IMW: Middleware

Stéphane Genaud
ENSIIE - Strasbourg
genaud@unistra.fr

Middleware

Outline

1. Introduction
2. Sun RPC
3. Java RMI
4. Corba

Table of Contents

1 Introduction

2 Sun RPC

3 Java RMI

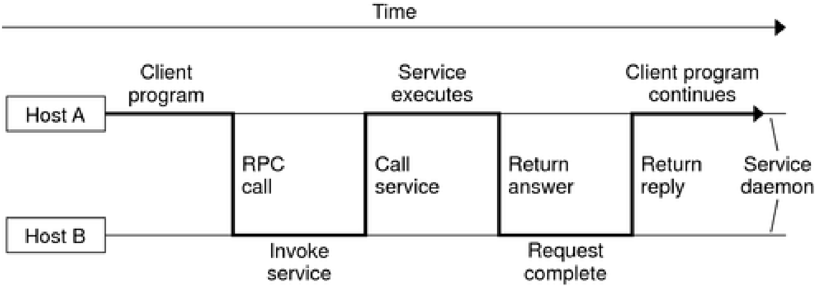
4 Corba

Technologies for Distributed Systems

- Extremely fast evolution since 1985: about a technology every 5 years.
- Implementations adapt to up-to-date technology
  e.g If networks go faster, it is possible to convey bigger messages.
  If the cost of some hardware becomes low, no need to spare it.

Technologies change but . . . concepts stay

- Client-server is the central concept:
  The **client** can make a request at any time,
  The **server** permanently waits for incoming requests



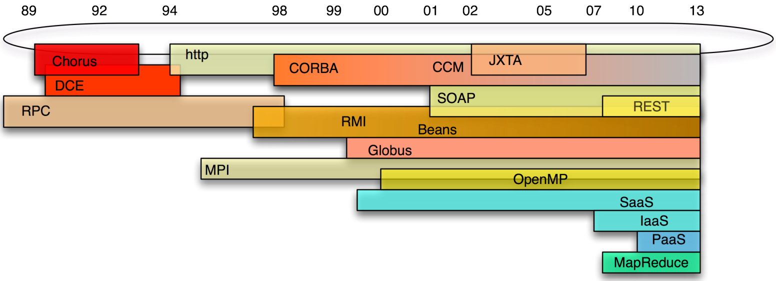
Middleware: definition

What is middleware?

- A sofware layer between the OS and the application allowing a set of distributed computers to communicate in a standardized way.

- Middleware provides inter-machines communication facilities, but may also include services, such as authentification services, resource directories, distributed file catalogs, ...

A Time-line of technologies



Principle Design Choices

- Abstraction vs. Performance
- Interoperability
- Versatility

Abstraction

- Abstraction of communication primitives
  - too low level: rapidly obsolete, lower programming productivity
  - too high level: difficult to optimize for performance
- Abstraction Trade-off
  - independent from the architecture: execute across different systems without **source code** modification
  - Hide details related to communication/synchronization management (e.g `Remote Procedure Calls` more abstract than `sockets`)

Interoperability

- Machine-independent
  e.g Sun RPC
- OS-independent
  e.g Java-RMI
- Language-independent
  e.g Corba, SOAP

Versatility

The more general, the more versatile

- Example 1: SOAP communicates through XML pieces of text
- ⇒ SOAP toolkits can be found for almost all languages.

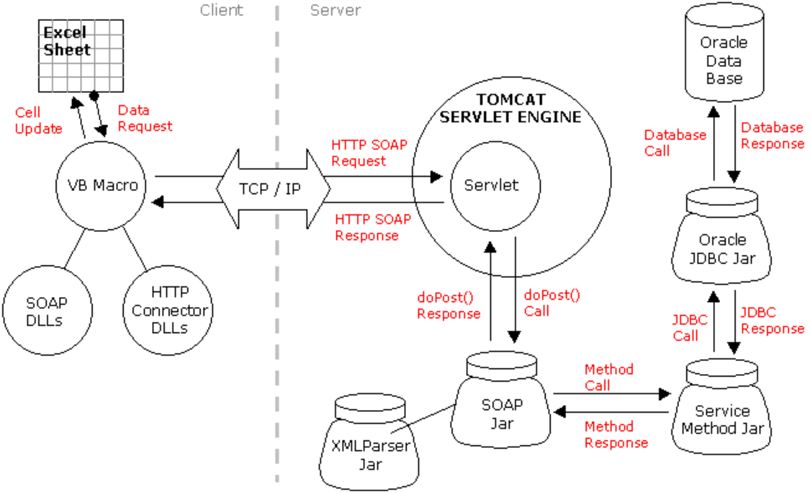


Table of Contents

1. Introduction
2. Sun RPC
3. Java RMI
4. Corba

Background

Sun RPC (aka RPC ONC (Open Network Computing))

- are the original RPC (RFC 1831), introduced by Sun in 1988
- motivation: provide a support to inter-machines services
- NFS as first target, NIS, ...
- is open source software (BSD license since 2009)

Interoperability

- ONC RPC allow programs on different OS and machines to communicate
- It may be in different languages but C in 99% cases.
- Relies on XDR (eXternal Data Representation)

RPC service identification

Services are identified by

1. the program name (`prog_name`)
2. the program version (`prog_ver`)
3. the function name

```
program MYPROG {
  version VERSION_ONE {
    void MYPROG_NULL(void) = 0;
    answer MYPROG_MYFUNC(data) = 1;
  } = 1;
} = 0x2000:0001;
```

Service Registration (portmap)

This service must be registered in a directory service generally called *portmapper*

- acts as a name server
- converts : <$\text{prog}_{\text{name}}$ + ver + protocol> to <portnumber>
- exact service name depending on sytem/distribution : `rpcbind` (or sometimes `portmap`, or `rpc.portmap`)
- attached to port 111

Standard RPC services

file /etc/rpc

```
portmapper 100000
rstatd 100001
rusersd 100002
nfs 100003
ypserv 100004
mountd 100005
ypbind 100007
walld 100008
```

Running Services

```
 % rpcinfo -p
   program vers proto port
   100000 2 tcp 111 portmapper
   100000 2 udp 111 portmapper
536870913 1 udp 58764
536870913 1 tcp 65106
```

Two last lines are one user program.

Programming with ONC RPC

Two layers:

- The **higher** layer: small set of functions to describe and call services in a simple way.
  - Essential primitives: `registerrpc()` and `callrpc()`
  - However, limitations: udp only, no auth, and encoding/decoding by hand.
- The **lower** layer: 20+ functions to fine tune the calls.
  - Much more complex, used for stressed services, for example to implement asynchronous RPC and authentification.

Server-side steps

The server must **register**: asks the local portmap to:

1. create a new entry so that clients can be routed
2. associate a service number and the address of the function that implements it, or the address of the *dispatcher*.

- The primtives are
  - `svc_register()` and `pmap_set()` (low level)
  - `rpcregister()` (high level)
  - on exit, `svc_unregister()`, `pmap_uset()`

Client-side steps

The client must initialize (1), lookup in remote portmap to find the service (2), then, several calls can be made afterwards (3):

1. `clnt_create()` / `clnttcp_create()` / `clntudp_create()`,
2. `pmap_getport()`
3. `clnt_call()`

The higher level `callrpc()` does steps 1, 2 and 3 in a row.

Example of high-layer usage (server side 1/2)

Define the service on the server:

```
#include <rpc/xdr.h>
#include <rpc/rpc.h>

int* my_function(int *n) {
  static int res;
  *n = *n + 1;
  res= *n;
  return (&res);
}
```

Example of high-layer usage (server side 2/2)

Register the service on the server:

```
#define PROGNUM 0x20000100
#define VERSNUM 1
#define PROCNUM 1

int main (void) {
  registerrpc( PROGNUM,
               VERSNUM,
               PROCNUM,
               my_function, /*ptr to function*/
               (xdrproc_t) xdr_int, /*encode input*/
               (xdrproc_t) xdr_int);/*decode output*/

   svc_run(); /* server starts listening ... */
}
```

Example of high-layer usage (client side 1/2)

Call the service from the client:

```
int main (int argc, char **argv) {
 int n=0x41424344;
 char *host = argv[1];
 int stat;
 stat = callrpc(host,
              PROGNUM,
              VERSNUM,
              PROCNUM,
              (xdrproc_t) xdr_int, //intput encoding
              (char *)&n, //input param
              (xdrproc_t)xdr_int, //output decoding
              (char *)&res); //return of func
}
```

Another way: `rpcgen`

- Taking care of conversion through XDR is difficult
- The `rpcgen` compiler automates the process of writing RPC applications
- `rpcgen` accepts interface descriptions in RPCL (RPC Language)
- and generates skeletons programs (C code)

Example with rpcgen

- Consider an *operation* `addition`, that adds up 2 `int` s
- Describe this service in a file `myservice.x`

```
struct data {
  int arg1; int arg2;
};
typedef struct data data;
struct response {
  int result; unsigned char error;
};
typedef struct response response;

program MYCOMPUTATION {
  version VERSION_ONE{
    void MYCOMPUTATION_NULL(void) = 0;
    response MYCOMPUTATION_ADDITION(data) = 1;
  } = 1;
} = 0x20000001;
```

Example with rpcgen (contd)

- Generate the skeletons

```
% rpcgen -a myservice.x
```

- The following files are generated

```
myservice.h /* parameter definitions */
myservice_xdr.c /* XDR conversion */
myservice_svc.c /* stubs server */
myservice_clnt.c /* stubs client */
myservice_server.c /* server code */
myservice_client.c /* client code */
```

RPCL in Brief (enumeration, constants & simple)

- Enumerations and Constants

```
enum colortype { RED = 0, GREEN = 1,BLUE = 2 };
const PI = 3.14;
```

- Simple Declarations

```
int length;
colortype c;
```

- Added types (bool and string)
  - `bool` : boolean, can take TRUE or FALSE values
  - `string`: translated to `char *` (See variable length array).

RPCL in Brief (arrays)

- Fixed-length arrays

```
int length[5];
color palette[8];
```

- Variable-length arrays
  - The maximum size is specified between angle brackets, or may be ommitted:

```
int notes_serie<20>; # at most 20
int heights<>; # unlimited
string message<256>;
```

each will translate to a C struct, e.g:

```
struct {
  u_int heights_len;/* # of items in array */
  int *heights_val; /* pointer to array */
} heights;
```

RPCL in brief (typedef)

- Type definitions

  Same syntax as C typedef

```
typedef string name_t<255>;
typedef string longstring<>;
```

will be translated into C code:

```
typedef char *name_t;
typedef char *longstring;
```

RPCL in Brief (pointers)

- Pointer declarations are as in C. Address pointers are not sent over the network. Instead, data pointed to are copied. This is useful for sending recursive data types such as lists and trees.

```
tree_t *t;
```

RPCL in Brief (struct)

- Translates as is in C, excepted that an extra typedef is generated.

```
struct coord { int x; int y; };
```

Translates to:

```
struct coord { int x; int y; };
typedef struct coord coord;
```

which allows to use `coord` instead of `struct coord`

Tips & Tricks

Linux

- Install: rpc lib provided by package `libtirpc-dev` (0.2.2-5 on ubuntu 12.04)
- Run: a portmapper is provided by package `rpcbind`
- Run: `svc_register()` might refuse to register ("credentials problem") ⇒ Start server as root or in sudo mode.
- Initialize array variables before calling remote functions ("Can't encode arguments" error).

MacOSX

- Install: the 'Command line tools' element from Xcode in the distrib or download it fom Apple .
- Use: `rpcgen -C` to force generation of ANSI-C code

Table of Contents

1. Introduction
2. Sun RPC
3. Java RMI
4. Corba

History

- Created by Sun in 1998
- Java only
- Available since JDK >= 1.1
- Since JDK 1.5, stubs are automatically generated (no `rmic`)

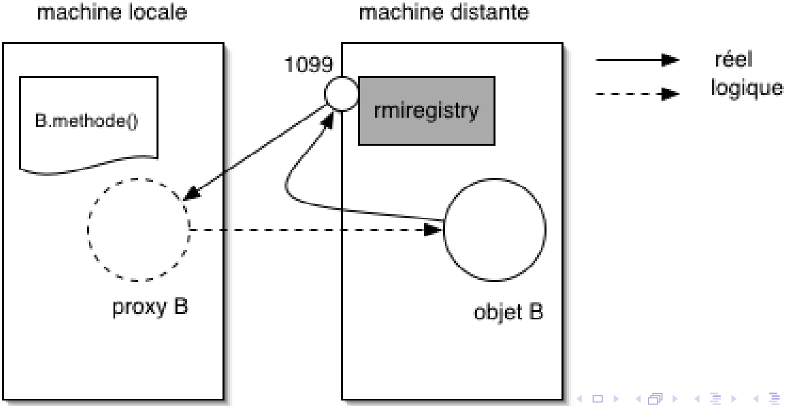
RPC in the world of RMI

- RMI provides access to **objects** and their **methods**
- In contrast to Sun RPC, not only data can be passed to remote computations, but also objects that can contain code and data.
- There are 2 ways to communicate in this object-oriented paradigm:
  1. through the `Remote` class
  2. through the `Serializable` class

The Remote class

definition: An object of the Remote class can be used remotely. It can be used:

- in the address space of the JVM that created it,
- in the address spaces of other JVMs through *handles* (aka *proxies*).



The Remote class and interface

- A Remote class must be defined in 2 parts
  - an interface
  - the class itself

Interface

```
public interface MyExample extends Remote {...}
```

Class

```
public class MyExampleImpl
  extends UnicastRemoteObject
  implements MyExample {
    ...
   }
```

The Serializable class

definition: an object of the class Serializable is an object that can be copied from one address space to another.

Registering the services

A process called **rmiregistry** is in charge of service registration (Equivalent of portmapper)

Characteristics of `rmiregistry`

- runs on the same host as the services
- default port is 1099
- can be started by program

Example 1: Remote object with primitive types

Example parameter passing using primitive types (e.g. int, float, ..) or arrays (e.g. String)

- In general, parameters just need to be **serializable** (java.io.Serializable).

The different pieces of code

- The service: description of the function prototype
- The service: the implementation of the service
- The server: a generic code which registers the service
- The client: the code that uses the service

Example 1: Service Description

A service is described by an **interface**.

- known by the client and the server.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Operation extends Remote {

   public int addition(int a, int b)
                  throws RemoteException ;
}
```

Example 1: Service Implementation

- Only the server **implements** the service.

```
import java.rmi.server.UnicastRemoteObject ;
import java.rmi.RemoteException ;
import java.net.InetAddress.* ;
import java.net.* ;

public class OperationImpl extends UnicastRemoteObject
  implements Operation {

    public OperationImpl () throws RemoteException {
        super();
    };

    public int addition(int a, int b)
                throws RemoteException {
      return( a + b) ;
  }
```

Example 1: Service Registration

- The first server task is to register the service in the rmiregistry under a name (here *Operation*)

```
import java.rmi.*;
import java.net.*;

public class Serveur {
  public static void main(String [] args) {
    try {
      OperationImpl une_op = new OperationImpl ();
      Naming.rebind("rmi://"+args[0]+"/Operation",une_op) ;
      System.out.println("Serveur␣pret");
    }
    catch (Exception e) {
        System.out.println(re) ;
    }
}
```

Example 1: Client code

- gets a reference to the the service in the registry (proxy)
- call the service using that reference

```
import java.rmi.* ;
import java.net.MalformedURLException ;
import java.io.*;

public class Client {
  public static void main(String [] args) {
    try {
        Operation o = (Operation)
            Naming.lookup("//"+args[0]+"/Operation");
        System.out.println("Client: 33+45= ?");
        int r = o.addition( 33, 45 );
        System.out.println("33+45="+ r );
     }
     catch (Exception e) { System.out.println(e) ; }
   }
```

Trouble shooting 1

Observation

The client experiences a `connection refused` error when contacting the server.

Why?

`$JAVA_HOME/lib/security/java.policy` is too restrictive wrt sockets

Solution

To override the standard, run

```
java -Djava.security.policy=more_perm Server
```

where `fichier` contains, for instance:

```
grant {
```

Trouble shooting 2

Observation

When calling the RPC (hence after the lookup), the client ends with: `java.rmi.ConnectException: Connection refused to host: 127.0.0.1`

Why?

In some linux distributions, the name resolution for hostname takes 127.0.0.1 from `/etc/hosts` instead of public IP.

Solution

run the server by overriding its IP

```
java -Djava.rmi.server.hostname=<my ip here> Server
```

Table of Contents

1 Introduction

2 Sun RPC

3 Java RMI

4 Corba

History

- Context
  - A specification defined by the *Object Management Group* (OMG), composed of about 1000 members
  - currently CORBA 3.0
  - Implementors then propose implementations
- Implemenations

  Commercial :
  - ORBIS, IONA, VisiBroker, ORBacus, ...

  Open source:
  - JDK, MICO, JacORB, TAO, . . .

Characteristics

CORBA = Common Object Request Broker Architecture

- A RPC framework
  - object oriented
  - multiple-OS, multiple languages can be involved
  - analogy of the "software bus"
- External Services
  - helper services, can connect to the bus
  - services: naming, transaction, persistence . . .

IDL

The Interface Definition Language: equivalent to the RPC Language.

- defines the **methods** a server proposes
- defines the **data** that can be accessed from the client (get/set)

From IDL, generation of concrete code to represent data and methods in the chosen language.

IDL structure

Three hierarchical elements:

1. `Module` : namespaces (correspond to Java packages)
2. `Interface` : logical groups of methods
3. *methods* : prototypes of the methods implemented by the server

Example:

```
module HelloApp
{
  interface Hello
  {
  string sayHello();
  oneway void shutdown();
  };
};
```

IDL types

Types and number of bytes between parenthesis:

- boolean ={TRUE,FALSE}
- octet (1)
- *signed* : short (2), long (4), long long (8)
- *unsigned* : unsigned short (2), unsigned long (4), unsigned long long (8)
- *floats* : float (4), double (8), long double (16)
- *characters*: char (1, iso-latin-1), string (var), string<n> (n), wchar (2, unicode), wstring (var of wchar)

IDL type mapping to Java

IDL	Java	IDL	Java
octet	byte	unsigned short	short
short	short	unsigned long	int
long	int	unsigned long long	long
long long	long	char	char
float	float	wchar	char
double	double	string	String
long double	N/A	wstring	String

IDL Methods

General Form

$<\text{return}_{\text{type}}>$ $\text{method}_{\text{name}}$([$<$mode$>$ $<$type$>$ $<\text{parameter}_{\text{id}}>$]*) [raises [exceptions]+];

- mode={in, out, inout} for input, output, and modified parameters resp.
- type: all primitive or constructed type with typedef (constructed before method call)

Method names must be unique (no overloading).

POA

OA and POA

- Object Adapter: mechanism that connects a request using an object reference with

the proper code to service that request.

- Portable Object Adapter: a particular type of object adapter that is defined by the CORBA specification.

- characteristics
  - Enables portability of object implementations between different ORB products
  - Provide support for objects with persistent identities
  - Provide support for transparent activation of objects
  - Allow a single servant to support multiple object identities simultaneously