Capítulo III:
Comunicación en Programación distribuida
Modelos, mecanismos y protocolos de comunicación

Prof. Dr.-Ing. Raúl Monge Anwandter ♦ 2º semestre 2025

Objetivos del capítulo:

Objetivo general:

- Comprender los aspectos de comunicación en la programación y desarrollo de sistemas de software distribuidos.

Objetivos de aprendizaje:

1. Explica los principales modelos de programación distribuida y diferentes mecanismos de comunicación entre programas distribuidos.
2. Aplica *sockets* con protocolos de transporte de datos en TCP/IP para comunicación entre procesos distribuidos.
3. Comprende los problemas y los métodos para intercambio de datos en ambientes heterogéneos de programación distribuida.
4. Aplica servicios de comunicación de *Middleware* del tipo invocación remota y mensajería asincrónica, para el desarrollo de software distribuido.

Organización del capítulo:

1. Modelos de programación distribuida y comunicación
2. Comunicación entre procesos mediante *sockets*
3. Representación e intercambio de datos en ambientes distribuidos
4. Invocación remota (MW)
5. Servicios Web (MW)
6. Servicios de mensajería (MW)

UNIVERSIDAD TECNICA FEDERICO SANTA MARIA | DEPARTAMENTO DE INFORMÁTICA

3.1 Modelos de programación distribuida y comunicación

Programación distribuida

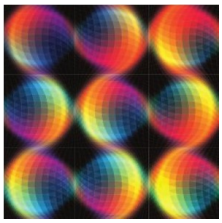
Desarrollo de software en Sistemas Distribuidos

DEFINICIÓN: Paradigma de desarrollo de software, donde múltiples programas o componentes independientes se ejecutan en diferentes computadores o nodos conectados en red, comunicándose y coordinándose entre sí para lograr algún objetivo común.

- Enfoque fundamental para desarrollar sistemas distribuidos, dividiéndose las tareas entre diferentes nodos para escalar, ser resiliente, tolerar fallos y/o compartir recursos.

Conceptos claves:

- Mecanismos y protocolos de comunicación (en este capítulo)
- Coordinación y sincronización (profundización en capítulos 4 y 5)
- Tolerancia a fallos (profundización en capítulo 6)
- Consistencia de datos (profundización en capítulo 7 y 8)



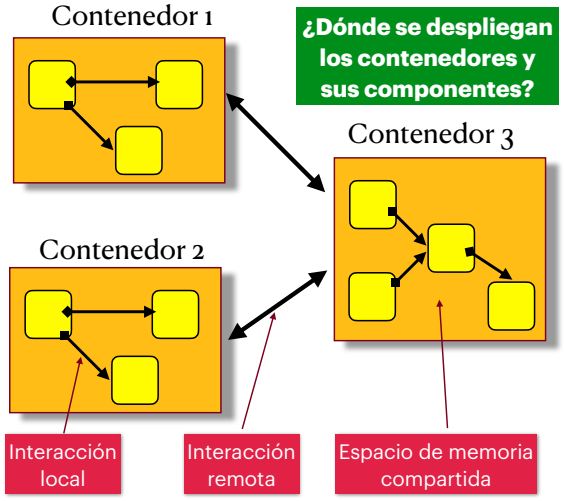
Programación concurrente

Programación concurrente: Modelo de programación de software basado en el diseño e implementación de múltiples tareas o procesos, que se traslapan en su ejecución y pueden interactuar entre sí, aun cuando no corran estrictamente en el mismo tiempo físico.

	Programación distribuida:	**Programación paralela:**
Estructura de procesamiento	Centrada en programas que se ejecutan en múltiples máquinas independientes conectadas en red (i.e., sistemas débilmente acoplados).	Centrada en ejecución simultánea de tareas en varios procesadores o núcleos dentro de una misma máquina (i.e., sistemas estrechamente acoplados).
Arquitectura típica	Cada máquina tiene su(s) propio(s) procesador(es) y memoria independiente, máquinas que se comunican en red por paso de mensajes.	Los procesadores comparten memoria o tienen acceso de muy baja latencia a la memoria de los demás.
Casos de uso	EIS, Sistemas de telemetría y control, DDBS, arquitectura de microservicios, sistemas P2P.	Computación científica (HPC); procesamiento de peticiones, eventos y grandes volúmenes de datos.
Desafíos	Fallas, latencia y partición de red; consistencia de datos; escalabilidad y coordinación.	Sincronización entre hebras/ procesos; uso eficiente de memoria compartida consistencia de *cache;* comunicación eficiente y sin contención.

Componentes y contenedores

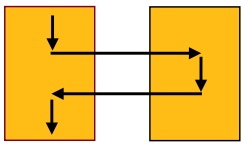
Diferentes niveles de abstracción y modelos de programación distribuida



Consideraciones de diseño:

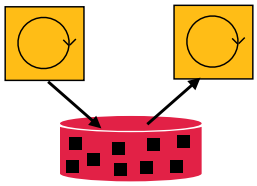
- Tipos de componentes, contenedores y conectores
- Localización y descubrimiento de componentes (co-localización & remoto)
- Movilidad y reubicación de componentes (para escalamiento, balance de carga, resiliencia y/o mantención)
- Lenguajes de programación y soporte de *runtime*
- Herramientas y servicios de apoyo para desarrollo de software (e.g. *Middleware*)
- Escalabilidad, fiabilidad, consistencia y seguridad.

Paradigmas de comunicación



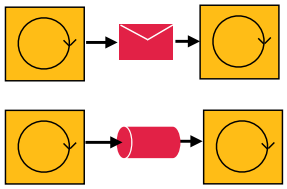
a) Control

- Comunicación directa por paso de parámetros y resultados (*piggybacking*).
- Enfoque de servicio que encapsula funcionalidad, permite anidamiento y otras variantes.
- *Multithreading* requiere control de concurrencia si existe estado.
- Usado en invocaciones remotas (RPC, RMI y Servicios Web)



b) Memoria compartida

- Comunicación indirecta usando un espacio de almacenamiento de datos compartido.
- Requiere control de concurrencia para garantizar consistencia de datos.
- Usado en memoria compartida distribuida y repositorio de datos (archivos y sistemas de BD).



c) Mensajes

- Dos estilos básicos: Paso de mensajes y *data streaming*.
- Se extiende a multiples destinatarios (e.g. *multicasting)*.
- Comunicación puede directa cuando no existe persistencia (e.g. *sockets* TCP/IP))
- Comunicación indirecta con sistemas de mensajería y de notificación de eventos.

Protocolos de redes de comunicación

Arquitecturas de referencia y caracterización de protocolos

Calidad de Servicio (QoS):

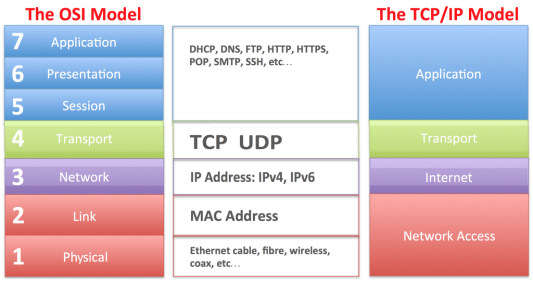
- **Desempeño**: caracterizado por *throughput*, latencia, *jitter*, tiempo de respuesta, etc.
- **Control de flujo:** regulación del flujo de mensajes.
- **Ordenamiento:** orden de entrega de mensajes.
- **Fiabilidad:** garantía de entrega de mensajes.
- **Durabilidad:** mensajes transientes o persistentes.
- **Seguridad :** autenticación de entidades y control de confidencialidad, integridad de mensajes.

Argumento end-to-end (Saltzer, 1984): *"Las funciones deben situarse en la capa más alta donde pueden aplicarse plenamente, en lugar de capas inferiores donde podrían ser redundantes o insuficientes."*

Fuente: J. H. Saltzer, et al.(1984). "End-to-end arguments in system design". *ACM ToCS* 2(4), Nov. 1984, 277–288.

Gestión de la comunicación:

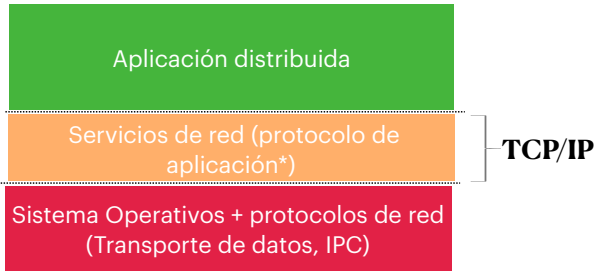
- ***Endpoints:*** identificación/dirección (e.g. *socket*, servicio).
- **Coordinación.** conexiones, sesiones y estado.
- **Enlaces de comunicación:** reserva de recursos.
- **Memoria:** *buffering*, asentimientos, colas.



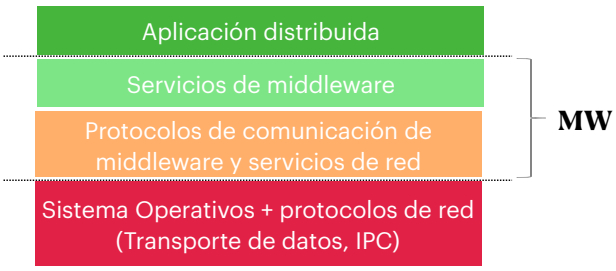
Protocolos de comunicación para Programación distribuida

Arquitecturas de TCP/IP vs. Middleware

- **Protocolos de transporte de datos (IPC)**
  - Paso de mensaje (UDP, RTP)
  - Flujo de datos (TCP, SCTP)
- **Protocolos de servicios de aplicación**
  - Telnet, FTP, SMTP DNS, HTTP



- **Servicios de comunicación de *middleware***
  - Invocación remota (RPC, RMI, WS)
  - Sistemas de Mensajería (Colas, MoM)
  - Multimedios (*data streaming*)
  - Memoria compartida (DSM, espacios de tuplas)
  - Computación paralela (OpenMP, MPI)



Comunicación basada en mensajes

Caracterización de la comunicación

Clasificación	Tipos/propósito	Ejemplos
Patrones de flujo de mensajes	**One-to-one** (*Unicast*, *Point-to-Point*); **One-to-many** (*Broadcast*, *Multicast*, *Anycast*, Pub-Sub)	TCP (*unicast*); UDP (*unicast* y *multicast*).
Dirección de la comunicación	**Unidireccional** o **bidireccional** (simultánea; o con inversión o respuesta)	UDP es unidireccional; TCP es bidireccional.
Sincronismo	**Sincrónico** (bloqueante al esperar ACK del receptor) **Asincrónico** (no bloqueante, más rápido)	UDP+TCP (asincrónico); MoM (sincrónico y asincrónico).
Ordenamiento de mensajes	**Sin ordenamiento**; **ordenamiento FIFO.** En grupos: **orden parcial** (e.g. causal) u **orden total**	UDP (sin ordenamiento); TCP (FIFO)
Control de flujo	Regula flujo de mensajes; previene rebasar (*overrun*) al receptor y evitar pérdida de mensajes.	UDP no controla flujo y TCP si lo hace
Garantía de entrega (fiabilidad)	**Maybe** (*best-effort*), **At-least-once**, **At-most-once**, **Exactly-once**.	UDP (maybe), gRPC (At-most-once), Exactly-once (MoM con transacciones)
Persistencia o durabilidad	**Transiente** (mensajes almacenados en *buffers*: volátil) **Persistente** (uso de medios de almacenamiento persistente)	UDP y TCP (transientes); e-mail y MoM (persistentes)
Seguridad	Soporte para autenticación de entidades, cifrado de datos y control de integridad en la comunicación.	SSL/TLS integra seguridad en TCP.

Sincronización de mensajes

Paso de mensaje unidireccional y bidireccional (Solicitud-Respuesta)

- **Sincronización en el iniciador (emisor o cliente)**
  - Paso de mensaje (unidireccional)
  - *Request-Reply* (bidireccional)

- **Sincronización en el receptor (receptor o servidor)**
  - Recepción explícita (*polling* / *pull*) o implícita (notificación - *push*)
  - Recepción selectiva (e.g. origen o contenido) o priorizada
  - Recepción con límite de tiempo (uso de *timer* para evitar bloqueo permanente)

Sincronismo del emisor en paso de mensaje

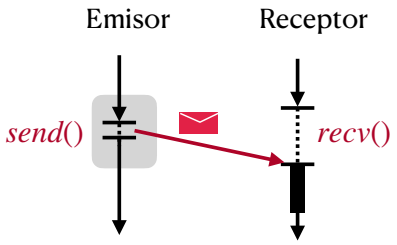
Emisor:

```
char msg[MAX];
int largo;
...
Preparar mensaje en msg
  de tamaño largo;
send(dest, msg, largo);
Hacer otra cosa...
...
```

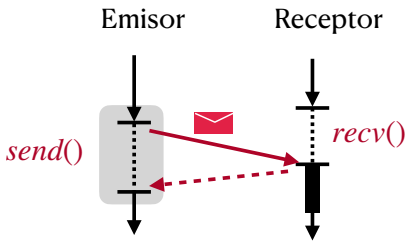
Receptor:

```
char msg[MAX];
int largo;
...
Preparar recepción.
recv(orig, msg, &largo);
procesar mensaje msg de
  tamaño largo;
...
```

a) Paso de mensaje asincrónico
(emisor no bloquea)



b) Paso de mensaje sincrónico
(emisor bloquea)



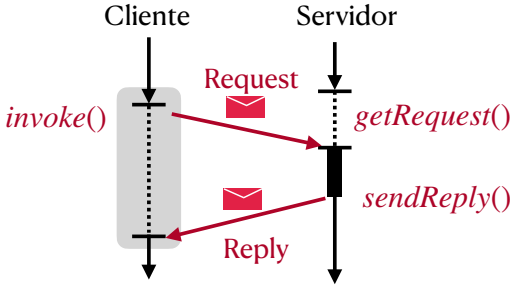
OBSERVACIONES:

- Paso de mensaje sincrónico requiere implícitamente ACK para desbloquear al Emisor.
- Receptor independiente con recepción explícita y sincrónica (bloqueante).

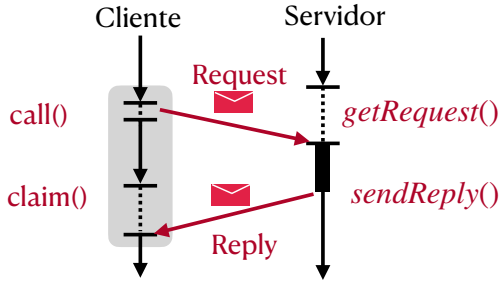
Sincronismo del invocador (cliente)

Protocolo básico tipo *Request-Reply*

a) Invocación sincrónica



b) Invocación asincrónica

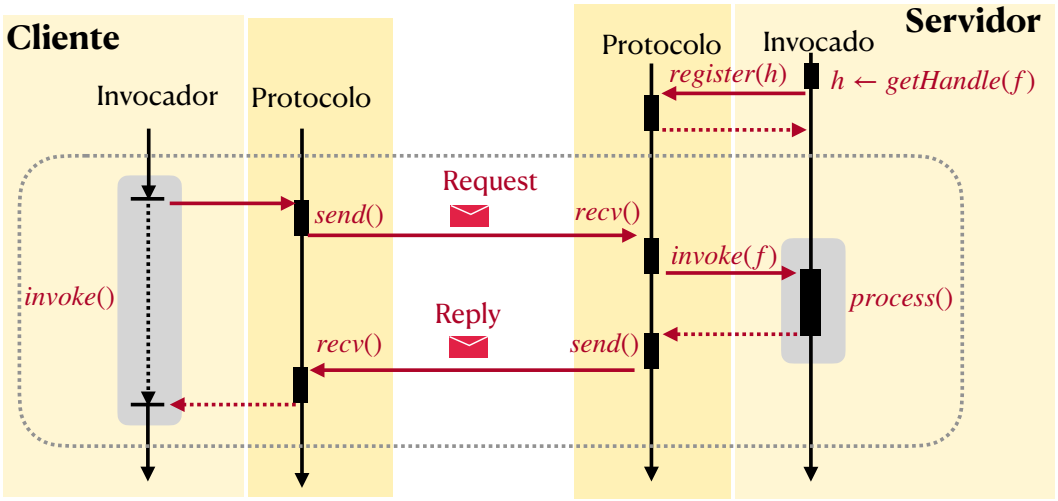


OBSERVACIONES:

- Este modelo de interacción se asocia a un **protocolo *Request-Reply*** (que podría estar implementado en el núcleo del SO o a nivel de proceso como sucede en protocolos de *Middleware*).
- Con *Multithreading*, el sincronismo no limita la concurrencia (i.e. no es necesario invocación asincrónica).
- En el ejemplo, recepción es sincrónica/bloqueante y explícita (pero podría ser diferente).

Recepción implícita (*push*)

Requiere registro previo y explicito en el soporte del protocolo (*upcall*)



Recepción implícita con Java RMI

Para el programador resulta transparente el uso de mensajes

```
public interface FileInterface extends Remote {
    public byte[] downloadFile(String fileName) throws RemoteException;
}
```

Lo que conoce el cliente:
Interfaz del servidor (i.e. un objeto remoto) .

```
public class FileImpl extends UnicastRemoteObject
                      implements FileInterface {
    private String name;
    public FileImpl(String s) throws RemoteException{
    …
    }

    public byte[] downloadFile(String fileName) throws RemoteException {
    ….
    }
}
```

Programación en el servidor:
La implementación de la interfaz en el servidor.

OBSERVACIÓN:

- **Recepción implícita** aplica también para recepción asincrónica en paso de mensaje unidireccional y notificación de eventos (tipo *push*).

Multicasting: comunicación grupal

DEFINICIONES:

- ***Multicasting***. Método de comunicación donde un emisor transmite un mensaje a un grupo de receptores simultáneamente, sin enviar copias separadas a cada uno. Corresponde a un patrón de comunicación *one-to-many*.
- ***Grupo***. Abstracción que agrupa a varios componentes (que poseen una propiedad común de la aplicación) en una unidad lógica, para una comunicación transparente con la agrupación de tipo *one-to-many*.

Propiedades útiles de aplicación de *Multicasting*:

- Grupo abstrae una unidad de interacción (útil para estructuración lógica de sistemas).
- Distribución y recolección de datos eficiente y escalable (i.e., uso de menor BW con menor costo de coordinación).
- Distribución de carga y procesamiento paralelo (e.g. grupo de trabajadores o *workers*).
- Descubrimiento y localización de recursos (e.g. ARP, contactar un determinado servidor en un *cluster*).
- Útil para tolerancia a fallos y proveer alta disponibilidad (e.g. usando grupos de replicación)

Mayores desafíos:

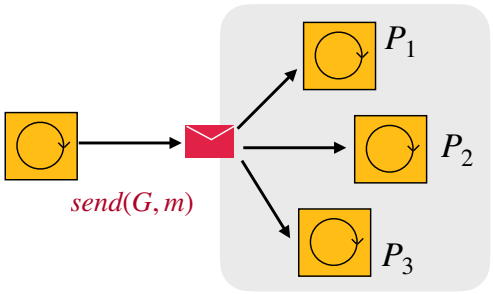
- Escalabilidad, rendimiento y soporte en grandes redes
- Fiabilidad y consistencia
- Seguridad

Aplicaciones de *Multicasting*

Ejemplos de casos de uso en Sistemas Distribuidos

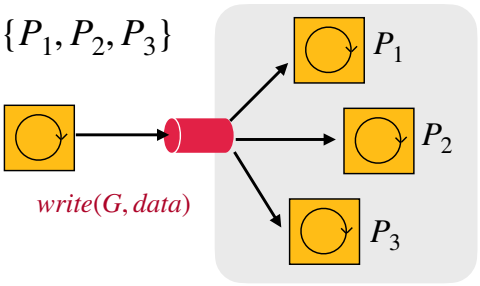
- **Comunicación de grupo** (e.g., sistemas de archivos distribuidos, bases de datos distribuidas, aplicaciones colaborativas).
- **Mensajería Pub-Sub** (e.g., arquitecturas basadas en eventos, datos bursátiles).
- **Streaming de vídeo/audio en tiempo real** (e.g., IPTV, videoconferencias).
- **Protocolos de consenso** (e.g., compromiso o acuerdo distribuido; Paxos, Raft).
- **Intercambio distribuido de estados** (e.g., sistemas de multijugador, analítica de datos en tiempo real).

Comunicación grupal (*Multicasting*)



a) Multicast de paso de mensaje (datagrama)

Grupo: $G = \{P_1, P_2, P_3\}$



b) Multicast con *data streaming*

Gestión de grupos:

- Deseable que G esté en el mismo espacio de direcciones de un *unicast*, para hacer transparente existencia de grupo.
- Mecanismos para unirse y dejar un grupo (*join* & *leave*)
- Para mayor QoS, se requiere conocer miembros del grupo.

Implementación de *Multicasting*:

1. Servicio básico de red (e.g. datagramas IP), con apoyo de redes de difusión (*broadcasting*) subyacentes.
2. Red sobrepuesta (*overlay network*).
3. Un servicio de *Middleware*.

Semántica de *Multicasting*

Entrega de mensajes y roles de miembros de grupo

SEMÁNTICA DE ENTREGA DE MENSAJES

- **Semántica de entrega (de ida*):**
  - Fiabilidad (*best-effort*, k-fiable, atómica)
  - Ordenamiento de mensajes (sin compromiso, FIFO, causal, total, etc.)
- **Semántica de respuesta*** (si existe, pues es menos frecuente):
  - Fiabilidad (e.g., k-respuestas)
  - Sincronización en el emisor (e.g., con la primera o última respuesta)

SEMÁNTICA DE GRUPO

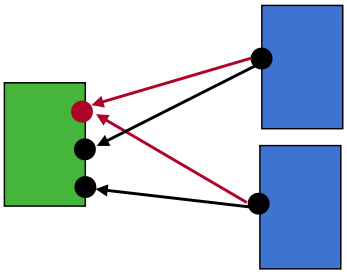
- **Clausura:**
  - **Grupos cerrados**: Mensajes sólo pueden provenir de miembros del mismo grupo (e.g., grupos cooperativos).
  - **Grupos abiertos**: El grupo puede recibir mensajes externos (e.g., un cliente de un grupo de servidores).
- **Simetría:**
  - **Grupos simétricos**: Miembros del grupo son pares (*peers*); control está distribuido.
  - **Grupos asimétricos**: Grupo tiene un coordinador o líder para contacto externo (e.g. coordina distribución interna de mensajes o respuestas).
    - Tolerancia a fallos requiere elección de líder.

Multicasting en TCP/IP

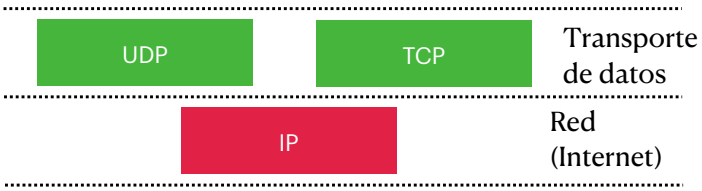
Dos niveles de abstracción a nivel de IPC

- **Protocolo de multicast de red** (nivel de capa de red e *internetworking*):
  - Uso de *multicast* UDP y dirección IP de *multicast* (e.g. 224.0.0.0 – 239.255.255.255).
  - Requiere de protocolos enrutamiento de *multicast* (e.g., PIM, DVMRP).
- **Multicast al nivel de aplicación o *Middleware***:
  - Implementado en la capa de aplicación (e.g., redes P2P, CDN).
  - Evita confiar en el soporte de la capa de red.
  - Usado en servicios de *streaming* de datos multimediales (e.g. HLS & DASH, RTMP, WebRTC)

3.2 Comunicación entre procesos mediante *sockets*



Protocolos de transporte de datos de TCP/IP



Característica	UDP	TCP
Tipo de servicio	datagrama (paso de mensaje)	*data streaming* (flujo de bytes)
Conexión	Sin conexión	Con conexión
Fiabilidad	*Best-effort* (sin garantía de entrega)	Confiable (sin pérdida de paquetes). Rompe conexión si no es posible recuperar error.
Ordenamiento	Solo paquetes de un mismo datagrama	FIFO (al *byte*)
Control de flujo	No	Si
Rendimiento	Liviano (eficiente para interacción rápida en un medio confiable)	Pesado (eficiente para transferir grandes volúmenes de datos)

API de Sockets

DEFINICIÓN: Una interfaz (API) para comunicación entre procesos o máquinas. Un *socket* es abstractamente un "terminal" (*endpoint*) para crear transparentemente conexiones entre programas a través del sistema de comunicación y se identifican con una dirección tipo $< IP, port >$.

- Aparece en versión 4.2 BSD de Unix (1983) y hoy es parte del estándar POSIX.1-2008.
- En UNIX existe similitud entre API de *sockets* y de archivos.

Tipo de socket	Protocolo	Característica	Casos de uso
Stream Socket (SOCK_STREAM)	TCP	Fiable, orientado a la conexión	Aplicaciones Web, mensajería
Datagram Socket (SOCK_DGRAM)	UDP	No fiable, sin conexión	VoIP, *streaming*, juegos
Raw Socket (SOCK_RAW)	IP	Acceso a protocolos de capa de red	*Packet sniffing*
Multicast Socket (SOCK_DGRAM)	UDP	*One-to-many*	Streaming de multimedia

Puertos de servicios (reservados)

Asignación estática (sistema): 0-1023 ($2^{10} - 1$)

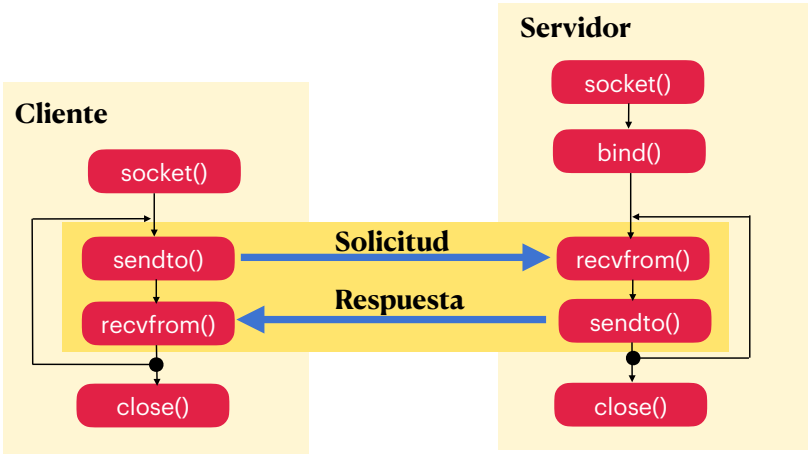
Puerto	Servicio	TCP	UDP
20	FTP (transferencia de archivos)	Si	Si
22	SSH (Secure Shell)	Si	(asignado)
23	Telnet (terminal remoto)	Si	(asignado)
25	SMTP (correo electrónico)	Si	(asignado)
53	DNS (servicio de nombres de dominio)	Si	Si
80	HTTP (transporte de datos en la Web)	Si	Si
111	ONC-RPC (invocación remota a procedimiento)	Si	Si
123	NTP (sincronización de relojes)	(asignado)	Si
161	SNMP (administración de componentes de red)	(asignado)	Si
220	IMAP (correo electrónico)	Si	Si
389	LDAP (servicio de directorio)	Si	(asignado)
443	HTTPS (transporte seguro de datos en la Web)	Si	Si

Rango:

$0 - (2^{16} - 1)$

Ver: IETF (2015). "Recommendations on Using Assigned Transport Port Numbers". Online: https://datatracker.ietf.org/doc/html/rfc7605

a) Sockets con UDP (SOCK_DGRAM)



API de Sockets (UDP)

Característica	C (BSD)	Python	Java
Biblioteca	<sys/socket.h>, <netinet/in.h>	socket	java.net.*
Crear un socket	int sockfd = socket(AF_INET, SOCK_DGRAM, 0);	sock = socket.socket (socket.AF_INET, socket.SOCK_DGRAM)	DatagramSocket socket = **new** DatagramSocket();
Cerrar un socket	close(sockfd);	sock.close()	socket.close();
Asociar dirección (*bind*)	bind(sockfd, ...);	sock.bind((host, port))	socket.bind(**new** InetSocketAddress(port));
Enviar datos	sendto(sockfd, buffer, size, flags);	sock.send(data, (host, port))	DatagramPacket packet = **new** DatagramPacket(data, length, address, port); socket.send(packet);
Recibir datos	recvfrom(sockfd, buffer, size, flags);	data, addr = sock.recvfrom(buffer_size)	DatagramPacket packet = **new** DatagramPacket(buffer, buffer.length); socket.receive(packet);

Ejemplo: UDP con Java

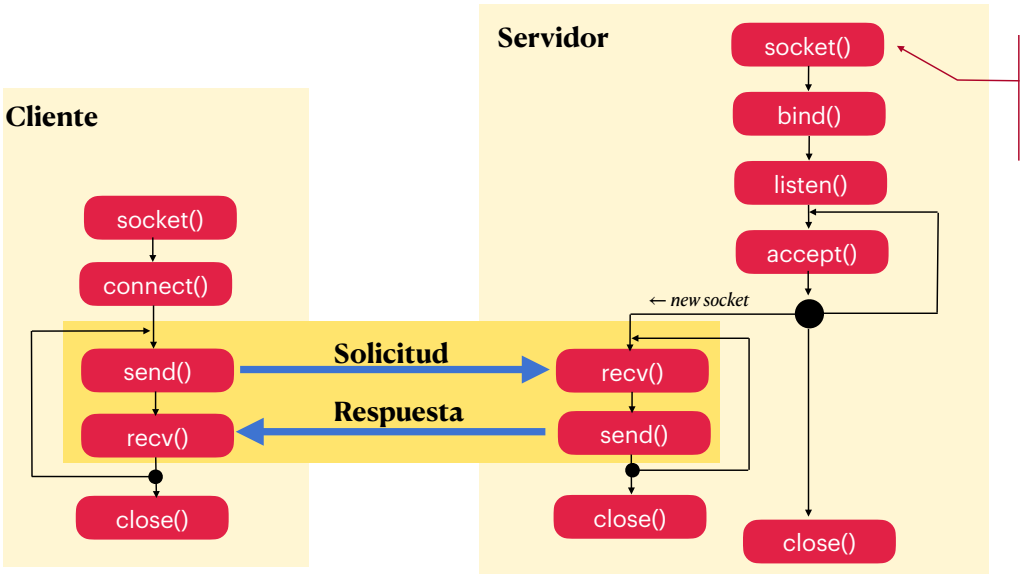
Cliente

```
DatagramSocket socket
DatagramPacket packet;
int  serverPort = 8080;
InetAddress address;
byte[] buffer;

socket = new DatagramSocket();      // Crear UDP socket
byte[] buffer = new byte[1024];     // Crear buffer

packet = new DatagramPacket(data, data.length, address, serverPort);
  // Crear paquete de envío
// preparar solicitud en paquete
socket.send(packet); // enviar solicitud
socket.recv(packet); // recibir respuesta
// procesar respuesta
// terminar la comunicación
socket.close();
```

Servidor

```
DatagramSocket socket
DatagramPacket packet;
int  serverPort = 8080;
InetAddress address;
byte[] buffer;

socket = new DatagramSocket(serverPort); // Crear y ligar UDP socket
byte[] buffer = new byte[1024];          // Crear buffer

for(; ;) {
   packet = new DatagramPacket(buffer, buffer.length);   // Crear paquete de recepción

   socket.receive(packet); // recibir solicitud
   address = packet.getAddress();   // obtener dirección IP del remitente
   int port = packet.getPort();     // obtener Nº puerto del remitente
   // procesar solicitud y preparar respuesta
   packet = new DatagramPacket(buffer, buffer.length, address, port);
   socket.send(packet); // enviar respuesta
}
// terminar la comunicación
socket.close();
```

b) Sockets con TCP (SOCK_STREAM)



API de Sockets (TCP)

MANEJO DE CONEXIONES

Característica	C (BSD)	Python	Java
Crear un socket	int sockfd = socket(AF_INET, SOCK_STREAM, 0);	sock = socket.socket (socket.AF_INET, socket.SOCK_STREAM)	Socket socket = **new** Socket(host, port);
Cerrar un socket	close(sockfd);	sock.close()	socket.close();
Asociar dirección (*bind*)	bind(sockfd, ...);	sock.bind((host, port))	ServerSocket server = **new** ServerSocket(port);
Escuchar una conexión	listen(sockfd, n);	sock.listen(n)	server.accept();
Aceptar una conexión	int client = accept(sockfd, ...);	client, addr = sock.accept()	Socket client = server.accept();
Conectarse a servidor	connect(sockfd, ...);	sock.connect((host, port))	Socket socket = **new** Socket(host, port);
Enviar datos	send(sockfd, buffer, size, flags);	sock.send(data)	InputStream in = socket.getInputStream(); in.read(buffer);
Recibir datos	recv(sockfd, buffer, size, flags);	data = sock.recv(size)	InputStream in = socket.getInputStream(); in.read(buffer);

Ejemplo: TCP con Python

UNIVERSIDAD TECNICA FEDERICO SANTA MARIA | DEPARTAMENTO DE INFORMÁTICA

Cliente

```
import socket

def run_client():
  # crear un objeto socket
  client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

  server_ip = "127.0.0.1"  #  dirección IP del servidor
  server_port = 8000       #  número de puerto del servidor
  # establecer conexión con el servidor
  client.connect((server_ip, server_port))

  while True:
     # ingresar mensaje y envier al servidor
     msg = input("Ingresar mensaje: ")
     client.send(msg.encode("utf-8")[:1024])

     # recibir mensaje del servidor
     response = client.recv(1024)

     response = response.decode("utf-8")
    # si el servidor envió"closed", entonces cerrar socket del cliente
     if response.lower() == "closed":
        break

     print(f"Recibido: {response}")   # e iterar con próximo entrada

  # cerrar conexión del cliente con el servidor
  client.close()
  print("Conexion cerrada por el servidor)

# ejecutar cliente
run_client()
```

Servidor

```
import socket

def run_server():
   # crear un objeto socket
   server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

   server_ip = "127.0.0.1"    # localhost
   port = 8000                # puerto del servidor

   # ligar (bind) el socket a la dirección y puerto especificada
   server.bind((server_ip, port))
   # escuchar a peticiones de conexiones
   server.listen(0)
   print(f"Escuchando en {server_ip}:{port}")

   # acepta una nueva conexión
   client_socket, client_address = server.accept()
   print(f"Conexión aceptada desde {client_address[0]}:{client_address[1]}")
   # recibir datos desde el cliente
   while True:
      solicitud = client_socket.recv(1024)
      solicitud = solicitud.decode("utf-8")     # convertir bytes en un string

      # si se recibe "close" desde el cliente, entonces cerrar la conexión
      if solicitud.lower() == "close":
         client_socket.send("closed".encode("utf-8")) # avisar al cliente que cierre conexión
         break

      respuesta = "conexion aceptada".encode("utf-8")   # convertir string a bytes
      # convertir y enviar respuesta al cliente
      client_socket.send(respuesta)

   # cerrar conexiones
   client_socket.close()
   server.close()

# ejecutar servidor
run_server()
```


UNIVERSIDAD TECNICA FEDERICO SANTA MARIA | DEPARTAMENTO DE INFORMÁTICA

c) Multicast IP

Características principales

- **Abstracción**. Multicasting IP es una abstracción de *multicasting* físico (en redes de difusión).
  - Datagramas IP se transmiten a miembros de un grupo dispersos en redes físicamente separadas.
  - Grupo tiene una dirección única IP de clase D en *IPv*4 (32*b*); tb. *IPv*6 (128*b*)
  - API similar a UDP (SOCK_DGRAM), pero agrega funciones para gestión de grupos.
- **Calidad de servicio**. Sin conexión, sin orden y entrega de mejor-esfuerzo (*best-effort*).
  - Como UDP, en *sockets* se hereda la misma QoS que tiene IP.
- **Gestión grupos**. Grupos dinámicos (máquinas pueden dinámicamente unirse o abandonar un grupo), sin sincronizar con otros los cambios de membresía.
  - Por lo tanto, un emisor no conoce (en principio) los destinatarios de sus mensajes.
  - Una misma máquina (o nodo) puede pertenecer a múltiples grupos.

Direcciones de Multicast IP

IPv4 **+ NAT sigue siendo opción más usada**

Protocolo *IPv4*

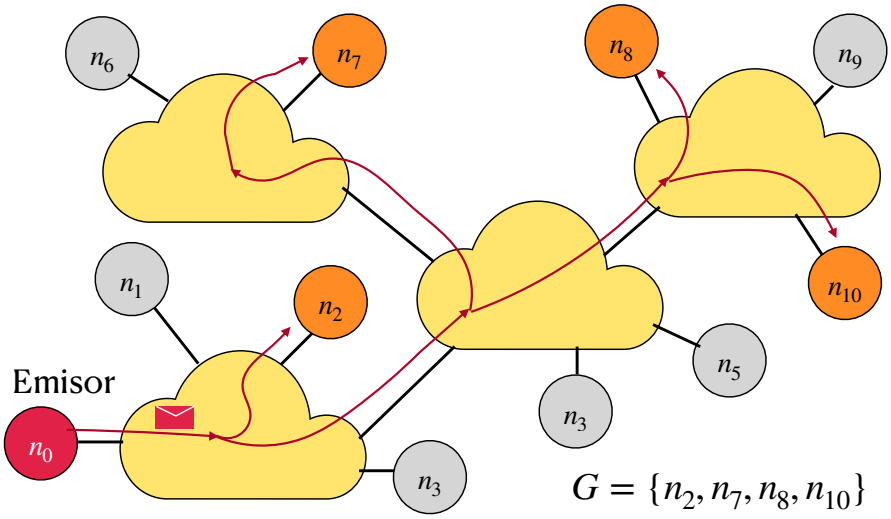
- Direcciones de 32*b* de Clase D (comienzan con 1110): 224.0.0.0 – 239.255.255.255
  - 224.0.0.0: No se usa
  - 224.0.0.1: El grupo de todos los grupos
  - 224.0.0.2 – 224.0.0.255: Reservadas (información de rutas)
- Existen grupos bien conocidos asignados por la autoridad central (permanentes); otras direcciones son transitorias.

Protocolo *IPv6*

- Direcciones de 128*b* que comienzan con 0*xff*.
- Provee también mecanismos de Anycast (se entrega a algún miembro, típicamente cercano).

Multicast e Internetworking

Requisito de *routing* entre redes con miembros de grupo



OBSERVACIONES:

- Se requiere definir una red sobrepuesta para rutear paquetes de *multicasting*.
- Se requiere limitar el alcance, para evitar sobrecargar las redes.
- Se puede usar *tunneling* para pasar por redes sin soporte de *multicasting*.

Gestión de Grupos en Multicast IP

- **Protocolo IGMP** (Internet Group Management Protocol: extensión de IP).
  - IGMP es usado por máquinas para reportar su membresía a *routers* próximos.
- **Routers** (de *multicast*):
  - Routers intercambian información periódicamente sobre estado de membresía en su red (un demonio por red; *mrouted* en Unix).
  - Se agrega PIM (Protocol-Independent Multicast) para ruteo de datagramas.
  - Uso de datagramas IP de *unicast* para encapsular mensajes de IGMP y para realizar *tunneling* en Internet a través de redes sin soporte para *multasting*.
- **Primitivas básicas:**
  - *Send*() y *Receive*(): envío y recepción de mensajes (datagramas)
  - *JoinHostGroup*() y *LeaveHostGroup*(): unirse y abandonar un grupo

Ejemplo de Multicast (IPv4) en Python

Emisor

```
import socket
import struct

MULTICAST_GROUP = '224.0.0.1'
PORT = 5007

# crear y configurar socket
sock = socket.socket(socket.AF_INET,
                     socket.SOCK_DGRAM, socket.IPPROTO_UDP)
sock.setsockopt(socket.IPPROTO_IP,
                socket.IP_MULTICAST_TTL, 2)  # TTL = 2

# enviar un mensaje de multicast al grupo IP y puerto especificado
message = b"Hola, multicast!"
sock.sendto(message, (MULTICAST_GROUP, PORT))

# cerrar socket
sock.close()
```

Receptor

```
import socket
import struct

MULTICAST_GROUP = '224.0.0.1'
PORT = 5007

sock = socket.socket(socket.AF_INET,
                     socket.SOCK_DGRAM, socket.IPPROTO_UDP)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind(('', PORT))       # vincular puerto a  interfaces de red

# unirse a un grupo multicast
mreq = struct.pack("4sl", socket.inet_aton(MULTICAST_GROUP),
                          socket.INADDR_ANY)
sock.setsockopt(socket.IPPROTO_IP,
                socket.IP_ADD_MEMBERSHIP, mreq)

# atender recepción de mensajes de multicast
while True:
    data, addr = sock.recvfrom(1024)
    print(f"Received {data} from {addr}")
```

Ejemplo de servidor de tiempo en Java

Servicio básico de notificación

```
MulticastSocket socket;
DatagramPacket packet;
InetAddress address;

address =
        InetAddress.getByName(args[0]);
socket = new MulticastSocket(PORT);

// Unirse a grupo de multicasting
socket.joinGroup(address);

byte[] data = null;



// sigue …
```

```
for (;;) {
      Thread.sleep(1000);
       System.out.println("Enviando fecha");
       String str = (new Date()).toString();

       data = str.getBytes();
       packet = new
                    DatagramPacket (data,str.length(),
                                          address, PORT);

        socket.send(packet);
}  // for

// abandonar grupo de multicasting
socket.leaveGroup(address);
....
```


Evaluación del uso de Multicast IP

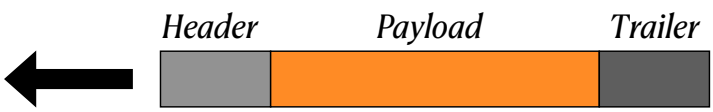
- **Soporte de red**. Multicast IP no es bien soportado en Internet.
  - No todas las redes dan buen soporte para *multicasting* (preferencia por *Middleware*).
  - Consume muchos recursos y susceptible de ser atacado.
  - La configuración de redes para soportar *multicast* no es simple de realizar (especialmente cuando existen múltiples dominios administrativos).
  - Muchos ISP usan multicast dentro de sus redes en un mismo dominio administrativo (e.g., para CDN y *Streaming* de multimedios).
- **Soporte de IPv6**. Multicast IP está mejor soportado por IPv6, con direcciones de 128 bits, pero es aun de uso poco frecuente.
  - En esta asignatura nos hemos limitado a ver ejemplos de programación con **Multicast** con *sockets* usando UDP / IPv4.

Conclusiones generales sobre uso de *sockets*

- **Programación compleja**. *Sockets* son un mecanismo para comunicación entre procesos (IPC) de bajo nivel, pues programador debe preocuparse por detalles como:
  - Transferencia de datos. Manejo de estructuras de mensajes; más complicado en ambientes heterogéneos.
  - Coordinación. Se requiere en interacciones *request-reply* (más fácil de realizarlo con TCP; e.g., en HTTP).
  - Recuperación de errores. Manejar errores de comunicación y establecer procedimientos de recuperación.
- **Calidad de Servicio (QoS)**. Desarrolladores pueden requerir mayor apoyo para abstraer la resolución de problemas complejos (e.g. tolerancia a fallos, escalabilidad, e interoperabilidad).
  - Soporte adicional. Interacciones asincrónicas y con persistencia de mensajes.
  - Consistencia. Se requiere a veces garantía de entrega (e.g. asentimiento de procesamiento o transaccional).
- **Modelo de programación**. Modelo con *sockets* se basa en paradigma de mensaje, que no es el modelo usual entre desarrolladores de software (i.e., normalmente se usa invocación).
  - Se produce alta "impedancia" entre los diferentes modelos de razonamiento, que requiere adaptación (*matching*).
  - Uso de una API abstrae detalles de la comunicación basada en mensajes usando un modelo de invocación, pero se debe entender la calidad de servicio comprometida por los protocolos.

OBSERVACIÓN: Uso de *sockets* son útiles para construir abstracciones y servicios de comunicación de mayor nivel (i.e. protocolos de comunicación de *Middleware*), facilitando el desarrollo de aplicaciones distribuidas.

3.3 Representación e intercambio de datos en ambientes distribuidos



Representación de datos en Sistemas Distrbuidos

El "intercambio de datos" es esencial en la comunicación

DESAFÍOS:

- **Fuentes de heterogeneidad**. Existen diferentes arquitecturas de computadores y formatos de comunicación; diferentes tipos de datos según lenguajes de programación, representación y precisión numérica, codificación de caracteres, etc.
- **Interoperabilidad**. Lograr un intercambio de datos entre diferentes sistemas en entornos heterogéneos, con una correcta interpretación.
- **Eficiencia vs. legibilidad**. Compromiso entre eficiencia (*parsing*, uso de memoria/BW; compresión) y legibilidad (entendible por humanos).
- **Estandarización**. Estándares comunes para resolver estos desafíos y facilitar la comunicación de datos.

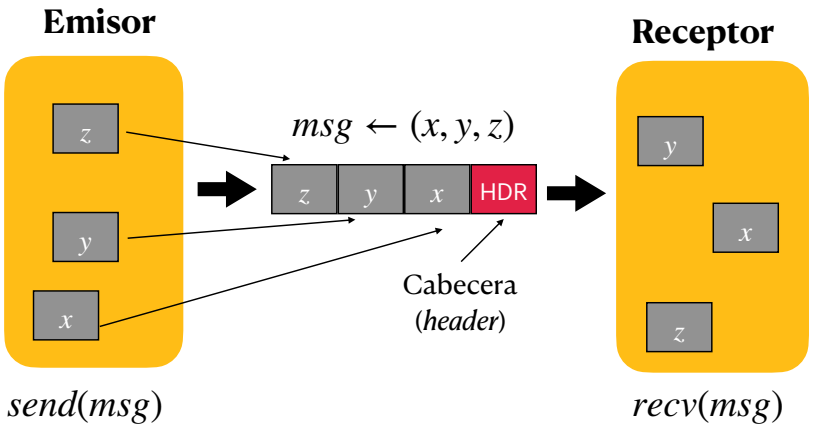
SOLUCIONES:

- Formatos estándares de representación de datos y métodos de conversión.
- Métodos de serialización (*marshalling*) y deserialización (*unmarshalling*) de los datos.

OBSERVACIÓN: Son funciones de capa 6 en modelo ISO/OSI)

Serialización y deserialización de datos

También *marshalling* y *unmarshalling*



DEFINICIONES:

- ***Marshalling*** (serialización): Proceso de ensamblar en el emisor los datos a ser intercambiados, como una secuencia de bytes en un formato que facilite su transmisión.
- ***Unmarshalling*** (deserialización): Proceso inverso que ocurre en el receptor, que determina para un programa los valores de variables en su propio espacio de memoria.

Observación: *msg* es finalmente una secuencia de bytes (o bits).

Ejemplo de heterogeneidad

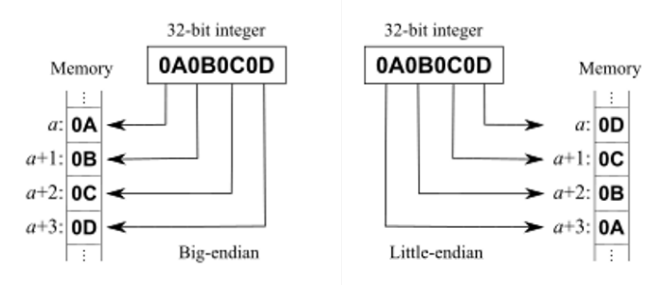
Representación e intercambio de números

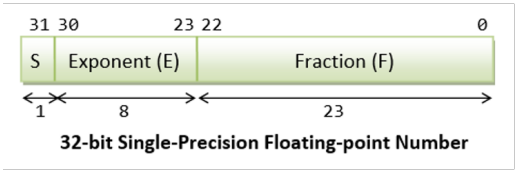
Números enteros:

- Largo (16, 32 o 64 bits)
- Representación (e.g. signo, C-1, C-2)
- Orden de *bytes* en memoria (*big endian* o *little endian*)

Números de punto flotante:

- Largo (32 o 64 bits)
- Representación (e.g. IEEE 754)
- Orden de *bytes* en memoria (*big endian* o *little endian*)





Protocolos de intercambio de datos

Serialización y deserialización

Formatos de representación de los datos

- **Binario**: Datos primitivos se representan en binario (e.g. IEEE 754 para punto flotante).
  - Representación compacta y eficiente, pero sólo legible por las máquinas.
  - Requiere *parsing* especial, pero más eficiente.
  - Java permite serialización binaria de objetos (incluyendo información para "introspección").
- **Textual**: Se representan como *string* de caracteres (e.g. XML y JSON).
  - Legibles por un humano.
  - Más verboso y menos eficiente (ocupan más memoria y BW, con mayor retardo).
  - Requiere mayor procesamiento para *parsing* de caracteres y extracción de información útil.

Técnica de codificación

- **Autocontenida**: Datos contenidos se auto describen.
  - Interpretación de mensaje es independiente del generador (emisor).
  - Agrega etiqueta/campo para especificar tipo antes de cada dato (mayor *overhead* de memoria y BW).
  - Pueden ser binaria (e.g. ASN.1, Java) o textual (e.g. XML, JSON).
- **Acordada**: Partes comparten previamente formato de codificación usado, para una correcta interpretación.
  - Típicamente binaria (e.g. Sun-XDR, CDR de Corba; Proto Buffer de Google).
  - Compilador genera automáticamente desde un esquema o lenguaje de descripción de interfaz (IDL) código para serializar y deserializar los datos.

Ejemplo: Representación textual (autocontenido)

INFORMACIÓN: 26 caracteres (sin incluir información estructural)

XML

```
<employees>
   <employee>
      <firstName>John</firstName>
      <lastName>Doe</lastName>
   </employee>
   <employee>
      <firstName>Anna</firstName>
      <lastName>Smith</lastName>
   </employee>
   <employee>
      <firstName>Peter</firstName>
      <lastName>Jones</lastName>
   </employee>
</employees>
```

Total: 244 caracteres (10,7% de eficiencia)

JSON

```
{"employees":[
  { "firstName":"John", "lastName":"Doe" },
  { "firstName":"Anna", "lastName":"Smith" },
  { "firstName":"Peter", "lastName":"Jones" }
]}
```

Total: 134 caracteres (19,4% de eficiencia)

OBSERVACIÓN:

- JSON está basado en JavaScript, se puede parsear más rápidamente y ocupa menos espacio que XML.

Fuente: https://www.w3schools.com/js/js_json_xml.asp

Ejemplo simple con XDR de Sun (ONC)

writer.c (escribe en `stdout`)

```
#include <stdio.h>
#include <stdlib.h>
#include <rpc/xrpc.h>   /* xdr is a sub-library of rpc */

int main()
{
    XDR xdrs;   /* XDR stream handle*/
    int i, n = 100000000;

    xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
    for (i = 0; i < 8; i++) {
        if (!xdr_int(&xdrs, &n)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        n++;
    }
    return 0;
}
```

```
% writer > tmp
% hexdump tmp
0000000 f505 00e1 f505 01e1 f505 02e1 f505 03e1
0000010 f505 04e1 f505 05e1 f505 06e1 f505 07e1
0000020
%
```

reader.c (escribe en `stdin`)

```
#include <stdio.h>
#include <stdlib.h>
#include <rpc/xdr.h>   /* xdr is a sub-library of rpc */

int main()
{
    XDR xdrs;  /* XDR stream handle */
    int i, n;

    xdrstdio_create(&xdrs, stdin, XDR_DECODE);
    for (i = 0; i < 8; i++) {
        if (!xdr_int(&xdrs, &n)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        printf("%d ", n);
    }
    printf("\n");
    return 0;
}
```

```
% reader < tmp
100000000 100000001 100000002 100000003
100000004 100000005 100000006 100000007
%
```

Fuente: https://docs.oracle.com/cd/E18752_01/html/816-1435/xdrnts-1.html

Ejemplo con Proto Buffer (binario y acordado)

Interoperabilidad entre diferentes lenguajes

Esquema

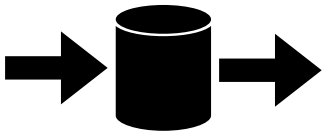
```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

Compilador a Java

Compilador a C++

Serialización en Java

```
Person john = Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .build();
output = new FileOutputStream(args[0]);
john.writeTo(output);
```



Deserialización en C++

```
Person john;
fstream input(argv[1], ios::in | ios::binary);
john.ParseFromIstream(&input);
int id = john.id();
std::string name = john.name();
std::string email = john.email();
```

Fuente: https://developers.google.com/protocol-buffers/docs/overview

Ejemplo: Serialización en Java (binario y auto-contenido)

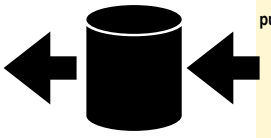
1) Definiendo una clase serializable (equivale a a IDL)

```
public class Employee implements java.io.Serializable {
   public String name;
   public String address;
   public int number;

   public void direccion() {
      System.out.println("Dirección de " + name + " es " + address);
   }
}
```

importar clase Employee

2) Serializando un objeto

```
import java.io.*;
Import Employee;

public class SerializeDemo {

  public static void main(String [] args) {
     Employee e = new Employee();
     e.name = "Ana Henriquez";
     e.address = "Avda. Vergara 110, Concón";
     e.number = 101;

     try {
        FileOutputStream fileOut = new FileOutputStream("/tmp/empleado.ser");
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(e);       //escribe y almacena en archivo  el objeto
        out.close(); fileOut.close();
        System.out.printf("Datos serializados guardados en /tmp/employee.ser");
     } catch (IOException i) {
        i.printStackTrace();
     }
  }
}
```



3) Guardar objeto serializado en archivo "/tmp/empleado.ser" (tb. transferir por socket)

3) Deserializando un objeto

```
import java.io.*
Import Employee;

public class DeserializeDemo {

  public static void main(String [] args) {
     Employee e = null;
     try {
        FileInputStream fileIn = new FileInputStream("tmp/empleado./ser");
        ObjectInputStream in = new ObjectInputStream(fileIn);
        e = (Employee) in.readObject();   // lee e instancia el objeto
        in.close();
        fileIn.close();
     } catch (IOException i) {
        i.printStackTrace();
        return;
     } catch (ClassNotFoundException c) {
        System.out.println("clase Employee no encontrada");
        c.printStackTrace();
        return;
     }

     System.out.println(" Empleado desserializado ...");
     System.out.println("Nombre: " + e.name);
     System.out.println("Dirección: " + e.address);
     System.out.println("Número: " + e.number);
  }
}
```

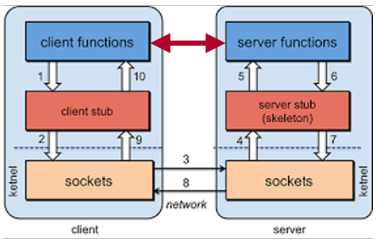
Fuente: https://www.tutorialspoint.com/java/java_serialization.htm

Formatos y Protocolos de Serialización de Datos

Resumen de estándares y métodos comunes en diferentes ámbitos

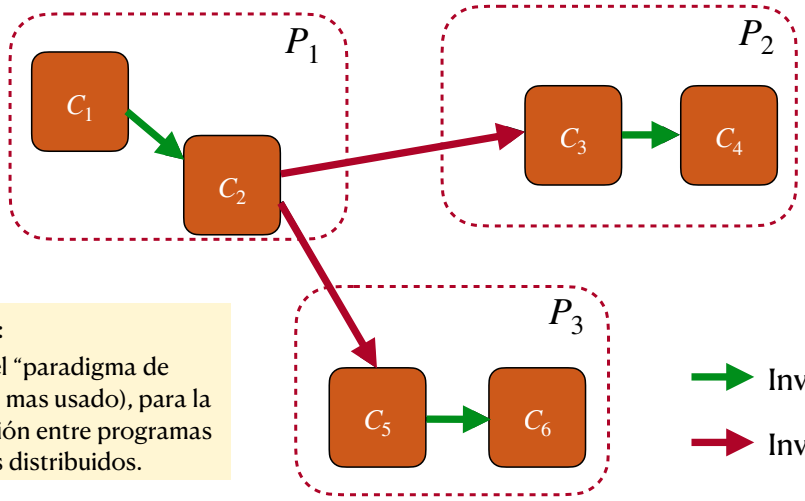
- **Redes de computadores (tradicionales):**
  - ASN.1 de ISO/OSI (binario)
  - Formato de caracteres (textual)
- **RPC (tradicionales):**
  - XDR en RPC de Sun (tb. ONC)
  - NDR en DCE/RPC de la OSF
- **Objetos distribuidos y componentes**
  - CDR en Corba de la OMG
  - Java RMI (soporta referencias remotas y paso de objetos)
  - NDR en RPC/DCOM de Microsoft
- **RPC (Binarios y HTTP, entre otros):**
  - Protocol Buffer de Google (gRPC, microservicios)
  - Apache Avro (*Data streaming* en Kafka)
  - Apache Thrift (múltiples lenguajes)
- **Textuales (basadas en HTTP y en documentos):**
  - XML (SOAP, XML-RPC, documentos)
  - JSON (REST Web API, NoSQL DB, GraphQL, configuraciones)
  - YAML (Kubernetes, CI/CD, configuraciones de infraestructura)

3.4 Invocación remota



Distribución de programas

Idea básica de invocaciones remotas



OBJETIVO:

- Mantener el "paradigma de control" (el mas usado), para la comunicación entre programas en sistemas distribuidos.

Invocación remota

Sobre un procedimiento, función o método

"**Llamada remota a procedimiento** (RPC) es un paradigma útil para comunicarse en una red para programas (distribuidos) escritos en un lenguaje de alto nivel."

— [A. Birrel & B. Nelson, 1984]

DEFINICIÓN: **Invocación remota** es una mecanismo de comunicación entre procesos (IPC), que permite a un programa ejecutar código o invocar funciones/métodos en otro proceso*, sin que el programador codifique explícitamente los detalles para esta interacción remota.

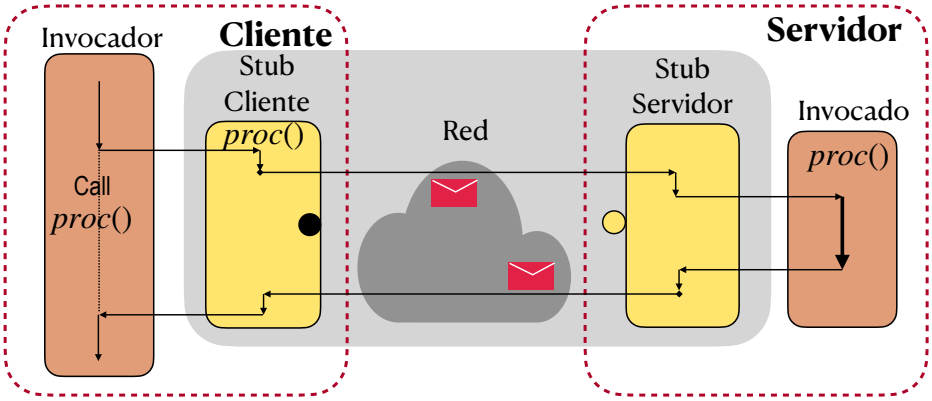
* Típicamente en un proceso de otra máquina en una red, que tiene otro espacio de direcciones

- Abstrae la comunicación de red, haciendo que las interacciones remotas aparezcan como si fueran llamadas locales ("transparencia de acceso").

Fuente: A. Birrel & B. Nelson (1984). "Implementing remote procedure calls". *ACM Trans. Comput. Syst.* 281), Feb. 1984, pp. 39-59

Invocaciones remotas

Concepto de *stub*



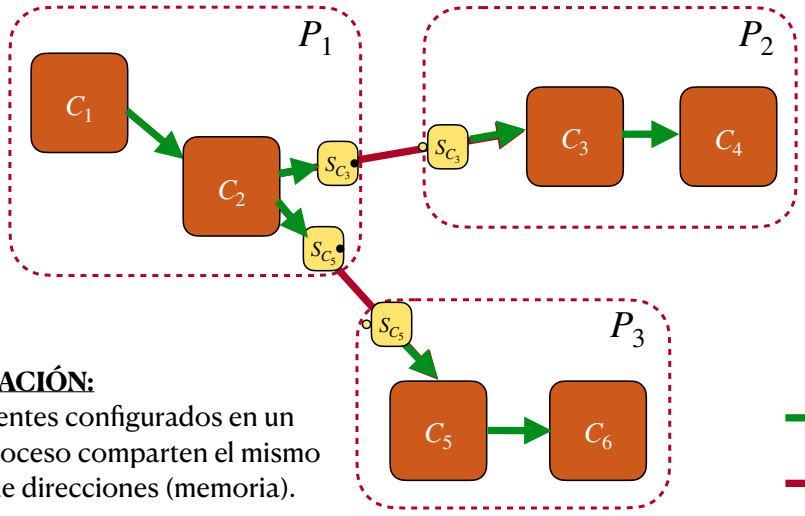
Nombres alternativos:
- Stub-Skeleton (Java)
- Proxy-Stub (MS)

Patrones relacionados:
- Broker
- Proxy

OBSERVACIÓN: Par de *stubs* hace transparente para el programador la invocación remota, ocultando la distribución del programa y la comunicación remota por la red por paso de mensajes.

Distribución de programas

Uso de *stubs*



Desafíos:
- Latencia
- Serialización
- Fallas parciales
- Seguridad

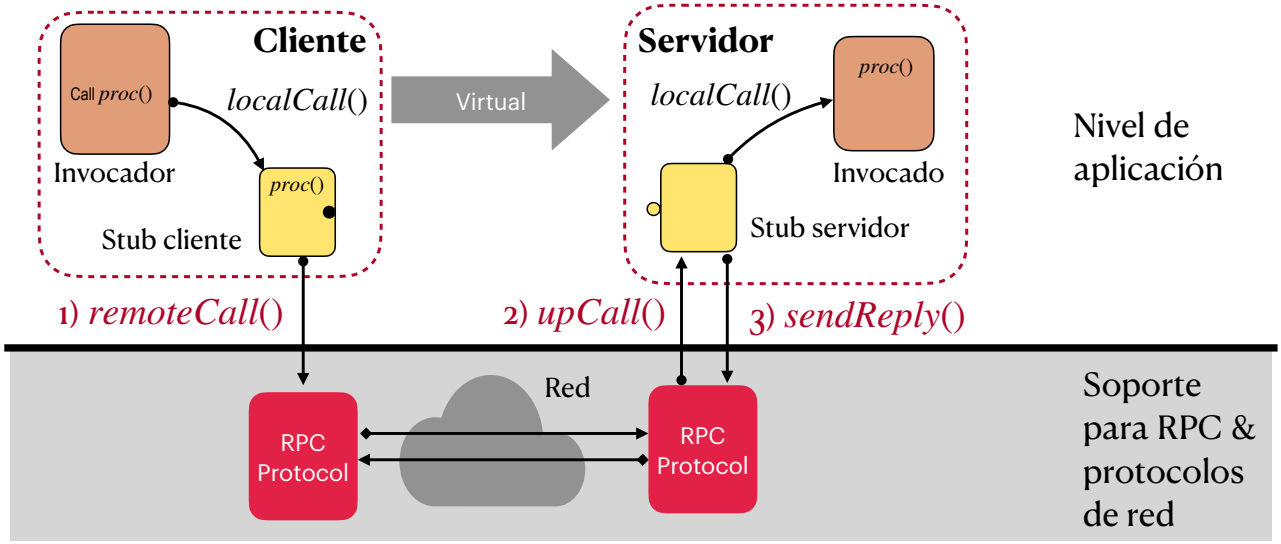
OBSERVACIÓN:
Componentes configurados en un mismo proceso comparten el mismo espacio de direcciones (memoria).

→ Invocación local

→ Invocación remota

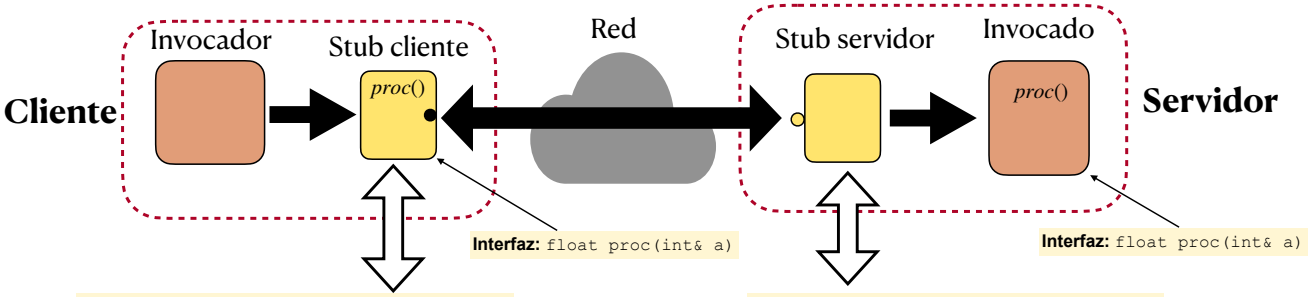
Protocolo de Invocación Remota

Esquema simple de flujo de control y datos



Implementación de Stubs

Ejemplo de esquema simple con pseudo-código para *stubs*




```
float proc(int& a)
{
  char msg[MAX];
  float res;

  definir cabecera de msg;
  agregar valor de a a msg;
  callrpc(server, msg); /* downcall */
  sacar resultado de msg y asignar a res;
  return res;
}
```

```
void proc_stub(char* msg)  /* upcall */
{
  int a;
  float res;

  sacar valor de msg y asignarlo a a;
  res = proc(a);   /* local call */
  preparar en msg cabecera de respuesta
  agregar res a msg;
   sendreply(msg); /* downcall */
}
```

Generación automática de *Stubs*

Uso de IDL (*Interface Description Language*) o Esquemas

CARACTERÍSTICAS:

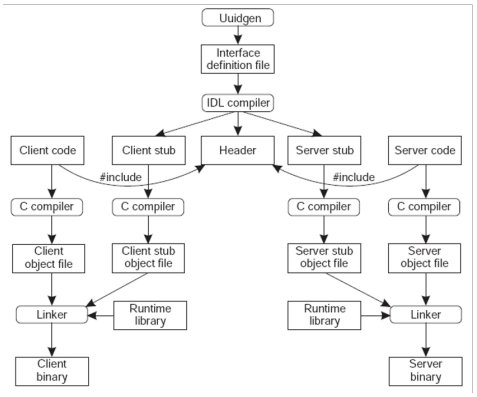
- Stubs pueden ser generados automáticamente a partir de la definición de su interfaz (IDL).
- La IDL (o esquema) depende del protocolo y lenguaje de programación.
  - Procedimientos (o función) tipo C (e.g. ONC-RPC)
  - Interfaces y clases de objetos (e.g. Java RMI)
  - Servicios Web (e.g. XML, RESTful)
- Existen generadores de *stubs* (o compiladores) que traducen a múltiples lenguajes, para soportar ambientes heterogéneos.
- Incluye generación de código para el proceso de *marshalling* o *unmarshalling*, para interoperabilidad en ambientes heterogéneos.

EJEMPLOS:

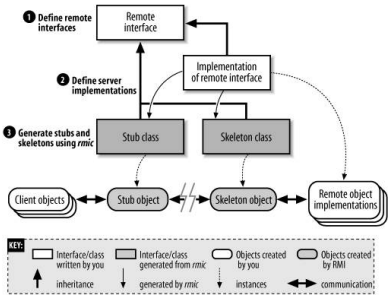
- **Sun RPC o ONC RPC** (ver: IETF, RFC 5531, 2009)
  - IDL con sintaxis similar a C
  - `rpcgen`: compilador de IDL que genera *stubs* en C.
  - Usado en servicios de red en Unix (e.g. comandos remotos, NFS)
- **Java RMI** (ver: docs.oracle.com)
  - IDL corresponde a una interfaz que extiende tipo `Remote`.
  - Compilador corresponde a comando `rmic` que genera *bytcode*.
  - Interoperabilidad a otros ambientes se logra con Corba.
- **XML-RPC y SOAP** (ver: xmlrpc.com y www.w3.org)
  - RPC con mensajes en formato XML (mensajes SOAP)
  - Protocolo XML-RPC basado SOAP para invocar servicios Web
- **gRPC** (ver: grpc.io)
  - IDL basado en Protocol Buffers de Google, con traducción a múltiples lenguajes (e.g. C++, C#, Java, Python, Go, etc.).

Ejemplos de generadores de *Stubs*

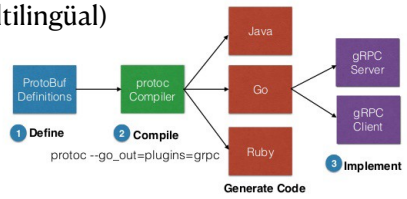
a) DCE-RPC (lenguaje C)



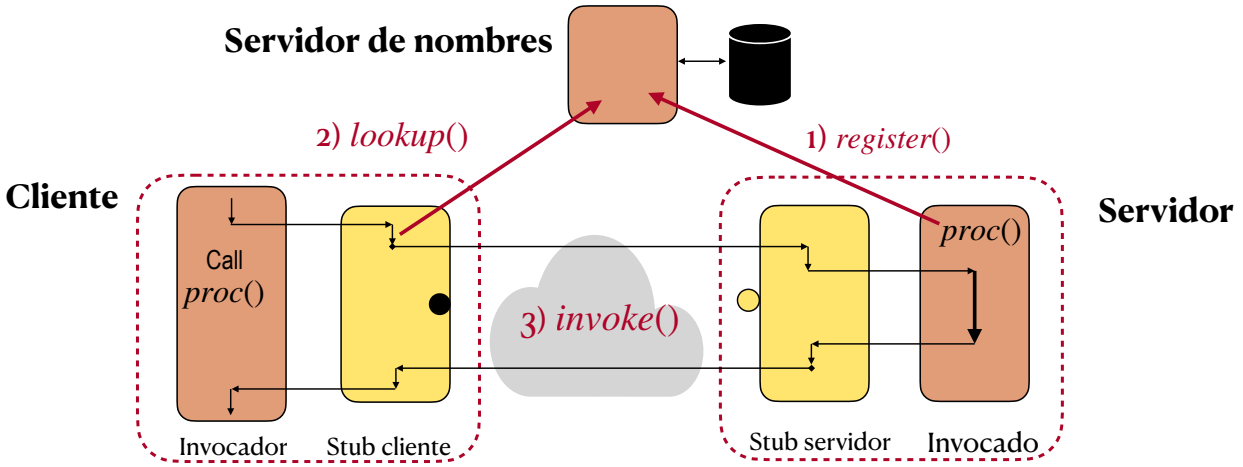
b) Java RMI



c) gRPC (multilingüal)



Ligado dinámico



Observación:

- Se agrega información de tipo y versión de servidor para calzar correctamente las interfaces y *stubs*.
- *Late binding*, con uso de *caching* evita volver resolver en invocaciones posteriores.

Descubrimiento y ligado en Invocaciones remotas

Métodos de descubriendo y ligado (estáticos o dinámicos)

Descubrimiento:

- Configuración estática. Direcciones fijas (*hard-coded*, no escala y es rígido)
- Portmapper. Mapea nombre del servicio en puerto en máquina fija (e.g. Sun RPC)
- Descubrimiento integrado : *built-in* (e.g. Java rmiregistry)
- Servicios de directorio (e.g. DNS, LDAP)
- Servicio de registro. BD centralizada tolerante a fallos (e.g. Consul, Zookeeper & etcd)

Tipos de ligado (*Binding*):

- Estático. No requiere resolver, inflexible (útil para servicios estándares). Uso de *portmapper* agrega flexibilidad.
- Dinámico. Dirección del servidor se resuelve en tiempo de ejecución (*runtime*): al inicio o en primera invocación (*late-binding*).
- Stub cliente. Actúa como *proxy* y resuelve transparentemente (típicamente con ligado dinámico, como en ejemplo anterior).

Ejemplos de descubrimiento y binding

Java RMI

```java
// Server
public class MyRemoteClass implements myInterface {

   public void myMethod()  throws RemoteException {
      ....
   }

   public static void main(String[] args) {
      .....
     MyRemoteClass obj = new MyRemoteClass();
     myInterface stub = (myInterface)
                    UnicastRemoteObject.exportObject(obj, 0);
     Naming.rebind("MyService", stub);
   }
}
```

```java
// Client
public class MyClientClass {
   ....
  public static void main(String[] args) {
      ....
     // Discover remote object and create stub (client)
     myInterface stub =
           (myInterface) Naming.lookup("MyService");
     // Do remote method invocation
     stub.myMethod();
     .....
   }
}
```

gRPC en Python

```python
# Server
import grpc
from generated_pb2 import MyRequest
from generated_pb2_grpc import MyServiceStub
....
class MyService(MyServiceServicer):
    # Define gRPC methods here
....
# register MyService & bind to port 50051
add_MyServiceServicer_to_server(MyService(), server)
server.add_insecure_port("[::]:50051")    # escucha interfaces IPv6
# iniciar el servicio
server.start()

....
```

```python
# Client
import grpc
from generated_pb2 import MyRequest
from generated_pb2_grpc import MyServiceStub

# Create a channel to the server and bind
channel = grpc.insecure_channel('localhost:50051').

# Create a stub (client)
stub = MyServiceStub(channel)

# Do remote method invocation
response = stub.MyMethod(MyRequest(param="value"))
```


Transparencia de invocaciones remotas

¿La diferencia entre invocación local versus invocación remota?

> *"La semántica de invocación de un mecanismo tipo RPC debe ser lo más semejante posible a una invocación local. Sin embargo, algunas diferencias son inevitables".*
>
> — [A. Birrel & B. Nelson, 1984]

- **Rendimiento**. Una diferencia evidente es el rendimiento.
  - Ver falacias de P. Deutsch en capítulo I sobre latencia, ancho de banda y costo de transporte.
- **Paso de parámetros**. Al no existir memoria compartida, se dificulta lograr transparencia.
  - Semántica de "paso por valor" (*call-by-value*) es fácil, pero no "paso por referencia" (*call-by-reference*).
  - Algunos lenguajes agregan otras semánticas para programación distribuida (e.g. *call-by-object-reference & call-by-move*)
- **Fiabilidad**. Procesos y comunicación pueden fallar (parcialmente).
  - **Semántica de fallas**. Comportamiento de una invocación remota puede ser diferente (e.g., semánticas: *at-least-once*, *at-most-once*, *exactly once*, etc.)
  - **Manejo de errores**. Mecanismos de detección y recuperación de errores debe ser abordado en programación distribuida (e.g. excepciones remotas, manejo de huérfanos, problemas de consistencia, etc.)

Semántica de fallas en Invocaciones remotas

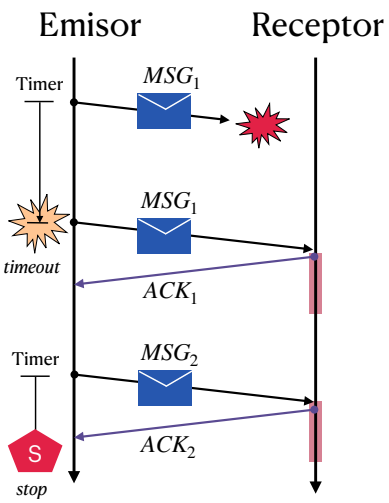
Su nombre se debe a Nº de ejecuciones posibles en el servidor

Semántica	Normal (Sin fallos ni retardos excesivos)	Sin fallos y gran retardo	Omisión de mensajes	Omisión de mensajes y falla de servidor (S)	Omisión de mensajes y falla de cliente (C)	Omisión de mensajes y fallas de C&S
Maybe	Op = 1 Res = 1	Op = 1 Res ≤ 1	Op ≤ 1 Res = 0	Op ≤ 1 Res = 0	Op ≤ 1 Res = 0	Op ≤ 1 Res =0
At-least-once	Op = 1 Res = 1	Op ≥ 1 Res ≥ 1	Op ≥ 1 Res ≥ 1	Op ≥ 0 Res ≥ 0	Op ≥ 0 Res ≥ 0	Op ≥ 0 Res ≥ 0
At-most-once	Op = 1 Res = 1	Op = 1 Res = 1	Op = 1 Res = 1	Op ≤ 1 Res = 0	Op ≤ 1 Res = 0	Op ≤ 1 Res = 0
All-or-nothing	Op = 1 Res = 1	Op = 1 Res = 1	Op = 1 Res =1	Op = 1 Res = 1	Op = 0 Res = 0	Op = 0 Res = 0

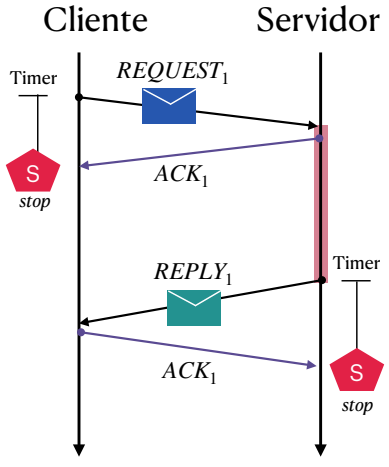
NOTACIÓN: Op: Nº de ejecuciones en el servidor; Res: Nº resultados recibidas por el cliente.

Recuperación de errores de comunicación

Uso de *Timeout* + Retransmisión



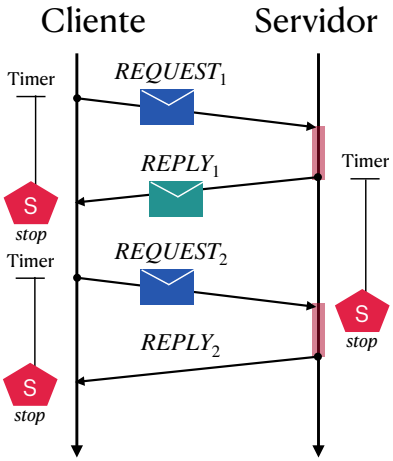
a) Paso de mensaje unidireccional



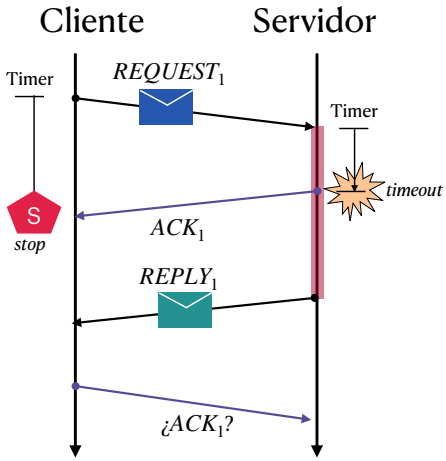
b) Protocolo *Request-Reply*

Protocolo *Request-Reply*

Duración de la invocación



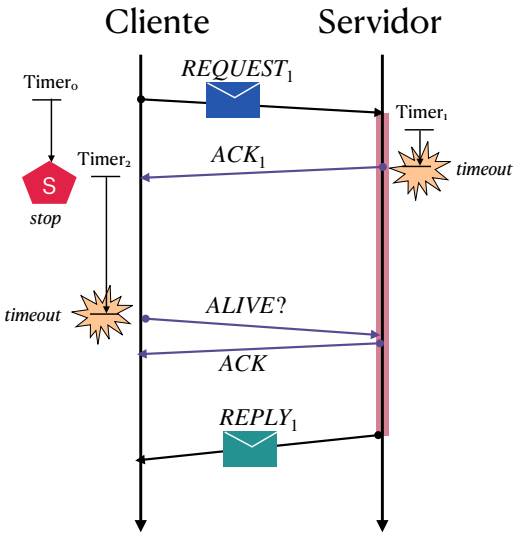
b) Invocación de corta duración



b) Invocación de larga duración

Protocolo *Request-Reply*

Prueba de sobrevivencia (*Alive-Test*)



OBJETIVOS:

- Mantener la sincronización entre cliente y servidor en una invocación de larga duración.
- Detectar y recuperarse ante una eventual falla del servidor o de la red.

TIMERS:

- $Timer_1$: ACK de REQUEST en larga duración
- $Timer_2$: Sondear al servidor si sigue procesando el REQUEST.

Variaciones al modelo de Invocación remota

Responder a necesidades de optimización o de proceso

- **Idempotencia**: Repetición del procesamiento de una petición produce el mismo resultado.
- **Renuncia a la respuesta**: Solo existe petición y no se espera respuesta (tipo notificación)
- **Delegación**: Respuesta puede ser generada por un tercero (sin anidamiento).
- **Llamada de vuelta** (*callback*): Se pasa una dirección del cliente para realizar (múltiples) invocaciones inversas.
- ***Streaming* de múltiples peticiones y/o respuestas**. Flujo de peticiones/respuestas sin sincronización (e.g. gRPC).
- **Invocación múltiple** (tipo *multicast*). Para formar grupos de replicación.

Conclusiones sobre Invocación remota

Evaluación crítica

VENTAJAS:

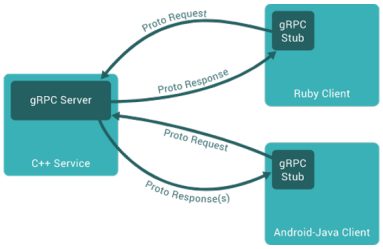
- Uso de un paradigma bien conocido por los programadores facilita programación distribuida.
- Abstrae comunicación remota (e.g. uso de paso de mensaje o protocolo *Request-Reply*).
- Modelo apoya bien uso de procesos y hebras, para mayor concurrencia o paralelismo.
- Reduce esfuerzo de desarrollo, al automatizar generación de código para lógica de comunicación y manejo de mensajes.
- Integra bien otros servicios de apoyo requeridos (e.g. nombre, seguridad, monitoreo, etc.)

DESVENTAJAS:

- Define modelo de comunicación fuertemente acoplado (*tightly coupled*).
- Requiere sincronizar cliente con servidor (ambos deben estar conectados simultáneamente).
- Uso de entornos de ejecución distribuidos complejiza a veces el modelo. Ejemplos:
  - Uso de datos masivos y/o grandes estructuras.
  - Latencia y fallas en la comunicación.
- No existen estándares comunes bien establecidos en la industria, más bien existe una diversidad de alternativas de uso (e.g. para RPC, RMI y WS).

Casos prácticos de invocación remota

a) gRPC de Google (2015)



Fuente: https://grpc.io/docs/what-is-grpc/introduction/

- Desarrollado por Google como RPC de alto desempeño (comparado con WS) y es de código abierto (2015).
- Usa Protocol Buffers como IDL como formato de intercambio de mensaje.
- Usa HTTP/2 como transporte de datos (por esta razón, se considera también como un servicio web de alto desempeño).
- Permite interoperar programas escritos en diferentes lenguajes (e.g., C#, C++, Go, Java, Python, Ruby, etc.).
- Usado para conexión de dispositivos y navegador Web con servicios de *backend*; tb. para conexión de microservicios.
- Soporte para seguridad, balance de carga y monitoreo de los servicios gRPC.

Invocaciones en gRPC

RPC orientado a servicios

Tipos de invocación en gRPC:

- **RPC unario**: invocación simple, con posible timeout para cliente.
- **Streaming de servidor**: El servidor puede mandar múltiples respuestas.
- **Streaming de cliente**: El cliente puede mandar múltiples peticiones (por adelantado).
- **Streaming bidireccional**: Permite múltiples mensajes en ambas direcciones.

Ejemplo simple de un servicio:

Interfaz en ProtoBuffer: helloworld.proto

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```


Ejemplo de gRPC en Python

Compilar la definición del servicio Greeter en ProtoBuffer en archivo helloworld.proto mediante el siguiente comando:

- **python** -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. helloworld.proto

Genera dos archivos:

- helloworld_pb2.py # mensajes en PB
- helloworld_pb2_grpc.py # stubs de Greeter en PB

Cliente

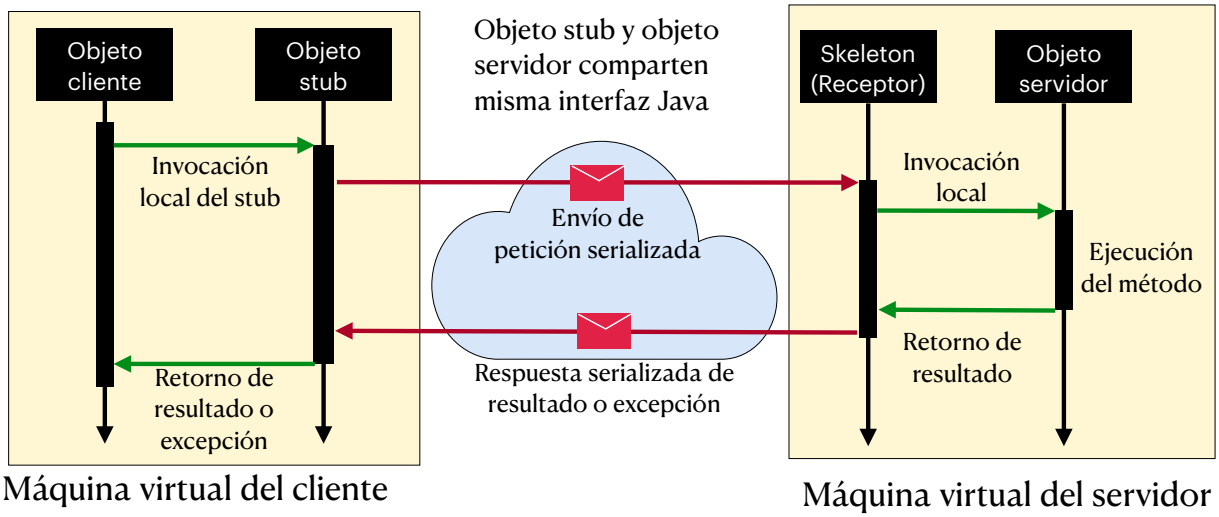
```
import grpc
import helloworld_pb2
import helloworld_pb2_grpc

def run():
    with grpc.insecure_channel('localhost:50051') as channel:
        # obtiene stub para servicio Greeter
        stub = helloworld_pb2_grpc.GreeterStub(channel)
        # invoca remotamente método SayHello
        response = stub.SayHello(
                helloworld_pb2.HelloRequest(name='World'))
    print("Greeter client received: " + response.message)

if __name__ == '__main__':
    run()
```

Servidor

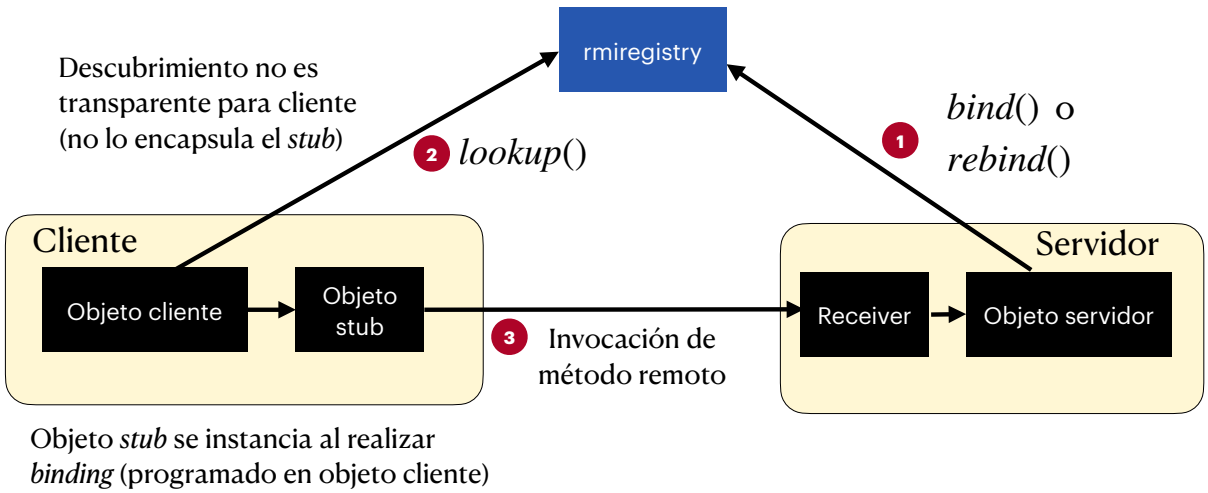
```
import grpc
from concurrent import futures
import helloworld_pb2
import helloworld_pb2_grpc

# implementa el servicio Greeter
class Greeter(helloworld_pb2_grpc.GreeterServicer):
    # implementa método SayHello
    def SayHello(self, request, context):
        return helloworld_pb2.HelloReply(
                                message=f'Hello, {request.name}!')

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    helloworld_pb2_grpc.add_GreeterServicer_to_server(Greeter(), server)
    server.add_insecure_port('[::]:50051')
    print("Server started, listening on 50051...")
    server.start()
    server.wait_for_termination()

if __name__ == '__main__':
    serve()
```

b) Java RMI (Remote Method Invocation)

Modelo de Invocación de métodos (Orientado a objetos)



Modelo de descubrimiento de Java RMI



Ejemplo de Java RMI

Servicio de cálculo remoto

Interfaz de objeto remoto (Calculo)

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

public interface ServicioCalculo extends Remote {
    <T> T ejecutarTarea(Tarea<T> t) throws RemoteException;
}
```

Interfaz de objeto parámetro (Tarea)

```
import java.io.Serializable;

public interface Tarea<T> extends Serializable {
   T ejecutar();
}
```

Fuente: http://docs.oracle.com/javase/tutorial/rmi/

Cliente del servicio de cálculo

Pasa tarea como objeto parámetro (cálculo de π)

```
import java.rmi.*;
Import java.math.*;

public class EjemploCalculoPI {
   private static String nombre = "MotorCalculo";

   public static void main(String args[]) {
          if (System.getSecurityManager() == null) {
               System.setSecurityManager(new RMISecurityManager());
          }
          try {
              Registry registry = LocateRegistry.getRegistry(args[0]);
              ServicioCalculo comp = (ServicioCalculo) registry.lookup(nombre);

              CalculoPI tarea = new CalculoPI(Integer.parseInt(args[1]));
              BigDecimal pi = comp.ejecutarTarea(tarea);
              System.out.println(pi);
          } catch (Exception e) {
               System.err.println("Excepción EjemploCalculoPI : " +  e.getMessage());
          }
   }
}
```

Autoriza acceso

Localiza y obtiene *stub* para contactar a rmiregistry

Obtiene *stub* de objeto remoto buscando con un nombre en rmiregistry

Invocación remota

Implementación del servidor de cálculo

Partida y registro del servicio en main()

```
import java.rmi.*;
import java.rmi.server.*;

// Implementación de motor de cálculo
public MotorCalculo implements ServicioCalculo
{
   private static String nombre = "MotorCalculo";

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

      // método de invocación remota
      public <T> T ejecutarTarea(Tarea<T> t)
                                  throws RemoteException {
            return t.ejecutar();
      }
// .... sigue
```

Método de invocación remota en la interfaz de ServicioCalculo

```
// ....
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
                System.setSecurityManager(new RMISecurityManager());
        }
        try {
            Calculo motor = new MotorCalculo();
            Calculo stub = (Calculo)
                                UnicastRemoteObject.exportObject(motor, 0);

            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(nombre, stub);
         } catch (Exception e) {
                System.err.println("Excepción de motor de cálculo: "
                                            + e.getMessage());
                e.printStackTrace();
        }
    }
}
```

Crea *stub* para el objeto remoto usando Socket TCP en puerto 0 (asigna automáticamente)

Registra *stub* en el servicio de registro con nombre "MotorCalculo"

3.5 Servicios Web

Servicios Web

DEFINICIÓN: Servicio Web. Formas estandarizadas para que aplicaciones distribuidas se comuniquen a través de la *web* utilizando protocolos abiertos (e.g., HTTP). En este entorno, su foco es la integración al facilitar la interacción entre plataformas y lenguajes diversos o heterogéneos.

Características clave:

- **Interoperabilidad**: Funciona en diferentes plataformas y lenguajes (e.g., Java, .NET, Python).
- **Acoplamiento flexible**: Los servicios son independientes entre sí (se conocen por una interfaz).
- **Reutilización**: Los servicios pueden fácilmente reutilizarse en múltiples aplicaciones al importr su interfaz.
- **Transporte de datos**: Se basan principalmente en HTTP(S) para conectar a aplicaciones y transferir datos.

Tipos de servicios web:

- **SOAP** (estándar de W3C y OASIS): Basado en XML y HTTP (aunque permite otros).
- **REST** (transferencia de estado representacional): Ligero, utiliza métodos HTTP.
- **Otros**: gRPC, GraphQL y WebSocket.

Casos de uso:

- Integración de sistemas (e.g., pasarelas de pago, API meteorológicas).
- SOA y Arquitectura de microservicios.
- *Backends* de aplicaciones móviles y web.

OBS: Revisar estilo arquitectónico orientado a servicios en Cap. 2

Estándares de servicios Web en XML

La base es definida por W3C y extendida por OASIS Open

W3C Web Service (www.w3c.org)

- **SOAP** (*Simple Object Access Protocol*): Protocolo de mensajería que permite el intercambio de información estructurada entre sistemas mediante mensajes en XML.
- **WSDL** (*Lenguaje de Descripción de Servicios Web*): Lenguaje basado en XML legible por máquina que se utiliza para describir servicios web (i.e., equivale a IDL en RPC).
- **UDDI** (*Universal Description, Discovery, and Integration*): API para acceder a un servicio de directorio para publicar y descubrir servicios web.

OASIS Web Service (www.oasis-open.org)

- **WS-Addressing** (Direccionamiento WS): Define cómo especificar los destinos de los mensajes en SOAP.
- **WS-ReliableMessaging**: Garantiza la entrega de mensajes en redes poco fiables.
- **WS-Security** (Seguridad de los servicios web): Marco para proteger los mensajes SOAP.
- **SAML** (Lenguaje de marcado de aserción para seguridad): Utilizado para autenticación y autorización.
- etc.

REST (REpresentational State Transfer)

Servicios Web (livianos)

DEFINICIÓN: Estilo arquitectónico, definido por R. Fielding (2000), para definir aplicaciones en red. Establece principios para el diseño, centrándose en la escalabilidad, ausencia de estado e interfaces uniformes.

Fuente: Fielding, Roy Thomas (2000). "*Representational State Transfer (REST): Architectural Styles and the Design of Network-based Software Architectures*" (PhD). University of California, Irvine.

PRINCIPIOS:

- **Modelo cliente-servidor**: Separación entre aspectos del cliente y el servidor.
- **Sin estado (*stateless*)**: Cada solicitud de un cliente debe contener toda la información necesaria; el servidor no almacena el estado de la sesión.
- **Almacenamiento en caché**: Las respuestas pueden almacenarse en caché para mejorar el rendimiento.
- **Interfaz uniforme**: Los recursos se identifican mediante URI y se manipulan con métodos (o comandos) estándares de HTTP (i.e. GET, POST, PUT, DELETE).
- **Sistema en capas (*Multi-tiered*)**: Un cliente no necesita saber si se está comunicando directamente con el servidor o a través de intermediarios (e.g., *proxies*, balanceadores de carga).

Observación:

- **RESTful** se refiere a servicios web que implementan el estilo arquitectónico REST (i.e., adhieren a sus principios y restricciones).

Mensajes de Servicios Web

Ejemplo de un Servicio con SOAP y REST

API del servicio

- **CreateUser**: Crea un nuevo usuario.
- **GetUser**: Recupera información de usuario por ID.
- **UpdateUser**: Actualiza información de un actual usuario.
- **DeleteUser**: Borra un usuario por ID.

SOAP

Request

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:usr="http://example.com/users">
  <soapenv:Header/>
  <soapenv:Body>
    <usr:GetUserRequest>
      <usr:id>123</usr:id>
    </usr:GetUserRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

Reply

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:usr="http://example.com/users">
  <soapenv:Header/>
  <soapenv:Body>
    <usr:GetUserResponse>
      <usr:user>
        <usr:id>123</usr:id>
        <usr:name>John Doe</usr:name>
        <usr:email>john.doe@example.com</usr:email>
      </usr:user>
    </usr:GetUserResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

RESTful

GET /users/123 HTTP/1.1
Host: example.com
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

```
{
  "id": 123,
  "name": "John Doe",
  "email":
"john.doe@example.com"
}
```

Comparación de SOAP y REST

	SOAP	REST
Protocolo	XML para mensajes usando HTTP, SMTP, etc.	Usa HTTP(S) con JSON (tb. XML)
Estándares	Estándares estrictos (WSDL, UDDI, WS-Security, etc.)	Flexible, siguiendo principios de arquitectura REST
Desempeño	Lento/pesado por XML (*parsing*, memoria, BW)	Más liviano y rápido; mas rápido de parsear
Seguridad	Integrada (e.g. WS-Security)	Se basa en HTTPS y OAuth
Casos de uso	Aplicaciones empresariales e interorganizacional (e.g. bancos, gobierno)	Aplicaciones web y móviles (e.g. redes sociales); microservicios.

3.6 Servicios de Mensajería (MoM)

Servicios de mensajería de Middleware

Soporte para estilo arquitectónico orientado a mensajes y/o dirigido por eventos

PROPIEDAD BÁSICA: Comunicación entre componentes por **interacción indirecta** (e.g. a través de un servicio de mensajería, tal como un *broker* o un servicios de colas de mensajes).

- **Productor** y **Consumidor** de mensajes son clientes del "servicio de mensajería".
- **Desacoplamiento** entre ambos tipos de componentes permite interacciones asíncronas e indirectas.
- **Patrones de interacción** se caracterizan por *fan-in* y *fan-out* y/o semántica de consumo de mensajes.

CARACTERISTICAS:

- **Entrega de mensajes**. Durabilidad, ordenamiento y garantía de entrega (e.g. *At-most-once*, *Exactly-once*).
- **Calidad de servicio**. Rendimiento, escalabilidad y tolerancia a fallos.
- **Enrutamiento (*routing*)**. Se requiere si servidores definen una red sobrepuesta (*overlay*).
- **Transformación**. Uso de pasarelas (*gateways*) apoya interoperabilidad entre diferentes protocolos.

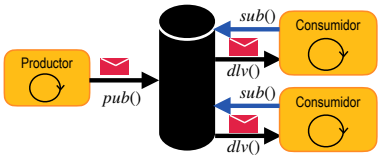
Ejemplos: Casos de uso

- **Arquitectura dirigida por eventos**. Notificación a múltiples servicios sobre eventos a observar y/o procesar (e.g., orden de compra realizada, nuevo usuario registrado).
- **Arquitectura de microservicios**. Habilitación de comunicación entre microservicios débilmente acoplados. También se usa para SOA.
- **Colas de tareas**. Distribución de tareas y balance de carga entre trabajadores (*workers*)
  - e.g., tareas como procesamiento de imágenes y envío de correo electrónico.
- **Monitoreo de un sistema**. Recopilación y procesamiento de registros de *log* desde múltiples fuentes (e.g., análisis de desempeño y detección de anomalías).
- **Análisis en tiempo real**. Recolección, procesamiento y análisis de datos en tiempo real (e.g., datos de sensores IoT, posicionamiento GPS en control de móviles o vehículos, sistemas de telemetría y control de procesos).

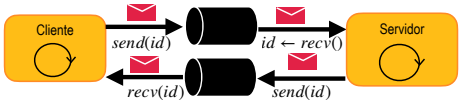
Patrones básicos de comunicación

- **Punto a punto** (PTP: patrón *Message Channel*). Cada mensajes es entregada y consumido por exactamente un único consumidor.
- **Publicar-Suscribirse** (PUB-SUB: patrón *Publish-Suscribe*). Mensajes se transmiten a múltiples consumidores a través de temas (*topics*) a los cuales están suscritos.
- **Solicitud- Respuesta** (REQ-REP: patrón *Request-Reply*). Productor envía un mensaje y espera respuesta de un consumidor. Para calzar mensajes se usa ID de correlación.

a) ***Queue*** (PTP)



b) ***Topic*** (PUB-SUB)



c) ***Request-Reply*** (REQ-REP)

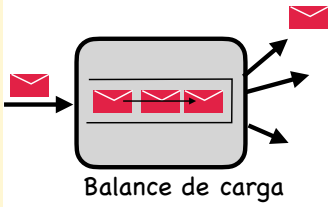
Ejemplo: Patrón *Competing Consumers* (tb. *pipeline*)

ZeroMQ: balanceador de carga para procesamiento paralelo

Productor (*push*)

```
import zmq

context = zmq.Context()
socket = context.socket(zmq.PUSH)
socket.bind("tcp://*:5557")

for task in range(10):
   print(f"Enviado tarea {task}...")
   socket.send_string(str(task))
```

Generador de tareas (chief)



Consumidor(*pull*)

```
import zmq

context = zmq.Context()
socket = context.socket(zmq.PULL)
socket.connect("tcp://localhost:5557")

while True:
   task = socket.recv_string()
   print(f"Recibiendo tarea: {task}")
   # Procesar la tarea aquí
```

Trabajador (worker)

Características:

- Existen varios consumidores, pero cada mensaje de la cola es consumido por uno único consumidor, con recepción explícita y sincronizada (sólo modo *pull*). Productor realiza envío asincrónico (modo *push*).
- Promueve mayor escalabilidad y balance de carga.
- ZeroMQ es sin Broker y sin persistencia de mensajes (i.e., se puede perder una tarea si un trabajador falla).

Estándares y API de MQS o MoM

Principales estándares en la industria

- **AMQP** (*Advanced Message Queuing Protocol*). Estándar abierto de OASIS & ISO.
  - Modelos PTP, PUB-SUB y REQ-REP. También incluye routing, transacciones, seguridad, clustering y federación.
  - Implementaciones: Apache ActiveMQ, Apache Qpid, IBM MQ, RabbitMQ, Azure Service Bus.
- **JMS** (*Jakarta Messaging Service*). API que es parte de plataforma JEE de Java/Oracle.
  - Modelos PTP y PUB-SUB. Idóneo para aplicaciones empresariales. Integra JNDI para nombramiento y descubrimiento de colas y proveedores.
  - Implementaciones: Amazon SQS, Apache ActiveMQ, JBoss Messaging de Red Hat, RabbitMQ (extendido). IBM WebSphere.
- **MQTT** (*MQ Telemetry Transport*). Estándar ISO/IEC 20922, presentado por IBM y apoyada por OASIS.
  - Provee transporte de mensajes liviano y fiable (*at-most-once*, *at-least-once* & *exactly-once*) de tipo PUB-SUB sobre TCP/IP. Idóneo para IoT y monitoreo en tiempo real.
  - Implementaciones: Eclipse Mosquitto, HiveMQ; AWS IoT Core, Google Cloud IoT Core & MS Azure IoT Hub

a) Ejemplo: PTP con RabbitMQ en Python

Productor

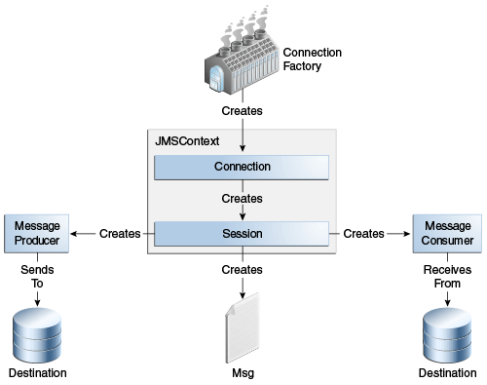
```python
import pika

# Establish a connection to RabbitMQ
connection = pika.BlockingConnection(
                        pika.ConnectionParameters('localhost'))
channel = connection.channel()

# Declare a queue (create if it doesn't exist)
queue_name = 'hello_queue'
channel.queue_declare(queue=queue_name)

# Send a message to the queue
message = "Hola, un mensaje en RabbitMQ!"
channel.basic_publish(
    exchange='',  # Use the default exchange
    routing_key=queue_name,  # Specify the queue name
    body=message
)
print(f"Sent: {message}")

# Close the connection
connection.close()
```

Consumidor

```python
import pika

# Establish a connection to RabbitMQ
connection = pika.BlockingConnection(
                                pika.ConnectionParameters('localhost'))
channel = connection.channel()

# Declare the same queue (create if it doesn't exist)
queue_name = 'hello_queue'
channel.queue_declare(queue=queue_name)

# Callback function to process received messages
def callback(ch, method, properties, body):
    print(f"Received: {body.decode()}")

# Start consuming messages from the queue
channel.basic_consume(
    queue=queue_name,
    on_message_callback=callback,
    auto_ack=True  # Automatically acknowledge messages
)

print("Waiting for messages. To exit, press CTRL+C")
channel.start_consuming()
```

b) Caso: JMS de JEE

Modelo de la API



Fuente: https://docs.oracle.com/javaee/7/tutorial/partmessaging.htm

- **Connection Factory**: Crea conexiones a un determinado proveedor.
- **Destination**: Especifica referencia para destino de los mensajes, que puede ser una cola (PTP) o un tópico (PUB-SUB), dos estilos de interacción.
- **Connection**: Representa una conexión con un determinado proveedor JMS, que permite crea múltiples sesiones con este proveedor.
- **Session**: Un contexto de una única hebra, para producir y consumir mensaje (mediante creación de productor, consumidor y mensaje).
- **Message Producer**: Objeto creado por la sesión para mandar mensajes a un destino especificado (i.e., corresponde al *endpoint* de un productor).
- **Message Consumer**: Objeto creado por la sesión para recibir mensajes de un destino especificado (i.e., corresponde al *endpoint* de un consumidor).
- **Message**: Objeto estructurado creado por la sesión que permite manejar diferentes tipos de datos en los mensajes.
- **Message Listener**: Objeto que en un consumidor permite recibir mensajes asincrónicamente, usando método `onMessage()`. Permite implementar el patrón de "consumidor dirigido por eventos" tipo *push*.

Ejemplo JMS: productor de mensajes

```
// 0. Obtener el contexto del JNDI
Context ctx = new InitialContext(ctxProps);

// 1. Buscar en JNDI la fábrica del driver para JMS
QueueConnectionFactory  qcf = (QueueConnectionFactory) ctx.lookup("ConnectionFactory");

// 2. Usar driver para crear una conexión de tipo PTP
QueueConnection con = qcf.createQueueConnection();
con.start();

// 3. Usar la conexión para crear una sesión
 QueueSession sesion = con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

// 4. Crear destino, buscando en JNDI por nombre de la cola
Queue queue = (Queue) ctx.lookup("queue/testQueue");

 // 5. Crear un endpoint para un productor de mensajes
QueueSender sender = sesion.createSender(queue);

// 6. Construir y enviar el mensaje
TextMessage msg = sesion.createTextMessage();
msg.setText("Este es el contenido del mensaje");
sender.send(msg);
```

Obtener contexto para servicio de directorio con JNDI

Obtener desde servicio de directorio *driver* para fábrica de conexiones PTP

Crear sesión para conexión, no transaccional y con ACK automático

Crear *stub* de destino para cola ligada a cierto nombre.

Crear *endpoint* para envío de mensajes al destino señalado

Enviar mensajes

Streaming de eventos

DEFINICIÓN: Sistema dirigido por eventos generados continuamente, que se almacenan en un *log* y se procesan en tiempo real (*near Real Time*).

Casos de uso:

- Sistemas IoT, Redes de sensores y sistemas de telemetría; microservicios.

Características claves:

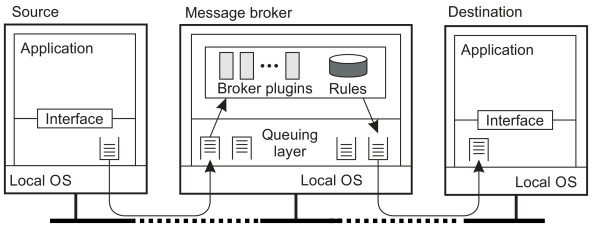
- Desacoplamiento: Productores y consumidores son independientes.
- Flujo continuo. Eventos se generan continuamente y se procesan a medida que se reciben.
- Procesamiento en tiempo real. Se requiere baja latencia (e.g. $10^{-3} - 10^{1}$ [s]).
- Persistencia. Eventos en el *log* o *stream* se pueden reproducir, siendo tolerante a fallos.
- Escalabilidad. Sistema puede manejar altos volúmenes de eventos y escalar horizontalmente.

Streaming de eventos con *Broker* basado en *Log*

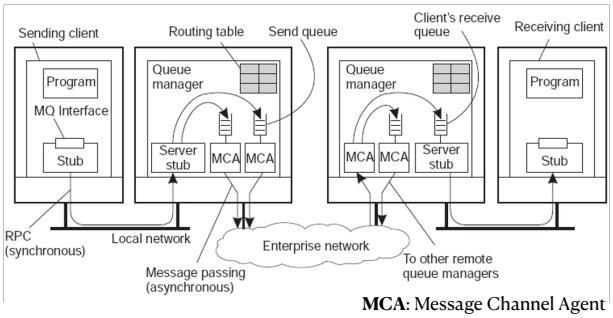
Generación y procesamiento continuo de eventos en tiempo real

Patrón	Descripción	Casos de uso
Pub-Sub	• Productor-consumidores suscritos reciben evento en una relación *one-to-many* (como *multicasting*). • Eventos persisten en el *log* para durabilidad.	• Notificaciones en RT • e.g., precios de acciones, actualización en RR.SS.
Consumidores competidores	• Consumidores se agrupan para procesamiento paralelo. • Cada evento es procesado por un único consumidor.	• Balance de carga • e.g. procesamiento de imágenes en sistema de vigilancia.
Fuente de eventos (*sourcing*)	• *Log* actúa como registro de eventos inmutables. • Consumidores reproducen eventos para reconstruir un estado o recuperarse de posibles fallos.	• Auditoría, recuperación de estado consistente • e.g. transacciones financieras

Arquitectura de Mensajería



a) Organización general de un *Broker* de mensajes



b) Arquitectura de IBM Websphere tipo MQS (MoM / JMS)

Fuente: van Steen & Tanenbaum (2017 & 2023)

Algunos Productos MQS o MoM

ABIERTOS

- **Apache ActiveMQ**: *Broke*r que admite JMS. Usado en ESB. Escrito en Java.
- **Apache Kafka**: Mensajería de eventos con alto rendimiento y baja latencia (*stream processing*). Escrito en Java+Scala.
- **RabbitMQ (VMware)**: *Broker* admite múltiples protocolos (e.g., AMQP, JMS y MQTT). Escrito en Erlang.
- **Apache Pulsar**: Parecido a RabbitMQ para mensajes y a Kafka para *stream processing*.

COMERCIALES

- IBM MQ, Microsoft MSMQ, Oracle Advanced Queuing, Solace PubSub+, Tibco EMS, Progress SonicMQ.

PROVEEDORES CLOUD

- **Amazon SQS**: un servicio administrado por AWS.
- **Google Pub/Sub**: un servicio escalable para sistemas basados en eventos.
- **Azure Service Bus**: un servicio multipropósito.

3.7 Conclusiones

Lecciones

1. Se ha reconocido los conceptos relevantes de programación distribuida, los paradigmas y los modelos de comunicación entre programas (e.g. control, memoria y mensaje).
2. Se ha revisado la programación con *sockets* tipo UDP, TCP y multicast UDP.
3. A nivel de comunicación de *Middleware*, se ha revisado la programación con invocaciones remotas, como base para la programación de servicios de Middleware.
4. Se ha revisado también a nivel de *Middleware*, programación con servicios Web y de mensajería o dirigidos por eventos.

Material de estudio complementario

- **van Steen (2023)**. Texto guía.
  - Cap IV: Communication (4.1-4.3: pp. 181-232, no se incluye MPI)
- **Coulouris (2012)**. Texto complementario (en muchos temas profundiza más).
  - Cap. IV: Interprocess Communication (pp. 145-173: excluye 4.4.2, 4.5 y 4.6)
  - Cap. V: Remote Invocation (pp. 185-228)
  - Cap. VI: Indirect Communication (6.1-6.2.1: pp. 229-235; 6.3 y 6.4: pp. 242-262)

UNIVERSIDAD TECNICA FEDERICO SANTA MARIA | DEPARTAMENTO DE INFORMÁTICA

Capítulo III:
Comunicación en Programación distribuida
Modelos, mecanismos y protocolos de comunicación

Prof. Dr.-Ing. Raúl Monge Anwandter ♦ 2º semestre 2025