Middleware:

Introducere

Paradigme:

- **RPC** (Remote Procedure Call); calitati / dezavantaje.
- **MOM** (Message Oriented Middleware); calitati / dezavantaje.
- **OOM** (Object Oriented Middleware); calitati / dezavantaje.

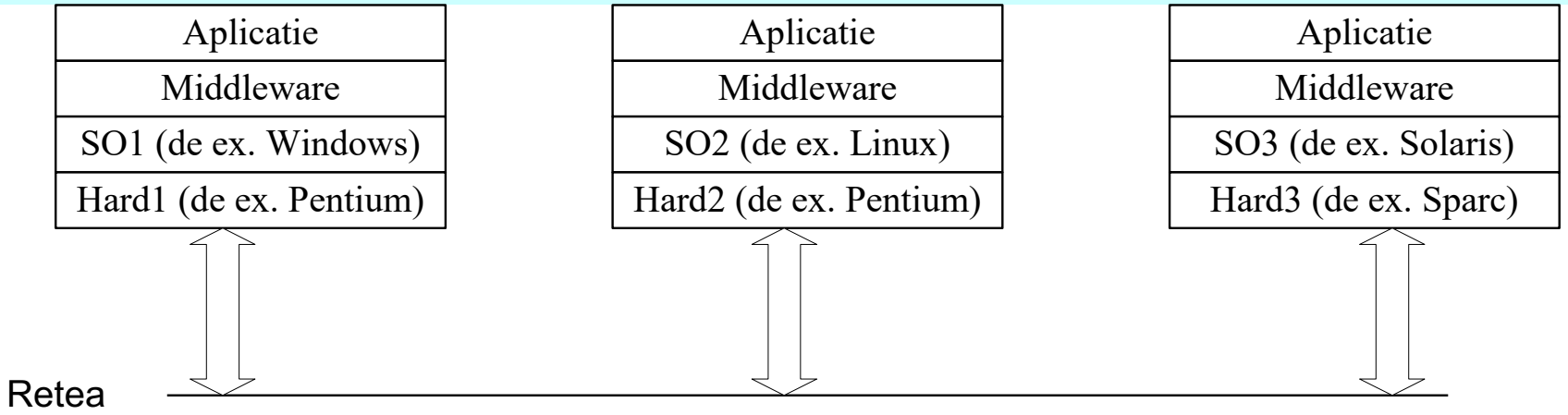
Implementari pe distributii simple ale exemplului suport

1. **RPC** apelul procedurilor la distanta in C (RPC).
2. **JMS** Serviciu de mesagerie Java, STOMP Python, NodeJs (MOM).
3. **Hessian** protocol binar pentru export de obiecte: Java, PHP, Python (OOM).
4. **Pyro** export de obiecte Python (OOM).
5. **RMI** export de obiecte Java (OOM).
6. **CORBA** export de obiecte: C++, Java etc. (OOM).

Legacy distributed computing

Proxies (Stub, Skeleton) (e.g. ONCRPC, RMI, CORBA)

Message Queue (e.g. JMS)



MOM – Message-Oriented Middleware. tranzacţiile să se poată desfăşura fără ca ambii parteneri să fie simultan operaţionali. Schimbul de informaţii între parteneri se realizează prin *mesaje.* Pentru realizarea schimbului de mesaje, trebuie să fie activ un *serviciu de mesagerie*.

RPC – Remote Procedure Call. Conceptual RPC extinde schemele clasice de apeluri de proceduri (subprograme, metode invocate etc.) din programarea clasică în sensul că programul apelator şi subprogramul se găsesc pe două maşini diferite, cu arhitecturi şi cu mecanisme de adresare diferite, cu SO-uri diferite etc.

OOM – *ObjectOrientedMiddleware* este in fapt o tehnologie RPC in care actiunile sunt incapsulate in obiecte de tipuri specifice.

Middleware ofera suport pentru:

- Servicii de nume; localizare; descoperirea de servicii; replicari
- Protocoale de manipulare; tratare esecuri de comunicare; QoS (Quality of Services)
- Memorare, tranzactionalitate, concurenta, cooperare, sincronizare
- Controlul accesului, autentificare.

Plaja de caracteristici (cu nivele de realizare relativa de la o distributie la alta)		
Request / reply	vs.	Mesagerie asincrona
Limbaj specific	vs.	Independenta de limbaj
Solutii proprietar	vs.	Solutii bazate pe standarde
Small - scale	vs.	Large - scale
Componente cuplate strans	vs.	Componente cuplate slab

Compatibilitati si interoperabilitati intre tehnologii middleware :

Server==> / Client V		RPC	MOM			Hessian		Pyro		RMI			CORBA
		Onc RPC	JMS	STOMP Python	STOMP NodeJs	Hess. Java	Hess. PHP	Pyro 3	Pyro4	JRMP	Lipe	IIOP	Java IDL
RPC	Onc-RPC	DA											
MOM	JMS		DA	DA	DA								
	STOMP Python		DA	DA	DA								
	STOMP NodeJs		DA	DA	DA								
Hessian	Hessian Java					DA	DA						
	Hessian PHP					DA	DA						
	Hessian Python					DA	DA						
	Hessian Android					DA	DA						
Pyro	Pyro3							DA					
	Pyro4								DA				
	Pyrolite C#								DA				
	Pyrolite Java								DA				
RMI	JRMP									DA			
	Lipe										DA		
	Lipe Android										DA		
	IIOP											DA	DA
CORBA	Java IDL											DA	DA

Extinde schemele clasice de apeluri de proceduri (subprograme, metode invocate etc.) din programarea clasică în sensul că programul apelator şi subprogramul se găsesc pe două maşini diferite, cu arhitecturi şi cu mecanisme de adresare diferite, cu SO-uri diferite etc.

Probleme:

1.Spaţii de adresare diferite: sensul *adresă de memorie?* identificarea procedurii? adresei de revenire la apelator?

2. "transmitere parametrii prin referinţă" sau "apel prin adresă"?

3.apelurile trebuie făcute numai prin **valoare** şi / sau prin **copie-rezultat**

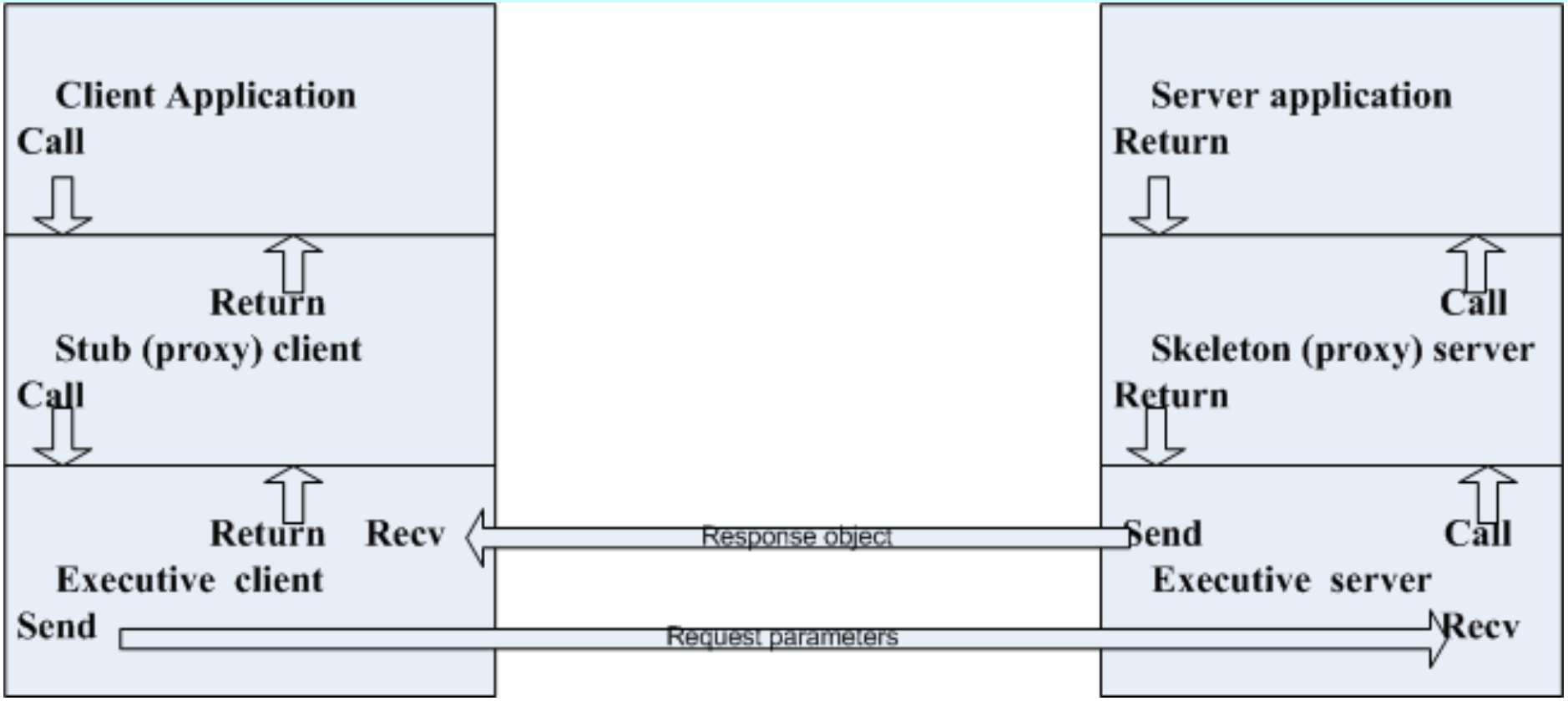
4.R*eprezentarea datelor*? Un exemplu simplu: dacă se transmite un întreg de pe o maşină *little-endian* reprezentat pe doi octeţi către o maşină *big-endian* care reprezintă întregul pe patru octeţi!

5. Fiecare tehnologie de tip RPC adoptă ***un sistem de reprezentare "universală" a datelor*****,** fiecare partener făcând conversiile în / din reprezentarea locală din / în cea universală.

6.Nu exista un control riguros al tipurilor

7.Nu există **control** într-o manieră naturală al **numărului şi tipurilor parametrilor**.

8.Nu există noţiunea de variabilă globală, comună pentru apelator şi pentru rutina apelată.



Implementari:

- ONC-RPC Unix: (Vezi **4Middleware/1ExecRpc/*.{c|h}** si [3] din pagina cursului)
  - https://man7.org/linux/man-pages/man3/rpc.3.html
  - https://docs.oracle.com/cd/E26502_01/pdf/E35597.pdf
- Microsoft RPC model: (Vezi **4Middleware/1ExecRpc/MsRpc.docx**)
  - https://docs.microsoft.com/en-us/windows/win32/rpc/rpc-start-page

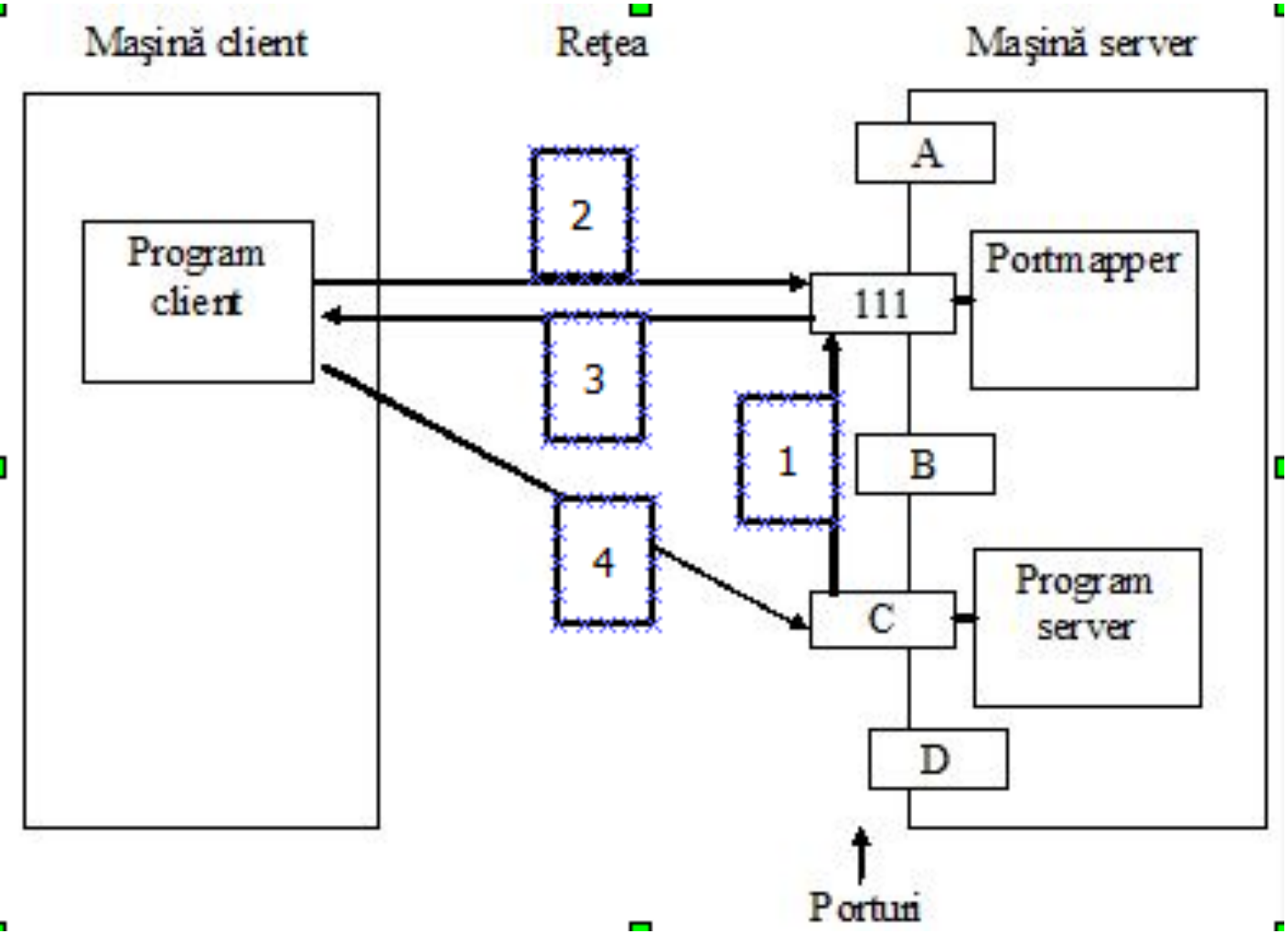
ONC - XDR simple filters

C	Filter	XDR
char	xdr_char()	int
short int	xdr_short()	int
unsigned short int	xdr_u_short()	unsigned int
int	xdr_int()	int
unsigned int	xdr_u_int()	unsigned int
long	xdr_long()	int
unsigned long	xdr_u_long()	unsigned int
float	xdr_float()	float
double	xdr_double()	double
void	xdr_void()	void
enum	xdr_enum()	int

Toate filtrele, inclusiv cele construite de user cu cele de mai sus, au prototipul de mai jos (bool_t este o renumire a lui de int):

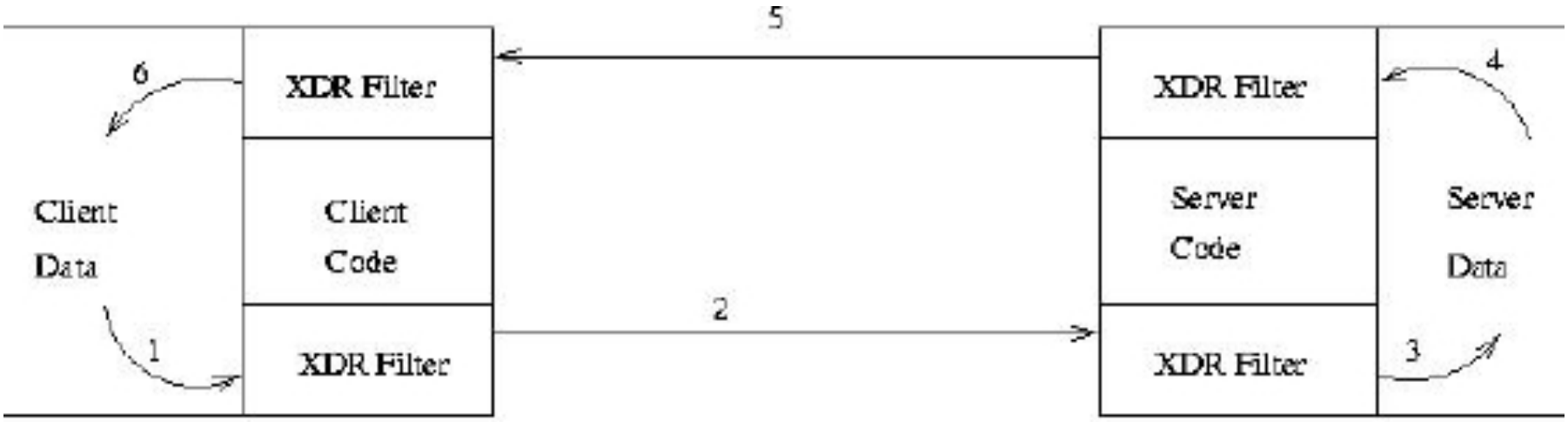
```
bool_t xdr_NAME(XDR *xdr, void *s)
```

Identificare – 3 longs: (numărprogram, numărversiune, numărprocedură)



1. Inregistrare procedura si atasare port propriu
2. Clientul cere de la portmapper portul procedurii
3. Portmapperul intoarce portul procedurii
4. Clientul trimite parametrii de intrare, asteapta rezolvarea, primeste rezultatul

ONC-RPC fluxul datelor



1. Clientul encodeaza intrarea cu filtrul XDR
2. Clientul transmite spre server intrarea encodata
3. Serverul decodeaza intrarea prin filtrul XDR

Se executa procedura remote

4. Serverul encodeaza rezultatul prin filtrul XDR
5. Serverul transmite spre client rezultatul encodat
6. Clientul decodeaza rezultatul prin filtrul XDR

RPC calitati, dezavantaje

Calitatati	Dezavantaje
Modelare **apel de functie** la nivelul limbajelor de programare clasice - ușor de înțeles pentru programatori Interactiune **sincronă request / response**: • naturala din p.d.v. limbaje de programare • potrivita pentru a pune in corespondenta raspunsurile la cererile corespunzatoare **Transparență privind distribuția** (în absenta eșecului), mascand complexitatea unui sistem distribuit Diverse **garanții de fiabilitate** prin tratarea implicita a unor aspecte ale eșecului	Interactiune **sincronă request / response**: • cere cuplare stransa intre client si server • clientul poate fi blocat mult timp daca serverul este prea incarcat, ceea ce impune uneori solutii multithreading • Asteptarile prea lungi pot conduce la caderea clientului. **Transparență privind distribuția** nu se pot masca toate problemele ce pot sa apara in complexitatea unui sistem distribuit **RPC nu este OO**, se invoca functii, nu metode ale unor obiecte – de aici un control slab al tipurilor parametrilor.

```
#define PROGRAM_EXEC   ((u_long)0x40000000)
#define VERSIUNE_EXEC  ((u_long)1)
#define EXEC_PING      ((u_long)1)
#define EXEC_UPCASE    ((u_long)2)
#define EXEC_ADD       ((u_long)3)

bool_t xdr_str(XDR *xdr, char *s) {
    int i;
    for (i = 0; i <= strlen(s); i++)
        if (xdr_char(xdr, &s[i]) == 0) return 0;
    return 1;
}

typedef struct doiInt {int a; int b; } doiInt;

bool_t xdr_doiInt(XDR *xdr, doiInt *s) {
    if (xdr_int(xdr, &(s->a)) == 0) return 0;
    if (xdr_int(xdr, &(s->b)) == 0) return 0;
    return 1;
}
```

```
#include "ExecOncRpc.h"
 - - -
char *ping() { - - - }

char *upcase(char *s) { - - - }

int *add(doiInt *s) { - - - }

main() {
    registerrpc(PROGRAM_EXEC, VERSIUNE_EXEC, EXEC_PING,
                ping, xdr_void, xdr_str);
    registerrpc(PROGRAM_EXEC, VERSIUNE_EXEC, EXEC_UPCASE,
                upcase, xdr_str, xdr_str);
    registerrpc(PROGRAM_EXEC, VERSIUNE_EXEC, EXEC_ADD,
                add, xdr_doiInt, xdr_int);
    svc_run();
}
```

ExecOncRpc fragmente de cod client

```
#include "ExecOncRpc.h"
main() {
    doiInt s;     int suma;   char linie[MAXSTRING];
    callrpc("localhost", PROGRAM_EXEC,VERSIUNE_EXEC,EXEC_PING,
           (xdrproc_t)xdr_void, NULL, (xdrproc_t)xdr_str, linie);
    printf("ping: \t%s\n", linie);
    strcpy(linie, "negru");
    callrpc("localhost", PROGRAM_EXEC,VERSIUNE_EXEC,EXEC_UPCASE,
           (xdrproc_t)xdr_str, linie, (xdrproc_t)xdr_str, linie);
    printf("upcase: \tnegru = %s\n", linie);
    s.a = 66;
    s.b = 75;
    callrpc("localhost", PROGRAM_EXEC,VERSIUNE_EXEC,EXEC_ADD,
           (xdrproc_t)xdr_doiInt, (char*)&s,
           (xdrproc_t)xdr_int, (char*)&suma);
    printf("add: \t66 + 75 = %d\n", suma);
    callrpc("localhost", PROGRAM_EXEC,VERSIUNE_EXEC,EXEC_PING,
           (xdrproc_t)xdr_void, NULL, (xdrproc_t)xdr_str, linie);
    printf("ping: \t%s\n", linie);
}
```

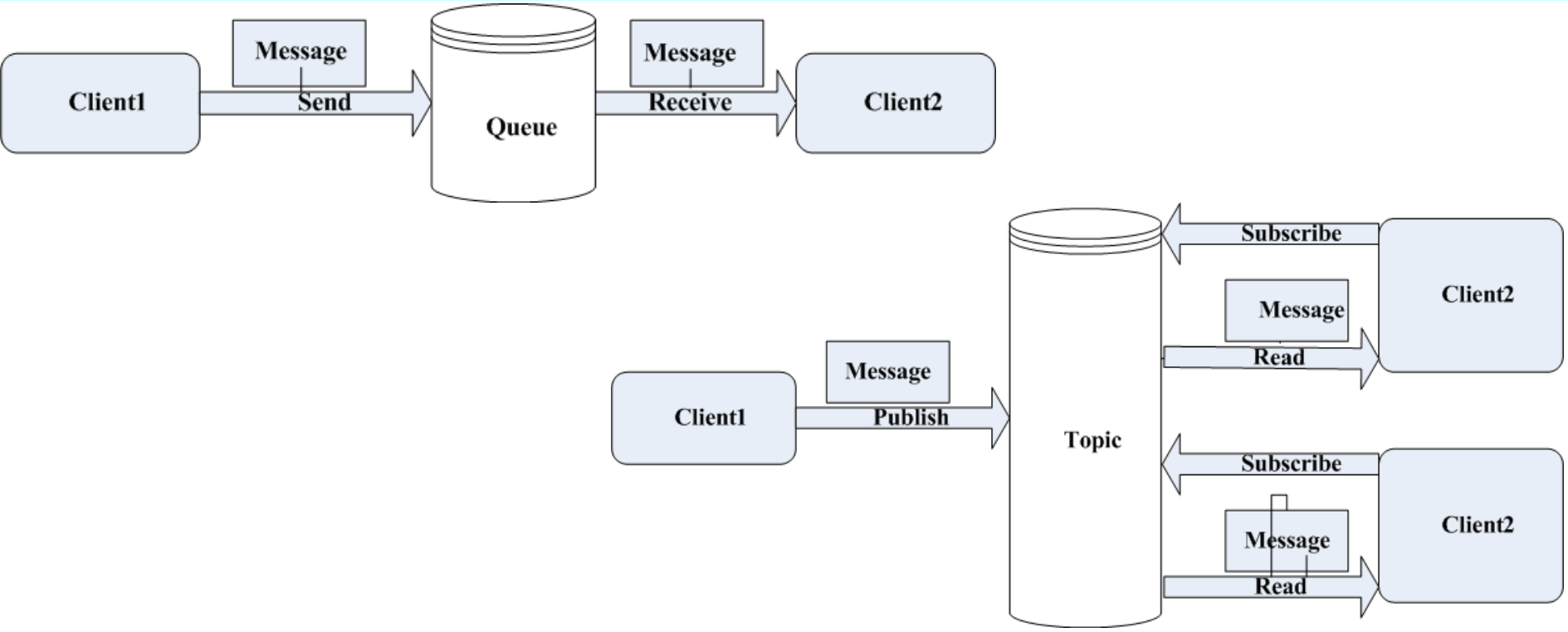
Un *serviciu de mesagerie* se ocupă de crearea, trimiterea şi recepţia mesajelor între persoane şi / sau procese de calcul. *Clienţii* abonaţi la serviciu sunt de două feluri:

- *clienţi cititori* care preiau mesaje, sincron sau asincron;
- *clienţi scriitori* care depun mesajele într-o coadă / topic de oferita de serviciu.

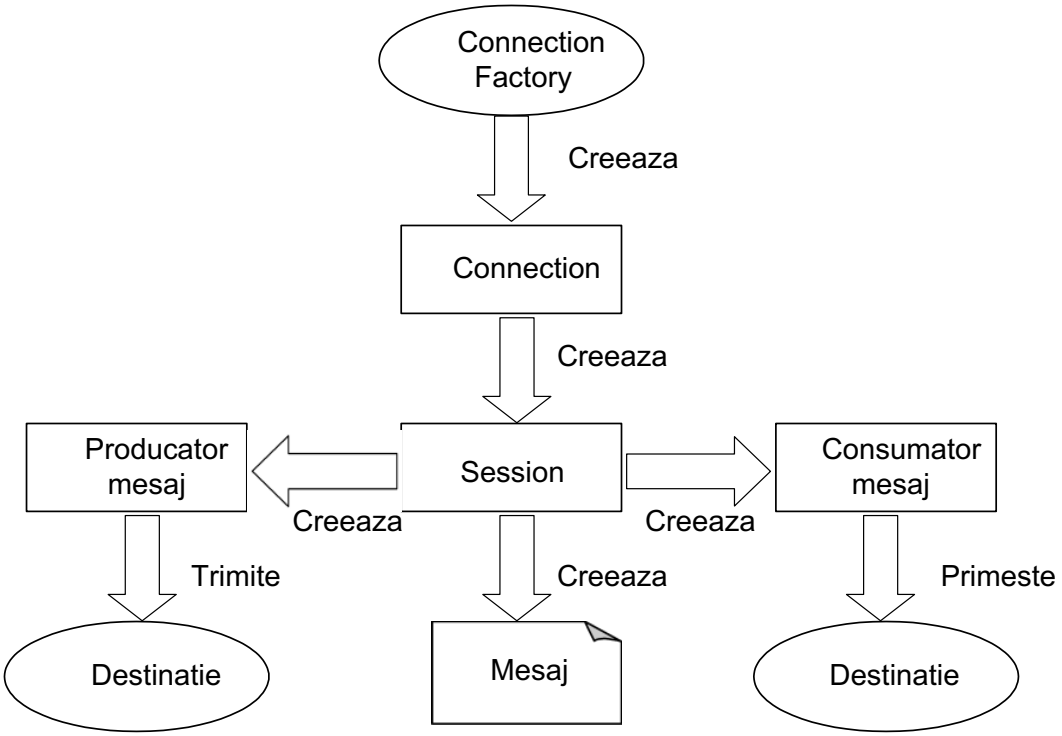
In serviciile de mesagerie nu este cazul ca scriitorul unui mesaj să fie în viaţă atunci când un cititor preia mesajul din coadă.

Componente JMS:

1.P*rovider JMS*. Implementează interfeţele JMS, furnizează facilităţile administrative şi de control ale serviciului. Principalele AS (**Jboss/WildFly**, **GlassFish** etc.) livrează şi JMS. Există distribuţii special destinate serviciului de mesagerie, de exemplu **ActiveMq**.
2.Clienţii JMS. Programe / componente Java care produc sau consumă mesaje.
3.Mesaje. Sunt obiecte prin intermediul cărora se comunică informaţii între clienţii JMS.
4.Obiecte administrative sunt create de provider: *fabricanţi de conexiuni* (**Connection Factories**) şi *destinaţii.* Destinatiile sunt: *cozi* (*queue*) sau *subiecte* (*topics)*.
5.Clienţi non JMS sunt programe care scriu / consumă mesaje din serviciu, dar care nu sunt implementate în Java, ci într-o manieră nativă, dependentă de platformă. Vom prezenta un exemplu cu distributia **stompest**, care emuleaza (cu mai mult sau mai putin succes), in diferite limbaje – Python, NodeJs s.a., un serviciu de tip JMS.



Consumarea mesajelor JMS se poate face în două moduri:
1.Consumare sincronă când clientul cititor specifică explicit obiectului destinaţie metoda **receive**. Metoda invocată blocheaza clientul cititor până la apariţia mesajului (eventual maximum un interval maximal de timp).
2.Consumare asincronă când clientul cititor defineşte un obiect aşteptător al mesajului - *message listener*. In momentul în care mesajul soseşte la destinaţie, providerul livrează mesajul către acest listener ca şi argument al unei metode numite **onMessage**, care trebuie rescrisa conform cerintelor de consum al mesajului.



Connection factory. Un obiect utilizat de client pentru a crea conexiunea cu un provider.Este o instanţă a uneia dintre interfeţele: **QueueConnectionFactory**, **TopicConnectionFactory**.

C*onnection* încapsulează conexiunea la un provider JMS. Implementează una dintre următoarele două interfeţe: **QueueConnection**, **TopicConnection**.

Session este un context destinat producerii sau consumării de mesaje. Se utilizează pentru crearea de producători de mesaje, consumatori de mesaje şi mesaje. Prin intermediul sesiunii se pot executa obiectele care aşteaptă evenimente, ca şi manevrarea tranzacţională a unor grupuri de emiteri şi receptări de mesaje.

Destinaţie este un obiect client utilizat pentru ţinta mesajelor produse, respectiv sursa mesajelor consumate. Fiecare destinaţie primeşte un *nume*. In acest nume se indică atât adresa Internet a maşinii care găzduieşte destinaţia, cât şi localizarea destinaţiei în cadrul maşinii. Numele este dat de un serviciu *JNDI* asociat providerului JMS.

Un *mesaj* este format din trei părţi: antet (singurul obligatoriu), proprietăţi şi corp.

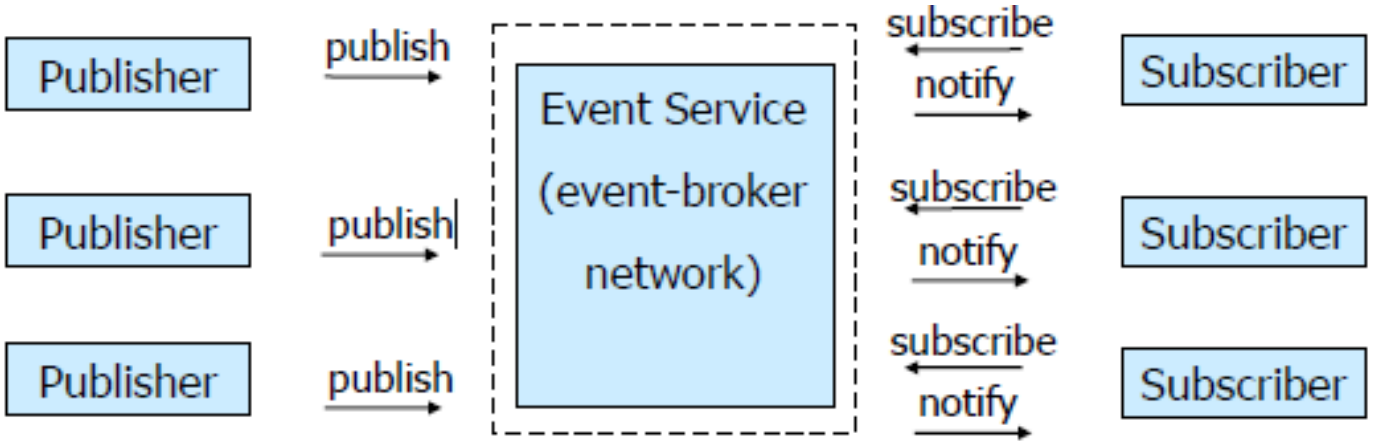
Antetul conţine un număr de câmpuri predefinite (vezi tabelul).

Proprietăţi mesaj se pot defini şi introduce pe lângă antet în vederea selectării mesajelor. API JMS oferă o serie de proprietăţi predefinite, precum şi un mecanism de definire a unor proprietăţi utilizator.
Corpul mesajului poate fi unul din următoarele tipuri (vezi tabelul).

Câmp antet	**Cine setează**
JMSDestination	`send` sau `publish`
JMSDeliveryMode	`send` sau `publish`
JMSExpiration	`send` sau `publish`
JMSPriority	`send` sau `publish`
JMSMessageID	`send` sau `publish`
JMSTimesstamp	`send` sau `publish`
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	JMS provider

Tip mesaj	**Conţinutul corpului**
TextMessage	Un String
MapMessage	Un set de perechi nume-valoare, unde nume sunt stringuri, iar valorile sunt tipuri de date primitive Java. Intrările pot fi accesate secvenţial prin enumerator sau direct prin numeI
BytesMessage	Un şir de octeţi fără interpretare
StreamMessage	Un şir de valori de tip primitiv Java, valori accesate secvenţial prin operaţii read
ObjectMessage	Serializarea unui obiect
Message	Conţinut vid, mesajul are doar antet şi proprietăţi

MOM: Paradigma publish / subscribe este un Event-base Middleware



- **Publisher** publica (face publicitate la) evenimente (mesaje)
- **Subscriber** isi exprima interesul in evenimente prin inscriere (subscribe)
- **Event-service** notifica subscriberii interesati in publicarea evenimentelor
- Evenimentele pot avea un continut arbitrar dar tipizat, sau perechi nume-valoare.

MOM calitati, dezavantaje

Calitatati	Dezavantaje
Se comunica folosind **mesaje** memorate in **cozi de mesaje** gazduite se **servicii de mesagerie**: • cuplare slaba intre client si server • Potrivit pentru integrarea de noi aplicatii Serviciile de mesagerie **decupleaza** clientul de server; cozile de mesaje sunt persistate pana la preluarea lor de catre destinatar. Se pot face diverse **ipoteze asupra continutului mesajelor**: se pot filtra, loga, transforma etc.	Desi a mai evoluat, exista inca un **nivel slab** de **abstractizare** a programarii: • dezvoltarea se face inca low / level • Corespondenta request / response este greu de realizat (desi se poate face) **Formatele mesajelor** nu sunt cunoscute de middleware, deci nu se poate face control de tipuri. Abstractia **queue** asigura numai comunicare unu-la-unu, ceea ce limiteaza scalabilitatea. Totusi, JMS depaseste acest impas prin implementarea **publish** / **subscribe**.

Implementari ExecJms folosind ActiveMQ (vezi **4Middleware/2ExecJms/**) (1/2)

S-a folosit serviciul de mesagerie **ActiveMq-5.16.0**. S-au implementat trei perechi de implementari: Java, Python si Node.js. Cele sase fisiere sursa sunt:

`ExecJmsServ.{java|py|js}` si **`ExecJmsClie.{java|py|js}`**

Toate programele sunt *clienti* ai acestui serviciu de mesagerie. Vom pastra totusi terminologia de **server Exec** si **client Exec**, asa cum am folosit-o la celelalte tehnologii.

Serverele si clientii se pot conecta la cate o pereche de cozi:

{J|P|N}cerere si **{J|P|N}raspuns**.

Pe prima dintre ele un client trimite cererea ca un string cu campurile separate prin |. Pe cea de-a doua coada clientul primeste raspunsul.

Cele sase implementari au fiecare cate o pereche de cozi implicita, dar la linia de comanda se pot da ca parametri adresa serviciului de mesagerie si numele unei perechi de cozi.

ActiveMq asculta la portul **61616** pentru java si **61613** pentru protocolul STOMPEST, folosit de implementarile Python si Node.Js

Ca elemente pregatitoare, trebuie definita variabila de mediu **ACTIVEMQ_HOME** ce indica radacina serviciului de mesagerie. Pentru Python trebuie instalat modulul **stompest**, iar pentru Node.Js trebuie instalat modulul **stomp-client**.

In directorul **...2ExecJms/** se afla:

- sursele celor sase fisiere (numele lor sunt in slide-ul precedent);
- sapte fisiere de comenzi pentru dirijarea implementarii si executiei.

Tabelul urmator precizeaza rolurile celor sapte fisiere de comenzi.

Nume fisier	Rolul lui
amq.bat	Lanseaza in lucru serviciul de mesagerie
j.bat	Compileaza si lanseaza in lucru serverul Java
y.bat	Lanseaza in lucru serverul python
n.bat	Lanseaza in lucru serverul NodeJs
jj.bat	Compileaza si lanseaza in lucru clientul Java
yy.bat	Lanseaza in lucru clientul python
nn.bat	Lanseaza in lucru clientul NodeJs

OOM calitati, dezavantaje

Calitatati	Dezavantaje
Obiectele si referintele la ele pot fi **locale** sau **remote.** **Obiectele remote** sunt vizibile prin **interfete remote** si mascate local prin **obiecte proxy**. Se ofera **suport OOP**: obiecte, metode, interfete, incapsulare, exceptii etc. **Interactiune sincrona request / response** ca si la RPC. **Transparenta locatiei** oferita de referintele ORB. **Servere specializate** ce ofera functionalitati ce simplifica dezvoltarea OOM.	Se ofera **numai** interactiune **sincronă request / response**, deci se cere cuplare stransa intre client si server. Se impune un mecanism de **Distributed garbage collection** care este dificil de realizat si nu ofera totdeauna solutii rezonabile. **OOM este static si dificil de proiectat si intretinut**, mai ales pentru sistemele embedded.

Hessian este un protocol binar usor (lighweight) de tip OOM . El isi defineste un mod propriu de reprezentare binara a unui set limitat de tipuri de date. Modul de reprezentare binar are la baza reprezentari Java, conversia reprezentand doar prefixarea, eventual postfixarea codului binar cu cate un caracter cu semnificatie speciala. Aceasta "imbracare" se face pentru a permite dezvoltarea de implementari si in alte limbaje decat Java. Deoarece este un protocol simplu si practic independent de limbaj, au aparut deja o serie de implementari non- Java. Diverse implementari se pot vedea la: **http://www.caucho.com/hessian/**

Implementarile pe care le avem in vedere:

- Servere si clienti:
  - **Java** `hessian-4.0.60.jar` + Jetty (sau Tomcat) embedded
  - **PHP** `HessianPHP_v2.0.3.zip`
- Numai clienti:
  - **Python** `hessianlib.py`
  - **Android** `hessdroid.jar`

Pentru moment, **Node.Js** nu ofera o implementare fezabila de hessian. Exista, totusi, un pachet pentru encoding / decoding date hessian si compatibilizarea cu java. Pentru aceasta se pot instala modulele **hessian.js** si **js-to-java**

Date primitive:

- **boolean** – serializat prin F sau T
- **int** - întreg pe 32 de biţi, serializat prin I ___
- **long** – întreg pe 64 de biţi, serializat prin L ___
- **double** – număr flotant pe 64 biţi, serializat prin D ___
- **date** – dată calendaristică pe 64 biţi, serializat prin d ___
- **string** – reprezentat UTF8, serializat prin S ___z
- **xml** – reprezentat UTF8, serializat prin X ___z
- **binary** – şir de octeţi, serializat prin B ___z
- **remote** – obiecte externe, serializat prin r ___ (URL-ul furnizorului)

Tipuri de colecţii:

- **list** – pentru liste şi pentru tablouri, serializat prin V ___ z
- **map** – pentru obiecte şi pentru hash tables, serializat prin M ___ z

C**onstructori speciali**:

- **null** – ce reprezintă pointerul null, serializat prin N
- **ref** – pentru referiri circulare la obiecte, serializat prin R ___

Fisierele implementarilor hessian in directorul ExecHess

Nume fisier	Rolul lui
`j.bat, h.bat`	Compilari si lansari in lucru servere Java + PHP
`jj1.bat, jj2.bat, hh.bat, yy.bat`	Compileaza/lanseaza/copiaza clientii Java, PHP, python
`ExecHessInte.java`	Interfata pentru server si client java
`ExecHessServ.java, ExecHessServ.php`	Sursele serverelor: Java cu arhiva war intr-un container Tomcat, PHP depus intr-un Apache
`ExecHessClie.java` `ExecHessClie.php` `ExecHessClie.py`	Sursele de implementare ale clientilor
`*.jar, *war`	Executabile server, client, arhiva war
`Android/` `initEHC.bat,saveEHC.bat` `AndroidManifest.xml` `activity_main.xml` `ExecHessInte.java` `MainActivity.java` `hessdroid.jar`	Surse: Initializare / salvare proiect android studio, descriptorul aplicatiei, descrierea interfetei grafice, interfata + sursa java Biblioteca hessian specifica android`HessianPHP_v2.0.3.zip`, `hessdroid.jar`, `hessianlib.py`, necesare implementarilor

```
import org.eclipse.jetty.server.*;
import org.eclipse.jetty.servlet.*;
import com.caucho.hessian.server.*; // hessian.jar
public class ExecHessServ_ extends HessianServlet
                           implements ExecHessInte{
    public String ping   - - -
    public String upcase - - -    public Integer add - - -
    public static void main(String[] args) throws Exception {
        Server service = new Server(9090);
        ServletHandler handler = new ServletHandler();
        service.setHandler(handler);
        handler.addServletWithMapping(ExecHessServ_.class,
                                                  "/exec");
        System.out.println("Start server jetty embedded");
        service.start();
        service.join();     }   }
```

```
import com.caucho.hessian.server.*; // hessian.jar
import javax.servlet.annotation.*;
@WebServlet("/Servlet")
public class ExecHessServ__ extends HessianServlet
                           implements ExecHessInte{
    public String ping   - - -     public String upcase - - -
    public Integer add - - - }
```

Hessian Java: interfata si fragmente de cod client

```
public interface ExecHessInte {
    public String ping();
    public String upcase(String s);
    public Integer add(Integer a,Integer b);
}
```

```
HessianProxyFactory hpf = new HessianProxyFactory();
hpf.setHessian2Request(true); // Altfel nu poate fi client php
ExecHessServ.phpExecHessInte proxy =
    (ExecHessInte) hpf.create(ExecHessInte.class, urlServ);
- - proxy.ping());
- - - proxy.upcase("negru"));
- - - proxy.add(66, 75)); - - -
```

```
class ExecHessServ {
    function ping () { - - - }
    function upcase ($s) { - - - }
    function add ($a, $b) { - - - }
}
include_once("hessianPHP/HessianService.php");//HessianPHP_v2.0.3
$service = new HessianService(new ExecHessServ());
$service->handle();
```

```
include_once("hessianPHP/HessianClient.php");//HessianPHP_v2.0.3
class ExecHessClie {
    function __construct($urlServ) {
        echo "Client Hessian PHP: " . $urlServ . "<br/>";
        $serviciu = new HessianClient($urlServ);
        echo "ping: " . $serviciu->ping() . "<br/>";
        echo "upcase: negru = " . $serviciu->upcase("negru");
        echo "add: 66 + 75 = " . $serviciu->add(66, 75)."<br/>";
        echo "ping: " . $serviciu->ping() . "<br/>";     }  }
if (isset($_GET["urlServ"]))
    new ExecHessClie($_GET["urlServ"]);
else
    new ExecHessClie("http://localhost/ExecHessServ.php");
```

```
from pyhessian.client import HessianProxyimport sys
class ExecHessClie:
    def __init__(self, urlServ):
        print "Client Hessian Python: " + urlServ
        proxy = HessianProxy(urlServ)
        print "ping: \t" + proxy.ping()
        print "upcase: \tnegru = " + proxy.upcase("negru")
        print "add: \t66 + 75 = " + `proxy.add(66, 75)`
        print "ping: \t" + proxy.ping()
if len(sys.argv) > 1:
    ExecHessClie(sys.argv[1]) # http://localhost:8080/exec
else:
    ExecHessClie("http://localhost/ExecHessServ.php")
```

Pyro

Pyro este o tehnologie pentru obiecte distribuite, scrisă în întregime în Python. Folosind doar câteva linii de cod în plus faţă de businessul aplicaţiei, Pyro oferă o reţea de comunicaţie puternică între obiecte răspândite pe mai multe maşini. Din punct de vedere conceptual, Pyro este o formă orientată obiect de tip RPC, foarte similară cu Java RMI dar mai puţin similară cu CORBA.

Prima varianta Pyro functionala este **Pyro3** http://www.xs4all.nl/~irmen/pyro3/ sau http://sourceforge.net/projects/pyro/. De la una dintre aceste adrese se poate downloada în prezent **Pyro-3.16.tar.gz** care este o distribuţie independentă de platformă, sau **Pyro-3.16.win32.exe** distribuţie specială numai pentru Windows.

Varianta actuala este **Pyro4** https://pypi.python.org/pypi/Pyro4 de unde se poate downloada **Pyro4-4.23.tar.gz**. Mai multe detalii si surse se pot obtine de la https://github.com/irmen/Pyro4 si http://www.razorvine.net
Pyro4 ofera facilitati de utilizare a unor clienti Java si C# prin pachetul **Pyrolite**. Pentru aceasta, se ofera un mecanism de serializare propriu numit **serpent.**

Instalarea **Pyro3** se poate face **numai pentru Python2!** cu: **python –m pip install pyro**
Instalarea **Pyro4** se poate face cu: **python –m pip install Pyro4**
Clientii Java Pyrolite folosesc arhivele **pyrolite.jar** si **serpent.jar**
Clientii C# Pyrolite folosesc arhivele (.NET-4.0) **Razorvine.Pyrolite.dll** si **Razorvine.Serpent.dll**

```
import Pyro.core
import socket
from datetime import datetime
class ExecPyroServ(Pyro.core.ObjBase):
    def __init__(self):
        Pyro.core.ObjBase.__init__(self)
    def ping(self):
        name = socket.gethostname()
        ip = socket.gethostbyname(name)
        return "Python Exec Pyro3" + ": " + name + \
            "(" +ip + ":7766" + "), " + `datetime.now()`
    def upcase(self, s):
        return s.upper()
    def add(self, a, b):
        return a + b
def start():
    Pyro.core.initServer()
    daemon = Pyro.core.Daemon()
    uri = daemon.connect(ExecPyroServ(), "exec")
    print "Python Exec Pyro3 waiting with name \"exec\" at: " + `uri`
    daemon.requestLoop()
start()
```

```
import Pyro.core
import sys
class ExecPyroClie:
    def __init__(self, urlServ):
        Pyro.core.initClient()
        proxy = Pyro.core.getProxyForURI(urlServ)
        print "Client RMI Pyro3 Python: "+urlServ
        print "ping: \t" + proxy.ping()
        print "upcase: \tnegru = " + proxy.upcase("negru")
        print "add: \t66 + 75 = " + `proxy.add(66, 75)`
        print "ping: \t" + proxy.ping()
if len(sys.argv) > 1:
    ExecPyroClie(sys.argv[1])
else:
    ExecPyroClie("PYROLOC://localhost:7766/exec")
```

```
import Pyro4
import socket
from datetime import datetime
@Pyro4.expose # Obiectul isi exporta metodele!
class ExecPyro4Serv(object):
    def ping(self):
        name = socket.gethostname()
        ip = socket.gethostbyname(name)
        return "Python ExecPyro4" + ": " + name + \
            "(" +ip + ":7543" + "), " + `datetime.now()`
    def upcase(self, s):
        return s.upper()
    def add(self, a, b):
        return a + b
def start():
    daemon = Pyro4.Daemon(port=7543)
    uri = daemon.register(ExecPyro4Serv(), "exec")
    print "Python ExecPyro4 waiting at: " + `uri`
    daemon.requestLoop()
start()
```

```
import Pyro4
import sys
class ExecPyro4Clie:
    def __init__(self, urlServ):
        proxy = Pyro4.Proxy(urlServ)
        print "Client Python Pyro4: "+urlServ
        print "ping: \t" + proxy.ping()
        print "upcase: \tnegru = " + proxy.upcase("negru")
        print "add: \t66 + 75 = " + `proxy.add(66, 75)`
        print "ping: \t" + proxy.ping()
if len(sys.argv) > 1:
    ExecPyro4Clie(sys.argv[1])
else:
    ExecPyro4Clie("PYRO:exec@localhost:7543")
```

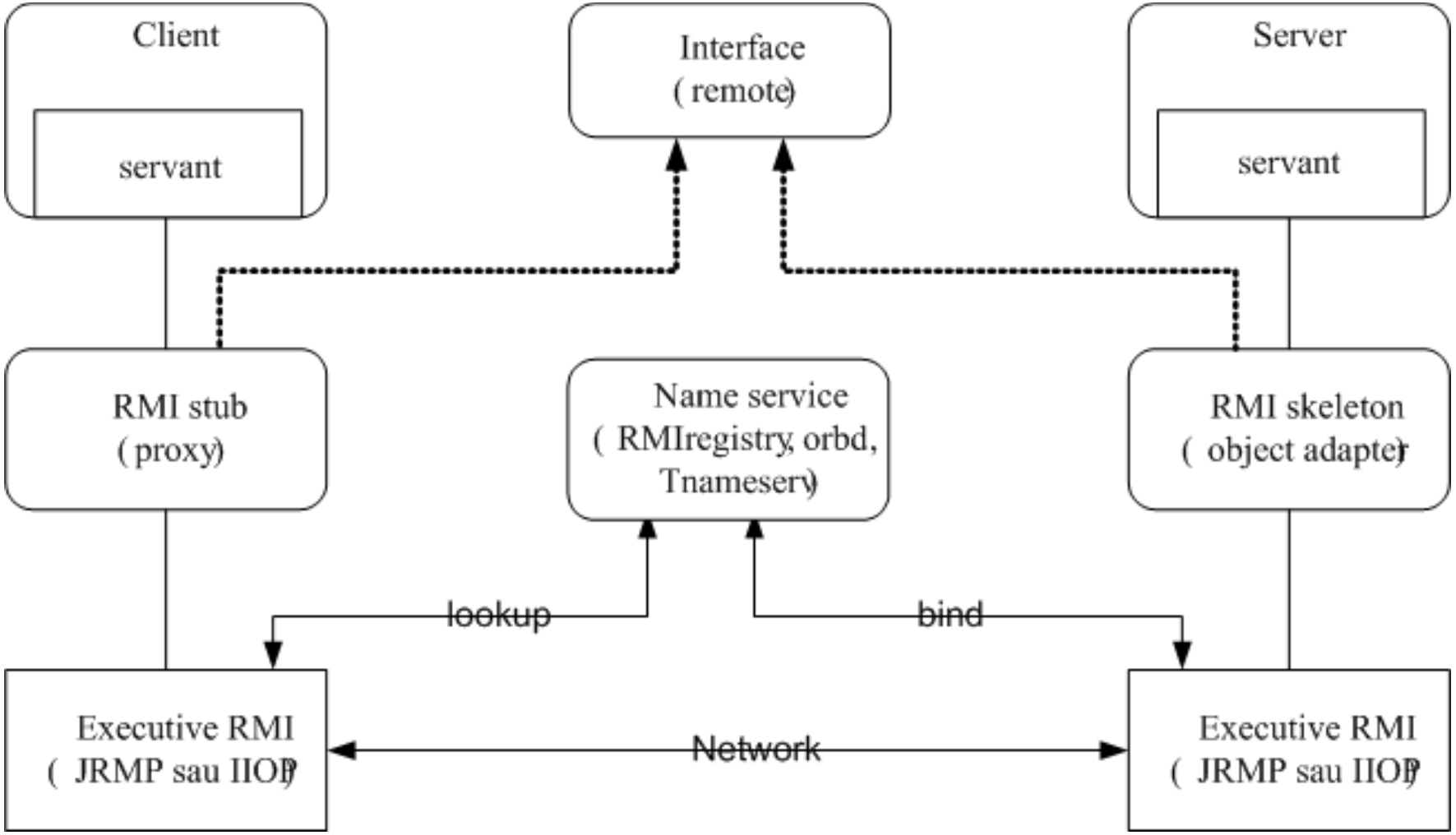
```
using Razorvine.Pyro;                          // Razorvine.Pyrolite.dll (.NET-4.0)
using Razorvine.Pickle;                        // Razorvine.Serpent.dll  (.NET-4.0)
using System;
using System.Collections;
class ExecPyro4Clie {
    public ExecPyro4Clie(String urlServ) {
        Console.WriteLine("Client C# Pyro (pyrolite): " + urlServ);
        Hashtable uri = Uri.uriFields(urlServ);
        String server = (String)uri["user"];
        String host = (String)uri["host"];
        int port = int.Parse((String)uri["port"]);
        PyroProxy proxy = new PyroProxy(host, port, server);
        Console.WriteLine("ping: \t" + proxy.call("ping"));
        Console.WriteLine("upcase: \tnegru = " + proxy.call("upcase", "negru"));
        Console.WriteLine("add: \t66 + 75 = " + proxy.call("add", 66, 75));
        Console.WriteLine("ping: \t" + proxy.call("ping"));
        proxy.close();
    }

    public static void Main(String[] args) {
        if (args.Length > 0)
            new ExecPyro4Clie(args[0]);
        else
            new ExecPyro4Clie("PYRO:exec@localhost:7543");
    }
}
```

```
import net.razorvine.pyro.*;                                    // pyrolite.jar
import net.razorvine.pickle.*;                                  // serpent.jar
import java.util.*;
public class ExecPyro4Clie {
    public ExecPyro4Clie(String urlServ) throws Exception {
        System.out.println("Client Java Pyro (pyrolite): " + urlServ);
        Hashtable<String, String> uri = Uri.uriFields(urlServ);
        String server = uri.get("user");
        String host = uri.get("host");
        int port = Integer.parseInt(uri.get("port"));
        PyroProxy proxy = new PyroProxy(host, port, server);
        System.out.println("ping: \t" + proxy.call("ping"));
        System.out.println("upcase: \tnegru = " + proxy.call("upcase", "negru"));
        System.out.println("add: \t66 + 75 = " + proxy.call("add", 66, 75));
        System.out.println("ping: \t" + proxy.call("ping"));
        proxy.close();
    }
    public static void main(String[] args) throws Exception {
        if (args.length > 0)
            new ExecPyro4Clie(args[0]);
        else
            new ExecPyro4Clie("PYRO:exec@localhost:7543");
    }
}
```

RMI furnizează o modalitate de executie unei aplicaţii Java pe mai multe hosturi, care discuta între ele prin intermediul obiectelor Java. Serverul RMI găzduieşte unul sau mai multe obiecte numite *obiecte servant* . Clientii vad aceste obiecte ca si obiecte locale.

RMI oferă o infrastructură care realizează exact acest lucru. Componentele necesare sunt:
1.Un *serviciu de nume* (rmiregistry, tnameserv, orbn etc.) necesar înregistrării obiectelor ce vor fi invocate de la distanţă;
2.O *interfaţă* care precizează legătura dintre client şi server (obiect care exporta);
3.Cate o clasă care implementează cate un obiect *servant* .
4.Un *server* care să găzduiască acest obiect servant. Este posibil ca serverul sa implementeze direct obiectele serant, fara a fi necesare clase implementatoare.
5.Pentru fiecare obiect servant , la server este necesar un obiect adaptor (*skeleton*) pentru relaţia de tip RMI.
6.Clasele client dorite.
7.Pentru fiecare host client este necesara o clasă proxy (*stub*), reprezentant al servantului pentru client.
8.In conexiune directă cu reţeaua şi în directă colaborare cu stub şi skeleton se află componenta *ExecutivRMI*; protocoalele prin care comunică acest executiv sunt **JRMP** (***J**ava **R**emote **M**ethod **P**rotocol*) sau **IIOP** (***I**nternet **I**nter-**O**RB **P**rotocol*).

RMI schema



RMI over JRMP (Java Remote Method Protocol) – close architecture

RMI over IIOP (Internet Interb ORB Protocol) – open architecture, CORBA compatible

ExecRmi: fragmente de cod interfata si implemntari servant

```
import java.rmi.*;
public interface ExecRmiInte extends Remote {
    public String ping() throws RemoteException;
    public String upcase(String s) throws RemoteException;
    public int add(int a, int b) throws RemoteException; }


import java.rmi.*; import java.net.*; import java.util.*;
public class ExecRmiImpl implements ExecRmiInte {
    public ExecRmiImpl() throws RemoteException {    }
    public String ping() { - - - }
    public String upcase(String s) { - - - }
    public int add(int a, int b) { - - - }     }


import javax.rmi.*; import java.rmi.*;
import java.util.*; import java.net.*;
public class ExecRmiImpl extends PortableRemoteObject
                                    implements ExecRmiInte {
    public ExecRmiImpl() throws RemoteException {super(); }
    - - - implementarile metodelor ping, upcase, add - - -   }
```

ExecRmi: fragmente de cod servere

```
import java.rmi.*;  import java.rmi.registry.*;
import java.rmi.server.*;
public class ExecRmiJrmpServ {
    public ExecRmiJrmpServ(String nume) throws Exception {
        LocateRegistry.getRegistry();
        ExecRmiInte stub =
           (ExecRmiInte) UnicastRemoteObject.exportObject(
                new ExecRmiJrmpImpl(), 0);
        Naming.bind(nume, stub);  }
- - - new ExecRmiJrmpServ( - - - ); - - - }      }
```

```
import javax.naming.*;  import java.util.*;
public class ExecRmiIiopServ {
public ExecRmiIiopServ(String host, String port, String nume)
           throws Exception {
        Properties props = new Properties();
        props.setProperty("java.naming.factory.initial",
                "com.sun.jndi.cosnaming.CNCtxFactory");
        props.setProperty("java.naming.provider.url",
                "iiop://" + host + ":" + port);
        Context ctx = new InitialContext(props);
        ctx.rebind(nume, new ExecRmiIiopImpl());  }
 - - - ExecRmiIiopServ( - - - ); - - -
```

```
import java.rmi.registry.*;
public class ExecRmiJrmpClie {
    public ExecRmiJrmpClie(String host, String nume)
                                        throws Exception {
        Registry registry = LocateRegistry.getRegistry(host);
        ExecRmiInte proxy = (ExecRmiInte) registry
                .lookup(nume);
- - - proxy.ping() - - - proxy.upcase(...) - - - proxy.add(...)
import javax.naming.*; import javax.rmi.*; import java.util.*;
public class ExecRmiIiopClie {
   public ExecRmiIiopClie(String host, String port,String nume)
                                      throws Exception {
        Properties props = new Properties();
        props.setProperty("java.naming.factory.initial",
                "com.sun.jndi.cosnaming.CNCtxFactory");
        props.setProperty("java.naming.provider.url",
                          "iiop://" + host + ":" + port);
        Context ctx = new InitialContext(props);
        Object ref = ctx.lookup(nume);
        ExecRmiInte proxy = (ExecRmiInte)PortableRemoteObject.
                                  narrow(ref, ExecRmiInte.class);
- - - proxy.ping() - - - proxy.upcase(...) - - - proxy.add(...)
```

```
start rmiregistry

javac -d . -cp . *.java

start java ExecRmiJrmpServ

java ExecRmiJrmpClie
```

```
javac -d . -cp . *.java

rmic -iiop ExecRmiIiopImpl

start orbd -ORBInitialPort 1050

start java ExecRmiIiopServ

java ExecRmiIiopClie
```

In cazul JRMP stub-ul si skeletonul sunt generate dinamic si incluse in codul serverului (implementarii servantului ?). Pentru IIOP, in urma comenzii **rmic** se genereaza fisiere pe post de stub (proxy) care trebuie sa insoteasca aplicatiile client (acestea vor fi utile il legarea cu CORBA):

```
_ExecRmiInte_Stub.class
_ExecRmiIiopImpl_Tie.class
```

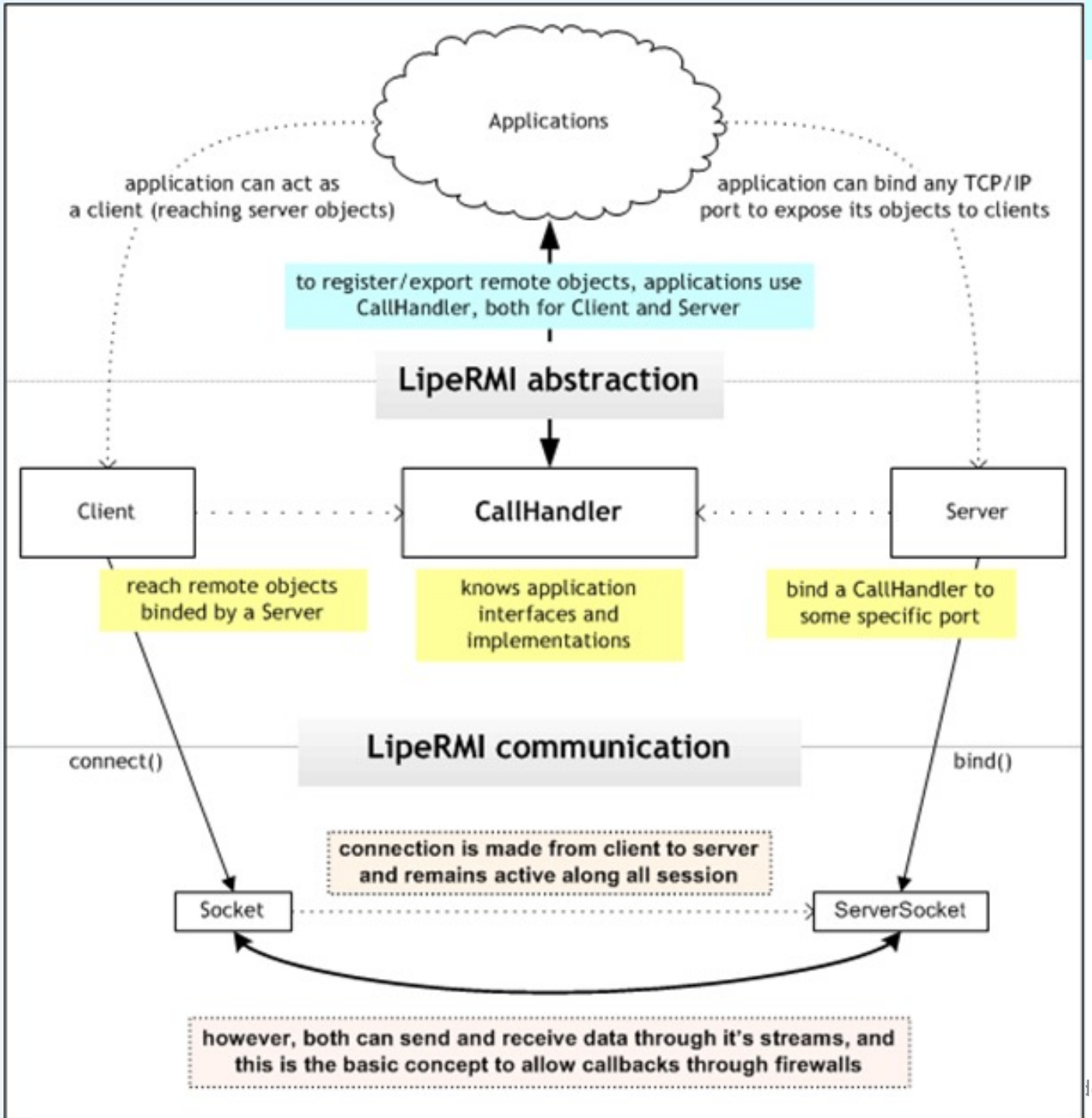
ExecRmi: plasarea fisierelor la servere si clienti

RMI JRMP	RMI IIOP
Spatiul server	**Spatiul server**
ExecRmiInte.class ExecRmiJrmpImpl.class ExecRmiJrmpServ.class	ExecRmiInte.class ExecRmiiopImpl.class ExecRmiiopServ.class _ExecRmiiopImpl_Tie.class
Spatiul client	**Spatiul client**
ExecRmiInte.class ExecRmiJrmpClie.class	ExecRmiInte.class ExecRmiiopClie.class _ExecRmiInte_Stub.class

LipeRMI este o implementare complet nouă a RMI care înlocuiește Java RMI nativ. Este complet independent de Java RMI nativ și utilizează o abordare optimizată pe Internet. Propunerea proiectului este de a rescrie Java RMI și de a elimina unele probleme ale RMI din implementarile standard (nu intram in detalii). Cel mai important neajuns al RMI ce Il face uneori impopular, este o arhitectură de comunicare proastă în Internet. Deoarece LipeRMI nu folosește importuri, el poate fi utilizat şi pe dispozitive mobile. Dezvoltarea LipeRMI are câteva puncte cheie:

- furnizeaza o comunicare abstractă între obiecte rezidente în diferite mașini virtuale;
- oferă un API clar,extensibil, similar cu RMI și o arhitectură simplă;
- nu necesita importuri de jar-uri externe;
- optimizeaza modul în care clienții ajung la server prin reutilizarea aceluiași socket și păstrarea lui în viață. Astfel, acesta poate sta in spatele unei retele locale, a unui router sau a unui firewall;
- ofera o modalitate de a ști când se întâmplă evenimente la conexiune;
- ofera o modalitate de a şti în orice moment și în orice metodă, care socket a făcut acest apel;
- portarea de la RMI este simplă;
- este open source.

Pentru a vedea diferentele, este suficient sa se compare sursele de la JRMP cu Lipe.
Vezi si documentul **LipeRMI.docx**


Applications
application can act as
a client (reaching server objects)
application can bind any TCP/IP
port to expose its objects to clients
to register/export remote objects, applications use
CallHandler, both for Client and Server
LipeRMI abstraction
Client
CallHandler
Server
reach remote objects
binded by a Server
knows application
interfaces and
implementations
bind a CallHandler to
some specific port
LipeRMI communication
connect()
bind()
connection is made from client to server
and remains active along all session
Socket
ServerSocket
however, both can send and receive data through it's streams, and
this is the basic concept to allow callbacks through firewalls

```
public interface ExecLipe { ca ExecRmiInte, fara nimic din RMI }

import lipermi.exception.* lipermi.handler.* lipermi.net.*;
public class ExecLipeServ implements ExecLipe {
    int port;
    public ExecLipeServ(int port) throws Exception {
        this.port = port;
        CallHandler callHandler = new CallHandler();
        callHandler.registerGlobal(ExecLipe.class, this);
        Server server = new Server();
        server.bind(port, callHandler);
    public String ping() { - - - }
    public String upcase(String s) { - - - }
    public int add(int a, int b) { - - - }     }

import lipermi.exception.* lipermi.handler.* lipermi.net.*;
public class ExecLipeClie {
  public ExecLipeClie(String host, int port) throws Exception {
    CallHandler callHandler = new CallHandler();
    Client client = new Client(host, port, callHandler);
    ExecLipe proxy = (ExecLipe)client.getGlobal(ExecLipe.class);
     - - - proxy.ping() - - - proxy.upcase(…) - - - proxy.add(…)
```

CORBA – Common Object Request Broker Architecture este o specificare software pentru **normalizarea semantică** a apelului între obiecte distribuite în spaţii de adrese diferite. Normalizarea presupune că **obiectele sunt elaborate pe platforme şi în limbaje de programare diferite,** motiv pentru care la transferul de obiecte acestea sunt supuse unei operaţii de normalizare.

CORBA foloseste **un limbaj IDL** (Interface Description Language), pentru a specifica interfeţele prin care obiectele se vor prezenta în lumea exterioară.

CORBA realizează **mapări ale IDL** în limbajele de implementare: C++, Java, Ada, C, Lisp, Ruby, Smalltalk, COBOL, PL/I, Python etc.

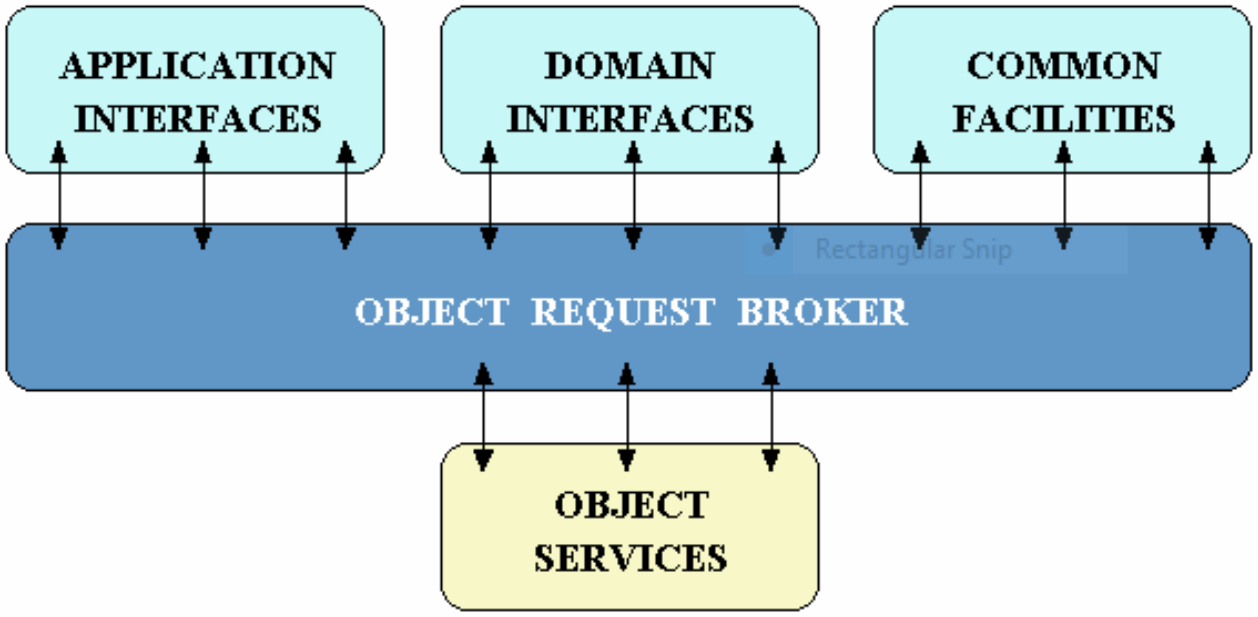
Toate aceste implementări pleacă de la specificaţiile IDL şi **generează clase de cod sursă în limbajul ţintă**. In acest mod se obţine materializarea de fapt a unor precursori pentru standardele actuale de împachetare de la serviciile web.

CORBA a fost conceput aşa încât **să fie independent de sistemul de operare**. El este disponibil si în Java (OS-independent), precum şi nativ pentru Linux / Unix, Windows, Sun, Mac ş.a.

Implementari CORBA open source

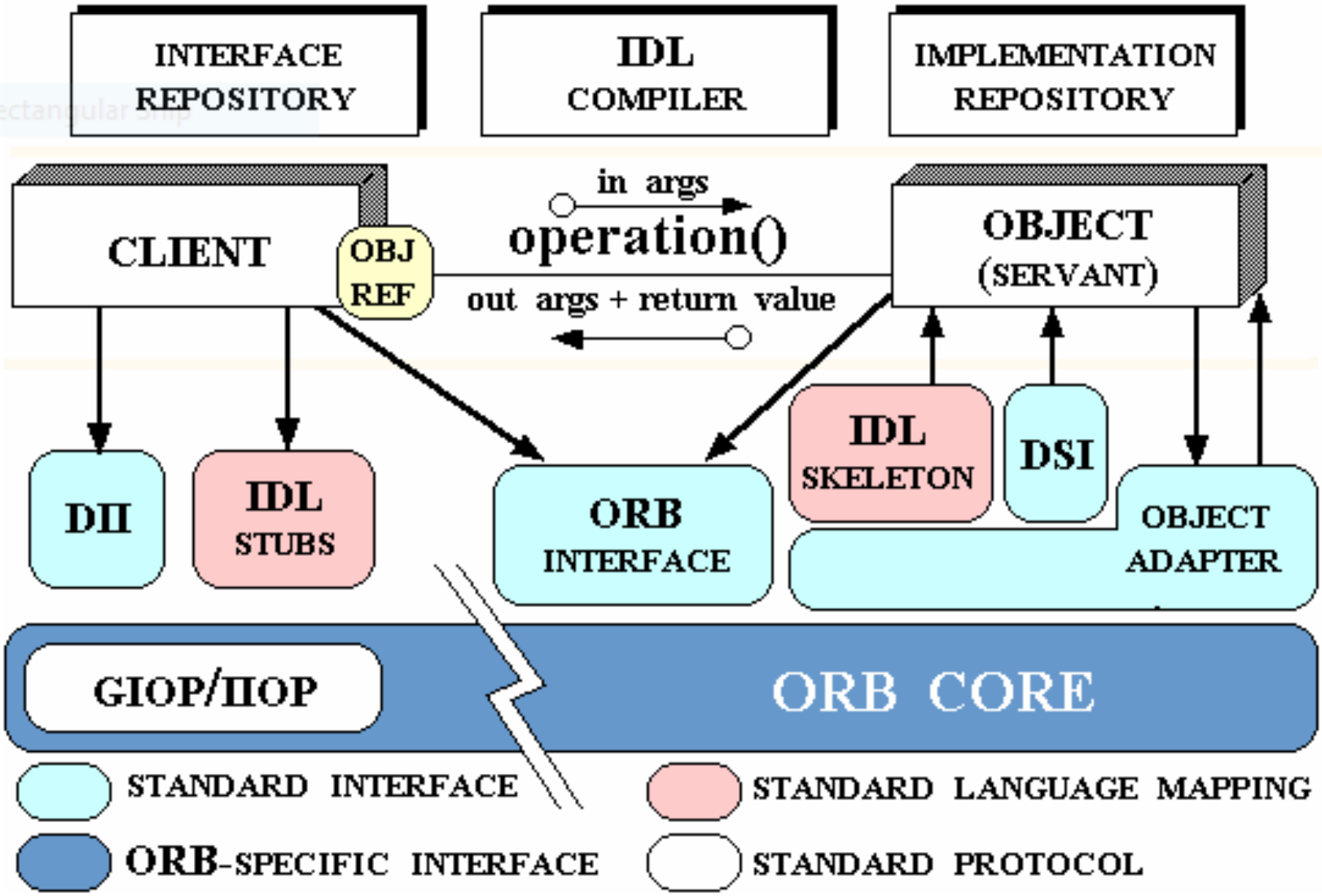
- **Fnorb** is a CORBA 2.0 ORB for Python first developed by DSTC (http://www.dstc.edu.au).
- **opalORB** (a Perl ORB) is an implementation of the OMG CORBA ORB standard is written completely in Perl.
- **MICO** is a mature, secure, robust, fully standards-compliant implementation of the CORBA standard.
- **OpenCCM** stands for the *Open CORBA Component Model Platform*: The first public available and open source implementation.
- **CCM** (Component Model) is the first vendor neutral open standard for *Distributed Component Computing* supporting various programming languages, operating systems.
- **JacORB** is a CORBA 2.3 compliant for Java Applications.
- **omniORB** is a robust high performance CORBA 2.6 compliant ORB for C++ and Python. See BOIAN F.M., BOIAN R.F. Tehnologii fundamentale Java pentru aplicaţii Web. Ed. Albastră, grupul Microinformatica, Cluj, 2005 for applications.
- **JavaIDL** - CORBA compliant included in JDK

Vom prezenta numai Java IDL Aceasta implementare este perfect compatibila cu Java RMI IIOP, in sensul ca fiecare clientii celor doua tehnologii pot importa obiecte gazduite de ambele servere.
Pentru aceasta sunt necesare transferuri ale unor fisiere .class care definesc stub-urile si skeleton-urile din cele doua implementari.



- **Object Services** – interfete (servicii) folosite de multe programe, cum ar fi: Naming Service – descoperirea obiectelor dupa nume, Trading Service – descopera obiectele dupa proprietati, servicii de securitate, tranzactii, evenimente etc.
- **Common Facilities** – servicii de uz general, folosite nu neaparat de userii finali. De exemplu, facilitati de elaborarea de documente cu OpenDoc.
- **Domain Interfaces** – orientate spre domeni specifice: finante, comunicatii, sanatate etc.
- **Application Interfaces** - dezvoltate pentru aplicatii specifice, nestandardizate.

CORBA: ORB Object Request Broker (1/3)



Object - Aceasta este o entitate de programare CORBA, care constă dintr-o identitate, o interfata si o implementare – numita **Servant**.

Servant - Aceasta este o implementare intr-un limbaj de programare ce suportă o interfață CORBA IDL: C, C ++, Java, Smalltalk, Ada.

Client - Acesta este entitatea program care invocă o operatie unui obiect. Accesarea serviciilor unui obiect la distanță este transparenta pentru apelant.

Object RequestBroker (ORB) - ORB oferă un mecanism pentru a comunica transparent intre clienți si implementari de obiecte. ORB distribuit decupleaza clientul de detaliile metodei invocate. Acest lucru face ca solicitările client par a fi apeluri de proceduri locale. Atunci când un client invocă o operație, ORB este responsabil pentru identificarea implementării obiectului, activând, dacă este necesar, livrarea cererii la obiect, returnarea unui răspuns la apelant.

ORB Interface - este o entitate logică care poate fi pusă în aplicare în diverse moduri (cum ar fi unul sau mai multe procese sau un set de biblioteci). Pentru a decupla aplicații de la detaliile de implementare, specificația CORBA definește o interfață abstractă pentru un ORB (folosind IDL).

CORBA IDL stub \ skeleton sunt intermediari între aplicațiile client / server și ORB. Transformarea intre definițiile CORBA IDL și limbajul de programare țintă este automatizat printr-un compilator CORBA IDL.

Dynamic Invocation Interface (DII) - permite unui client să acceseze direct mecanismele furnizate de un ORB. Aplicațiile folosesc DII ca să emită în mod dinamic solicitările către obiecte, fără a necesita definirea de interfete IDL specifice si legarea de stub / skeleton. DII permite clienților să facă sincronizari neblocante si apeluri oneway.

Dynamic Skeleton Interface (DSI) - Acesta este analog DII dar pe partea de server. DSI permite o ORB să livreze cereri către un obiect despre care nu are cunoștințe în timpul compilării.

Object Adapter - Aceasta ajută ORB cu livrarea de solicitări către obiect și cu activarea obiectului. Un astfel de adaptor poate fi specializat pentru a oferi suport pentru anumite stiluri de implementare.

Fisierul de descriere `ExecIdlInte.idl`

```
interface ExecIdlInte {
    string  ping();
    string  upcase(in string s);
    long add(in long a, in long b);
};
```

```
import java.net.*;
import java.util.*;
public class ExecIdlImpl extends ExecIdlIntePOA {
    public ExecIdlImpl() {   super();    }
    public String ping() { - - - }
    public String upcase(String s) { - - -}
    public int add(int a, int b) { - - - }
}
```

Clasa ExecIdlIntePOA este generata (sursa Java) de catre comanda **idlj** in urma prelucrarii fisierului **ExecIdlInte.idl**

```
import org.omg.CORBA.*; import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.PortableServer.*;
public class ExecIdlServ {
  public ExecIdlServ(String host,String port,String nume)
            throws Exception {
    String t[]={"-ORBInitialHost", host, "-ORBInitialPort", port };
    ORB orb = ORB.init(t, null);
    POA rootpoa = POAHelper.narrow(orb
                .resolve_initial_references("RootPOA"));
    rootpoa.the_POAManager().activate();
    ExecIdlImpl execImpl = new ExecIdlImpl();
    org.omg.CORBA.Object ref=rootpoa.servant_to_reference(execImpl);
    ExecIdlInte href = ExecIdlInteHelper.narrow(ref);
    org.omg.CORBA.Object objRef = orb
                .resolve_initial_references("NameService");
    NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
    NameComponent path[] = ncRef.to_name(nume);
    orb.run();
  }

 - - - ExecIdlServ( - - - ); - - -
```

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class ExecIdlClie {
  public ExecIdlClie(String host, String port, String nume)
            throws Exception {
    String t[]={"-ORBInitialPort",port, "-ORBInitialHost",host};
    ORB orb = ORB.init(t, null);
    org.omg.CORBA.Object objRef = orb
                .resolve_initial_references("NameService");
    NamingContextExt ncRef =
                NamingContextExtHelper.narrow(objRef);
    ExecJavaIdlInte proxy = ExecJavaIdlInteHelper.narrow(ncRef
                .resolve_str(nume));

 - - - proxy.ping() - - - proxy.upcase(...) - - - proxy.add(...)
```

```
idlj -fall ExecIdlInte.idl

javac *.java

start orbd -ORBInitialPort 3000

start java ExecIdlServ

java ExecIdlClie
```

In urma comenzii **idlj** se genereaza urmatoarele surse Java, care vor fi apoi compilate si vor face parte din client si / sau server:

ExecIdlInte.java
ExecIdlInteHelper.java
ExecIdlInteHolder.java
ExecIdlInteOperations.java
ExecIdlIntePOA.java
_ExecIdlInteStub.java

CORBA – ExecJavaIdl
Spatiul server
ExecIdlInte.class ExecIdlImpl.class ExecIdlServ.class ExecIdlInteHelper.class ExecIdlInteHolder.class ExecIdlInteOperations.class ExecIdlIntePOA.class _ExecIdlInteStub.class
Spatiul client
ExecIdlInte.class ExecIdlClie.class ExecIdlInteHelper.class ExecIdlInteHolder.class ExecIdlInteOperations.class _ExecIdlInteStub.class

Legare Rmiliop - CORBA plasarea fisierelor la server si client

RMI IIOP	CORBA – ExecJavaIdl
Spatiul server	**Spatiul server**
ExecRmiInte.class ExecRmiliopImpl.class ExecRmiliopServ.class _ExecRmiliopImpl_Tie.class **_ExecIdlInteStub.class**	ExecIdlInte.class ExecIdlImpl.class ExecIdlServ.class ExecIdlInteHelper.class ExecIdlInteHolder.class ExecIdlInteOperations.class ExecIdlIntePOA.class _ExecIdlInteStub.class **_ExecRmiliopImpl_Tie.class**
Spatiul client	**Spatiul client**
ExecRmiInte.class ExecRmiliopClie.class _ExecRmiInte_Stub.class **_ExecIdlInteStub.class**	ExecIdlInte.class ExecIdlClie.class ExecIdlInteHelper.class ExecIdlInteHolder.class ExecIdlInteOperations.class _ExecIdlInteStub.class **_ExecRmiInte_Stub.class**

Fisierele ale caror nume sunt colorate, sunt preluate fiecare de la cealalta implementare, pentru a permite legarea Idl liop.

TITLU