Desarrollo de Sistemas Distribuidos

Resumen Completo — Temas 2 y 3

Universidad de Granada · Grado Ingeniería Informática · Curso 2025–26

Basado íntegramente en las diapositivas oficiales (T2: slides 52–101; T3: slides 105–149) y complementado con Coulouris *et al.*, *Distributed Systems: Concepts and Design* (5ª ed.)

Realizado por **@jcartov** · Más recursos en **repasaYA**

TEMA 2 — Comunicación y Sincronización de Procesos

Un sistema distribuido (SD) necesita mecanismos para que procesos en máquinas distintas **se comuniquen** (intercambien información) y **se sincronicen** (coordinen su avance). El tema estudia cuatro paradigmas, de menor a mayor nivel de abstracción:

1. Paso de mensajes → 2. Llamada Remota a Procedimiento (RPC) → 3. Comunicación Cliente/Servidor (petición/respuesta) → 4. Invocaciones remotas o citas (*rendez-vous*).

1. Paso de mensajes

Es el mecanismo básico: dos procesos intercambian datos explícitamente a través de un **canal**, sin memoria compartida.

Conceptos fundamentales

- **Formar / empaquetar (*marshalling*)**: poner una colección de datos en un formato adecuado para transmitirlos en un mensaje. Consiste en:
  - **Aplanar (*to flat*)** las estructuras de datos, convirtiéndolas en secuencias básicas de bytes.
  - **Traducir** los elementos básicos a una **representación de datos estándar** independiente de la máquina (p.ej. *eXternal Data Representation*, **XDR** de SUN) para superar la heterogeneidad (distinto *endianness*, tamaños de tipos, etc.).
  - Las operaciones de *marshalling*/*unmarshalling* se pueden **generar automáticamente** a partir de la definición de la interfaz.
- **Canal**: abstracción de una red de comunicación física que proporciona un **camino de comunicación** entre procesos y su **sincronización**, mediante dos primitivas: `send` y `receive`.

Notaciones: en qué se diferencian

Las distintas notaciones de paso de mensajes difieren en:

- **Ámbito y denominación** de los canales (globales a todos los procesos o asociados a un subconjunto).
- **Uso**: flujos de información bidireccionales o unidireccionales.
- **Sincronización**: síncrona/bloqueante, asíncrona/no bloqueante con *buffer* ilimitado, o asíncrona con *buffer* limitado.
- Una operación **no bloqueante** nunca retrasa al proceso que la invoca. Normalmente no bloquea el `send`, aunque también existe el `receive` no bloqueante (más complejo: requiere **sondeos** o **interrupciones**).

La mayoría de las notaciones son **equivalentes** (un programa escrito en una se puede reescribir en otra), pero cada una es más adecuada según el tipo de problema.

Notación aceptada (admite variantes con *timeouts*, etc.):
`send <puerto [o canal]>(<mensaje>)` • `receive <puerto>(<mensaje>)` • `empty <puerto>` *(comprueba si la cola del canal está vacía)*

Tipos de comunicación según la sincronización

Tipo	Bloqueo	Consecuencias
Síncrona	`send` y `receive` bloqueantes	Orden estricto, cita implícita emisor/receptor; bloquea el resto de operaciones (comportamiento lineal).
Asíncrona	normalmente sólo bloquea `receive`	Sin orden estricto, permite concurrencia y optimización del tiempo; *buffer* conceptualmente ilimitado.
Asíncrona con *buffer* finito	`receive` bloqueante; `send` bloquea sólo si el *buffer* está lleno	Compromiso realista entre las dos anteriores.

Destino de los mensajes

El destino debe ser **conocido** por el emisor e **independiente de la localización** (**transparencia**), como ocurre p.ej. con los nombres del **DNS**. Tipos de destino según su cardinalidad e indirección:

Destino	Cardinalidad	Descripción
Proceso (*referencia directa*)	1 a 1	Punto a punto **sin** indirección.
Enlace (*link*)	1 a 1	Punto a punto **con** indirección.
Puerto (*port*)	muchos a 1	Con indirección (varios emisores, un receptor).
Buzón (*mailbox*)	muchos a muchos	Con indirección.
Difusión (*broadcast*)	muchos a muchos	Con indirección (a todos).
Selección (*multicast*)	muchos a muchos	Con indirección (a un grupo).

Comunicación de grupos

Protocolos que envían un mensaje a un **conjunto** de procesos (*multicast*).

- **Usos**: tolerancia a fallos en servicios replicados; localización de objetos en servicios distribuidos (p.ej. ficheros); mejor rendimiento con servicios replicados; actualización múltiple notificando eventos a varios procesos a la vez.
- **Propiedades** deseables:
  - **Atomicidad**: el mensaje es recibido por **todos** los miembros o por **ninguno**.
  - **Ordenación**: todos ejecutan las operaciones en el **mismo orden**.

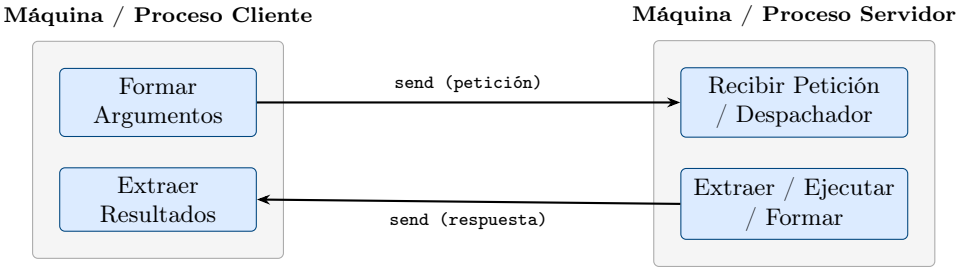
Fiabilidad

- **Fiable**: los datos que emite el cliente llegan al servidor **sin errores** y **en el mismo orden** en que se emitieron.
- Un paso de mensajes fiable se puede **construir sobre uno no fiable** (p.ej. mediante **acuses de recibo** y retransmisiones).
- Ejemplos: **TCP/IP** (fiable, orientado a conexión) frente a **UDP** (**no fiable**, sin garantías de entrega ni orden).

2. Llamada Remota a Procedimiento (RPC)

RPC ofrece **transparencia**: el programador invoca un procedimiento remoto **como si fuera local**. Los ***stubs*** ocultan el *marshalling* y la comunicación por red.

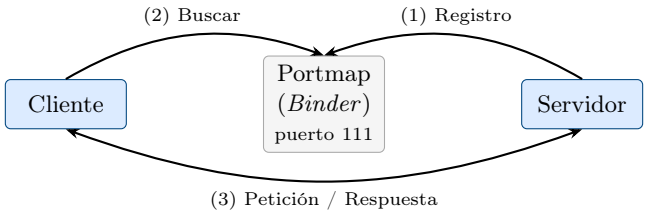
Implementación



Componentes / ficheros generados (nomenclatura estilo Sun RPC): Cliente (`_client.c`), *Stub* del cliente (`_clnt.c`), Módulos de comunicaciones (`libnsl`), Despachador y *Stub*/*Wrapper* del servidor (`_svc.c`), Procedimiento remoto (`_server.c`).

Los tres elementos de un sistema RPC

1. **Procesamiento de la interfaz**: integra el mecanismo RPC con los programas cliente y servidor formando/extrayendo argumentos y resultados. Se compila una especificación escrita en un **Lenguaje de Definición de Interfaces (IDL)** que genera cabeceras, plantillas y *stubs*:
   - En el **cliente**, el *stub* convierte la llamada **local** en **remota**.
   - En el **servidor**, el despachador **selecciona y llama** al procedimiento adecuado.
2. **Módulo de comunicaciones**: implementa el **protocolo petición–respuesta**.
3. **Servicio de ligadura (*binding*)**: mecanismo para **localizar el servidor**.
   - Asocia un **nombre** a un **identificador de comunicación** (puerto). El mensaje de petición se dirige a un puerto concreto.
   - Se evalúa **cada vez** que el cliente lo requiere, porque el servidor puede haber sido **relocalizado**.
   - Es un servicio del que dependen otros ⇒ debe ser **tolerante a fallos**.
   - Los servidores **exportan (registran)** sus servicios y los clientes los **importan (buscan)**.



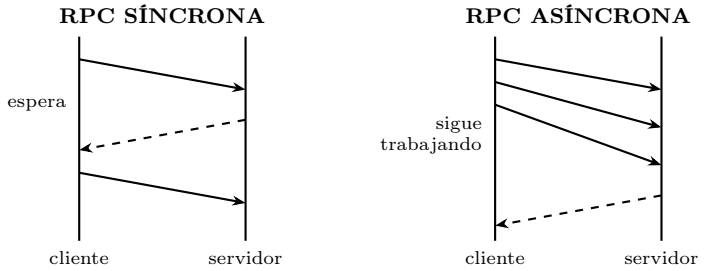
Operaciones del *binder*: `Registro(<nombre>,<puerto>,<versión>)`, `Retirar(<nombre>,<puerto>,<versión>)`, `Buscar(<nombre>,<versión>) : <puerto>`. *Ejemplo*: `Registro<Youtube,1004,v1.0>`; una consulta `Buscar<Youtube,v1.0>` devuelve `1004`.

Alternativas para localizar el propio ligador:

1. **Dirección conocida**: cliente/servidor deben **recompilarse** si el ligador se relocaliza (rígido).
2. El **sistema operativo** proporciona la información en tiempo de ejecución (p.ej. variables de entorno).
3. Al lanzarse, cliente/servidor envían mensajes de **difusión (*broadcast*)** y el ligador (*binder*) responde con su dirección.

RPC asíncrona

- **Requisitos típicos**: el cliente envía **muchas** peticiones (muchos `send` no bloqueantes) y **no** necesita respuesta a cada una.
- **Ventajas**: el servidor puede **planificar** las operaciones más eficientemente; el cliente **trabaja en paralelo**; se facilita el **cálculo de peticiones paralelas** sobre varios servidores (+paralelismo, +concurrencia).
- **Optimizaciones**: agrupar varias peticiones en una sola comunicación, almacenándolas hasta que (a) se cumple un **plazo de tiempo**, o (b) se realiza una petición que **requiere respuesta**. El cliente puede continuar si no espera una respuesta que podrá obtener más tarde.



3. Comunicación Cliente/Servidor: protocolo petición/respuesta

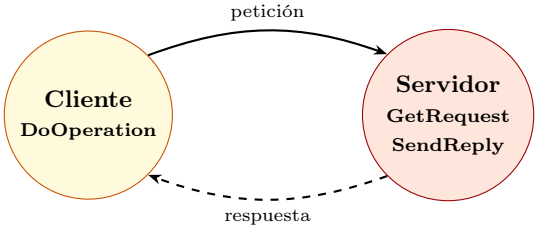
Idea general

Comunicación típicamente **síncrona** y **segura/fiable**: la **respuesta** del servidor sirve como **acuse de recibo** de la petición.

Alternativas de implementación:

- Con **primitivas** `send`/`receive`. Inconvenientes: **sobrecarga** (más canales usados explícitamente), **correspondencia** manual entre cada `send` y su `receive`, y garantizar el **reparto** de mensajes si la red no lo proporciona.

- Con **operaciones de comunicación** de más alto nivel, que combinan aspectos de **monitores** (exportación de operaciones) y de **paso de mensajes síncrono** (las peticiones bloquean al cliente). Tres primitivas:




```
public byte[] doOperation(RemoteObjectRef o, int methodId, byte[] arguments);
public byte[] getRequest();
public void sendReply(byte[] reply, InetAddress clientHost, int clientPort);
```

Formato del mensaje: *tipoMensaje* (0=petición, 1=respuesta), *idPetición*, *refObjeto* (en **RMI**; en RPC este campo se **omite**), *idMétodo*, *argumentos*.

Los tres protocolos petición/respuesta

Existen tres protocolos con **semánticas distintas en presencia de fallos**:

Protocolo	1º Cliente	2º Servidor	3º Cliente
R	Petición	—	—
RR	Petición	Respuesta	—
RRA	Petición	Respuesta	Reconocimiento (*ack*)

Modelo de gestión de fallos

La implementación depende de las **garantías de envío/entrega**. Mecanismos:

- **Plazo de tiempo (*time-out*)** tras `doOperation`. Al cumplirse: (a) **devolver fallo** (aunque la operación finalmente se ejecute), o (b) **repetir** la petición y, eventualmente, devolver fallo.
- **Filtrar mensajes de petición duplicados** usando el *idPetición*.
- **Mensajes de respuesta perdidos**: las **operaciones idempotentes** pueden reejecutarse dando el mismo resultado.
- **Gestión de históricos**: retransmitir respuestas **sin reejecutar** la operación. La **nueva petición** actúa como reconocimiento de la respuesta previa; los *acknowledgments* del cliente ayudan a **descartar** entradas de la historia (también se descartan tras un periodo de tiempo).

Reintentar petición	Filtrar duplic.	Reejecutar / Retransmitir	Semántica de invocación	Ejemplo
No	N/A	N/A	**Quizás (*maybe*)**	Peticiones donde un fallo es tolerable.
Sí	No	Reejecutar procedimiento	**Al menos una vez (*at-least-once*)**	Sun RPC (sólo válido si es idempotente).
Sí	Sí	Retransmitir respuesta	**Como mucho una vez (*at-most-once*)**	RPC, RMI, CORBA.

4. Invocaciones remotas o citas (*rendez-vous*)

Concepto

- Las invocaciones remotas se sirven mediante una **instrucción de aceptación** (par de instrucciones ***call*–*in***). En entornos distribuidos se llaman **citas extendidas**.
- A diferencia de RPC (que es una **comunicación intermódulo**), en las citas el servidor es un **proceso activo** que decide cuándo atender.
- Limitación de las instrucciones básicas: a menudo un proceso desea comunicarse con **más de un** proceso, quizás en puertos distintos, y **no sabe el orden** en que los otros querrán comunicarse con él ⇒ se necesita **no determinismo**.

No determinismo: instrucciones guardadas (Dijkstra)

Una **instrucción guardada** tiene la forma:

$$B\ ;\ C\ \rightarrow\ S$$

donde B es una **expresión lógica opcional**, C una **instrucción de comunicación opcional** (juntas forman la **guarda** $B;C$) y S una **lista de instrucciones**.

Semántica de la guarda:

1. **Tiene éxito** si B es verdad **y** la ejecución de C no produce retardo.
2. **Falla** si B es falso.
3. **Bloquea** si B es verdad pero C no puede ejecutarse sin producir retardo.

B no puede cambiar hasta ejecutar otras asignaciones (no hay **variables globales**). La guarda puede incluir instrucciones de comunicación de entrada o de salida.

Construcciones que combinan instrucciones guardadas:

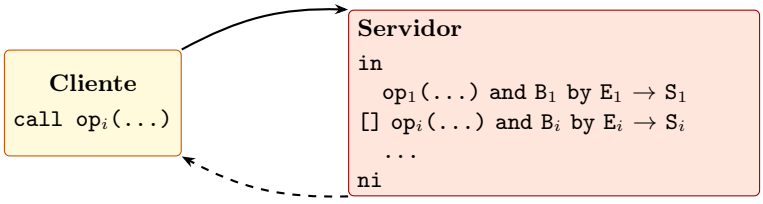
- **Alternativa (*IF*)**: si al menos una guarda tiene éxito, se escoge **una no determinísticamente** y se ejecuta su C y S. Si **todas fallan**, `IF` termina/falla. Si ninguna tiene éxito pero algunas bloquean, la ejecución se **retrasa** hasta que la primera tenga éxito.
- **Repetitiva (*DO*)**: como `IF` pero **itera** hasta que todas las guardas fallen.

Ejemplo (ordenar 3 valores con no determinismo):

```
a, b, c := A, B, C;
do  a > b → a, b := b, a;  []  b > c → b, c := c, b;  od
```

Órdenes guardadas de comunicación (citas extendidas)

- Un proceso **exporta operaciones** (similar a RPC). Otro las invoca con `call <proceso>.<operación>(<args>);`
- El servidor las atiende en su contexto mediante **instrucciones de aceptación**: `in <operación>(<params>) and <expr_lógica> by <expr> → S ni`
- El **ámbito** de los parámetros formales es el de la operación guardada. Una guarda de operación **tiene éxito** cuando (a) se ha **invocado** la operación y (b) su **expresión lógica** es verdad. Con varias, la ejecución se **retrasa** hasta que una tenga éxito ⇒ no determinismo.



Características de las citas extendidas:

- Puntos de comunicación de **muchos a uno** en el contexto del proceso.
- **Sin parámetros** hay **sincronización** pero **no comunicación** de datos.
- El servidor puede definir **distintas guardas** para una misma operación exportada ⇒ **efectos diferentes** ante la misma invocación.
- Las invocaciones se sirven en los **instantes que desee el servidor**.
- A diferencia de RPC, el servidor es un **proceso activo** que ya se está ejecutando **antes y después** de servir la invocación.

Realización en lenguajes reales (tarea obligatoria T2): en el estándar **POSIX**, las llamadas al sistema `select` y `poll` implementan la espera no determinista sobre varios descriptores. En **Ada**, el mecanismo de cita se expresa con `entry`/`accept` y la selección no determinista con `select ... or ... or terminate`.

```
task Buffer1 is
   entry Escribir (Elem: TElemento);
   entry Leer     (Elem: out TElemento);
end Buffer1;

task body Buffer1 is
   ElemLocal: TElemento;  nElementos: integer;
   tamBuffer: constant := 1;
begin
   loop
      select when (nElementos < tamBuffer) =>      -- guarda
         accept Escribir (Elem: TElemento) do
            ElemLocal := Elem;  nElementos := nElementos + 1;
```

```
         end Escribir;
      or when (nElementos > 0) =>                    -- guarda
         accept Leer (Elem: out TElemento) do
            Elem := ElemLocal;  nElementos := nElementos - 1;
         end Leer;
      or terminate;
      end select;
   end loop;
end Buffer1;
```

TEMA 3 — Coordinación

Estudia cómo procesos sin memoria común ni reloj global **ordenan eventos** y **acuerdan acciones**: (1) **tiempo lógico** y (2) **algoritmos distribuidos** de **exclusión mutua**, **elección** y **consenso**.

1. Tiempo

¿Por qué es importante el tiempo en un SD?

1. **Medida** que se desea obtener con precisión:
   - **Sincronización externa**: saber **cuándo** ocurrió un evento (p.ej. una transferencia bancaria). Requiere sincronizar la hora con un **reloj o fuente externa autorizada** (relojes atómicos por radio/satélite, red telefónica, UTC).
   - **Sincronización interna**: obtener las **mismas referencias** de tiempo o el **intervalo** entre dos eventos que ocurren en máquinas diferentes, con **precisión conocida** (p.ej. tiempos de transmisión de un mensaje: dos marcas de tiempo, una en origen y otra en destino).
2. **Problemas lógicos** debidos a la distribución (mantener la **consistencia** en un gestor de transacciones bancarias, auditorías...).

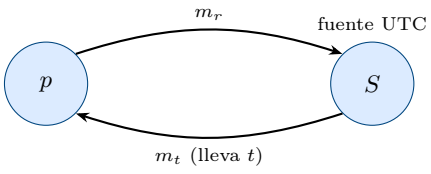
- **Evento**: acción que **parece ocurrir indivisiblemente** (p.ej. el envío de un mensaje: acción atómica).
- El **orden de ocurrencia** de los eventos puede ser **crítico** (p.ej. un servidor de datos replicado).

Tipos de aplicaciones

- **Centralizadas**: sólo necesitan conocer el **orden** de los eventos ⇒ basta asociar un **reloj** (tiempo absoluto) o un **contador** (tiempo relativo) a cada evento.
- **Distribuidas**: conocer el desplazamiento relativo del reloj de una máquina respecto de otra, con idéntica velocidad de pulso, es **casi imposible**. Opciones: un **reloj físico compartido** (sistemas síncronos) o un **servidor de tiempo** sobre peticiones (sistemas asíncronos). **Problema** del servidor: su fallo, o una **réplica/impostor** que responda a los mensajes *multicast*.

Método de Cristian (sincronización de relojes)

El proceso p pide la hora a un servidor S con acceso a **UTC**; mide el tiempo de ida y vuelta T_{round} y ajusta su reloj compensando la mitad del trayecto:



$$T_{round} = T_{recibir\_m_t} + T_{enviar\_m_r}, \qquad p \text{ fija su reloj a } \; t + \frac{T_{round}}{2}$$

Asume que los tiempos de envío y recepción son **pequeños y prácticamente iguales** respecto a la precisión requerida. La **exactitud** conseguida es del orden de $\pm(T_{round}/2 - min)$, siendo min el tiempo mínimo de transmisión.

2. Tiempo lógico

Como no hay un reloj físico global fiable, **Lamport** propuso ordenar eventos por **causalidad** en lugar de por tiempo real.

Relación *ocurrió-antes* (*happened-before*, $\rightarrow$)

Esquema de ordenación basado en dos reglas observables más la transitividad:

1. Si dos eventos ocurren en el **mismo proceso**, ocurren en el **orden en que se observan**: si $x \xrightarrow{p} y$ en p, entonces $x \to y$.
2. Si se **envía** un mensaje m: $send(m) \to receive(m)$ (el envío ocurre antes que la recepción).
3. **Transitividad**: si $x \to y$ e $y \to z$, entonces $x \to z$.

Si **no** se cumple ni $x \to y$ ni $y \to x$, los eventos son **concurrentes** ($x \parallel y$): no hay nada que los relacione causalmente.

Relojes lógicos de Lamport

Un **reloj lógico** C es un **contador software** que crece **monótonamente**. Notación: C_p (reloj del proceso p), $C_p(a)$ (marca del evento a en p), $C(b)$ (marca del evento b donde ocurriera). Reglas de actualización:

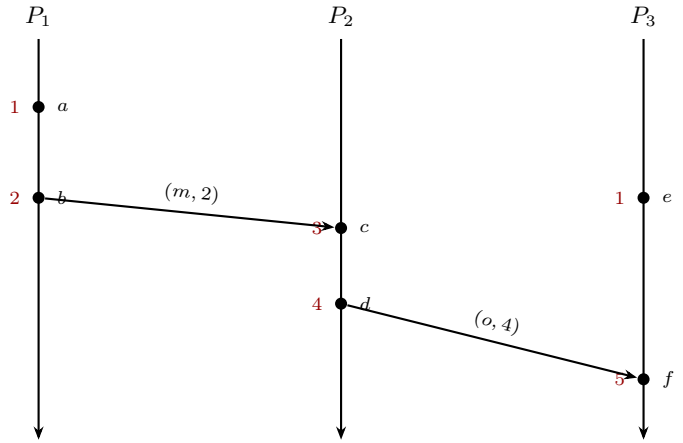
LC1. C_p se **incrementa** antes de cada evento que ocurre en p: $C_p := C_p + 1$.
LC2. Cuando p **envía** un mensaje, le añade el valor $t = C_p$. Cuando q lo **recibe**, calcula $C_q := \max(C_q, t)$ y luego aplica LC1 antes de marcar el evento $receive(m, t)$.

Propiedad clave: $a \to b \Rightarrow C(a) < C(b)$. El **recíproco NO se cumple**: $C(a) < C(b)$ no implica $a \to b$ (pueden ser **concurrentes**). Los relojes de Lamport **capturan** la causalidad pero **no la caracterizan** por completo.

Extensión a orden total: se desempata con el identificador de proceso.

$$(C(a), p) < (C(b), q) \iff C_p(a) < C_q(b) \quad \vee \quad \big(C_p(a) = C_q(b) \ \wedge \ p < q\big)$$

Ejemplo (marca +1 por evento; los mensajes transportan su marca):



$a \to b$, $b \to c$ (mensaje) $\Rightarrow a \to c$. En cambio a (en P_1, $C = 1$) y e (en P_3, $C = 1$) tienen **la misma marca** y **no** hay nada que los conecte $\Rightarrow$ son **concurrentes**: el orden total los desempata por id de proceso, $(1, P_1) < (1, P_3)$.

Complemento (Coulouris §14.4): relojes vectoriales. Superan la limitación de Lamport: cada proceso mantiene un vector V_i con una entrada por proceso. Se cumple la **bicondicional** $a \to b \iff V(a) < V(b)$, por lo que **sí** detectan la concurrencia ($V(a) \parallel V(b)$ cuando ninguno domina al otro). Coste: se transmite un vector de tamaño N en cada mensaje.

3. Algoritmos distribuidos

Necesarios en telecomunicaciones, procesamiento distribuido, computación científica, control en tiempo real, etc. Requieren **diseño**, **implementación** y **análisis**. Se clasifican según:

- **Modelo de temporización** de eventos: asíncrono / completamente síncrono / parcialmente síncrono.
- **Método de comunicación (IPC)**: memoria compartida, punto-a-punto, grupos, RPC...
- **Modelo de fallos**: completamente fiable o tolerante a ciertos fallos (de procesador o de comunicaciones).
- **Problemas abordados**: asignación de recursos, comunicación de datos, consenso, control de concurrencia, detección de bloqueos...

Distinción principal: modelo de temporización

Modelo	Características	Compromiso
Síncrono	Los componentes dan pasos **simultáneamente** (límites conocidos en retardo de mensajes y deriva de reloj). El más simple de describir, programar y razonar; **imposible o ineficiente** de implementar en muchos SD.	Poco realista, +fácil, −eficiente
Asíncrono	Los componentes dan pasos **arbitrariamente** (sin suposiciones de tiempo). Simple de describir pero con complicaciones de **vivacidad**; más difícil por la **incertidumbre** en el orden; solución más **general y portable**, a veces poco potente o ineficiente.	Realista, general
Parcialmente síncrono	Con **ciertas restricciones** en la temporización relativa (p.ej. plazos de tiempo). El más **realista** pero difícil; puede ser eficiente aunque **frágil** (falla si no se cumplen las restricciones).	+realista, +difícil, +eficiente

A) Exclusión mutua distribuida

Necesaria cuando **no existe un núcleo central** donde basar la exclusión mutua en variables compartidas (p.ej. **NFS** se diseña **sin estado** y no admite bloqueo de archivos, por eso UNIX añade un demonio `lockd` para las peticiones de bloqueo; o la coordinación en servicios replicados).

Requisitos:

- **ME1 – Seguridad (*safety*): a lo sumo un** proceso en la sección crítica (SC) a la vez (*"algo malo nunca sucederá"*). Si hay dos, se rompe.
- **ME2 – Vivacidad (*liveness*)**: **ausencia de interbloqueo e inanición**; todo proceso que quiera entrar en la SC **eventualmente lo logra** (*"algo bueno sucederá"*).
- **ME3 – Orden causal** (adicional/deseable): las entradas a la SC respetan la relación **ocurrió-antes**.

Tres soluciones: **servidor centralizado**, **algoritmo distribuido** (relojes lógicos) y **algoritmo en anillo**.

A.1) Servidor centralizado. Un servidor gestiona un **testigo (*token*)**.

- Para **entrar**, un proceso pide el testigo al servidor y **espera** la respuesta.
- Si el servidor **no** tiene el testigo (SC ocupada), **encola** la petición.
- Al **salir**, el proceso devuelve el testigo (mensaje de **liberación**); el servidor se lo entrega al **primero** de la cola.
- **Coste**: **2 mensajes** para entrar (petición + concesión) y **1** para liberar.
- **Inconvenientes**: el servidor es un **cuello de botella** y un **punto crítico de fallo**; hay que **regenerar el testigo** si el cliente que lo tiene falla.

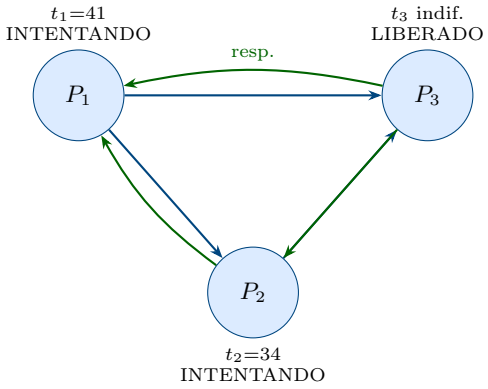
A.2) Algoritmo distribuido basado en relojes lógicos (Ricart–Agrawala). **Idea**: quien quiere entrar hace *multicast* de una petición **con marca de tiempo** a los otros $n-1$ procesos; **entra** cuando **todos** responden. **Suposiciones**: los procesos **conocen las direcciones** de los demás, el **paso de mensajes es fiable** y cada proceso mantiene su **reloj lógico**. Cada proceso P_i tiene un estado en {`LIBERADO`, `INTENTANDO`, `EN_SECCION_CRITICA`}.

Solicitar entrada (P_i): `estado := INTENTANDO`; envía la petición con marca T_i a los demás; `wait until (nº de respuestas recibidas = n-1)`; `estado := EN_SECCION_CRITICA`.

Al recibir una petición $\langle T_j, P_j\rangle$ en P_i ($i \neq j$):
`if estado=EN_SECCION_CRITICA` **o** `(estado=INTENTANDO` **y** $(T_i, P_i) < (T_j, P_j)$`)`
`then` encolar la petición de P_j;
`else` enviar respuesta a P_j;

Liberar (P_i): `estado := LIBERADO`; responder a **todas** las peticiones encoladas y eliminarlas.

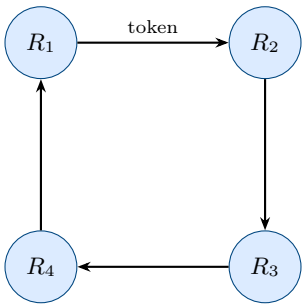
El **desempate** usa el par (T, id): a igual marca de tiempo, gana el proceso de **menor id**.



En el ejemplo: P_2 (t=34) tiene **prioridad** sobre P_1 (t=41) porque $34 < 41$. P_3 está liberado y responde a ambos. P_2 reúne las $n-1$ respuestas y entra; al salir, responde a P_1, que entonces entra.

- **Coste**: obtener el testigo requiere $2(n-1)$ **mensajes** ($n-1$ peticiones + $n-1$ respuestas). **Más costoso** que el centralizado.
- **Inconvenientes**: **cualquier** proceso es punto crítico de fallo; cada proceso recibe peticiones y envía respuestas $\Rightarrow$ puede aparecer **cuello de botella**.
- **Variante – algoritmo de votación de Maekawa**: cada proceso pide permiso sólo a su **subconjunto** asociado (*voting set*); cada proceso responde sólo a la **primera** petición recibida; las **intersecciones** de los conjuntos **no pueden ser vacías**.

A.3) Algoritmo basado en anillo. Los procesos se configuran en un **anillo lógico** (cada uno conoce a su **vecino por la derecha**); un **testigo** circula en **una sola dirección** y el proceso que lo posee puede entrar en la SC (si no, espera y lo reenvía). **Suposiciones**: cada proceso conoce a su vecino derecho y el paso de mensajes es **fiable**.



- **Seguridad**: hay **un solo testigo** en el anillo. **Vivacidad**: todos entran, aunque **no** se verifica el orden **ocurrió-antes**.
- **Coste**: obtener el testigo requiere entre **1 y** N mensajes; el testigo **circula continuamente** (consume ancho de banda aun sin peticiones).
- **Inconvenientes**: si un proceso falla hay que **reconfigurar el anillo**; hay que **regenerar el testigo** si se pierde; latencia alta en el peor caso ($N-1$ saltos).

Solución	Mensajes/entrada	Retardo	Problemas
Servidor centralizado	2 (entrar) + 1 (salir)	2 mensajes	Cuello de botella y punto único de fallo.
Distribuido (Ricart–Agrawala)	$2(n-1)$	$2(n-1)$ mensajes	Cualquier proceso es punto de fallo; n puntos de fallo.
Anillo	de 1 a N (continuo)	de 0 a N	Testigo circula siempre; reconfigurar/regenerar; no respeta orden causal.

B) Elección

Método para **escoger un único proceso** que desempeñe un **rol concreto** (p.ej. elegir un nuevo **servidor primario** replicado cuando el anterior falla). **Requisito principal**: el elegido debe ser **único** (el de **mayor identificador**) aunque **varios** soliciten la elección **simultáneamente**. Dos soluciones: **algoritmo del Valentón (*Bully*)** y **algoritmo basado en anillo (Chang–Roberts)**.

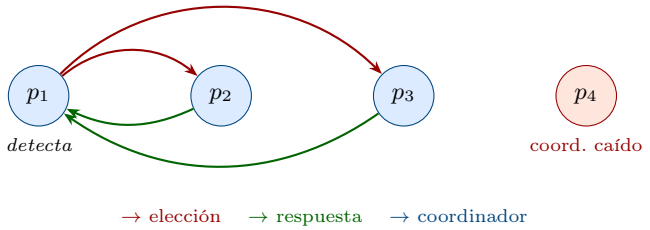
B.1) Algoritmo del Valentón (*Bully*). **Requisitos**: los miembros del grupo **se conocen** (puede haber caído más de un proceso además del coordinador); sistema **casi-síncrono** (*time-outs*) y paso de mensajes **fiable**. **Tres**

tipos de mensaje: **elección** (anuncia una elección), **respuesta** (responde a un mensaje de elección) y **coordinador** (anuncia el id del nuevo coordinador).

Pasos:

1. Un proceso inicia la elección al detectar que el coordinador ha fallado ⇒ envía **"elección"** a los procesos con **id mayor** que el suyo.
2. Espera un mensaje **"respuesta"**. Si **no** llega, se proclama coordinador y envía **"coordinador"** a todos los de id **menor**; si llega, espera el **"coordinador"** del proceso elegido.
3. Si no llega "coordinador", **reinicia** la elección; si llega, graba el id del nuevo coordinador.
4. Si recibe un "elección", devuelve "respuesta" e inicia una elección (salvo que ya la hubiera iniciado).
5. Al **restablecerse** un proceso caído, inicia una nueva elección; si tiene el id más alto, se convierte en el nuevo coordinador (el **"valentón"** se impone).

Coste: en el peor caso (el proceso de **menor id** detecta el fallo) es del orden de $O(n^2)$ mensajes.



p_1 detecta el fallo de p_4 y manda "elección" a los superiores p_2, p_3; al responderle, p_1 sabe que no será coordinador. La elección continúa hacia arriba hasta que el mayor superviviente se proclama coordinador.

B.2) Algoritmo basado en anillo (Chang–Roberts). **Requisitos**: procesos organizados en **anillo lógico** (el orden **no** tiene por qué coincidir con el de los id); paso de mensajes **fiable** y los procesos **no fallan durante** la elección. **Mensajes**: **elección** (elección en curso) y **coordinador** (anuncia el id del nuevo coordinador). Cada proceso está marcado como **participante** o **no participante**.

Pasos:

- Inicialmente todos **no-participantes**. Quien inicia se marca **participante** y envía "elección" con su id al vecino **derecho**.
- Al recibir "elección", compara el id recibido con el suyo:
  1. id recibido **mayor** ⇒ **reenvía** el mensaje;
  2. id recibido **menor** y estaba **no participante** ⇒ **sustituye** el id por el suyo y lo envía (si ya era **participante**, **no** reenvía). En ambos casos se marca **participante**;
  3. id recibido **igual** al suyo ⇒ es el de **mayor id**: se marca **no-participante** y envía "coordinador" con su id.
- Al recibir "coordinador", se marca **no-participante** y lo **reenvía** (grabando el nuevo coordinador).

Coste: en el peor caso $3N - 1$ mensajes ($N-1$ para que el id ganador alcance a su predecesor + N para completar el circuito de elección + N del mensaje coordinador).

Algoritmo	Topología	Coste (peor caso)	Notas
Valentón (*Bully*)	Todos con todos	$O(n^2)$	Casi-síncrono (*time-outs*); tolera caídas durante la elección; el de mayor id se impone.
Anillo (Chang–Roberts)	Anillo lógico	$3N - 1$	Fiable; los procesos **no** deben fallar durante la elección.

C) Consenso

Resolver de forma **precisa y generalizada** los **problemas de acuerdo**: cada proceso propone un valor, lo comunica a los demás y se fija un **valor de decisión que ya no puede cambiar**. Problemas relacionados: solución a los **generales bizantinos** (un proceso propone un valor), **consistencia interactiva** (se acuerda un **vector**) y ***multicast*** *totalmente ordenado*. Casos concretos ya vistos: transacciones (acordar una transferencia), exclusión mutua (qué proceso entra en la SC), elección (acordar el proceso elegido) y comunicación en grupo (acordar el orden de los mensajes).

Requisitos/propiedades:

- En sistemas **asíncronos no se puede garantizar** el consenso si algún proceso falla (resultado de imposibilidad FLP). Por eso se asume un sistema **casi-síncrono**.
- Comunicaciones **fiables**; los procesos pueden fallar **arbitrariamente** (***Byzantine***: caer, omitir pasos o darlos incorrectamente).
- **Terminación**: todo proceso correcto acaba decidiendo.
- **Acuerdo**: todos los correctos deciden el **mismo valor**.
- **Validez / Integridad**: si todos los correctos propusieron el mismo valor, cualquier proceso correcto escoge ese valor.

C.1) Algoritmo general de consenso. Sistema **casi-síncrono** (*multicast* básico con *timeouts*); de N procesos, hasta f pueden caer durante las rondas. Procede en $f+1$ **rondas**:

Init: $Values_i^1 := \{v_i\}$; $Values_i^0 := \{\}$.
En la ronda r $(1 \leq r \leq f+1)$:
  `B-multicast`$(g,\ Values_i^r - Values_i^{r-1})$; *// enviar sólo lo no enviado aún*
  $Values_i^{r+1} := Values_i^r$;
  al recibir V_j de p_j: $Values_i^{r+1} := Values_i^{r+1} \cup V_j$;
Tras $f+1$ **rondas**: $d_i :=$ minimum$(Values_i^{f+1})$.

El valor de decisión es el **mínimo** del conjunto común (podría ser cualquier función determinista acordada). Con $f+1$ rondas se garantiza que, aunque caiga un proceso mientras difunde, algún proceso correcto propagó ya el valor.

r	**Values**	**P1**	**P2**	**P3**	**P4**	**P5**
1	1	7	4	5	×	3
2	2	7 5 4 3 ×	7 5 4 3	7 5 4 3		7 5 4 3
3	3		7 5 4 3 **1**	7 5 4 3		7 5 4 3
Resultado (d_i)			$d = \text{mín}(7,5,4,3,\dots) = 3$ (para los correctos)			

(× indica que el proceso cae mientras difunde. Con $f=2$ caídas se necesitan $f+1=3$ rondas.)

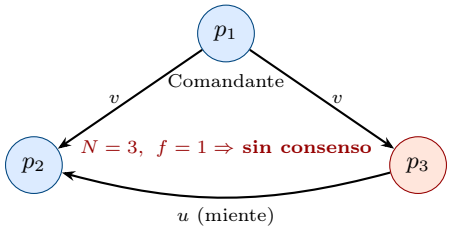
Propiedades: **Terminación** (obvia: sistema al menos parcialmente síncrono por *timeouts*); **Acuerdo** (todos llegan al mismo conjunto de valores tras la ronda final y aplican la misma función); **Integridad** (se aplica a los valores propuestos por **todos**, no sólo los correctos).

C.2) Problema de los Generales Bizantinos. Tres o más generales deben acordar **atacar o retirarse**; uno o más pueden ser **traidores** (defectuosos). El **comandante** emite la orden y los **tenientes** deben decidir:

- **Comandante traidor**: propone a uno atacar y a otro retirarse.
- **Teniente traidor**: miente sobre la orden que dio el comandante.

Es un **caso especial** de consenso: **sólo un proceso** (el comandante) propone el valor inicial, con **fallos arbitrarios** de los procesos (cualquier mensaje/valor e incluso omisión) y procesos correctos con *timeouts* (casi-síncrono). **Requisito de integridad**: si el comandante es correcto, **todos** acuerdan su valor.

Resultado de Lamport (imposibilidad): con **3 procesos y 1 traidor no hay solución**; en general **no hay solución si** $N \leq 3f$. La solución existe sólo si $N > 3f$ (más de $3f$ procesos para tolerar f traidores), requiere $f+1$ **rondas** de mensajes y los procesos aplican la **función *majority*** a los valores recibidos, devolviendo un valor especial $\perp$ cuando el resultado es indefinido (el comandante falla).



Complemento (ejercicio de la tarea T3): Bizantinos en Blockchain. El problema de los generales bizantinos modela el **consenso descentralizado** entre nodos que no confían entre sí. **Blockchain** lo resuelve de forma probabilística/económica: p.ej. **Proof-of-Work** (el coste computacional hace inviable que una minoría imponga una cadena falsa; se acepta la **cadena más larga**) o protocolos **BFT** deterministas (p.ej. *PBFT*) que toleran hasta f nodos maliciosos con $N > 3f$. Así se logra **acuerdo** sobre el orden de las transacciones sin autoridad central.

Tablas resumen

Semánticas de invocación (T2)

Semántica	Garantías	Cuándo
Quizás (*maybe*)	No reintenta	El fallo es tolerable; sin garantías.
Al menos una vez	Reintenta, no filtra, reejecuta	Válida sólo si la operación es **idempotente** (Sun RPC).
Como mucho una vez	Reintenta, filtra duplicados, retransmite respuesta	La más fuerte: RPC, RMI, CORBA.

Fórmulas y costes clave (T3)

Concepto	Expresión
Cristian	p fija su reloj a $t + T_{round}/2$; exactitud $\pm(T_{round}/2 - min)$.
Lamport	LC1: $C_p{:=}C_p{+}1$ por evento. LC2: $C_q{:=}\max(C_q, t)$ al recibir. $a \to b \Rightarrow C(a){<}C(b)$ (no recíproco).
Orden total	$(C(a), p) < (C(b), q) \iff C_p(a){<}C_q(b) \vee (C_p(a){=}C_q(b) \wedge p{<}q)$.
E.M. centralizado	2 mensajes entrar, 1 liberar.
E.M. Ricart–Agrawala	$2(n-1)$ mensajes por entrada.
E.M. anillo	1 a N mensajes; testigo en circulación continua.
Elección Valentón	$O(n^2)$ mensajes (peor caso).
Elección anillo	$3N - 1$ mensajes (peor caso).
Consenso general	$f + 1$ rondas; $d_i = \min(Values^{f+1})$.
Bizantinos	Sin solución si $N \leq 3f$; solución si $N > 3f$, con $f + 1$ rondas y función *majority*.

Paradigmas de comunicación (T2) de un vistazo

Paso de mensajes	Explícito, `send`/`receive`, *marshalling*, canales; síncrono/asíncrono; fiable (TCP) o no (UDP).
RPC	Transparencia local/remoto; *stubs*, IDL, *binding* (portmap); síncrona o asíncrona.
Cliente/Servidor	Protocolo petición/respuesta (R, RR, RRA); semánticas de fallo; `doOperation`/`getRequest`/`sendReply`.
Citas (*rendez-vous*)	Servidor activo; no determinismo con instrucciones guardadas; `in`/`accept` (Ada), `select`/`poll` (POSIX).