Sistemas Distribuidos I
(75.74)

Arquitecturas Distribuidas Simples

Cliente-Servidor, Peer-to-Peer, RPC, Distributed Objects

Docentes

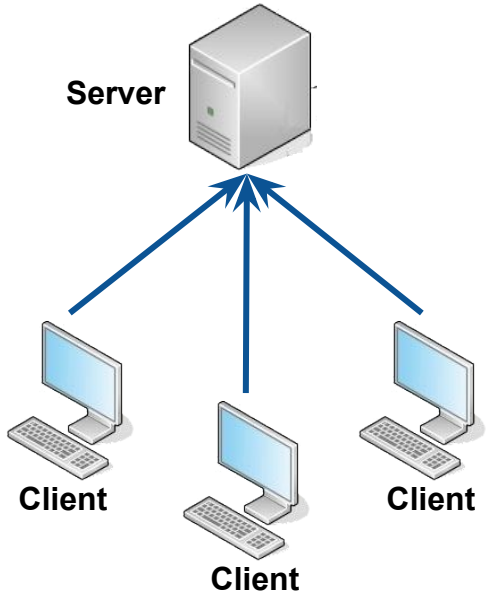
- Pablo D. Roca
- Ezequiel Torres Feyuk
- Ana Czarnitzki
- Cristian Raña

Agenda

- **Cliente-Servidor**
- Peer-to-Peer
- RPC
- Distributed Objects

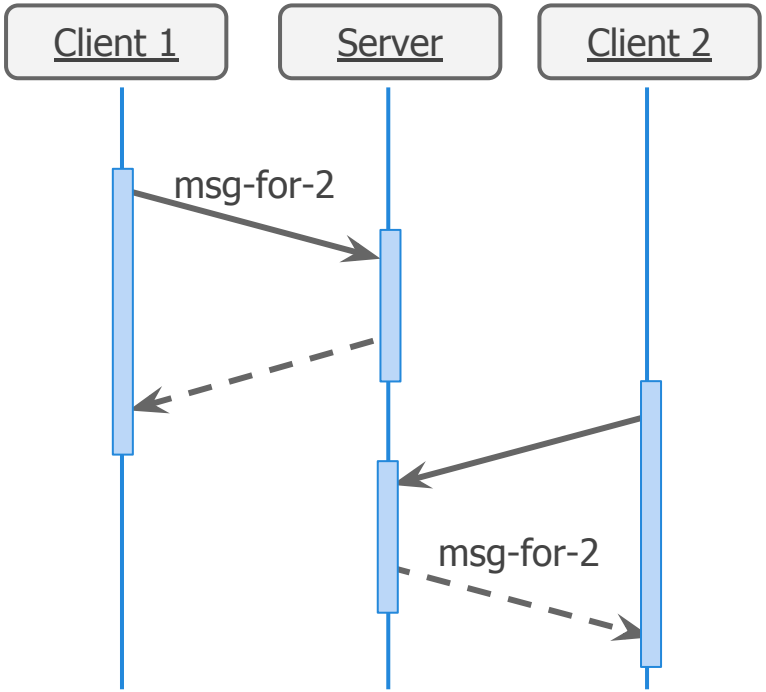
Cliente-Servidor

- Se definen roles para los participantes:
  - Servidor como elemento pasivo y provee servicios
  - Clientes activos que envían pedidos al servidor
- Permite centralización en toma de decisiones
- Suele asumirse que los servidores tienen más capacidades de *hardware* que los clientes



Cliente-Servidor | Flujos de Comunicación

- Los clientes deben conocer la ubicación del servidor para poder utilizarlo
- Los clientes no entablan comunicaciones entre sí, salvo a través del servidor
- Se pueden utilizar modelos de *callback* aunque no es su carácter natural:
  - *Long polling*
  - Push notifications

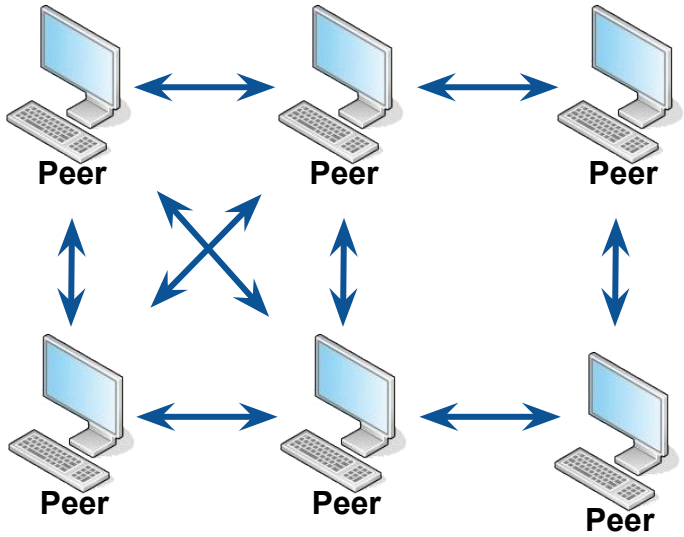


Agenda

- Cliente-Servidor
- **Peer-to-Peer**
- RPC
- Distributed Objects

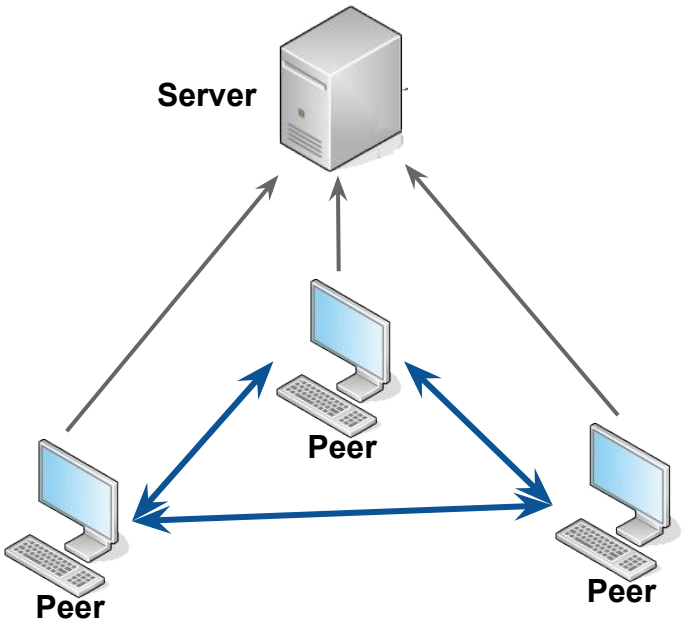
Peer to Peer

- Se establece una red de nodos que se consideran pares entre sí
- Asume capacidades de recursos similares entre los pares
- Muy útil cuando existen objetivos de colaboración por parte del negocio
  - Protocolo acordado entre partes
  - La lógica distribuida requiere coherencia entre los nodos
- Auge en internet a partir de la invención de Napster, BitTorrent, etc



Peer to Peer | Flujos de Comunicación

- Muy difícil de establecer la comunicación entre pares:
  - Esquema mixto como cliente-servidor para proveer un servicio de nombres
  - Grupo de comunicación donde se comparten dirección de miembros
- Requieren mayores permisos de *networking* (reglas de *firewall* de entrada, rangos de puertos, etc)

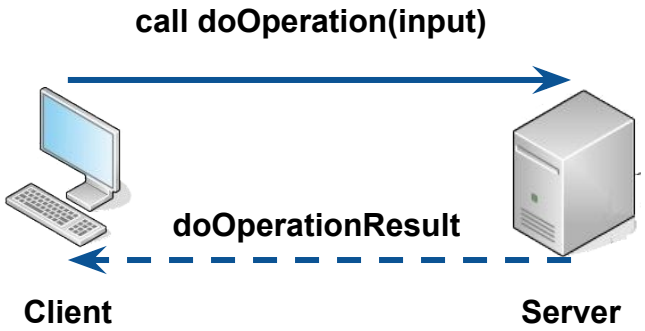


Agenda

- Cliente-Servidor
- Peer-to-Peer
- **RPC**
- Distributed Objects

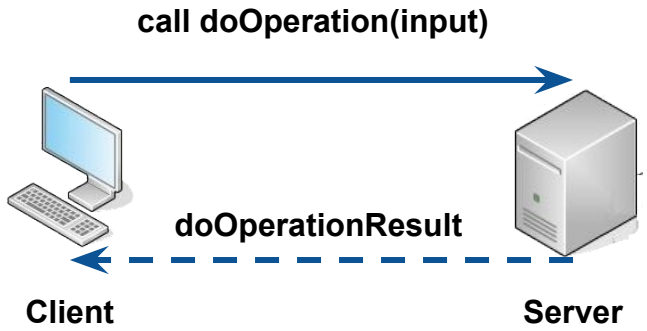
Remote Procedure Call (RPC)

- Ejecución remota de procedimientos
- Modelo Cliente-Servidor
  - Cliente realiza una llamada a un Procedimiento
  - Servidor responde con el resultado de la operación
- Comunicación remota transparente para el usuario
- Portabilidad a través de implementación de interfaces bien definidas



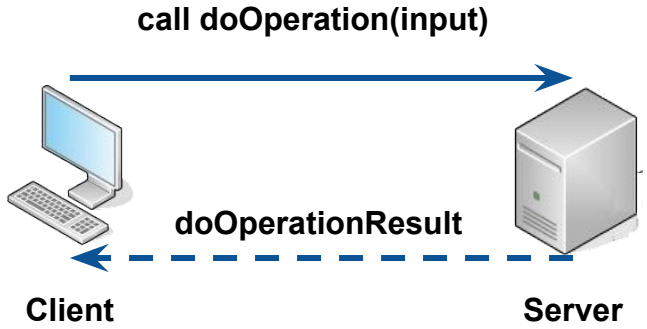
RPC | IDL (*Interface Domain Language*)

- Diseñados para permitir que diferentes lenguajes puedan invocarse entre sí
- Interfaz definida en función de datos de entrada (Input) y datos de salida (output)
  - Acceso a métodos permitido
  - Pasaje de variables por valor
  - Punteros no permitidos
- Definición de tipos de mensajes a enviar como para del IDL
- Ejemplo: google protocol buffers



RPC | Tolerancia a Fallos

- A diferencia de Local Procedure Calls (LPCs), un procedimiento puede o no ser ejecutado
- Diferentes estrategias para garantizar Delivery de mensajes:
  - *Request-Retry* con *Timeout*
  - Filtrado de operaciones duplicadas
  - Retransmisión / Re-ejecución de operación si se pierde *retry*



RPC | *Call Semantics*

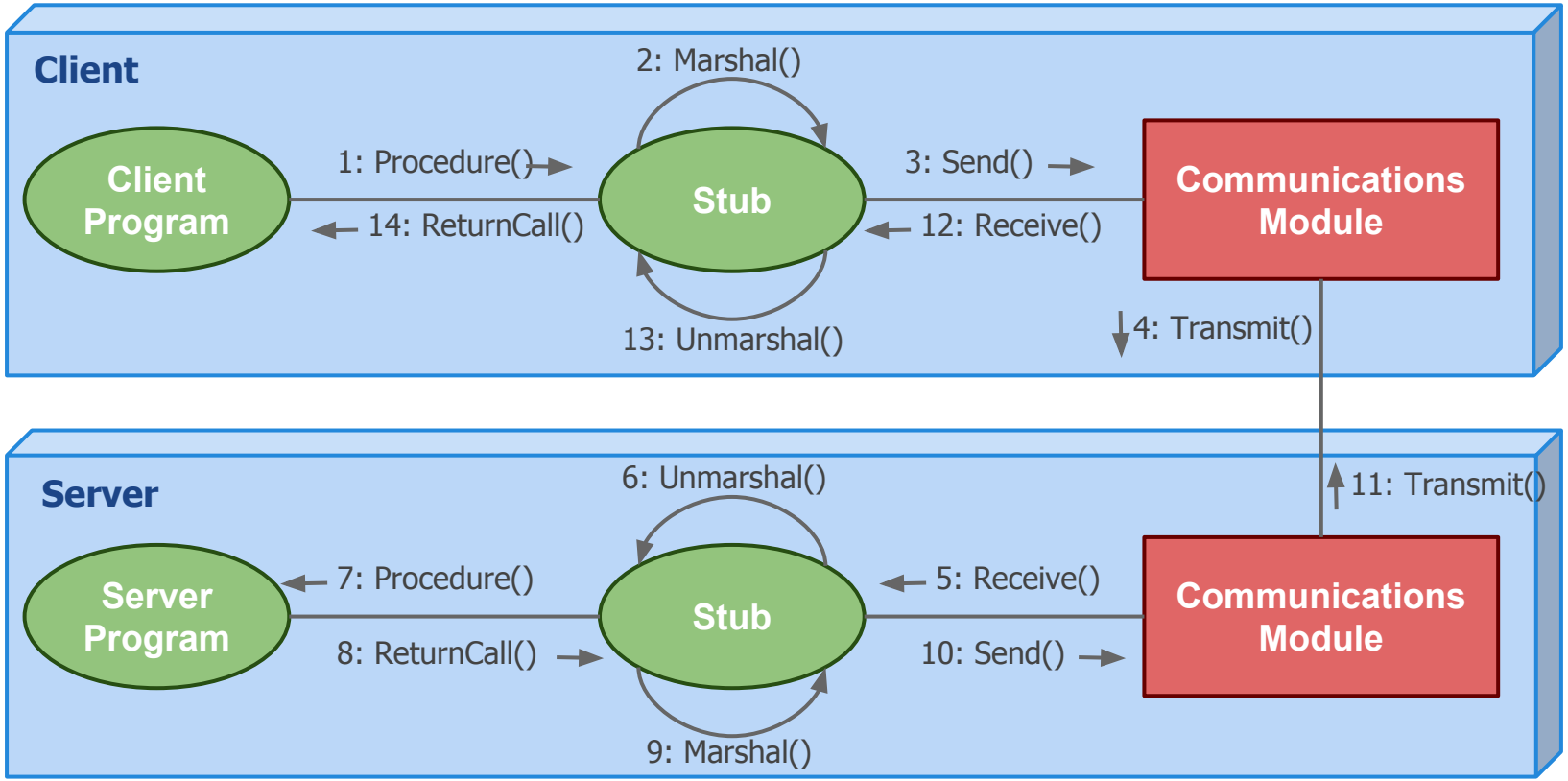
Según las estrategias adoptadas para asegurar el *delivery* de mensajes, los mensajes pueden llegar a ser recibidos 0, 1 o muchas veces:

Estrategia	¿Mensaje recibido?	Tipo de Control	Retry - Request	Filtro Duplicados
#1	***Maybe***	Sin control	No	No implementable
#2	***At Least Once***	Re-ejecución	Si	No
#3	***Exactly Once***	Retransmisión	Si	Si

RPC | Implementación

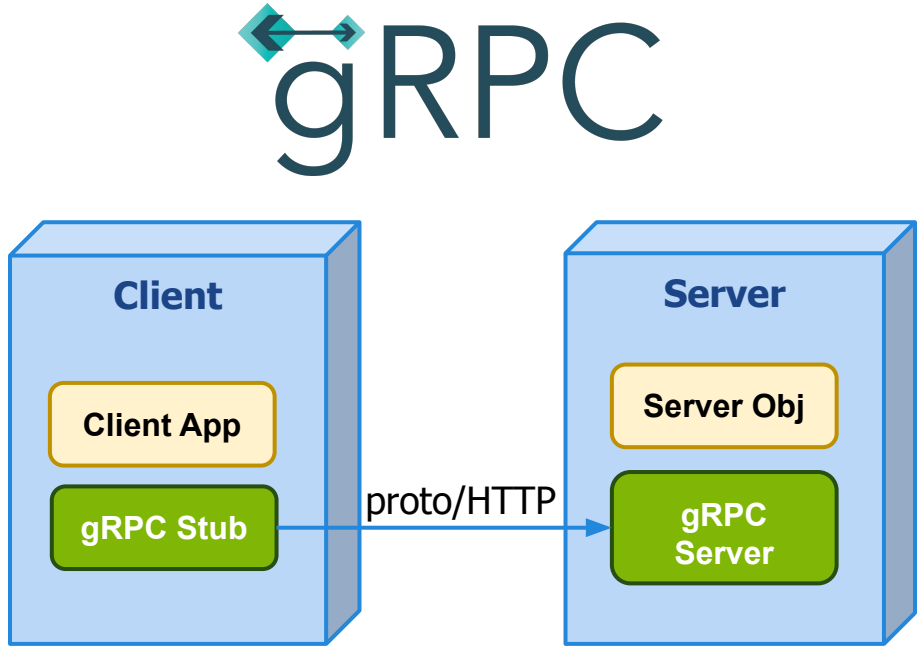
- **Cliente**
  - Se encuentra conectado a un *stub*
  - Realiza llamadas de forma transparente al servidor (o no tanto)
- **Servidor**
  - Se encuentra conectado a un *stub* del cual recibe parámetros
  - Posee lógica particular del *remote procedure*
- **Stubs**
  - Administra el *marshalling* de la información
  - Envía información de llamadas (***calls***) al módulo de comunicación y al cliente / servidor
- **Módulo de comunicación**
  - Abstrae al *stub* de la comunicación con el servidor

RPC | Implementación



gRPC

- Definición de RPC basada en:
  - HTTP2 para transporte
  - Protocol Buffers para encoding
  - Conexión punto a punto basada en server:port
- Definición de Servicios y Mensajes en archivos .proto
- Generación de código en distintos lenguajes
- Diseñado para alta performance y microservicios

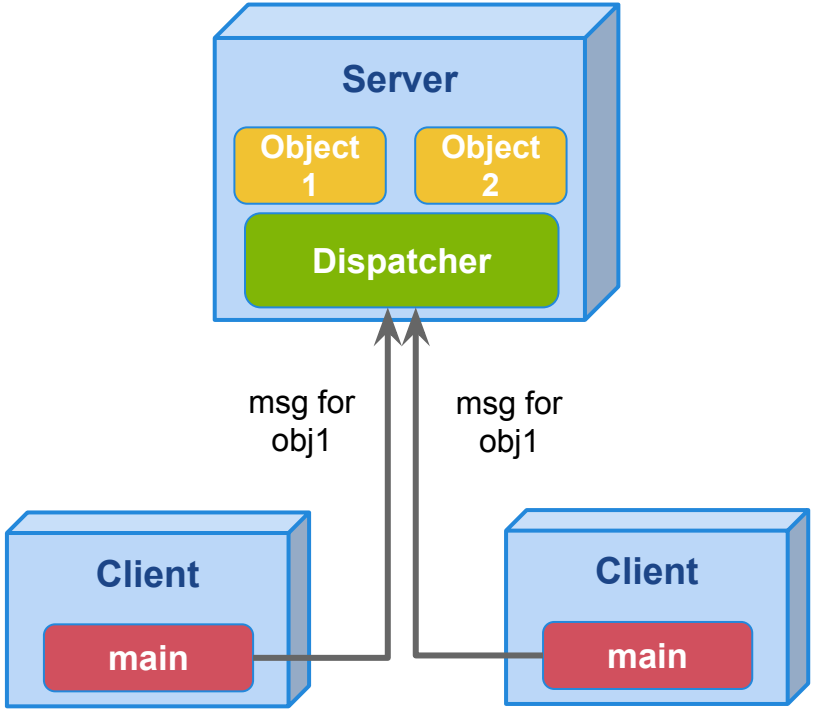


Agenda

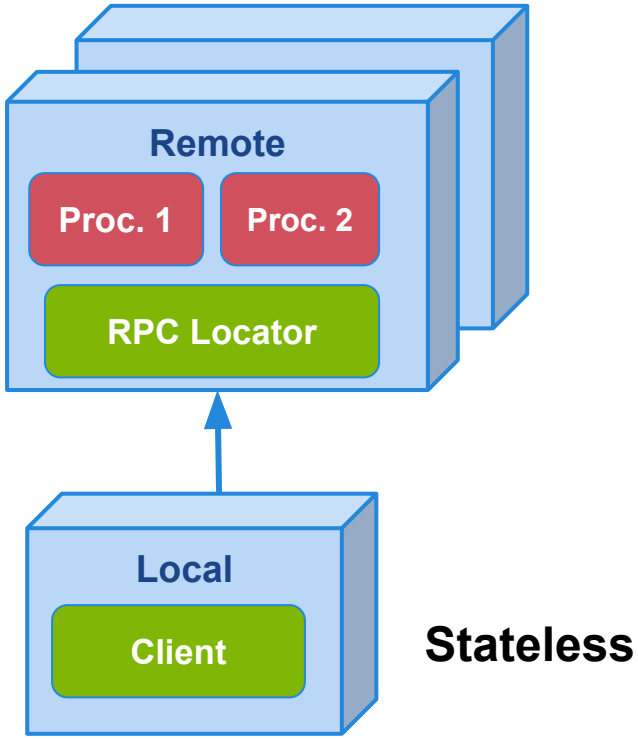
- Cliente-Servidor
- Peer-to-Peer
- RPC
- **Distributed Objects**

Objetos Distribuídos

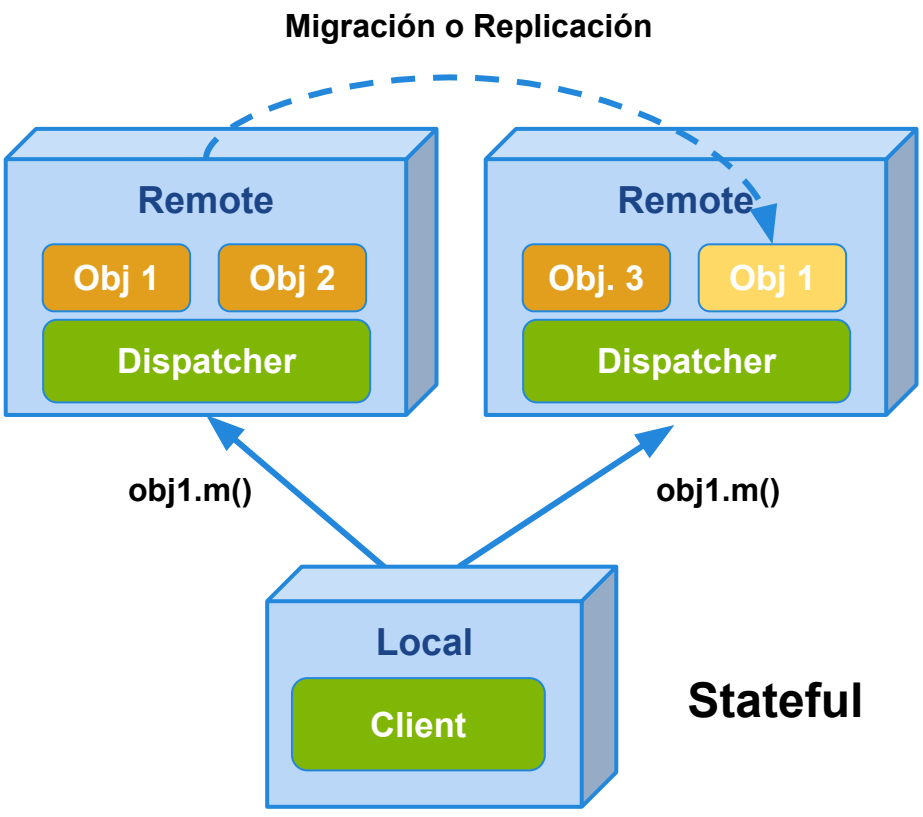
- Los servidores ya no proveen servicios sino objetos
- Existe un middleware que oculta la complejidad de:
  - Referencias a Obj. remotos
  - Invocación de acciones
  - Errores (excepciones)
  - Recolección de basura



RPC vs Objetos Distribuidos



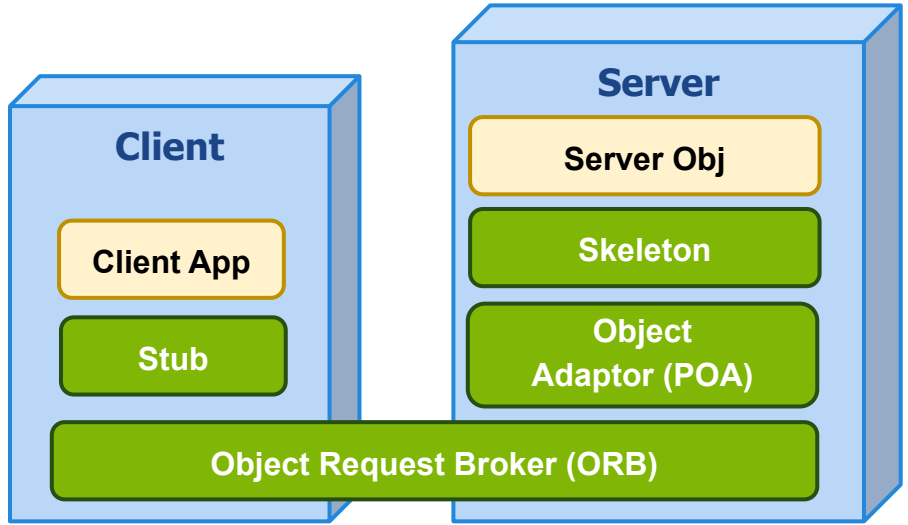
Stateless



Stateful

CORBA

- Estandar definido por comité
- Soporte en múltiples lenguajes
- En vías de deprecación
- Provee:
  - Protocolo y serialización
  - Transporte
  - Seguridad
  - *Discovery* de Objetos



CORBA | *Interface Definition Language* (IDL)

BankAccount.idl

```
module distribuidos {
    interface Money {
        double getValue();
        string getCurrency();
        void setValue(in double v);
        void setCurrency(in string c);
    };

    interface BankAccount {
        void add(in Money m);
        void substract(in Money m);
        Money getBalance();
    };
};
```

```
$ idlj BankAccount.idl
```

- MoneyOperations.java
- Money.java
- MoneyHelper.java
- MoneyHolder.java
- _MoneyStub.java
- BankAccountOperations.java
- BankAccount.java
- BankAccountHelper.java
- BankAccountHolder.java
- _BankAccountStub.java

```
$ idlj BankAccount.idl -fserver
```

- MoneyOperations.java
- Money.java
- MoneyPOA.java
- BankAccountOperations.java
- BankAccount.java
- BankAccountPOA.java

CORBA | Lado Cliente (autogeneradas)

BankAccountOperations.java

```
public interface BankAccountOperations
{
  void add(Money m);
  void substract(Money m);
  Money getBalance();
}
```

BankAccountHelper.java

```
abstract public class BankAccountHelper
{
  public static BankAccount extract(Any a)
  {
    return read(a.create_input_stream());
  }
  public static void insert(Any a,
                            BankAccount that)
  { ... }
}
```

BankAccount.java

```
public interface BankAccount extends
     BankAccountOperations,CORBA.Object
...
{ }
```

_BankAccountStub.java

```
public class _BankAccountStub extends
org.omg.CORBA... implements BankAccount
{
  public void add(Money m) {
   try {
     OutputStream $out = _request("add",..
     MoneyHelper.write($out, m);
     InputStream $in = _invoke($out);
   } catch (ApplicException $ex) {...
   } finally {
     _releaseReply($in);
  }
}
```

CORBA | Lado Cliente (Invocación)

ClientApp.java

```
public class ClientApp {
  public static void main(String argv[]) {
    try {
      ORB orb = ORB.init(argv, null);
      CORBA.Object obj = orb.resolve_initial_references("NameService");
      NamingContext nc = NamingContextHelper.narrow(obj);
      NameComponent path[] = { new NameComponent("BankAccount", "") };
      CORBA.Object sobj = nc.resolve(path);
      BankAccount account = BankAccountHelper.narrow(sobj);
      Money m = new _MoneyStub();
      m.setValue(173.39); m.setCurrency("Pesos");
      account.add(new MoneyHolder(m));
      Money balance = account.getBalance();
      System.out.println("Balance = " + balance.getValue());
    }
    catch (Exception e) { ... }
  }
```

CORBA | Lado Servidor (autogeneradas)

BankAccountPOA.java

```
public abstract class BankAccountPOA extends Servant implements BankAccountOperations {
  private static java.util.Hashtable _methods = new java.util.Hashtable ();
  static {
    _methods.put("add", new Integer(0)); _methods.put("substract", new Integer(2));
    _methods.put("getBalance", new Integer(3));
  }
  public OutputStream _invoke(String $method, InputStream in, ResponseHandler $rh)
  {
    Integer __method = (Integer)_methods.get($method);
    if (__method == null)
      throw new BAD_OPERATION (0, CompletionStatus.COMPLETED_MAYBE);
    switch (__method.intValue()){
       case 0: // distribuidos/BankAccount/add {
         Money m = MoneyHelper.read(in);
         this.add (m);
         OutputStream out = $rh.createReply();
        break; }
       case 1: // distribuidos/BankAccount/substract { ...
}}
```

CORBA | Lado Servidor (Invocación)

BankAccountImpl.java

```
public class BankAccountImpl extends BankAccountPOA {
  public void add(Money m) { ... }
  public void substract(Money m) { ... }
  public Money getBalance() { ... }

  public static void main(String argv[]) {
    try {
      ORB orb = ORB.init(argv, null);
      orb.connect(new BankAccountImpl());
      CORBA.Object ncObj = orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(ncObj);
      NameComponent path[] = { new NameComponent("BankAccount", "") };
      ncRef.rebind(path, solver);
      java.lang.Object dummySync =new java.lang.Object();
      synchronized (dummySync) { dummySync.wait(); } // Wait for clients
    } catch(Exception e) { ... }
  }}
```

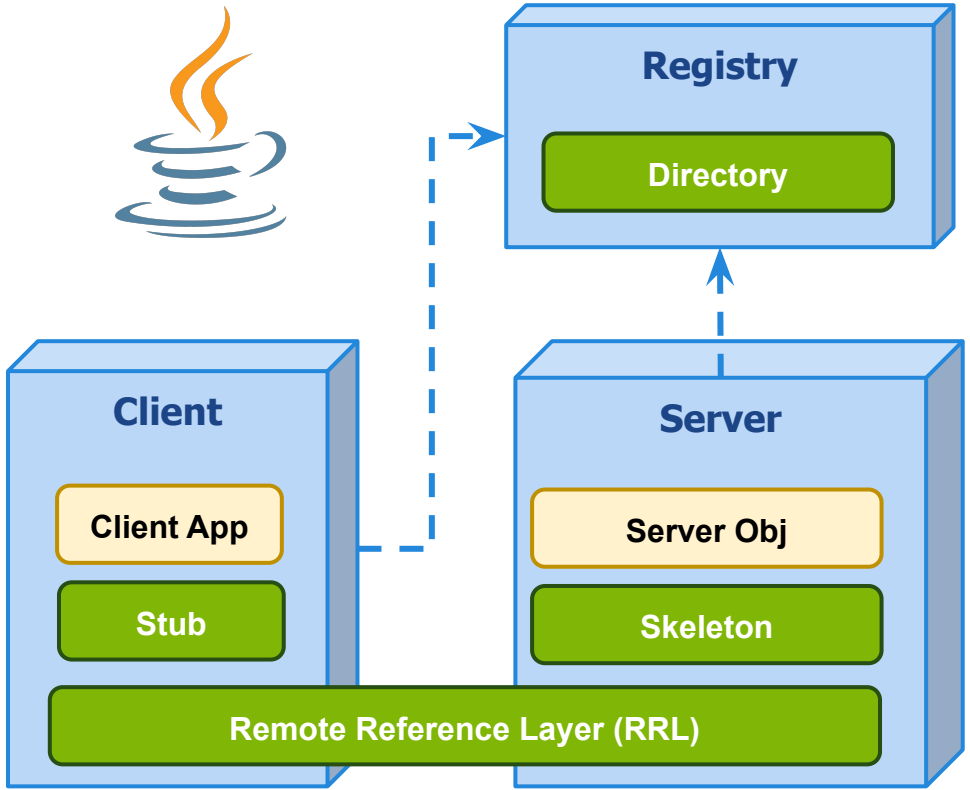
RMI

- Versión optimizada de Distributed Objects en Java
- Requiere los siguientes pasos:

  1- Registro del servidor en un directorio de servicios.

  2- Consulta del registro por parte del cliente

  3- Invocación desde el cliente al servidor



RMI | Interfaz Cliente-Servidor

Money.java

```
package distribuidos;
import java.rmi.Remote;
...
public interface Money extends Remote {
  double getValue() throws RemoteException;
  String getCurrency() throws RemoteException;
  void setValue(double v) throws RemoteException;
  void setCurrency(String c) throws RemoteException;
}
```

BankAccount.java

```
package distribuidos;
...
public interface BankAccount extends Remote {
  void add(Money m) throws RemoteException;
  void substract (Money m) throws RemoteException;
  Money getBalance() throws RemoteException;
}
```

RMI | Implementación Servidor

MoneyImpl.java

```
package distribuidos;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class MoneyImpl extends UnicastRemoteObject implements Money {
  private double value;
  private String currency;
  public MoneyImpl() throws RemoteException {
    this.value = 0;
    this.currency = "Pesos";
  }
  public double getValue () throws RemoteException {
    return this.value;
  }
  public String getCurrency() throws RemoteException {
    return this.currency;
  }
  ...
}
```

RMI | Implementación Servidor (II)

BankAccountImpl.java

```
...
public class BankAccountImpl extends UnicastRemoteObject implements BankAccount {
  private Money balance;
  public BankAccountImpl(double initialBalance) throws RemoteException {
    this.balance = new MoneyImpl(initialBalance, "Pesos");
  }
  public void add(Money m) throws RemoteException {
    String currency = this.balance.getCurrency();
    if (!m.getCurrency().equals(currency))
      throw new RemoteException("Invalid currency type");
    double value = this.balance.getValue() + m.getValue();
    this.balance = new MoneyImpl(value, currency);
  }
  ...
  public Money getBalance() throws RemoteException {
    return new MoneyImpl(this.balance.getValue(), this.balance.getCurrency());
  }
}
```

RMI | Cliente - Servidor

ServerApp.java

```
import distribuidos.BankAccountImpl;
import java.rmi.Naming;

public class ServerApp {
  public static void main(String[] args){
    try {
      Naming.rebind(
         "//localhost/BankAccount",
         new BankAccountImpl(100));
      System.out.println("Server ready");
    } catch (Exception e) {
      System.err.println("Error:" + e);
    }
  }
}
```

ClientApp.java

```
import distribuidos.BankAccount;
import distribuidos.Money;
import java.rmi.Naming;
public class ClientApp {
  public static void main(String[] args) {
    try {
      BankAccount a = (BankAccount)Naming
        .lookup("//localhost/BankAccount");
      Money m0 = a.getBalance();
      a.add(m0);
      Money m1 = a.getBalance();
      System.out.println("Original:" +
        m0.getValue() + " - New:" +
        m1.getValue());
    } catch (Exception e) { ... }
  }
}
```

Bibliografía

- G. Coulouris, J. Dollimore, t. Kindberg, G. Blair: Distributed Systems. Concepts and Design, 5th Edition, Addison Wesley, 2012.
  - Capítulo 5: Remote Invocation
  - Capítulo 8: Distributed Objects and Components
- P. Verissimo, L. Rodriguez: Distributed Systems for Systems Architects, Kluwer Academic Publishers, 2001.
  - Capítulo 3.6: Client Server with RPC
  - Capítulo 4.4: Object Oriented Environments (CORBA)
- Ejemplos de código:
  - https://github.com/7574-sistemas-distribuidos/grpc-example
  - https://github.com/7574-sistemas-distribuidos/rmi-example