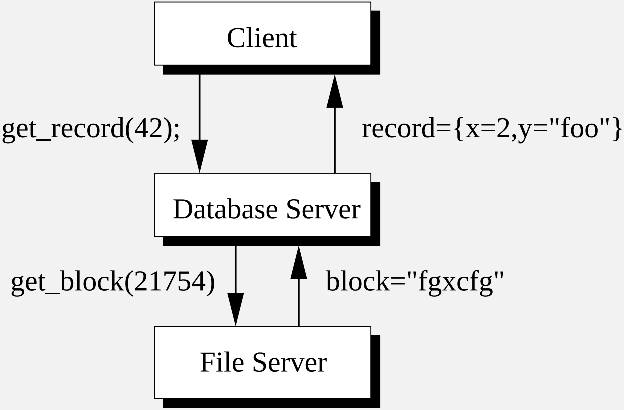
The Client-Server Model

- This is the most used model for organizing distributed applications
- Servers implement specific services
  - Example: a file system, a database service
- Clients request services from the servers, and wait for the response before continuing



Chained Client-Server Interactions

- A server can itself be a client to another server:
  - You just have to be careful about loops (A→B→C→A)

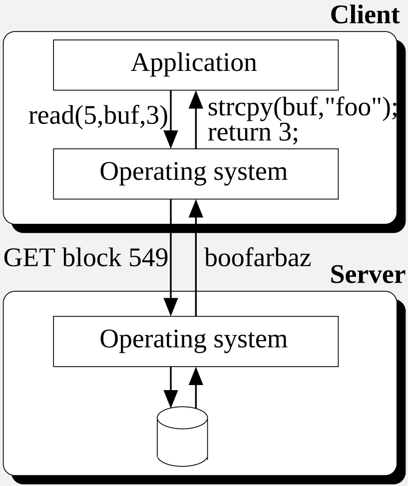


Why Use the Client-Server Model?

- A client-server application (usually) does not run faster than a centralized application
  - The client is waiting while the server works
  - Additional delay due to communication
- But it has several advantages. . .
  - Splitting an application
    - If it is too big to fit in one computer (memory space, disk, etc.)
  - Benefitting from specialized resources
    - One computer has a special device that is accessible from other machines (e.g., a file server, a print server, etc.)
    - Use cheap clients, and run CPU-intensive parts on a fast computer
  - Sharing information between multiple clients
    - A file server allows file sharing between multiple clients
    - Same for a database server (data sharing), etc.

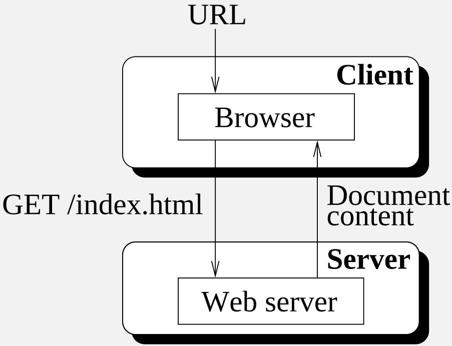
- A whole system is built specifically for one given application
- Example: distributed file systems
  - The operating system contains specialized code

1. Convert file-related system calls to requests to the file server
2. Convert server replies into system call return values



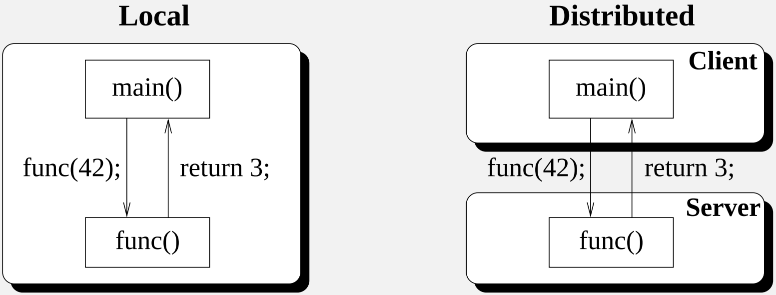
Ad-hoc Client-Server Implementations [2/2]

- Another example: the World-Wide Web
  - A specialized client-server protocol has been defined: HTTP
  - (More details about this in chapter 4. . .)



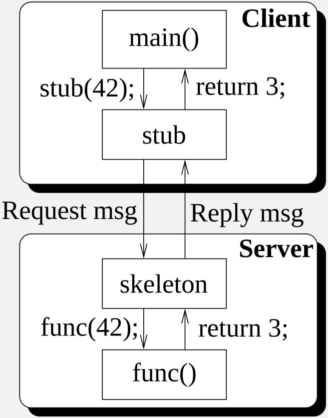
Remote Procedure Call [1/2]

- There already exists a way to represent a task in a local application: procedures (or functions)
- Let's extend the model to remote procedures



Remote Procedure Call [2/2]

- Of course, you need to convert invocations into network messages and vice-versa
  - A **stub** is a function with the same interface as `func()`: it converts function calls into network requests, and network responses into function returns
  - A **skeleton** converts network requests into function calls and function responses into network replies



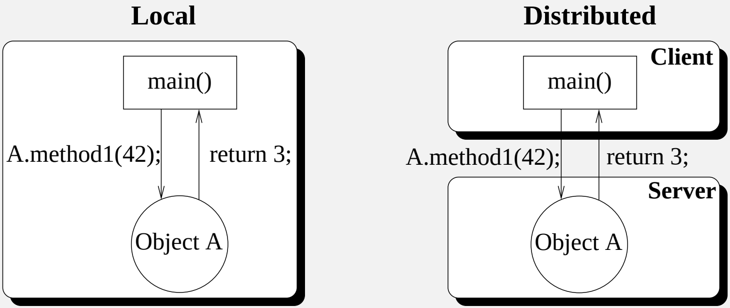
- An RPC system is used to generate the stub and the skeleton (more or less) automatically
  - Based on a description of the interface of `func()`

Limitations of the Remote Procedure Call Model

- Clients and servers do not share the same address space
- Contrary to non-distributed programs, clients and servers:
  - do not share global variables
  - do not share file descriptors
    - Therefore the server cannot directly access a file open by the client
  - cannot use pointers as function parameters
    - Because the server will not be able to follow such pointers
- This sets constraints on *which parts* of a program you can separate and run as a server
  - The server must have a clear interface (a set of function prototypes)
  - The server can have internal data, but the clients cannot access them directly
  - The client can have internal data, but the servers cannot access them directly
- These constraints look familiar, don't they?

Remote Method Invocation

- The equivalent to RPC in the object-oriented world is RMI (remote method invocation)



- Like for RPC, you must have stubs and skeletons...
- There are several Remote Method Invocation systems:
  - Sun RMI (entirely in Java), Corba (language-independent), etc.

Table of Contents

Presentation

- Sun RPC is a **Remote Procedure Call** system
  - Officially called **ONC-RPC**
  - "Open Network Computing Remote Procedure Call"
- It is very widely used
  - It was originally developed by Sun Microsystems, but it is now implemented in most (all?) Unix systems
  - Also implemented on Windows
  - The NFS distributed file system is based on Sun RPC
- It is platform-independent
  - A Linux computer can call a procedure on a Solaris box, etc.
- But not language independent
  - Designed to call remote C procedures
  - But you can also use other languages that have gateways to C

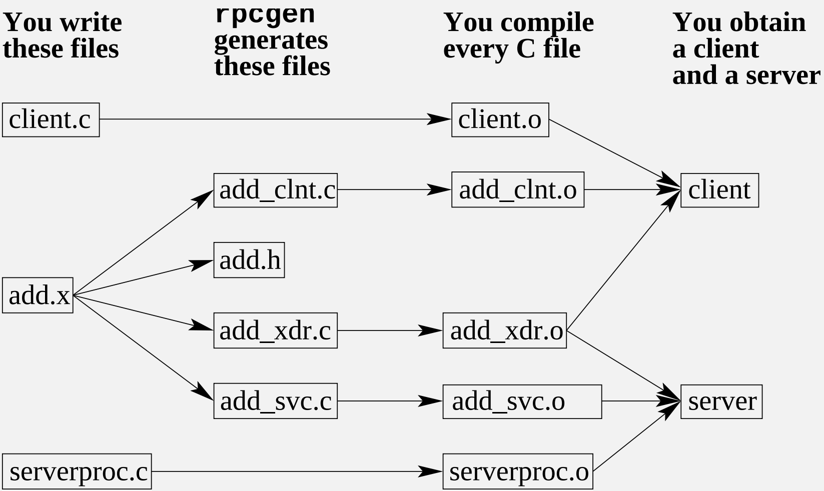
Writing an RPC Program [1/3]

- To write a minimalist RPC program, you must write:
  - A C procedure to be remotely called: `remoteproc.c`
  - A specification of the procedure: `remoteproc.x`
  - A client program to request the procedure: `client.c`

Writing an RPC Program [2/3]

- Based on `remoteproc.x`, the program `rpcgen` generates:
  - A header file that you will include in your programs: `remoteproc.h`
  - A server program (which will call your procedure when a request is received): `remoteproc_svc.c`
  - A client stub (that your client program can use to send an RPC): `remoteproc_clnt.c`
  - Internal functions to convert the procedure parameters into network messages and vice-versa: remoteproc_xdr.c
  - ☞ **Beware:** on Solaris, rpcgen generates K&R code by default. Use `rpcgen -C` to generate ANSI code.

Writing an RPC Program [3/3]



A Bit of Terminology

- One computer can be a server for multiple procedures
  1. A server may host several **programs** (identified by a **program number**)
  2. Each program may have several subsequent **versions** (identified by a **version number**)
  3. Each version of a program may contain one or more **procedures** (identified by a **procedure number**)
- **Program numbers** are 32-bit hexadecimal values (e.g. 0x20000001)
  - As a user, you can choose any program number between `0x20000000` and `0x3FFFFFFF`
  - But make sure program numbers are unique!
    - You cannot have several programs with the same number on the same machine
- **Version** and **procedure numbers** are integers (1, 2, . . .)

An RPC Example: add(x,y)=x+y;

1. Start by writing a specification file: add.x

```
struct add_in {    /* The arguments of the procedure */
    long arg1;
    long arg2;
};

typedef long add_out;  /* The return value of the procedure */

program ADD_PROG {
  version ADD_VERS {
    add_out ADD_PROC(add_in) = 1; /* Procedure number = 1 */
  } = 1;                          /* Version number = 1 */
} = 0x3543000;                    /* Program number = 0x3543000 */
```

An RPC Example (continued)

- This file contains specifications of:
  - A structure `add_in` containing the arguments
  - A typedef `add_out` containing the return values
  - A program named `ADD_PROG` whose number is `0x3543000`
  - The program contains one version with value `ADD_VERS = 1`
  - The version contains one procedure with value `ADD_PROC = 1`
    - This procedure takes an `add_in` as parameter, and returns an `add_out`
- Remark: your procedures can only take one input argument, and return one output return value
  - If you need more arguments or return values, group them into a structure (like `add_in`)

An RPC Example (continued)

2. Generate stubs: rpcgen add.x

- add.h contains various declarations:

```
#define ADD_PROG 0x3543000          /* Program nb */
#define ADD_VERS 1                  /* Version nb */
#define ADD_PROC 1                  /* Procedure nb */
add_out * add_proc_1(add_in *, CLIENT *);
add_out * add_proc_1_svc(add_in *, struct svc_req *);
```

  - add_proc_1 is the stub (i.e., the procedure that the client program will call)
  - add_proc_1_svc is the actual procedure that you will write and run at the server

- add_clnt.c contains the implementation of add_proc_1
- add_svc.c contains a program which calls your procedure add_proc_1_svc when it receives a request
- add_xdr.c: marshall/unmarshall routines

An RPC Example (continued)

3. Write your server procedure: serverproc.c

```
#include "add.h"
add_out *add_proc_1_svc(add_in *in, struct svc_req *rqstp) {
  static add_out out;
  out = in->arg1 + in->arg2;
  return(&out);
}
```

- rqstp contains some information about the requester
  - Its IP address, etc.

An RPC Example (continued)

4. Compile your server

- You need to compile together your procedure, the (generated) server program, the (generated) marshall/unmarshall procedures and the `nsl` library
  - The `nsl` library contains the RPC runtime

```
$ gcc -c serverproc.c
$ gcc -c add_svc.c
$ gcc -c add_xdr.c
$ gcc -o server serverproc.o add_svc.o add_xdr.o -lnsl
```

- You can start your server:

```
./server
```

An RPC Example (continued)

5. Write a client program: `client.c`

```
#include "add.h"
int main(int argc, char **argv) {
  CLIENT *cl;
  add_in in;
  add_out *out;

  if (argc!=4) { printf("Usage: client <machine> <int1> <int2>\n\n"; return 1; }

  cl = clnt_create(argv[1], ADD_PROG, ADD_VERS, "tcp");
  in.arg1 = atol(argv[2]);
  in.arg2 = atol(argv[3]);
  out = add_proc_1(&in, cl);
  if (out==NULL) { printf("Error: %s\n",clnt_sperror(cl,argv[1])); }
  else { printf("We received the result: %ld\n",*out); }
  clnt_destroy(cl);
  return 0;
}
```

An RPC Example (continued)

- You must first create a client structure thanks to `clnt_create`

```
#include <rpc/rpc.h>
CLIENT *clnt_create(char *host, u_long prog, u_long vers, char *proto);
```

  - `host`: the name of the server machine
  - `prog`, `vers`: the program and version numbers
  - `proto`: the transport protocol to use (`"tcp"` or `"udp"`)
- Then you can call the (generated) client procedure `add_proc_1` to send the RPC
- When you are finished, you destroy the client structure
  - ☞ A client structure can be used multiple times without being destroyed and re-created

An RPC Example (end)

6. Compile your client

```
$ gcc -c client.c
$ gcc -c add_clnt.c
$ gcc -c add_xdr.c
$ gcc -o client client.o add_clnt.o add_xdr.o -lnsl
```

7. Try it all

- Start your server

```
$ ./server
```

- Send a request:

```
$ ./client renard.cs.vu.nl 8 34
We received the result: 42
```