04 - Comunicación

IIC2523 - Sistemas Distribuidos

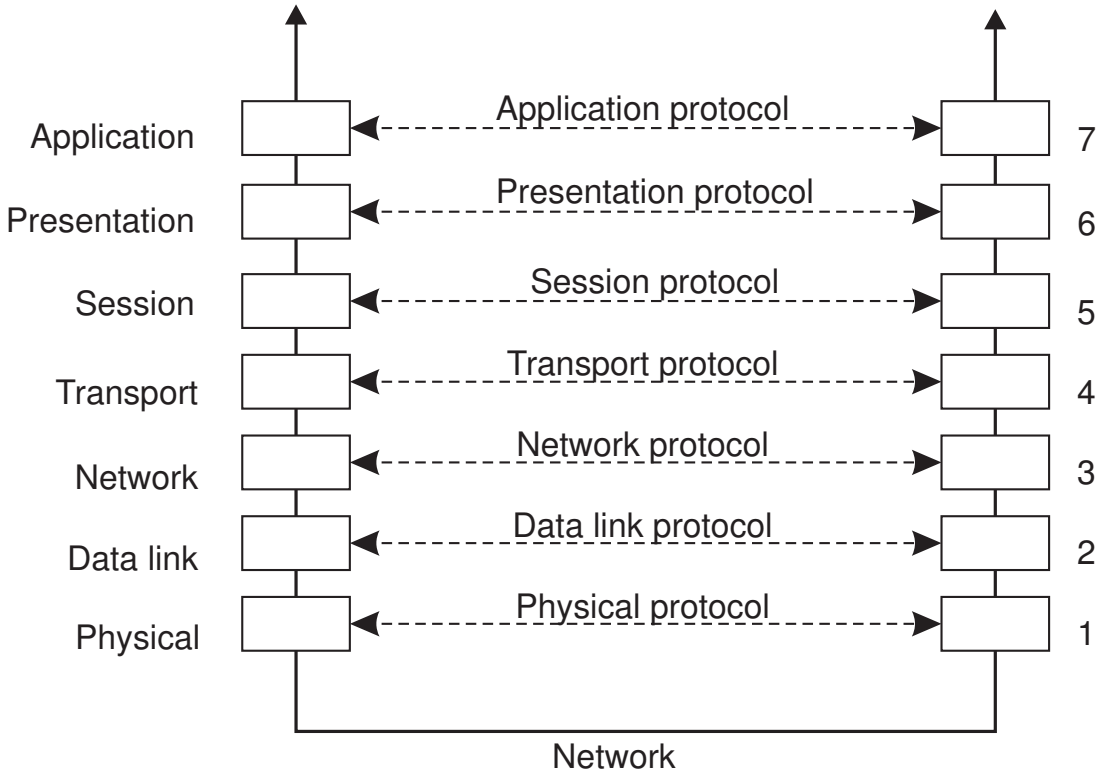
Cristian Ruz – cruz@ing.puc.cl

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

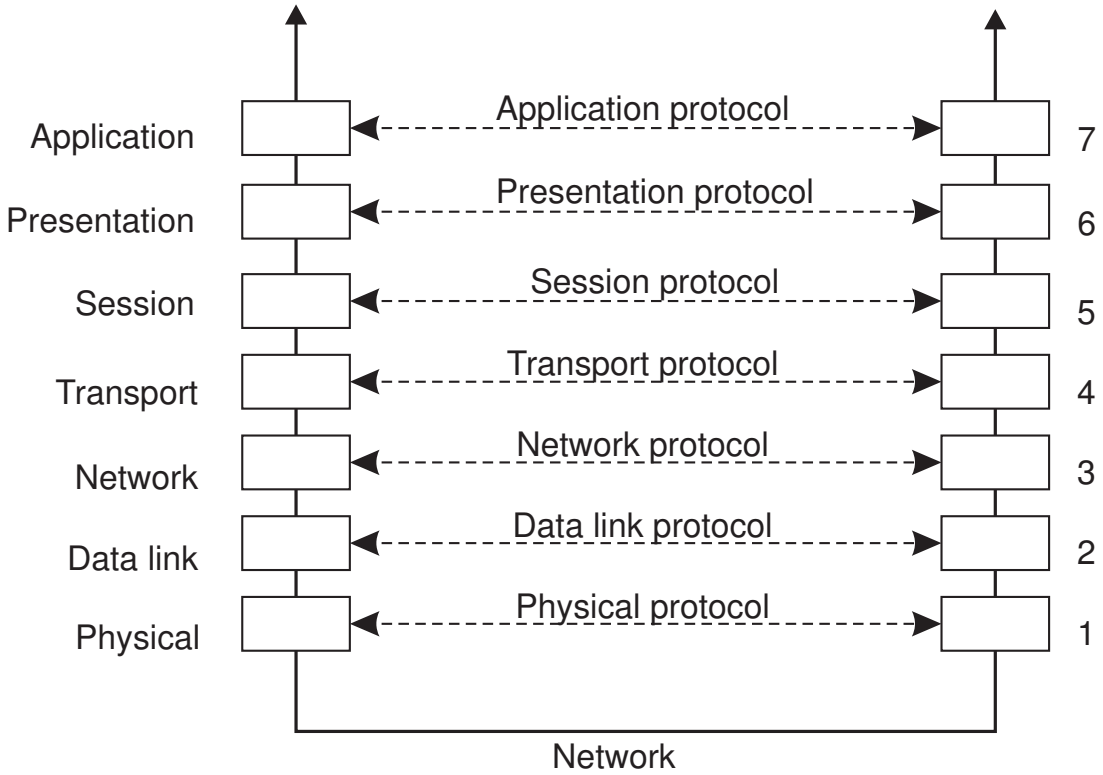
Semestre 2-2020

Contenidos

Modelo de capas



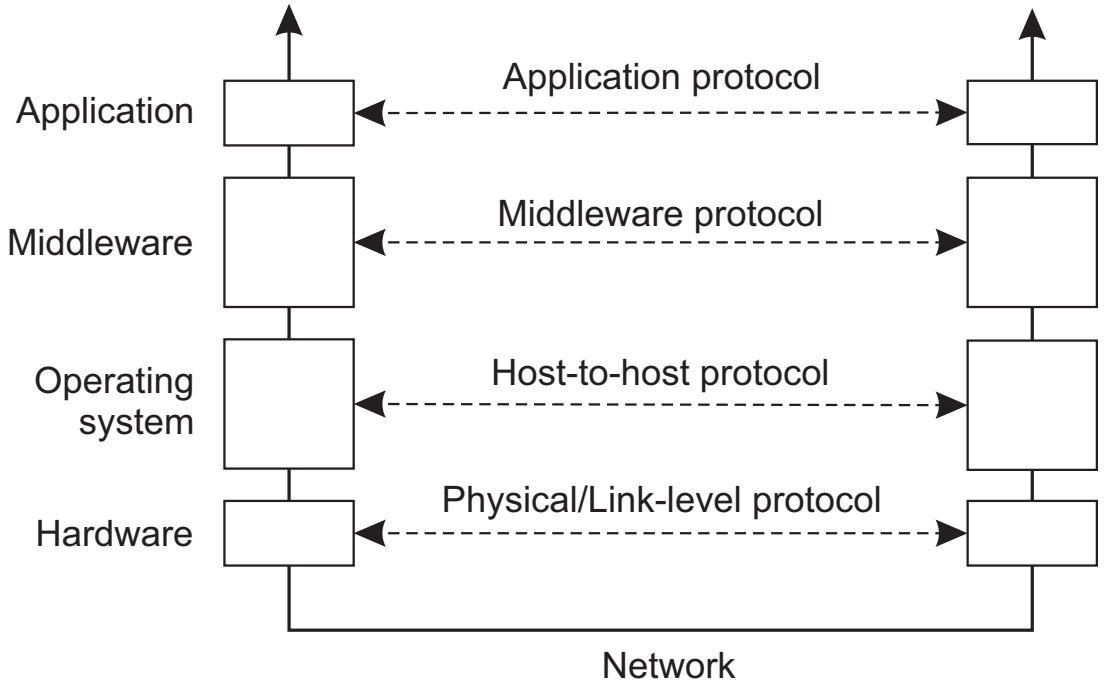
Modelo de capas



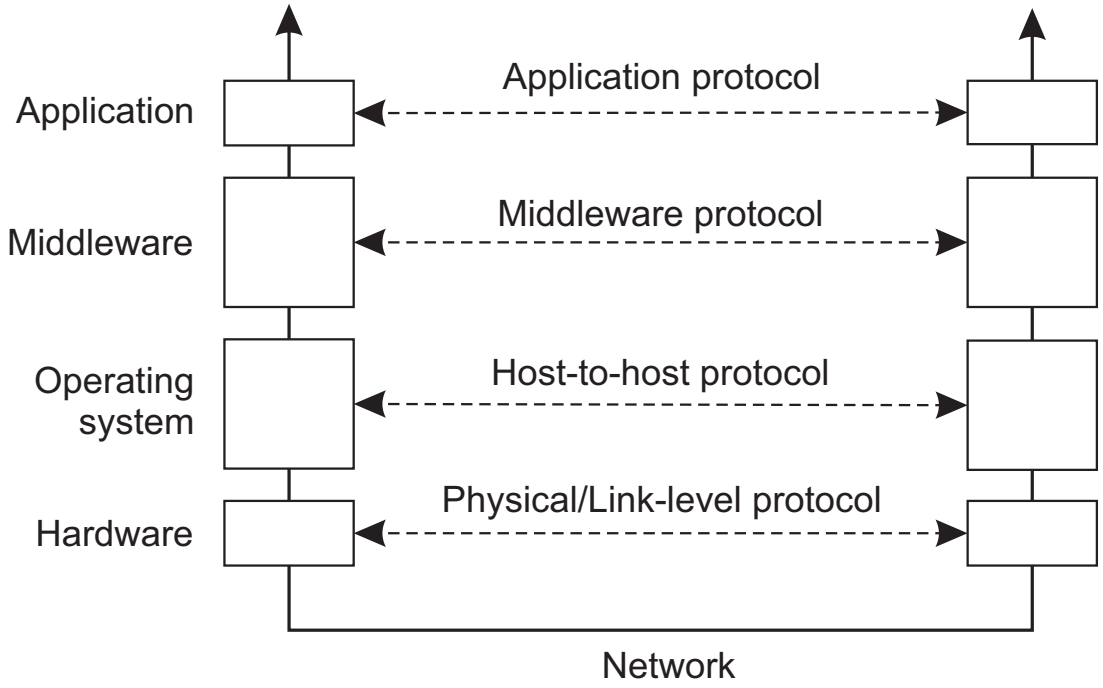
Modelo orientado a transmisión de mensajes

- ¿Dónde se implementa el *middleware*?
- ¿Provee la transparencia que necesitamos?

Middleware


Application
Application protocol
Middleware
Middleware protocol
Operating system
Host-to-host protocol
Hardware
Physical/Link-level protocol
Network

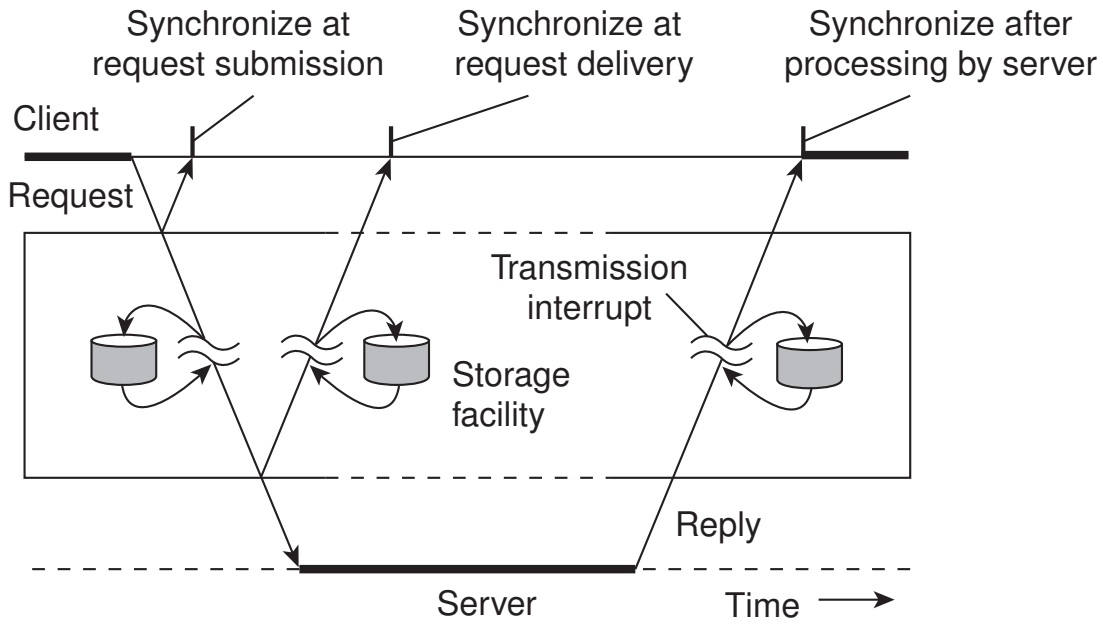
Middleware



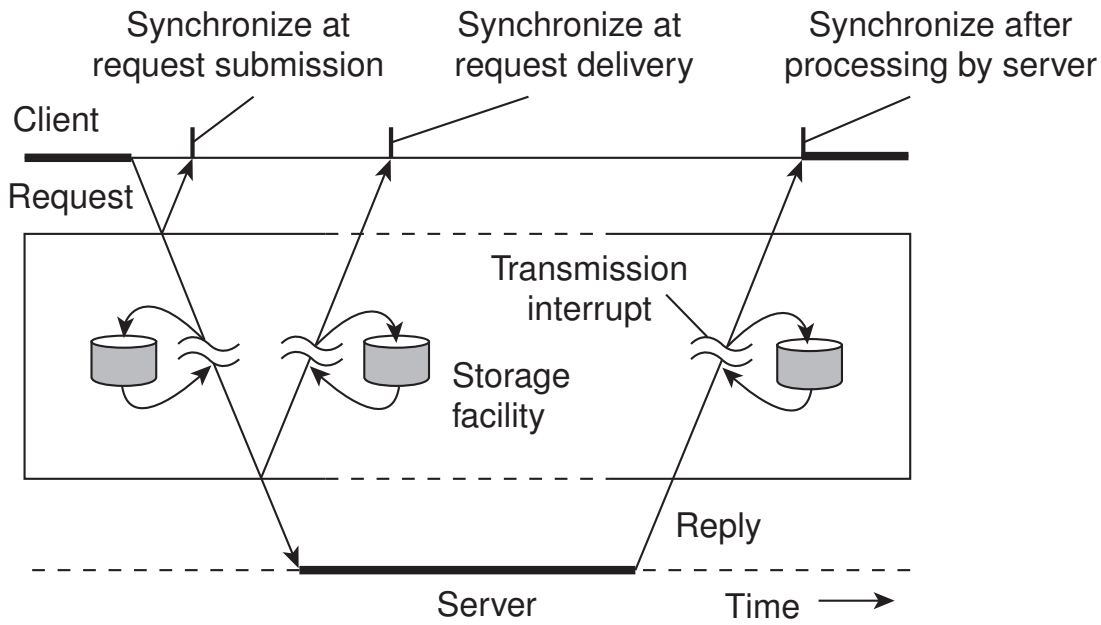
Trabajo del *middlware*

- Protocolos de comunicación
- *Marshalling* / *Unmarshaling* de mensajes
- Esquemas para **nombrar** elementos y poder compartirlos
- Protocolos de **seguridad**
- Protocolos de **escalamiento** (replicación, *caching*, . . .)

Tipos de comunicación



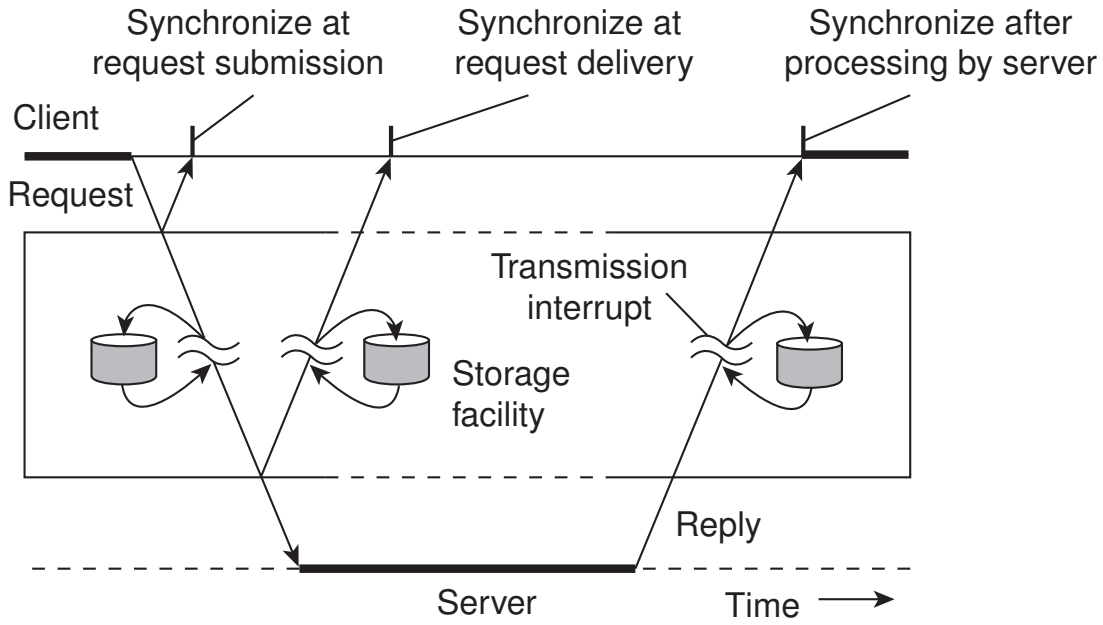
Tipos de comunicación



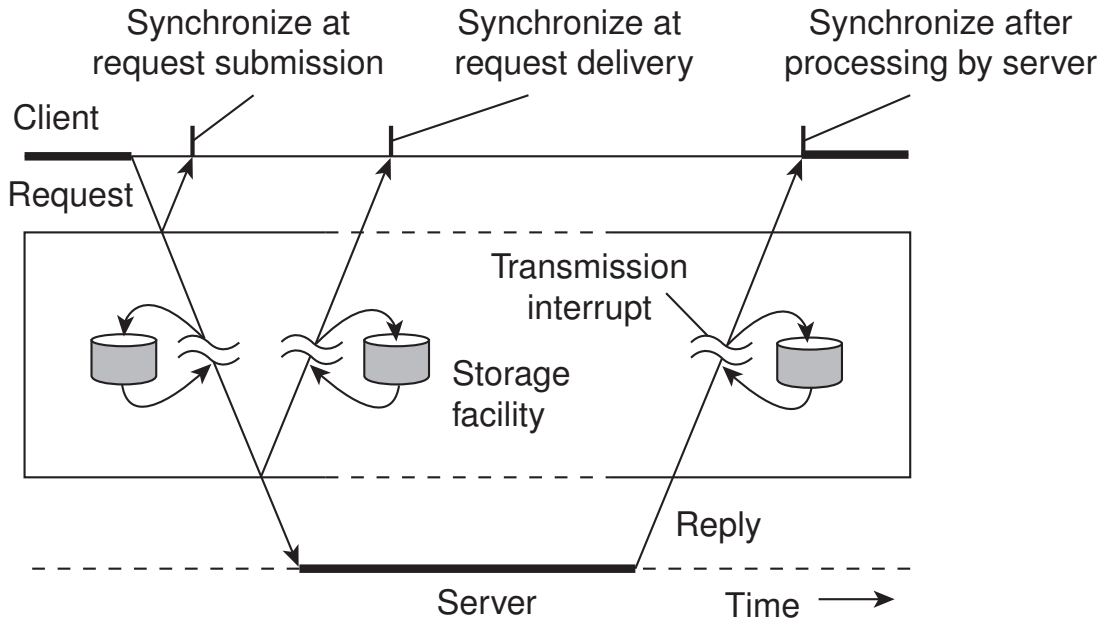
Comunicación transiente o persistente

- **Transiente**: servidor descarta el mensajes si no puede ser transmitido
- **Persistente**: servidor mantiene el mensaje hasta que el receptor está disponible

Tipos de comunicación



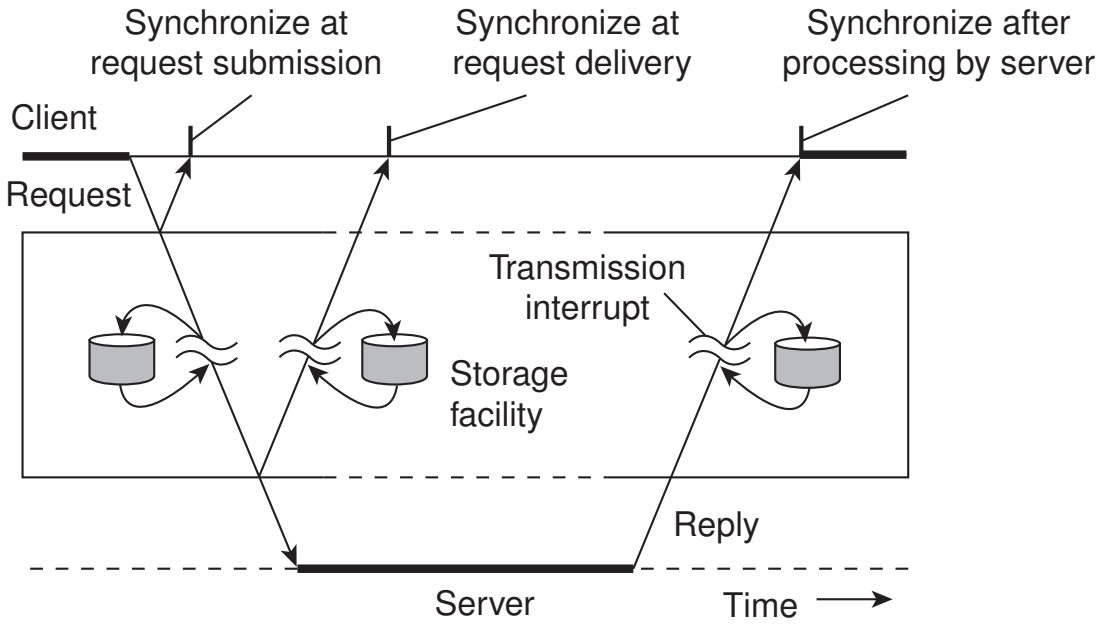
Tipos de comunicación



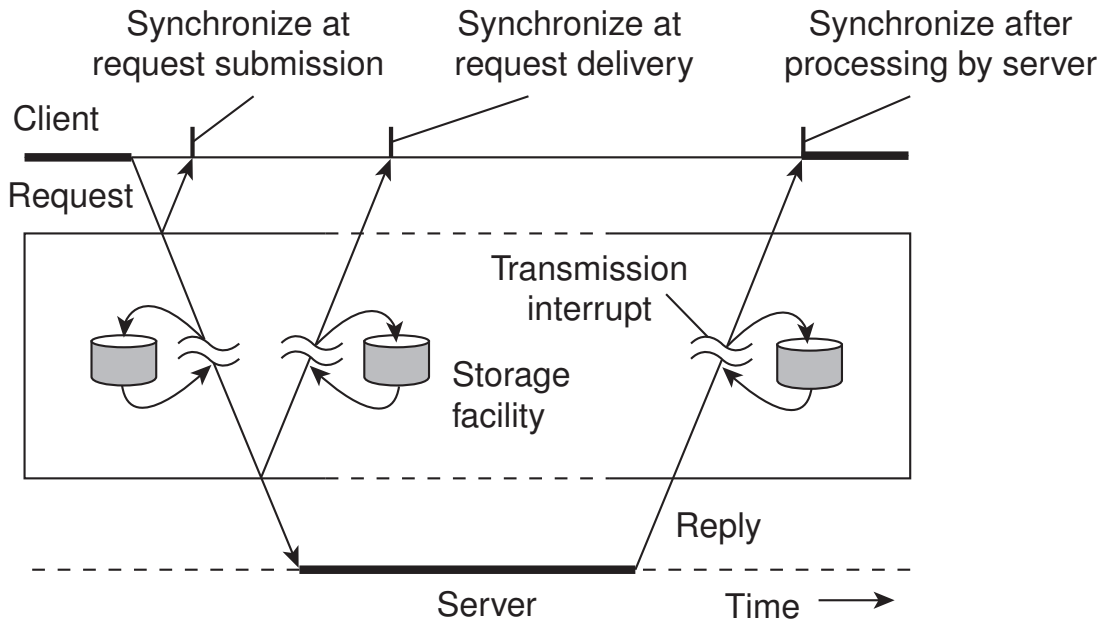
¿Dónde se produce la sincronización?

- Después que el mensaje **se envía**
- Después que el mensaje **se entrega**
- Después que el mensaje **se procesa**

Tipos de comunicación



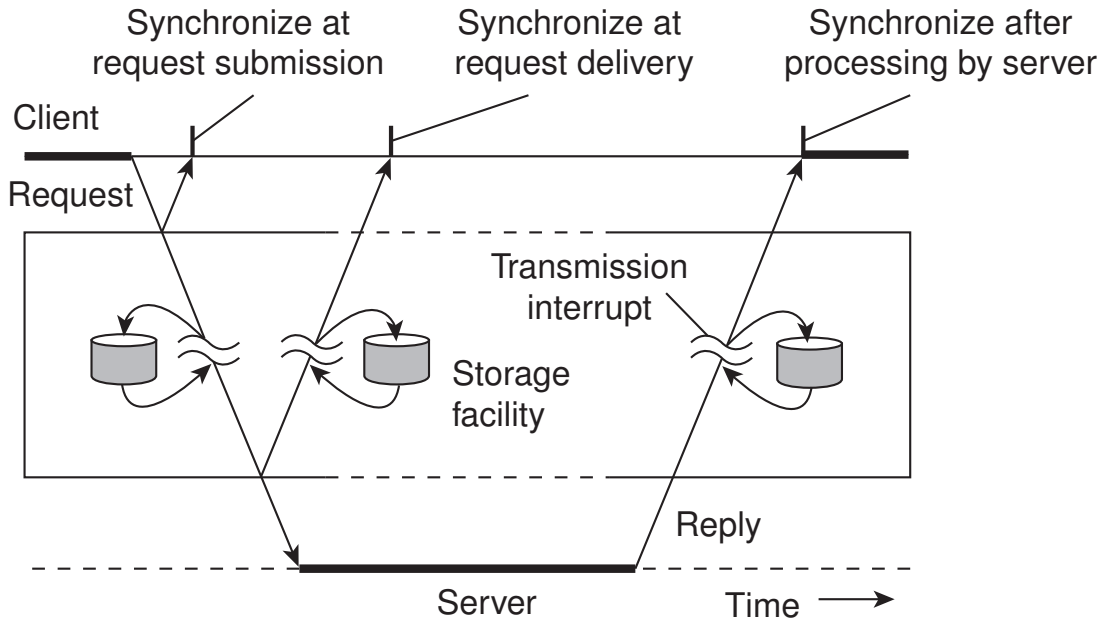
Tipos de comunicación



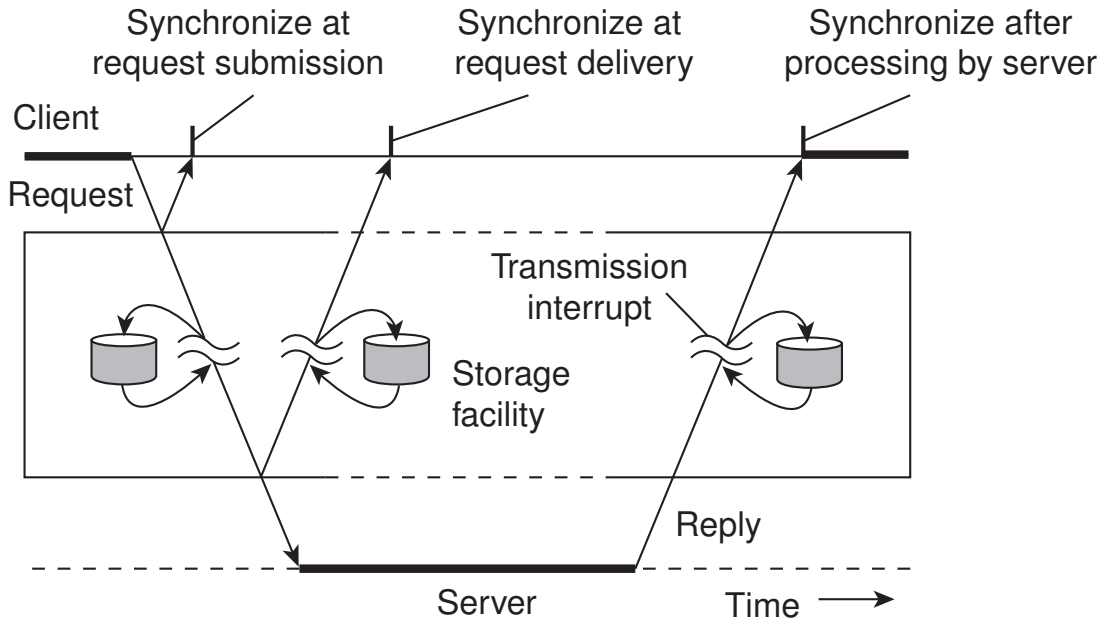
Comunicación tradicional: modelo **síncrono transiente**

- Cliente y servidor deben estar activos
- Cliente se bloquea hasta recibir respuesta
- Servidor espera solicitudes y las procesa
- Cliente no puede hacer nada mientras servidor no responda
- Fallas deben manejarse inmediatamente
- ¿Qué aplicaciones funcionan así?

Tipos de comunicación



Tipos de comunicación

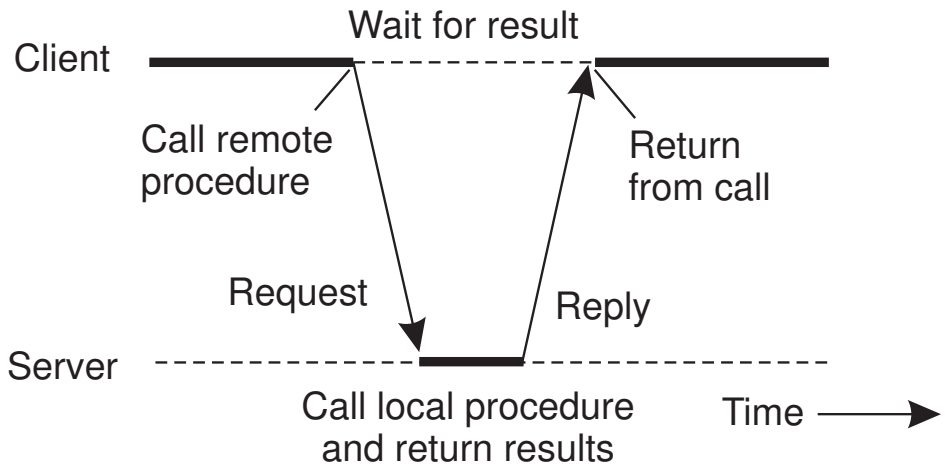


Message-oriented middleware: **asíncrono persistente**

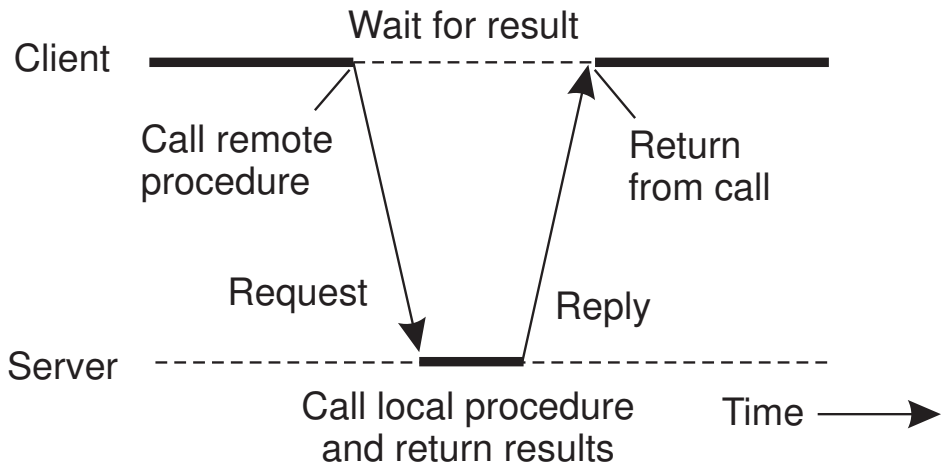
- Servidor encola mensajes
- Cliente puede hacer otras cosas mientras espera respuesta
- *Middleware* puede ocuparse la tolerancia a fallas

Contenidos

RPC: *Remote Procedure Call*



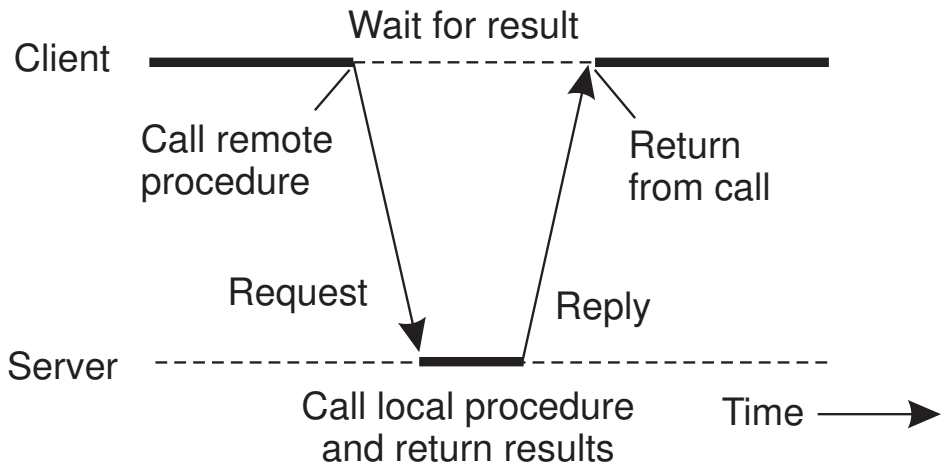
RPC: *Remote Procedure Call*



RPC: *Remote Procedure Call*

- Modelo familiar para el programador
- Procesos "bien diseñados" pueden ejecutar de manera aislada (*black box*)

RPC: *Remote Procedure Call*

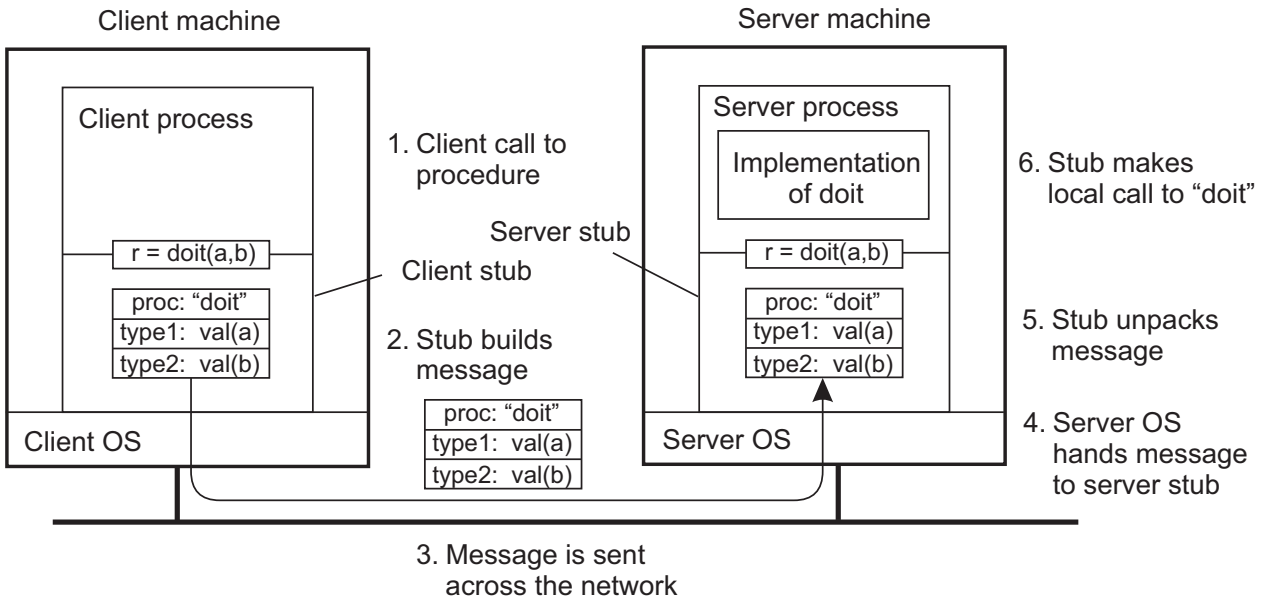


RPC: *Remote Procedure Call*

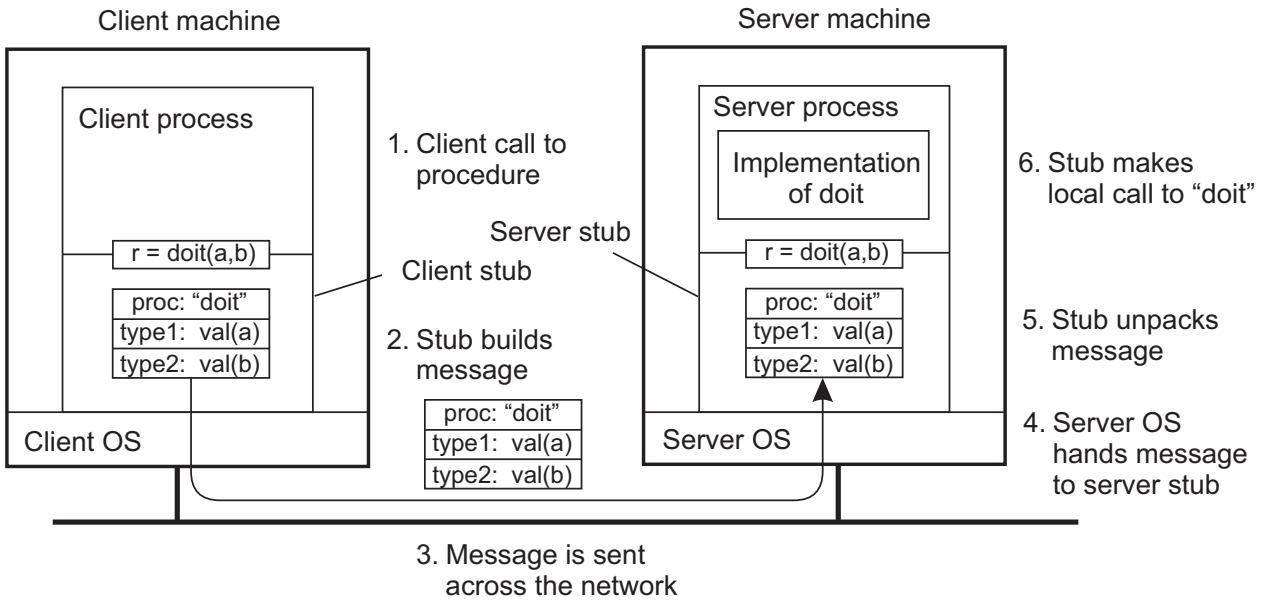
- Modelo familiar para el programador
- Procesos "bien diseñados" pueden ejecutar de manera aislada (*black box*)

Esconder la comunicación entre **emisor** y **receptor** detrás de un mecanismo de **llamadas a procedimientos**

RPC: *Remote Procedure Call*

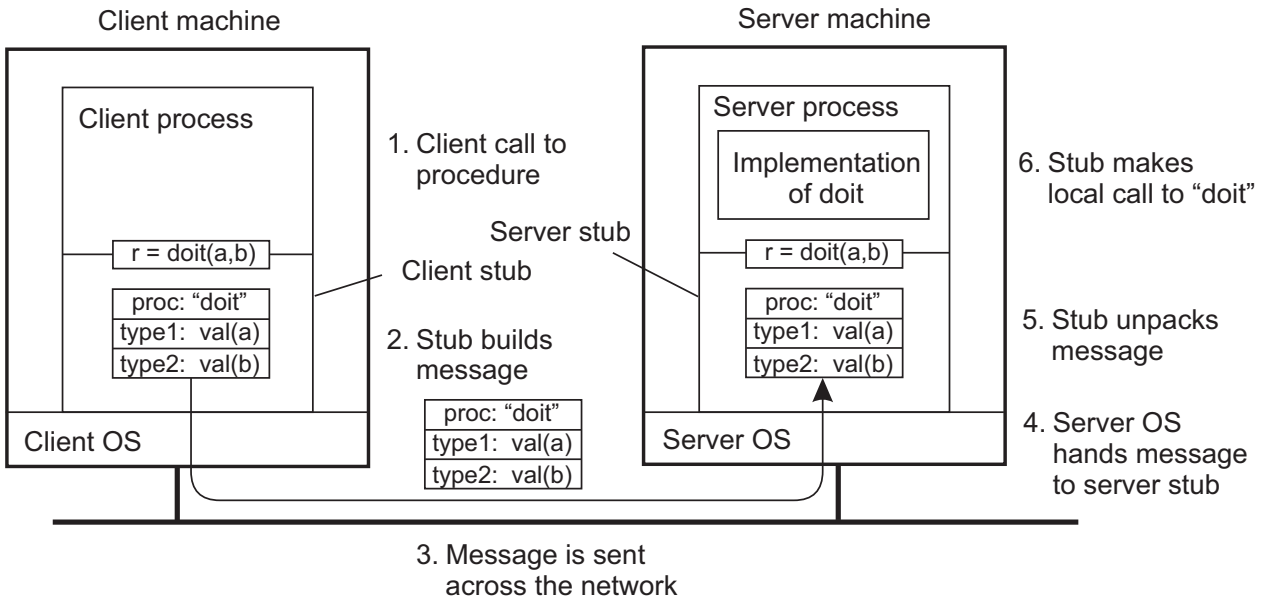


RPC: *Remote Procedure Call*



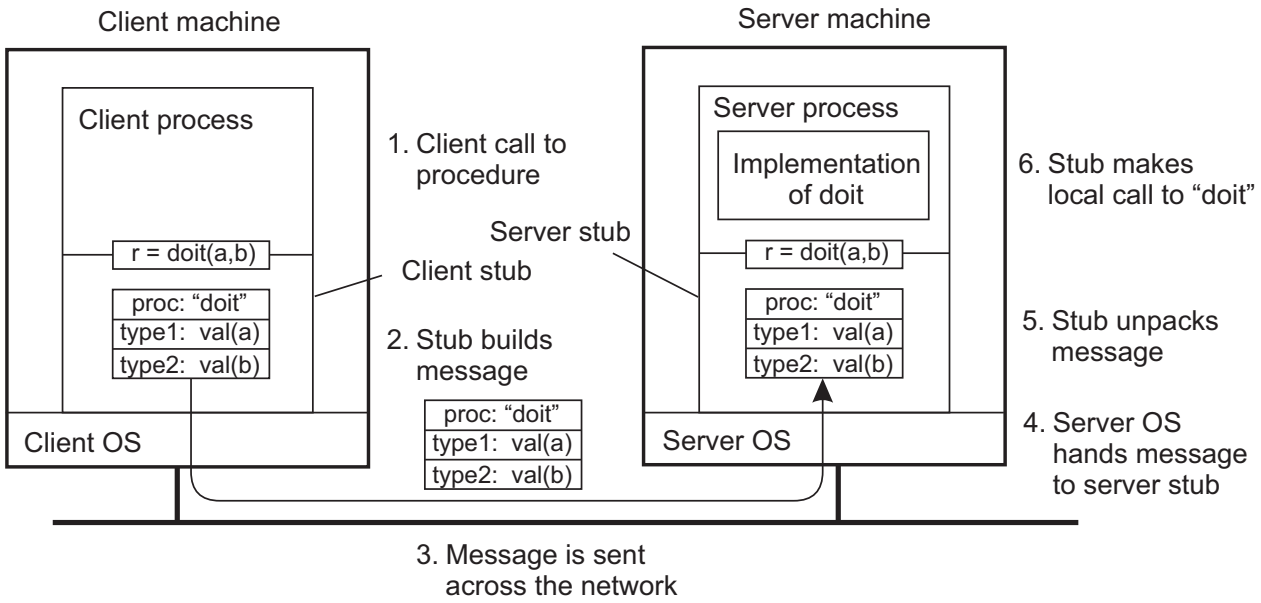
1. Procedimiento de cliente llama a **client stub**
2. **Client stub** arma mensaje (*marshall*) y llama a **S.O. cliente**
3. **S.O. cliente** envía mensaje a **S.O. remoto**
4. **S.O. remoto** pasa mensaje a **server stub**
5. **Server stub** desempaqueta (*unmarshall*) argumentos y llama a servidor

RPC: *Remote Procedure Call*

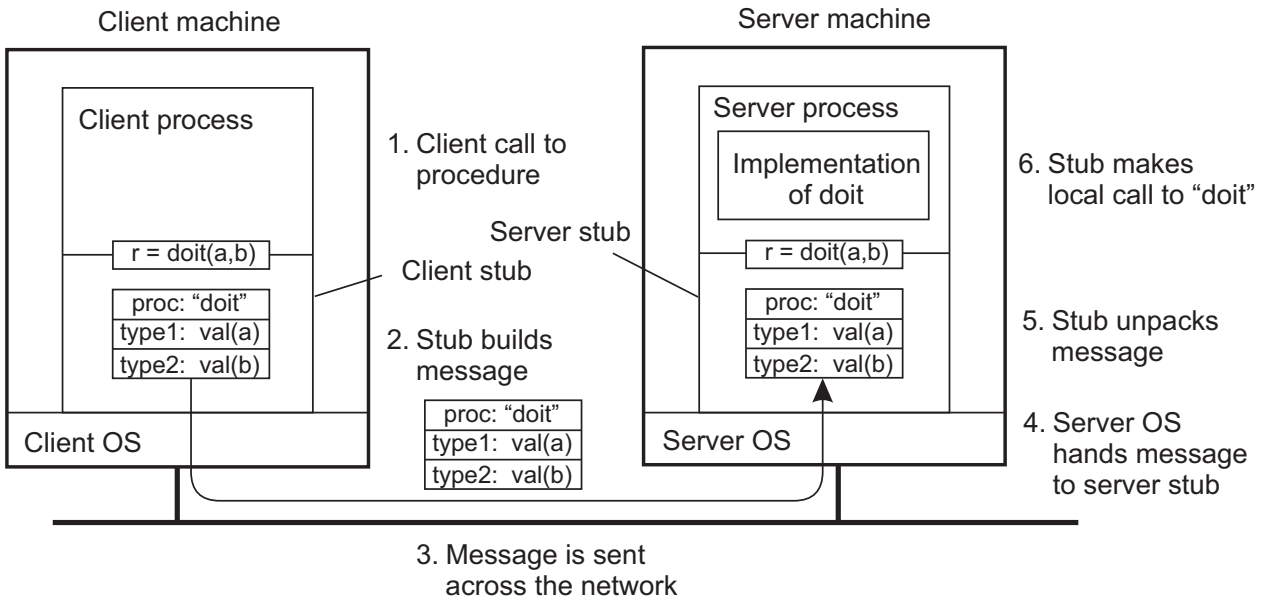


1. Servidor ejecuta llamado local, y entrega resultado a **server stub**
2. **Server stub** arma mensaje (*marshall*) y llama a **S.O. remoto**
3. **S.O. remoto** envía mensaje a **S.O. cliente**
4. **S.O. cliente** pasa mensaje a **client stub**
5. **Client stub** desempaqueta (*unmarshall*) argumentos y retorna al cliente

RPC: Paso de parámetros



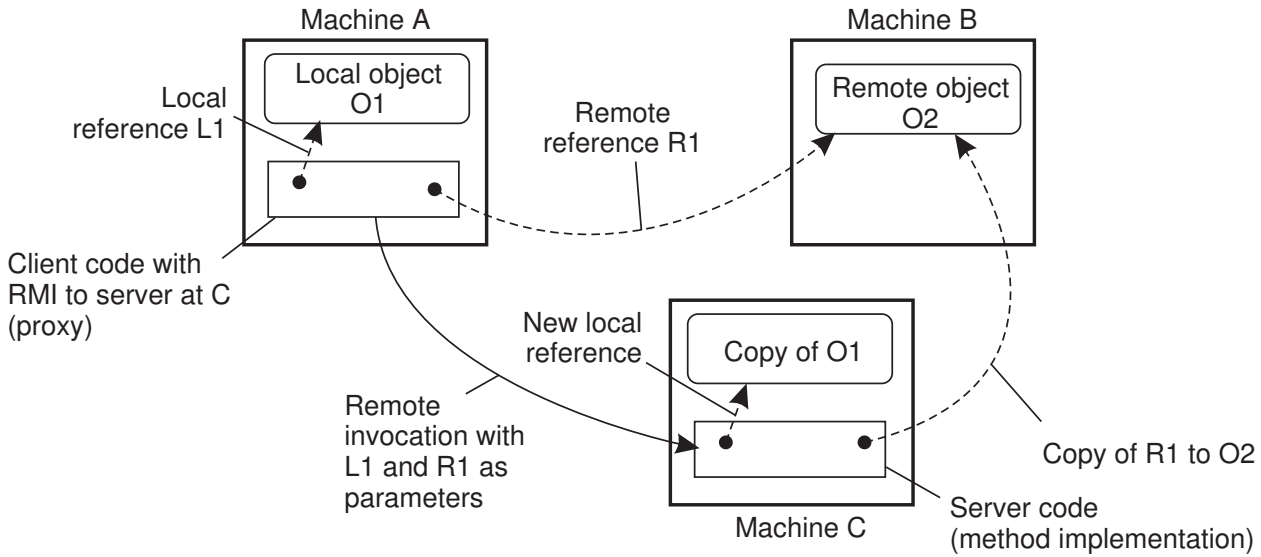
RPC: Paso de parámetros



No es cosa de solo empaquetar parámetros

- ¿Cómo representan los datos cliente y servidor?
- Se debe acordar la representación y el *encoding*
- ¿Cómo representar tipos primitivos?
- ¿Cómo representar tipos complejos?
- ¿Puedo pasar direcciones de memoria o referencias?

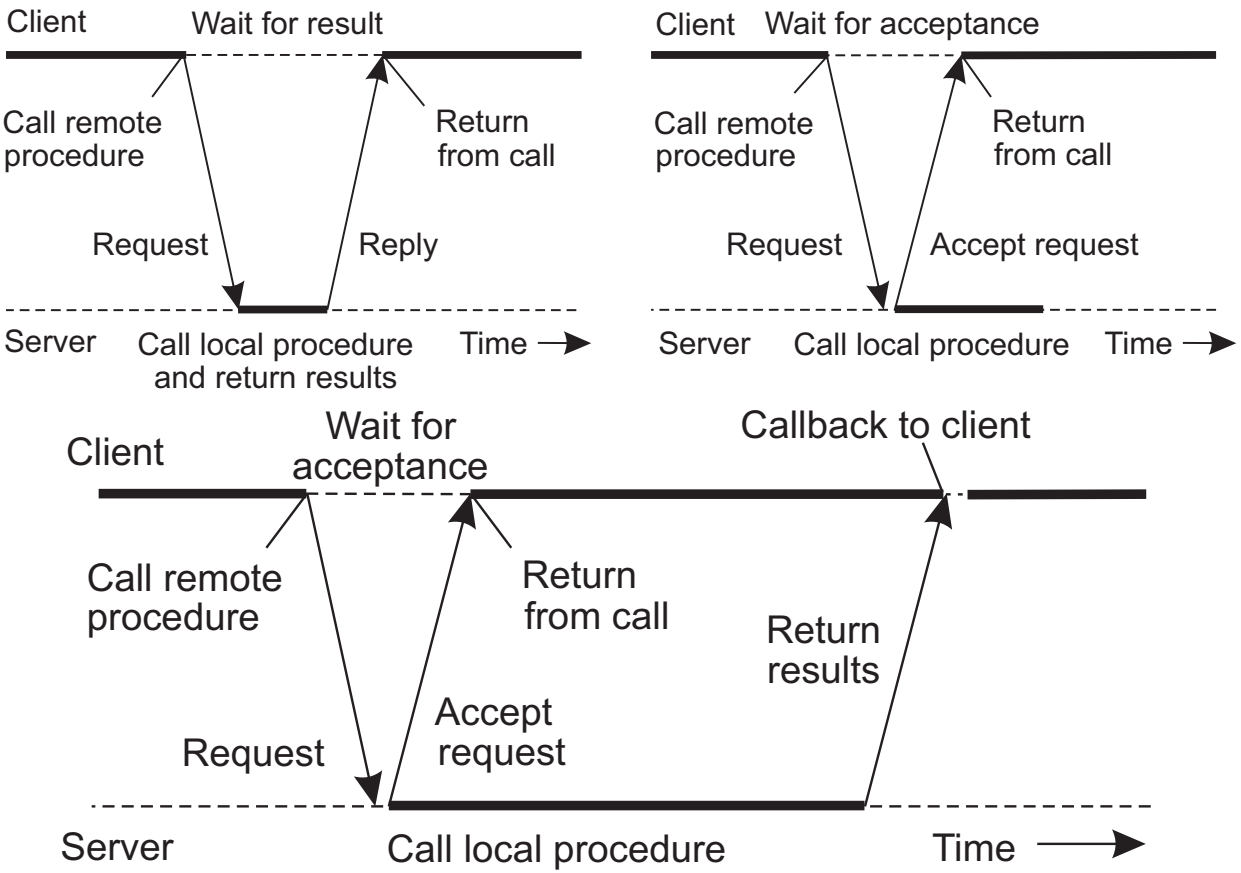
RPC: Paso de parámetros



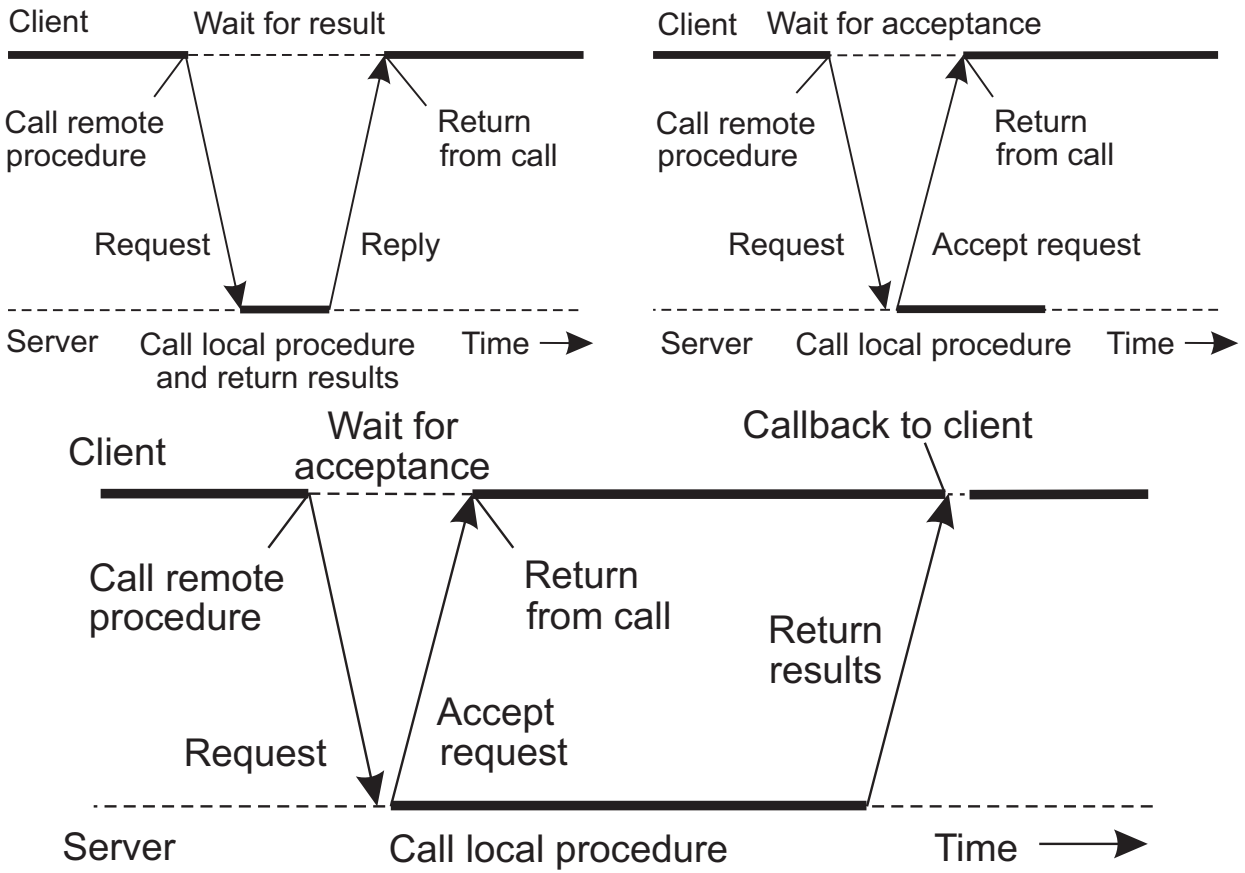
No es cosa de solo empaquetar parámetros

- Semántica **copy in/copy out**
- Todo se copia
- No se puede asegurar transparencia total de acceso
- ¿Cómo manejar referencias remotas?
- ¿Puedo pasar referencias remotas?

RPC Asíncrono



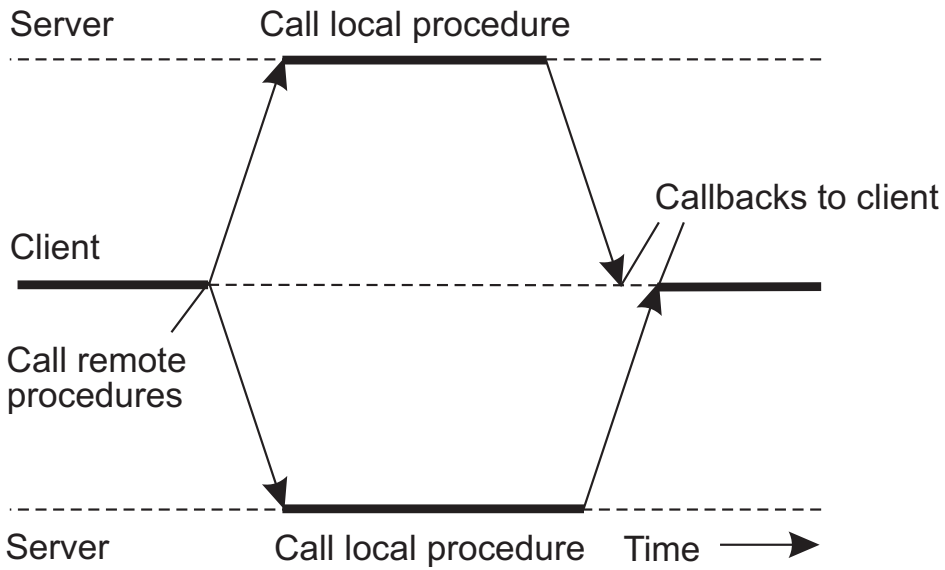
RPC Asíncrono



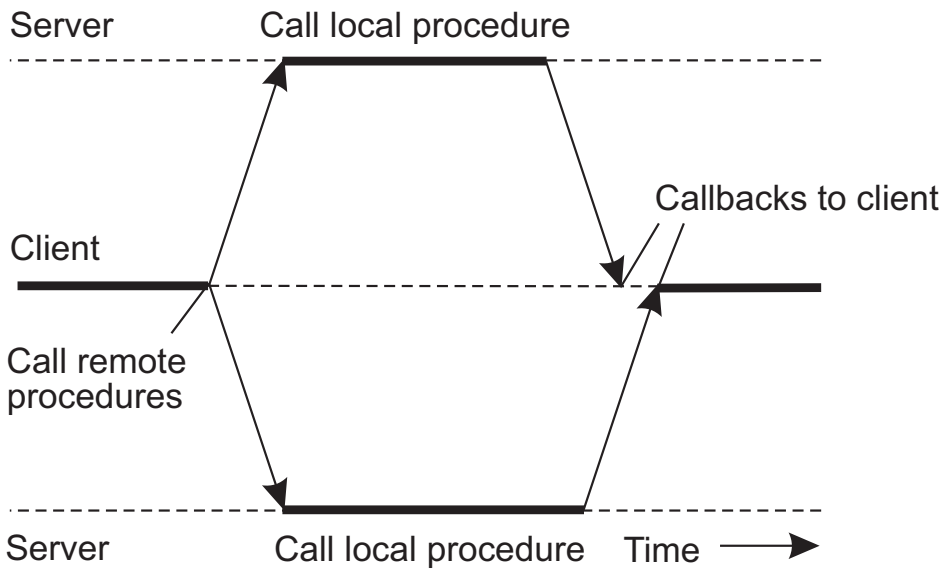
RPC Asíncrono

- Cliente debe poder continuar mientras el servidor trabaja

RPC múltiple



RPC múltiple

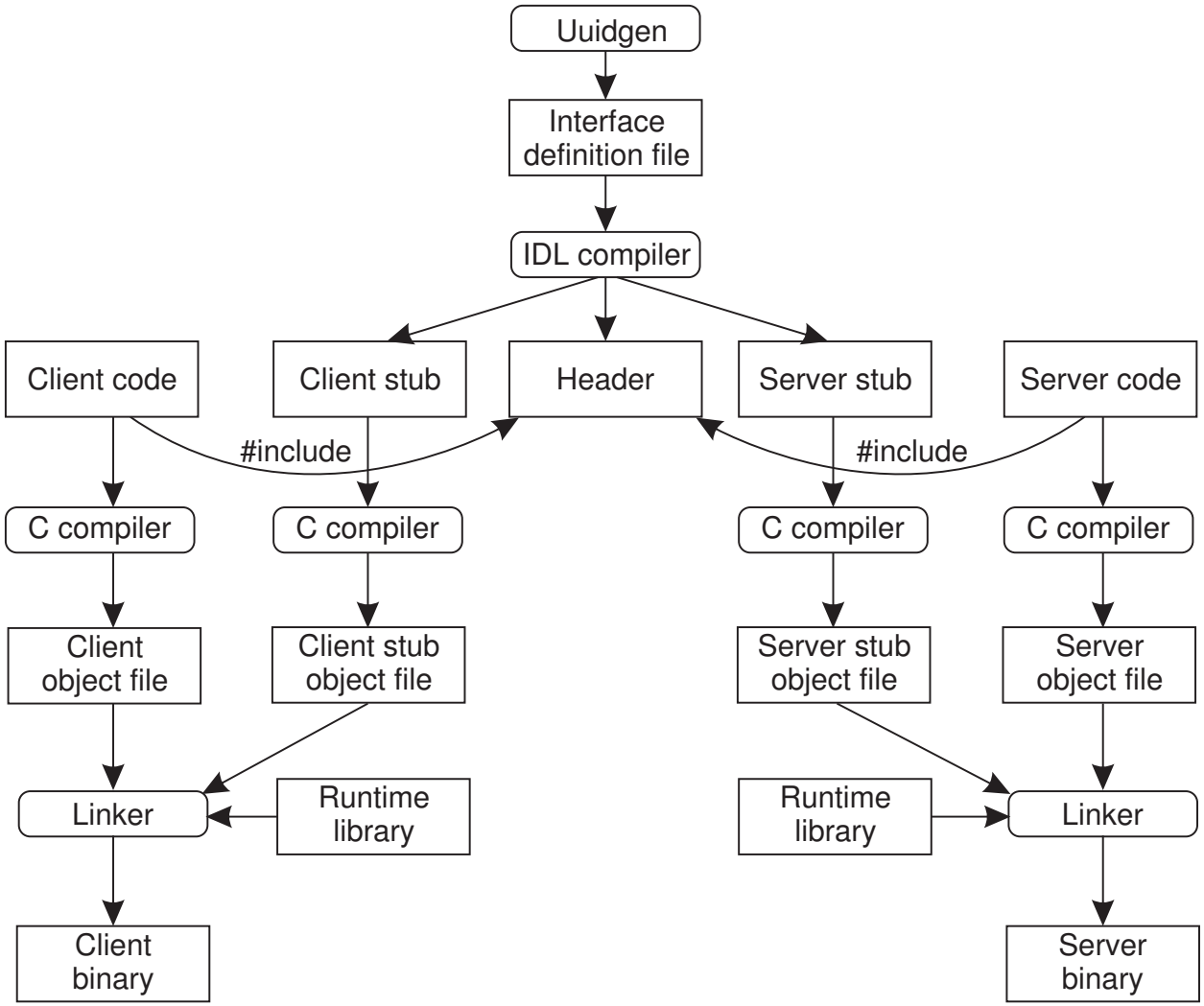


RPC múltiples

- *Request* se envía a un grupo de servidores

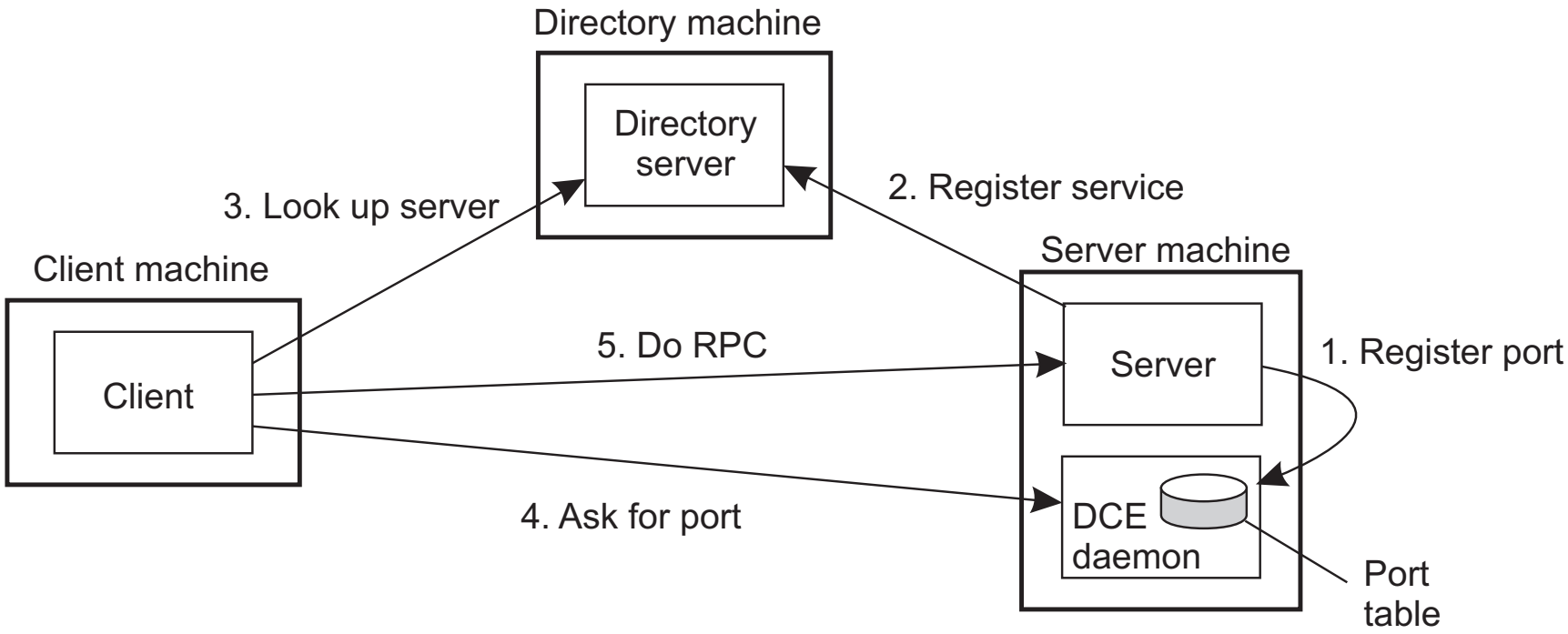
Ejemplo: DCE RPC

Componentes



Ejemplo: DCE RPC

Binding

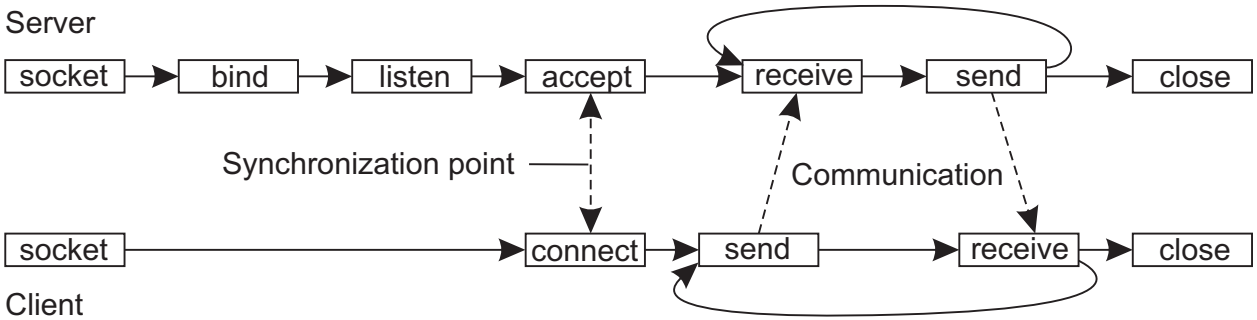


Contenidos

Comunicación por paso de mensajes

Comunicación **síncrona transiente**, tradicionalmente mediante *sockets*

Operation	Description
socket	Create a new communication end point
bind	Attach a local address to a socket
listen	Tell operating system what the maximum number of pending connection requests should be
accept	Block caller until a connection request arrives
connect	Actively attempt to establish a connection
send	Send some data over the connection
receive	Receive some data over the connection
close	Release the connection



Comunicación por paso de mensajes

¿Cómo implementar comunicación transiente?

Comunicación por paso de mensajes

¿Cómo implementar comunicación transiente?

Un ejemplo: ZeroMQ

Nivel mayor de abstracción mediante *pairing* de *sockets*.

- Un *socket* para enviar desde P, asociado a uno que recibe en Q
- Cada par puede utilizar su propio **patrón de comunicación**
- **Toda** la comunicación es **asíncrona**

Comunicación por paso de mensajes

¿Cómo implementar comunicación transiente?

Un ejemplo: ZeroMQ

Nivel mayor de abstracción mediante *pairing* de *sockets*.

- Un *socket* para enviar desde P, asociado a uno que recibe en Q
- Cada par puede utilizar su propio **patrón de comunicación**
- **Toda** la comunicación es **asíncrona**

Patrones posibles

- *Request/Reply*
- *Publish/Subscribe*
- *Pipeline*

Comunicación transiente síncrona

Server

```
1  from socket  import *
2  s = socket(AF_INET, SOCK_STREAM)
3  s.bind((HOST, PORT))
4  s.listen(1)
5  (conn, addr) = s.accept()   # returns new socket and addr. client
6  while True:                 # forever
7    data = conn.recv(1024)    # receive data from client
8    if not data: break        # stop if client stopped
9    conn.send(str(data)+"*") # return sent data plus an "*"
10 conn.close()                # close the connection
```

Client

```
1  from socket  import *
2  s = socket(AF_INET, SOCK_STREAM)
3  s.connect((HOST, PORT)) # connect to server (block until accepted)
4  s.send('Hello, world')  # send same data
5  data = s.recv(1024)     # receive the response
6  print data              # print the result
7  s.close()               # close the connection
```

ZeroMQ: Patrón Request/Reply

Server

```python
import zmq
context = zmq.Context()

p1 = "tcp://"+ HOST +":"+ PORT1 # how and where to connect
p2 = "tcp://"+ HOST +":"+ PORT2 # how and where to connect
s  = context.socket(zmq.REP)    # create reply socket

s.bind(p1)                      # bind socket to address
s.bind(p2)                      # bind socket to address
while True:
  message = s.recv()            # wait for incoming message
  if not "STOP" in message:     # if not to stop...
    s.send(message + "*")       # append "*" to message
  else:                         # else...
    break                       # break out of loop and end
```

Client

```python
import zmq
context = zmq.Context()

php = "tcp://"+ HOST +":"+ PORT # how and where to connect
s   = context.socket(zmq.REQ)   # create socket

s.connect(php)                  # block until connected
s.send("Hello World")           # send message
message = s.recv()              # block until response
s.send("STOP")                  # tell server to stop
print message                   # print result
```

ZeroMQ: Publish/Subscribe

Server

```
1  import zmq, time
2
3  context = zmq.Context()
4  s = context.socket(zmq.PUB)          # create a publisher socket
5  p = "tcp://"+ HOST +":"+ PORT        # how and where to communicate
6  s.bind(p)                            # bind socket to the address
7  while True:
8    time.sleep(5)                      # wait every 5 seconds
9    s.send("TIME " + time.asctime()) # publish the current time
```

Client

```
1  import zmq
2
3  context = zmq.Context()
4  s = context.socket(zmq.SUB)            # create a subscriber socket
5  p = "tcp://"+ HOST +":"+ PORT          # how and where to communicate
6  s.connect(p)                           # connect to the server
7  s.setsockopt(zmq.SUBSCRIBE, "TIME")    # subscribe to TIME messages
8
9  for i in range(5):   # Five iterations
10   time = s.recv()    # receive a message
11   print time
```

ZeroMQ: Patrón Pipeline

Source

```
1  import zmq, time, pickle, sys, random
2
3  context = zmq.Context()
4  me  = str(sys.argv[1])
5  s   = context.socket(zmq.PUSH)          # create a push socket
6  src = SRC1  if me == '1' else SRC2      # check task source host
7  prt = PORT1 if me == '1' else PORT2     # check task source port
8  p   = "tcp://"+ src +":"+ prt           # how and where to connect
9  s.bind(p)                               # bind socket to address
10
11 for i in range(100):                    # generate 100 workloads
12   workload = random.randint(1, 100)     # compute workload
13   s.send(pickle.dumps((me,workload)))   # send workload to worker
```

Worker

```
1  import zmq, time, pickle, sys
2
3  context = zmq.Context()
4  me = str(sys.argv[1])
5  r  = context.socket(zmq.PULL)      # create a pull socket
6  p1 = "tcp://"+ SRC1 +":"+ PORT1    # address first task source
7  p2 = "tcp://"+ SRC2 +":"+ PORT2    # address second task source
8  r.connect(p1)                      # connect to task source 1
9  r.connect(p2)                      # connect to task source 2
10
11 while True:
12   work = pickle.loads(r.recv())    # receive work from a source
13   time.sleep(work[1]*0.01)         # pretend to work
```

MPI: Paso de mensajes genéricos

Operation	Description
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until transmission starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

Message-Queueing Systems

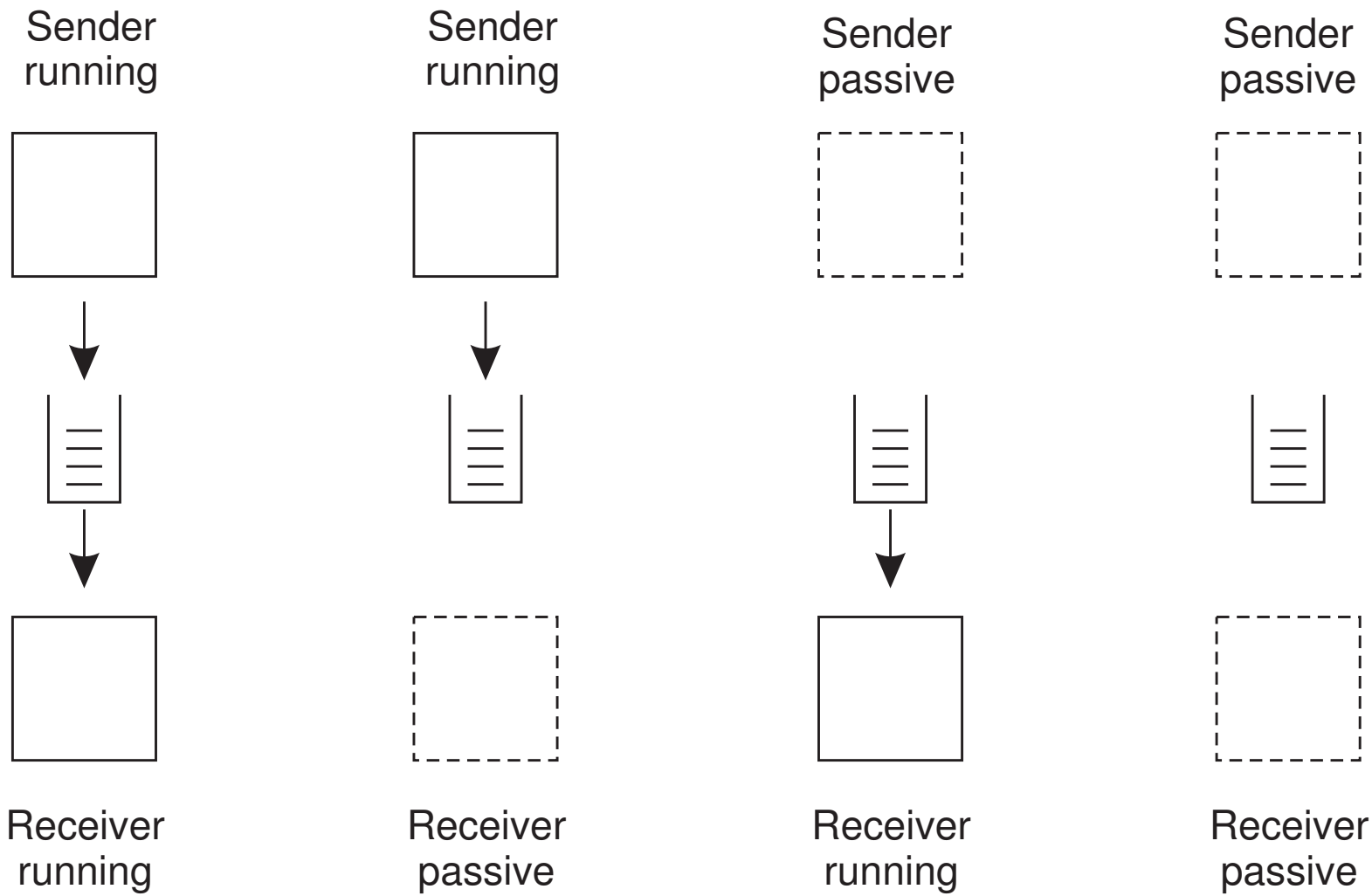
Message Oriented Middleware

Comunicación **persistente** y **asíncrona** mediante colas.

Operation	Description
put	Append a message to a specified queue
get	Block until the specified queue is nonempty, and remove the first message
poll	Check a specified queue for messages, and remove the first. Never block
notify	Install a handler to be called when a message is put into the specified queue

Message-Queueing Systems

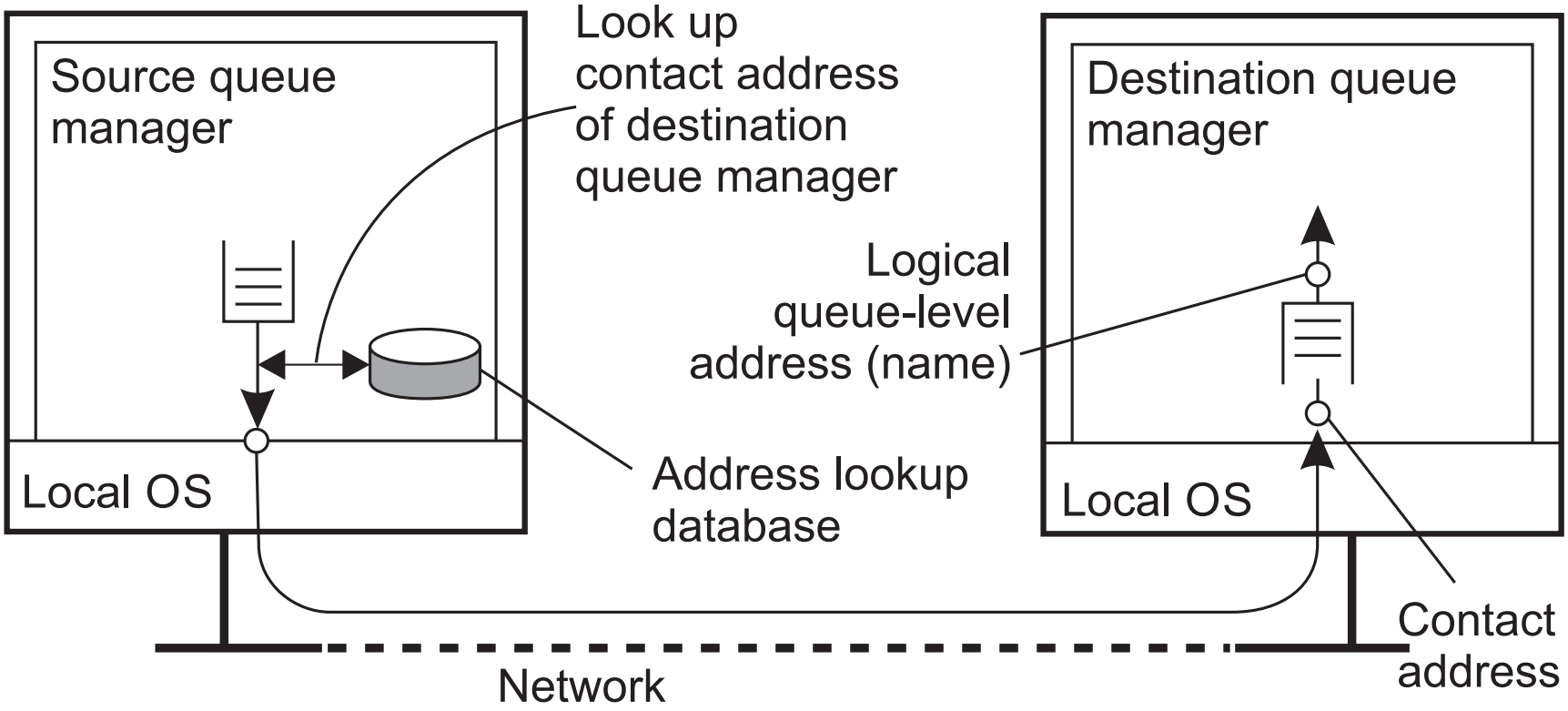
Comunicación *débilmente acoplada*



Message-Queueing Systems

Modelo de **Queue Managers**

Administran colas locales y rutean mensajes entre colas remotas

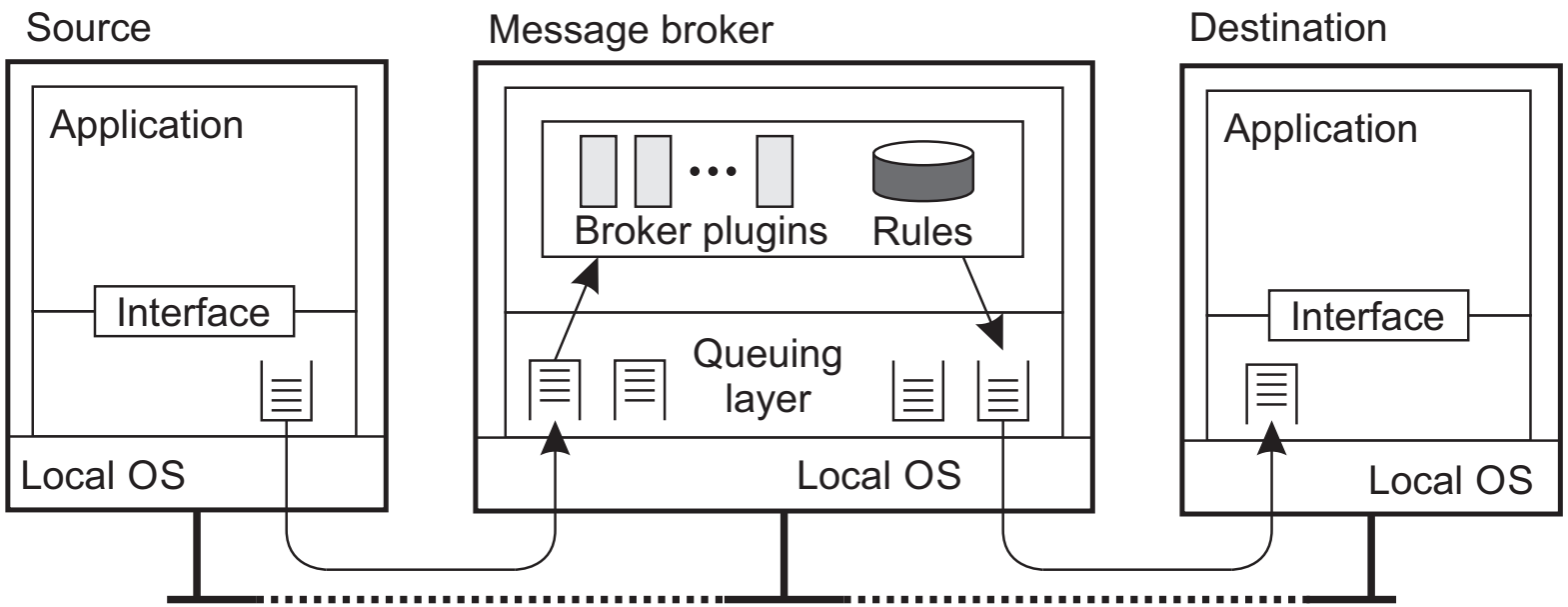


Message Brokers

Cuando puede haber distintos formatos y representaciones

Message Brokers permiten interconectar distintos sistemas de colas

Funciona como *gateways* entre aplicaciones



Contenidos

Multicast en redes

¿Cómo transmitir un mensaje **eficientemente** a un grupo de nodos?

Multicast a nivel de aplicación

Organizar los miembros (nodos) de un sistema distribuido, mediante una *overlay network*.

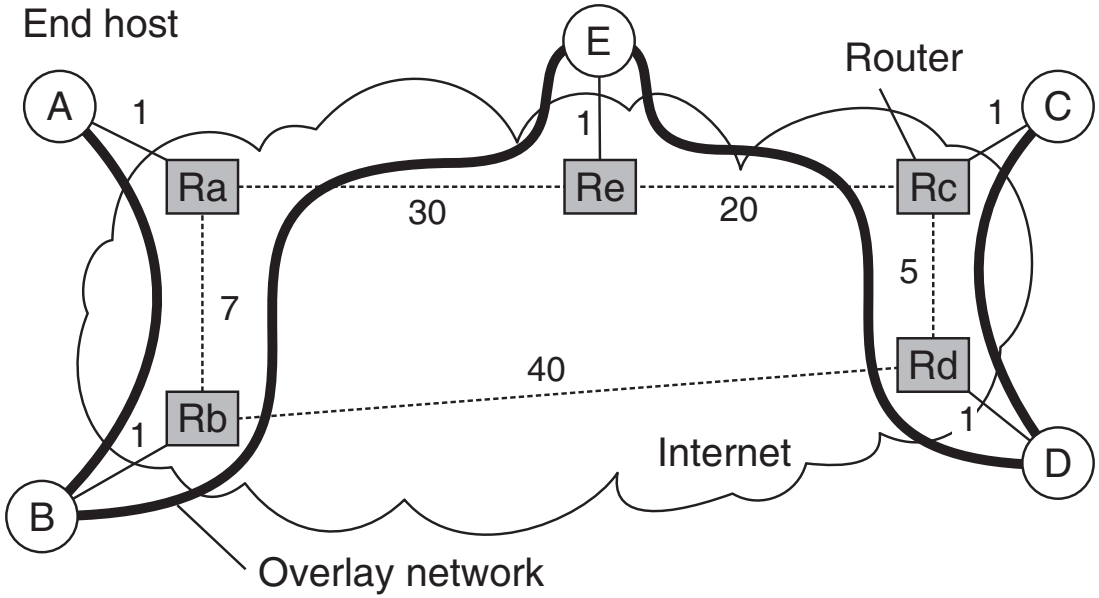
- Organización de árbol: caminos únicos, conexiones no robustas
- Organización de *mesh*: múltiples caminos, conexiones redundantes. Requiere de algún algoritmo de *routing*

Multicast en red *Chord*

Un nodo s desea iniciar un envío *multicast*

1. s genera un valor `mid` (*multicast identifier*)
2. Buscar (*lookup*) a succ(`mid`). Nodo responsable de `mid`
3. Solicitud de *multicast* se envía a $r =$ succ(`mid`), que se convierte en *root*
4. Si algún nodo p quiere unirse al grupo de *multicast* `mid`, debe enviar una solicitud a lookup(`mid`)
   - Con esto se envíe un mensaje de tipo `lookup` a través de la red, pasando por varios nodos
5. Cuando un nodo q recibe una solicitud `lookup`, puede que q sea o no parte del grupo
   - Si q no ha visto nunca un mensaje lookup(`mid`), se convierte *forwarder*, y reenvía el mensaje
   - Se genera un serie de *forwarders*
   - p será hijo de q
   - Si q ya era parte del grupo de `mid`, entonces agregar a p como hijo. Mensaje ya no se reenvía.

Costo de *multicast* a nivel de aplicación



Enlaces de la *overlay network* no son necesariamente los más eficientes respecto a los enlaces físicos

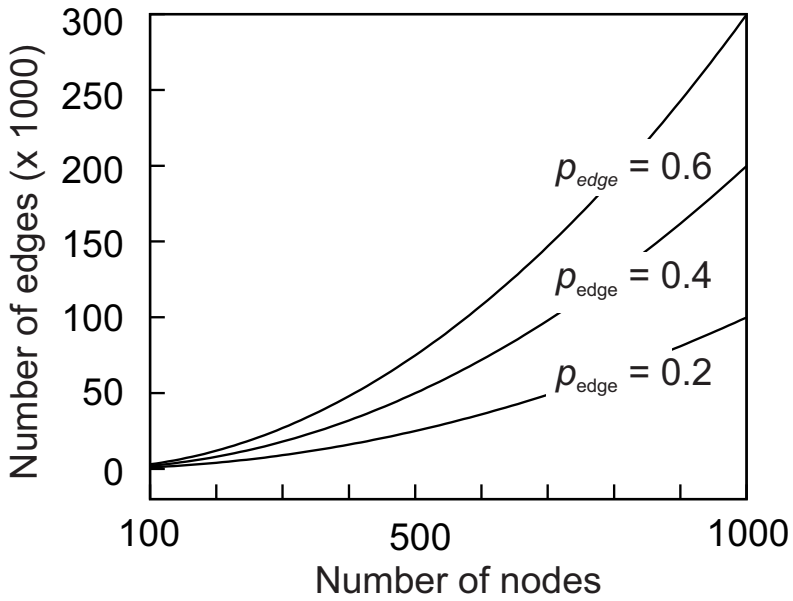
- **Link stress**. Algunos enlaces se repiten. Ejemplo: ruta desde A hasta D
- **Stretch**, o **RDP**, Relative Delay Penalty. Relación entre costo de enlace virtual versus enlace físico. Ejemplo: desde B hasta C
- **Tree cost**. Suma de los costos de las aristas. ¿Qué árbol conviene más construir?
- Ante múltiples nodos, ¿quién conviene que sea la raíz?

Multicast basado en *flooding*

Flooding

Cada nodo P envía el mensaje m a todos sus vecinos.
Si P recibue un mensaje repetido, lo ignora.
Desventaja: por cada enlace pasan aproximadamente dos mensajes

¿Cuántos enlaces hay?



Multicast basado en *flooding*

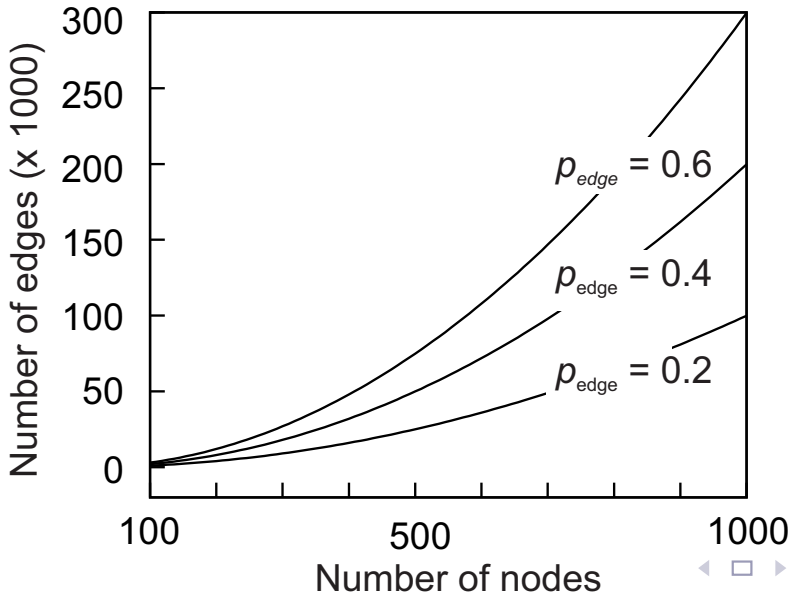
Envío probabilístico

Cada nodo envía un mensajes con una probabilidad p_{flood}

p_{flood} puede depende del grado del nodo (cantidad de vecinos), o del grado de sus vecinos

Ventaja: Cantidad de mensaje baja a tamaño lineal

Desventaja: probabilidad que no todos reciban en el mensaje si p_{flood} es muy bajo



Multicast basado protocolos epidémicos

Propagación epidémica

Idea: propagar información basado solamente en información local

- **Supuesto**: las actualizaciones de un data vienen siempre del mismo origen, por lo tanto no hay conflictos *write*/*write*
- Mensaje sólo se reenvía a algunos vecinos
- Propagación *lazy* (no inmediata)

Dos tipos de protocolos epidémicos

- **Anti-entropy**. Cada nodo elige un nodo al azar, e intercambian su estado.
- **Rumor spreading**. Un nodo que recibe un mensaje escoge un conjunto de vecinos al azar, y propaga.

Multicast basado en protocolos epidémicos

Anti-entropy

P elige nodo Q al azar.
Interactúan de tres maneras posibles.

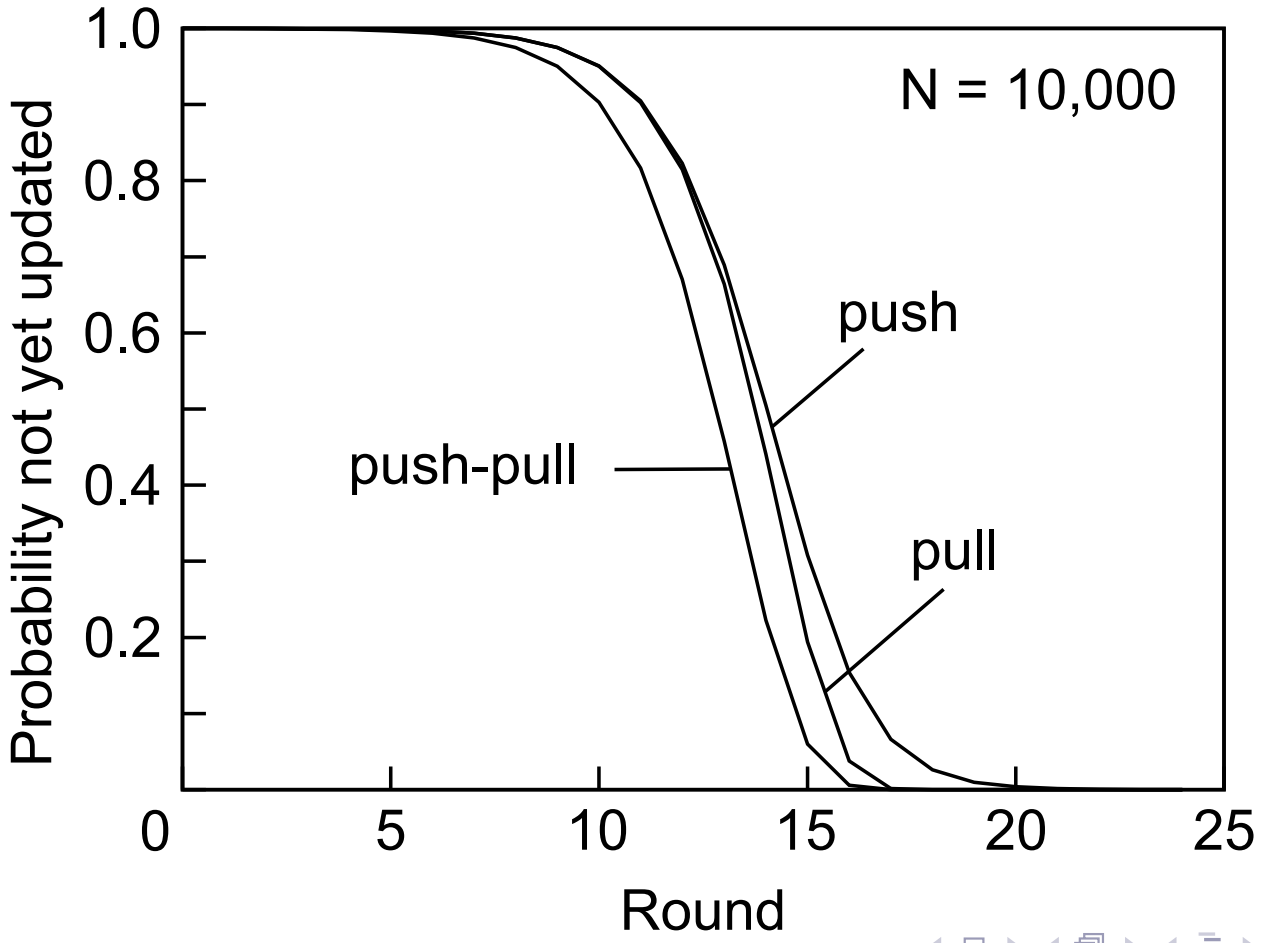
- **Pull**: P solo obtiene mensajes desde Q
- **Push**: P solo envía mensajes a Q
- **Push-pull**: P se envían mensajes mutuamete

round: período en el cual nodo puede decidir intercambiar mensajes

Tiempo de propagación para N nodos con *push-pull* es $\mathcal{O}(\log N)$

Multicast basado en protocolos epidémicos

Considerando un único nodo origen que desea propagar un mensaje. p_i es la probabilidad que después de *i rounds* el mensaje no se haya recibido.



Multicast basado en protocolos epidémicos

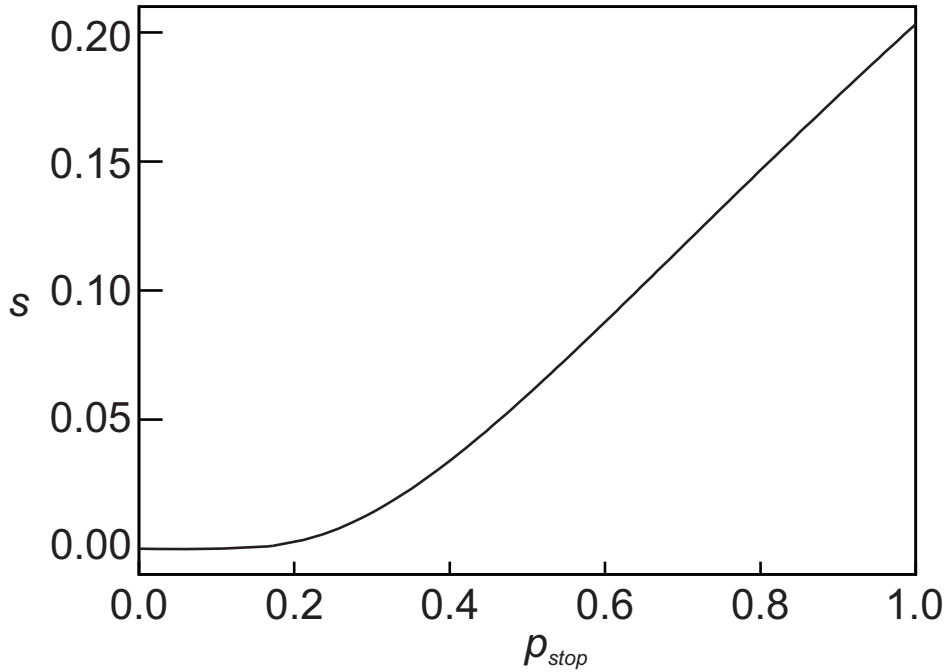
Rumor-spreading

Nodo S debe enviar un mensaje y contacto a otros nodos.
Si encuentra un nodo que ya ha recibido el mensaje, S deja de contactar nodos, con probabilidad p_{stop}

Si s es la fracción de los que no reciben el mensaje en este modelo, se puede demostrar que con suficientes nodos:

$$s = e^{-(\frac{1}{p_{\text{stop}}}+1)(1-s)}$$

Multicast basado en protocolos epidémicos



Consider 10,000 nodes		
$1/p_{stop}$	s	N_s
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3