Tema 4.2: Coordinación y consenso

Sistemas Distribuidos

2022-2023

Introducción

La coordinación incluye problemas como:

- La exclusión mutua en el acceso a recursos
- Algoritmos de elección (acuerdo)
- Multidifusión en grupos de procesos
- Consenso

Veremos algunos de ellos.

Exclusión mutua

Exclusión mutua

- Existe un recurso que no puede ser accedido por varios procesos a la vez.
- Debe garantizarse que sólo uno de ellos tiene el recurso (ejecuta la **sección crítica**)

Este problema es clásico en los sistemas no-distribuídos y se resuelve con:

- *Mutexes* (cerrojos)
- Semáforos
- Monitores

Pero ¿y en un sistema distribuído?

La solución con semáforo, mutex, etc funciona porque el Sistema Operativo se encarga de crear estos objetos y controlar su acceso a ellos.

> El sistema operativo es el árbitro

Exclusión mutua en sistemas distribuídos

En un sistema distribuído ¿quién sería el árbitro?

- Solución **centralizada**: Designar a un proceso como árbitro
  - Los demás procesos le "piden permiso"
- Solución **descentralizada**: No hay árbitro.

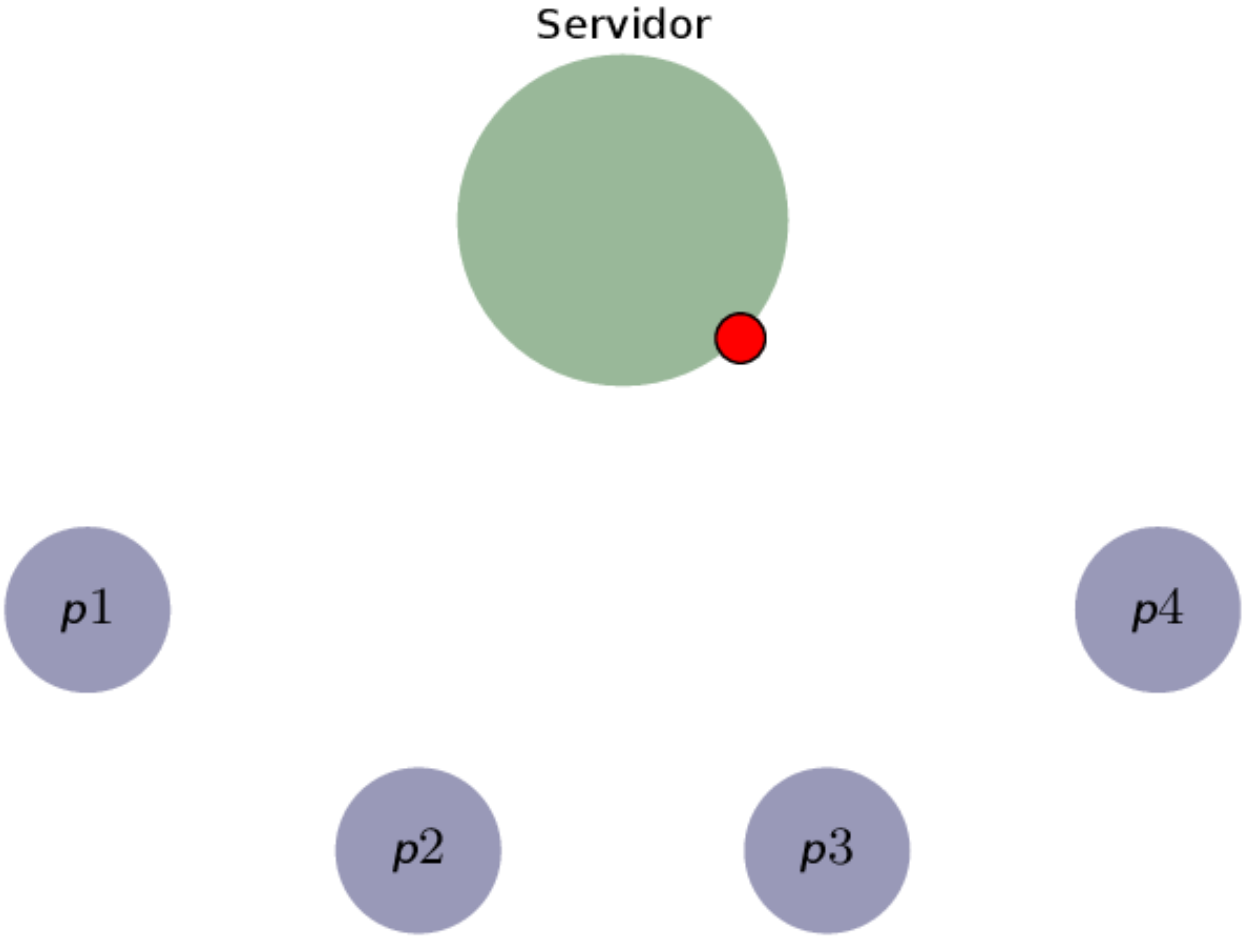
Propiedades que debería cumplir el algoritmo

1. **Exclusión mutua**. Un proceso como máximo puede estar en su sección crítica.

2. **Pervivencia**. Ningún proceso "*muere de hambre*". Todo el que pida entrar en su sección crítica consigue el permiso tarde o temprano.

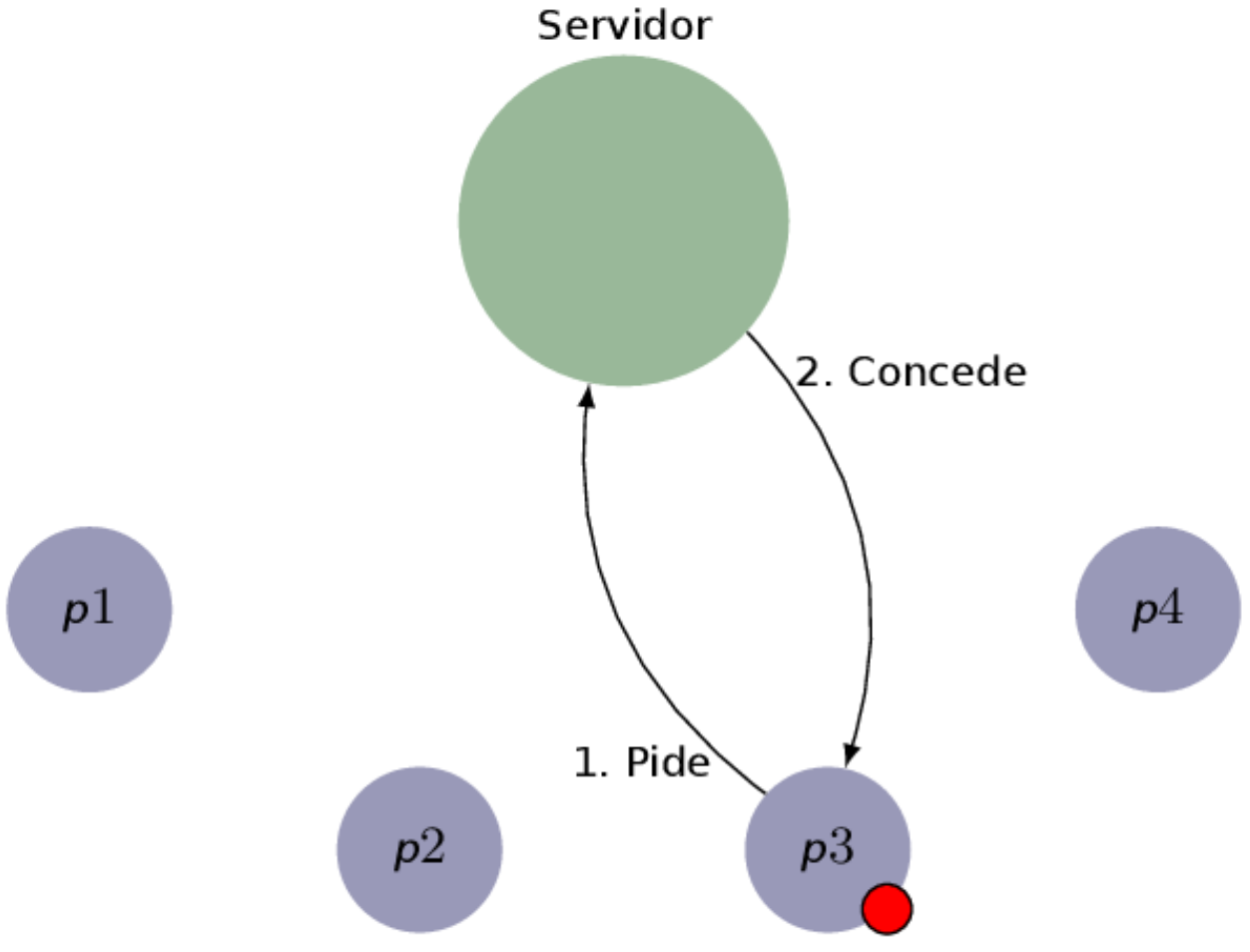
3. (Opcional) **Ordenación**. Se respeta el orden de las peticiones.

Solución con servidor-árbitro



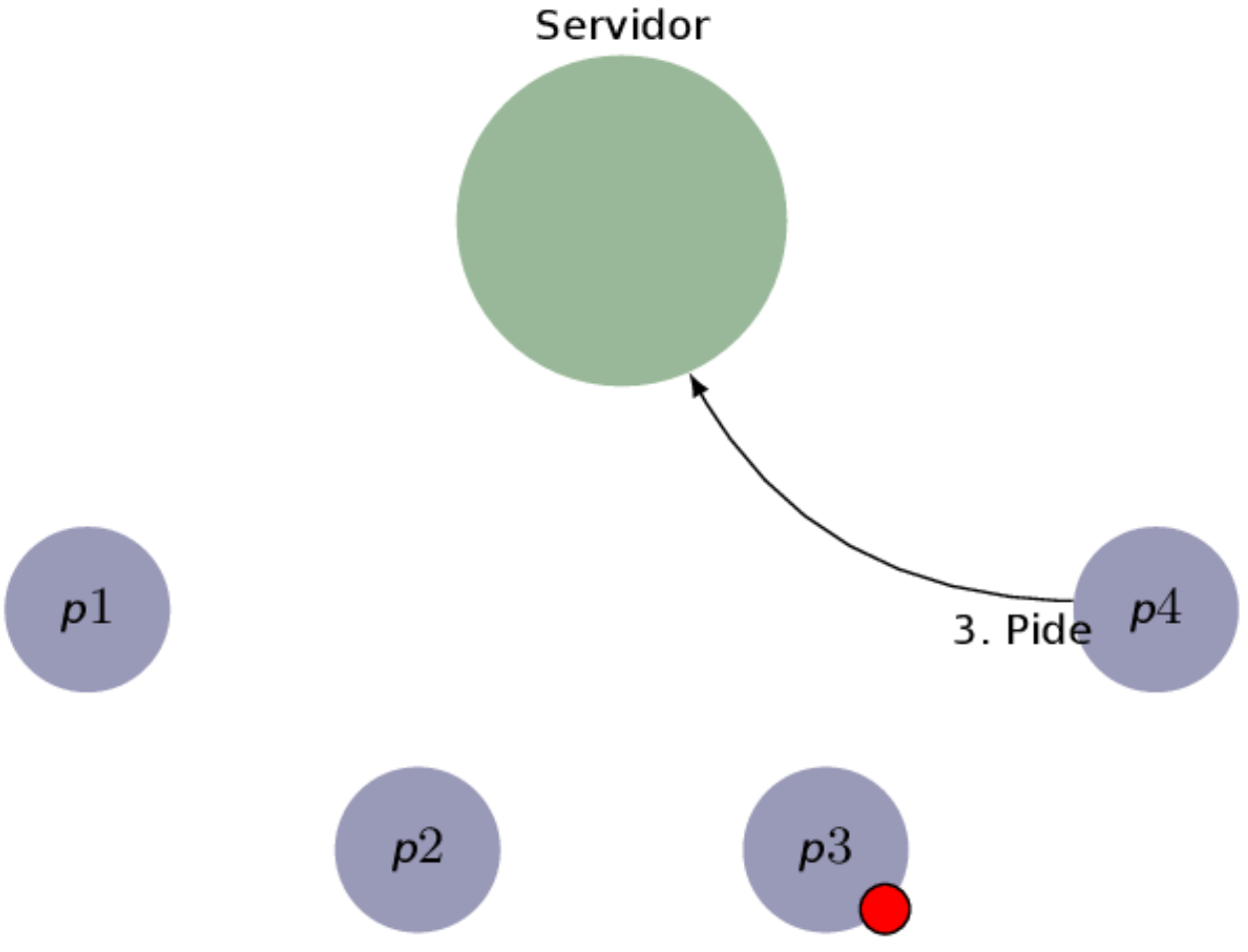
El servidor mantiene un "token" que pasa al primero que lo pida, y una lista de quienes esperan por él.

Solución con servidor-árbitro



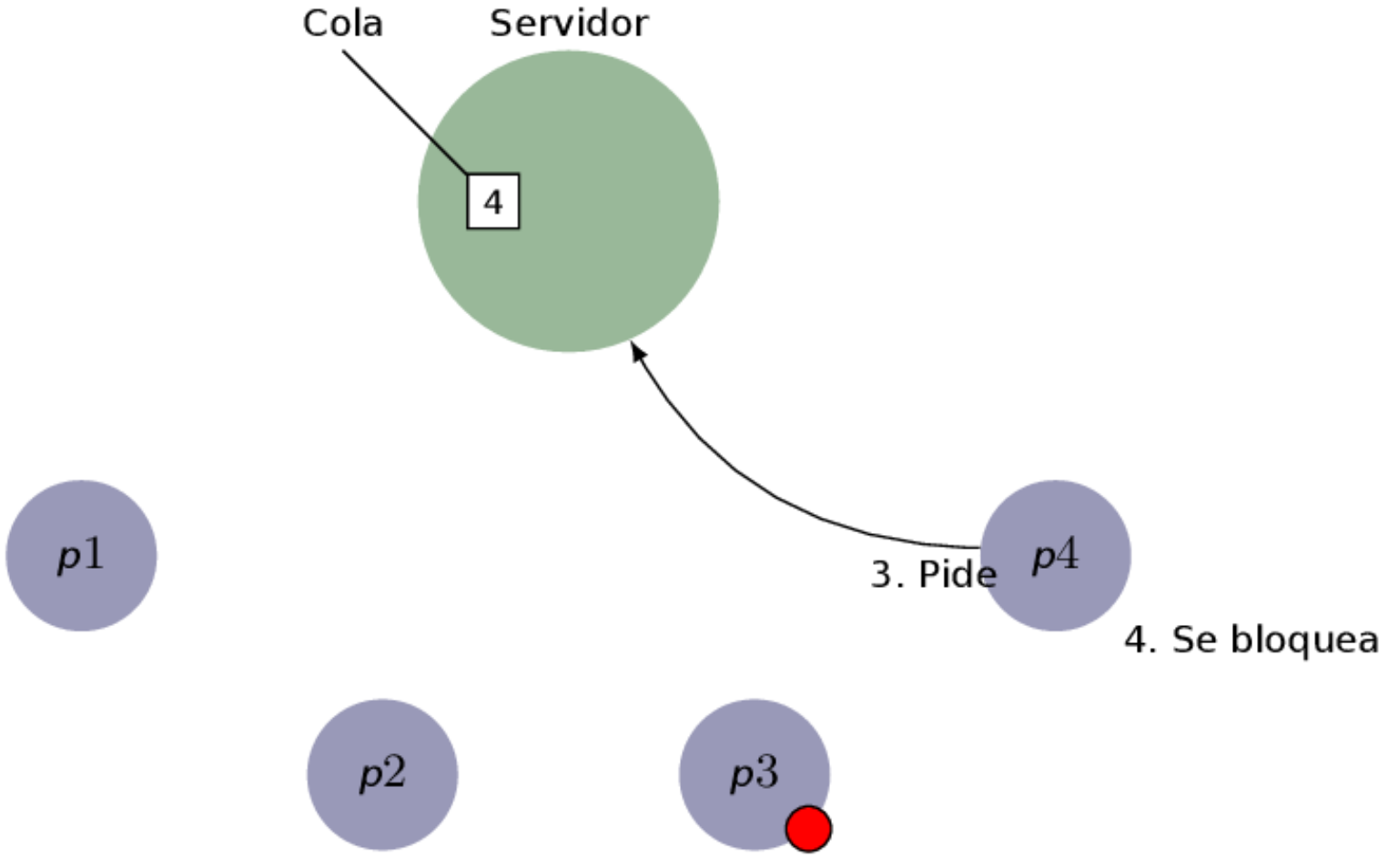
Por ejemplo, el cliente $p3$ pide el token, y el servidor se lo concede.

Solución con servidor-árbitro



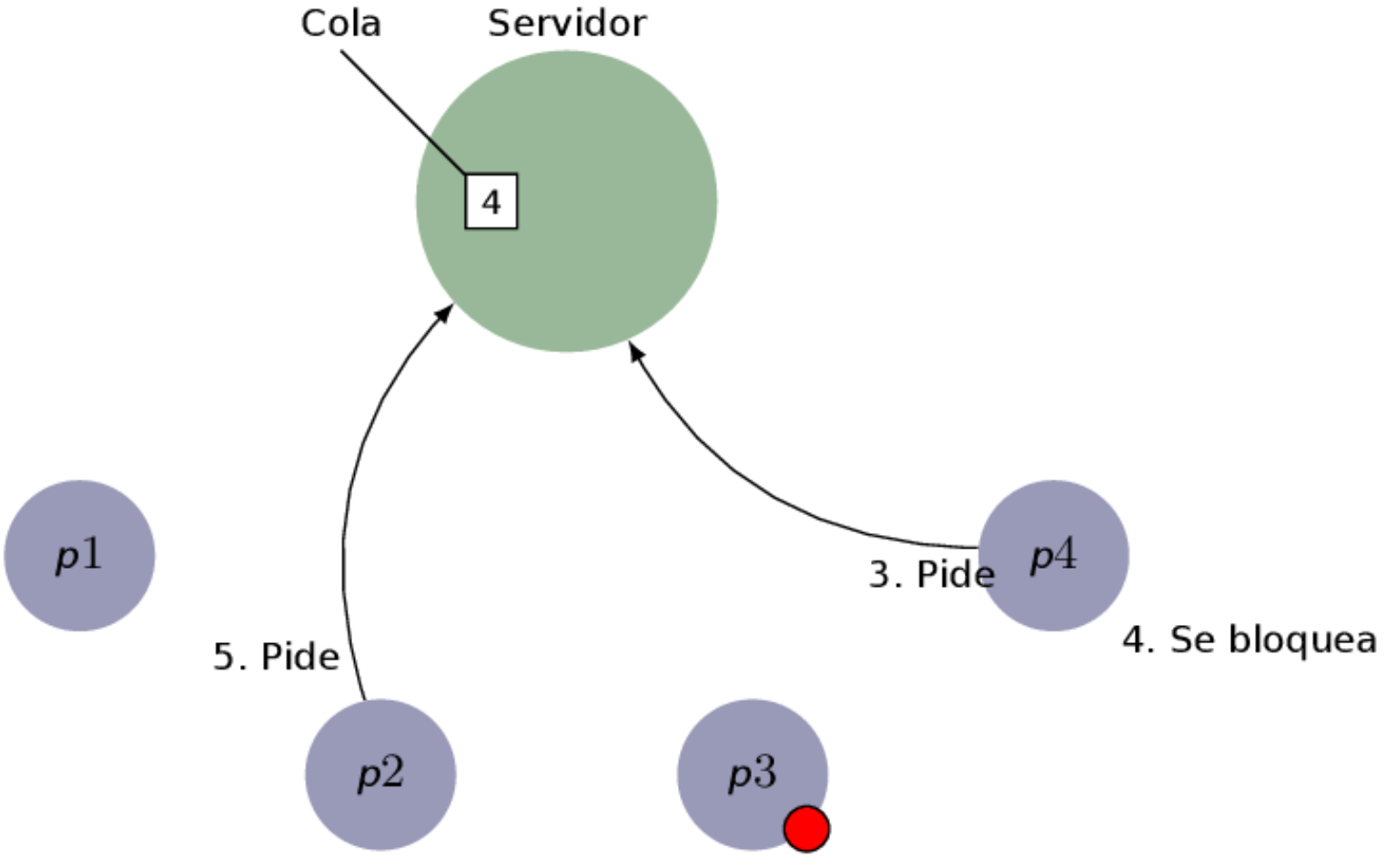
Cuando otro cliente solicite el token ...

Solución con servidor-árbitro



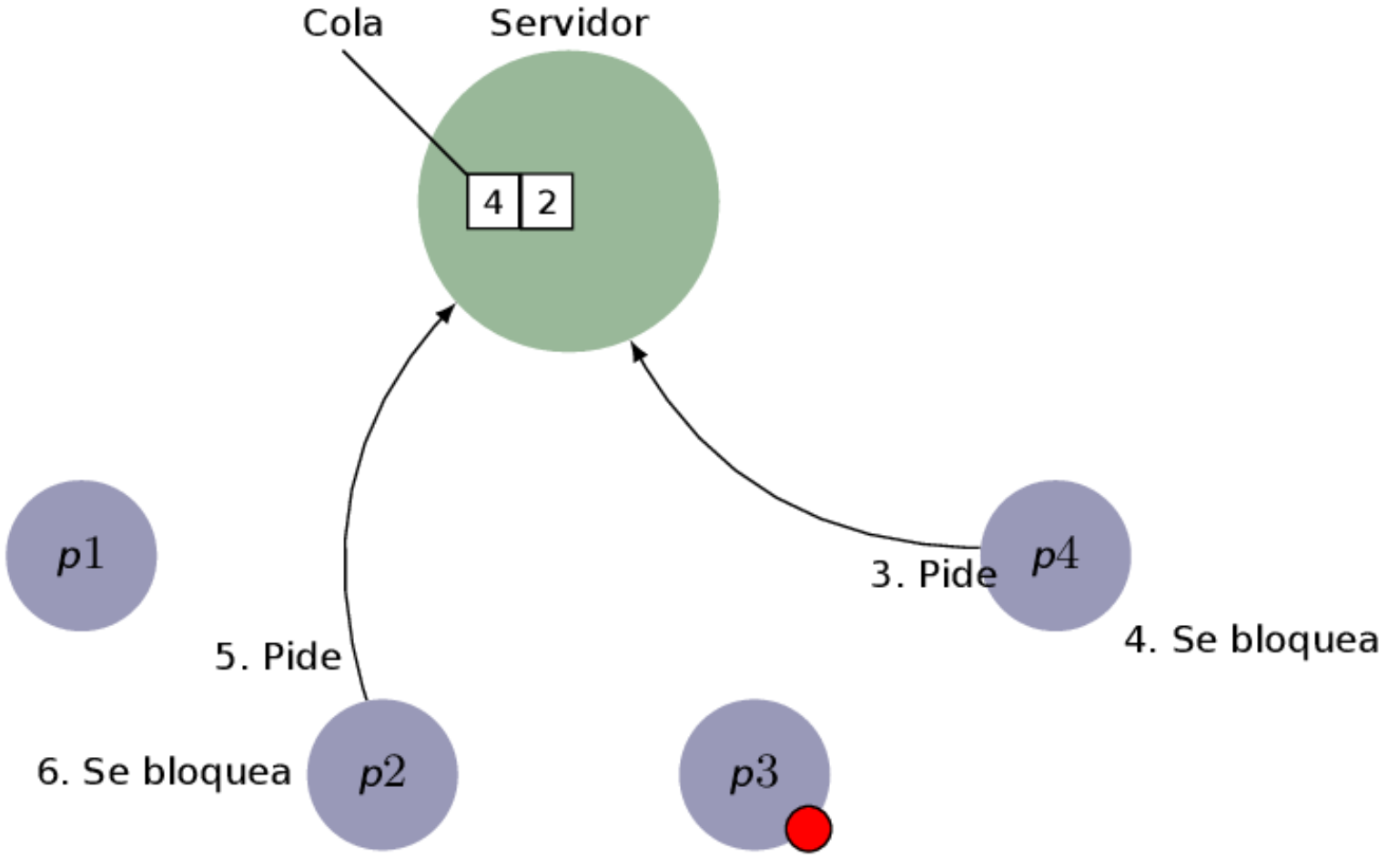
... el servidor no le responde, con lo que el cliente queda bloqueado y no entra en su sección crítica. El servidor "recuerda" que $p4$ lo había pedido.

Solución con servidor-árbitro



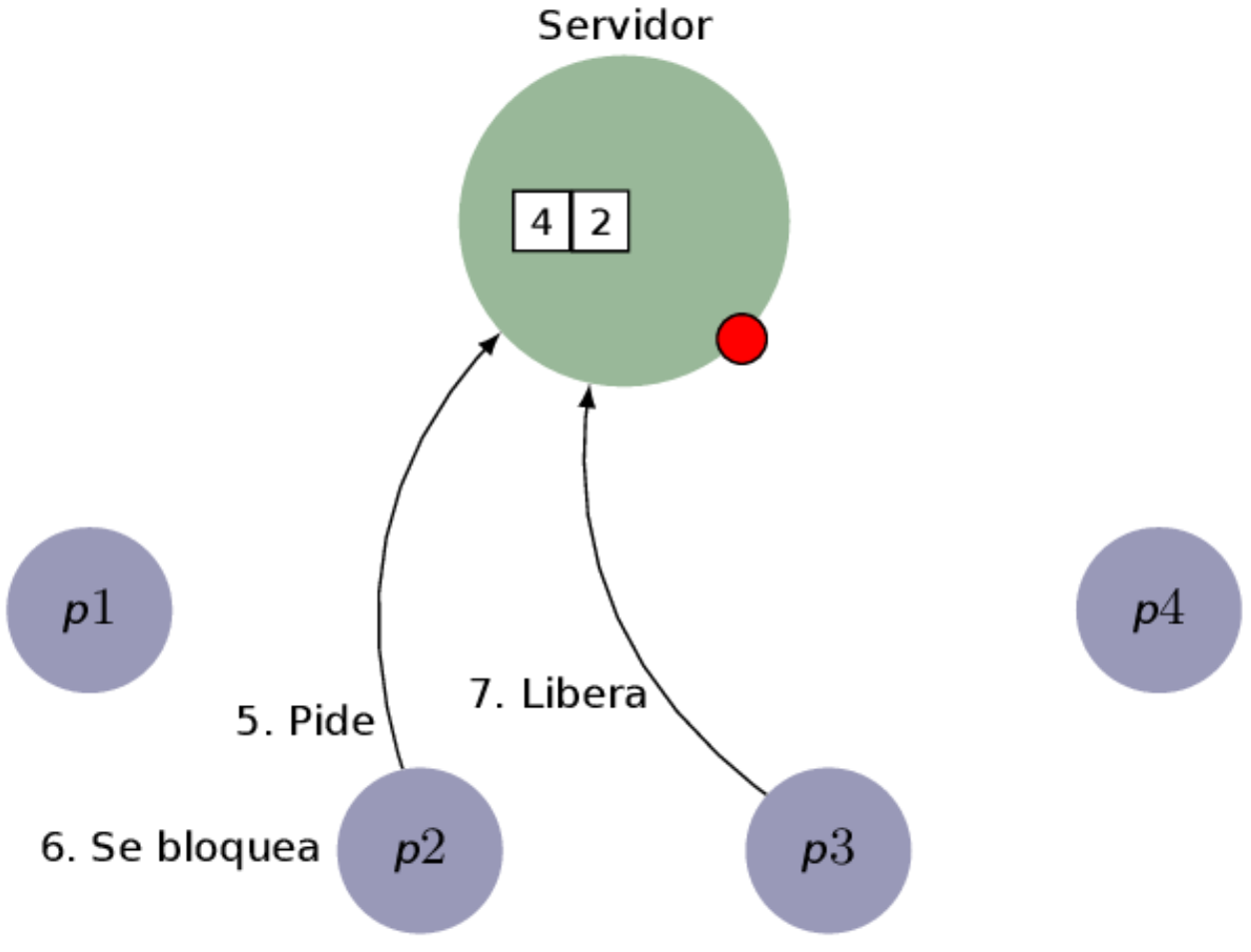
Cada nuevo cliente que solicite el token...

Solución con servidor-árbitro



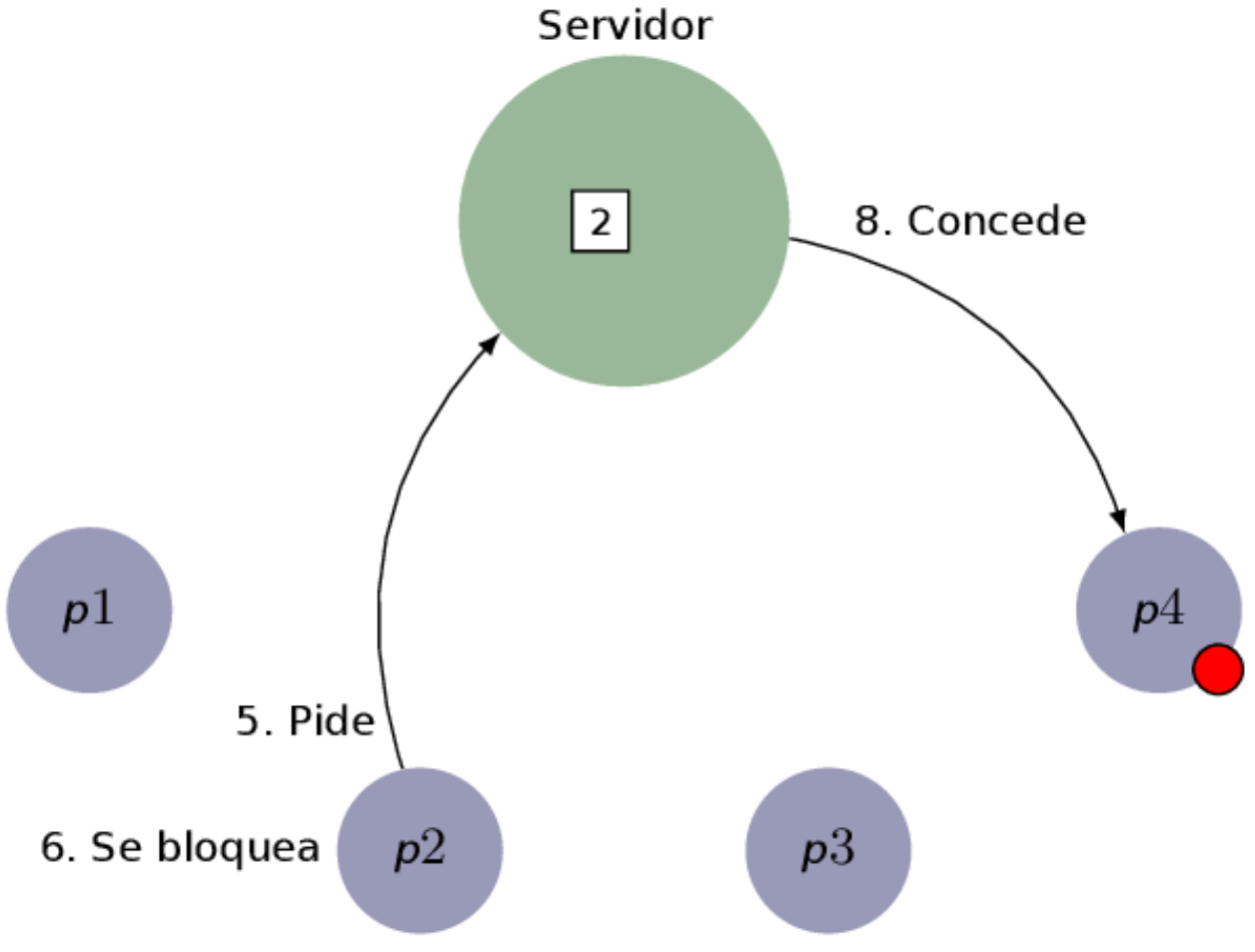
... es recordado por el servidor en una cola

Solución con servidor-árbitro



Cuando $p3$ salga de su sección crítica, "devuelve" el token al árbitro.

Solución con servidor-árbitro



El árbitro consulta su lista y le responde al que lleva más tiempo esperando ($p4$)

Ventajas

- Garantiza las propiedades 1 y 2
- Según cómo se implemente el servidor puede garantizarse también la 3 (cola FIFO)

Inconvenientes

- El servidor es un *punto de fallo único* (si falla, no funciona nada)
- Si se pierde un *token* (fallo de red) hay un bloqueo infinito.

Soluciones sin servidor

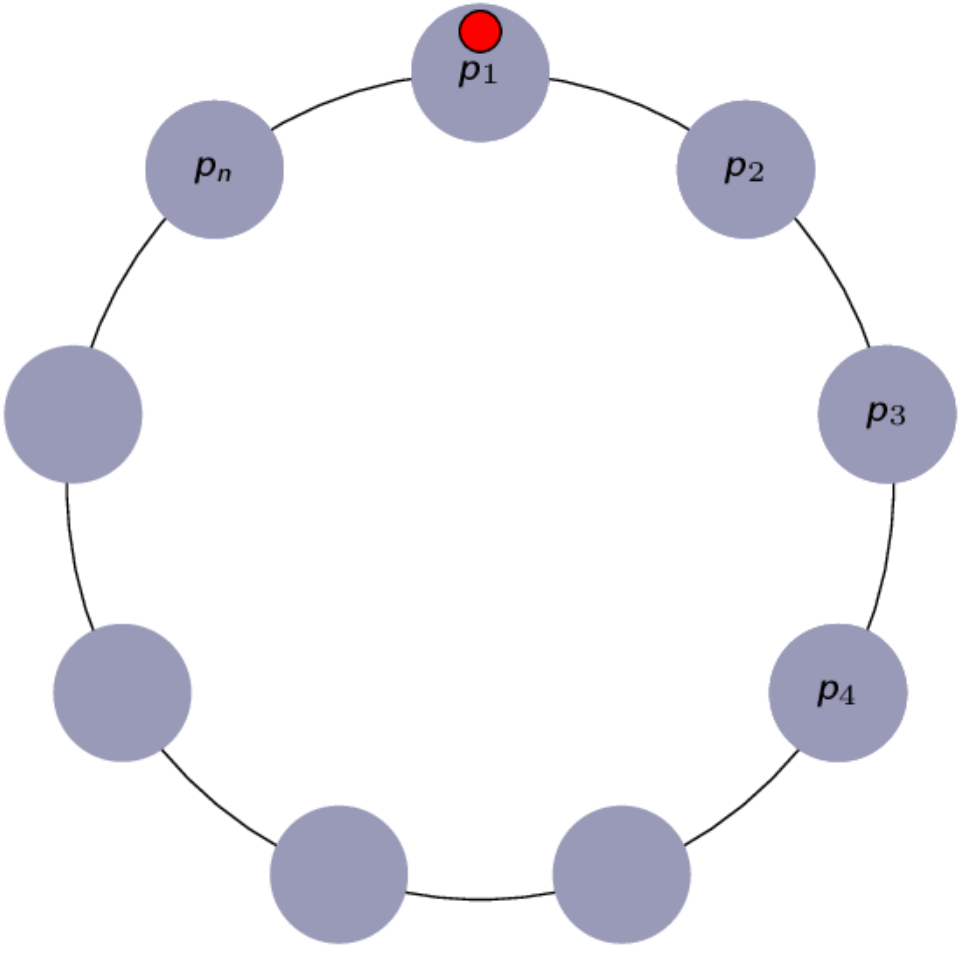
- Paso de un *token* "en anillo"
  Lo veremos a continuación
- Algoritmos P2P basados en Tablas Hash Distribuidas
  Una clave en la tabla hash puede contener quién tiene el token
- Algoritmos de "votación"
  En vez de un solo token hay n, mantenidos por n nodos. Para entrar en la sección crítica un nodo debe conseguir al menos $n/2$ nodos.

Sólo estudiaremos el primer caso

Paso de *token* en anillo

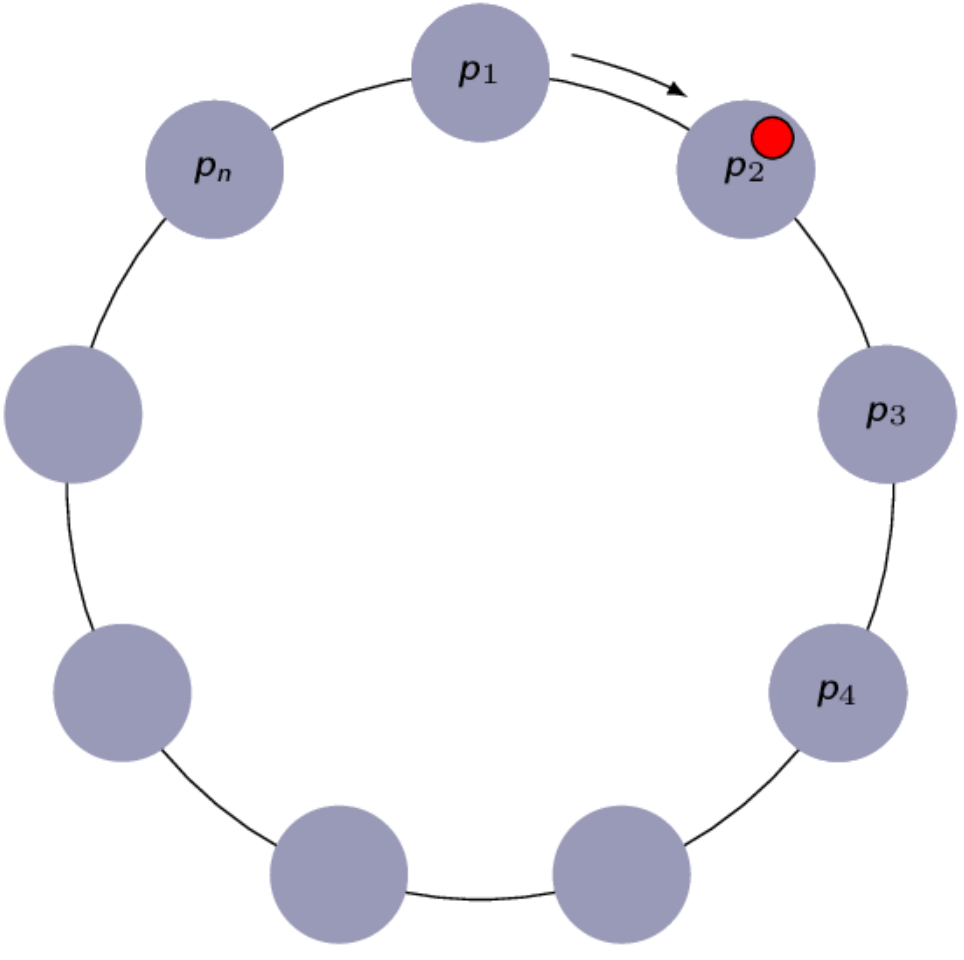
- Se asigna a cada proceso un número
- De forma que puedan ser ordenados
- Cada nodo conoce la IP y puerto de su nodo siguiente
- Mediante mensajes punto-a-punto, se pasan un *token*
- El que tenga el *token* puede:
  - Entrar en su sección crítica si estaba esperando por ello
  - Pasar el token al siguiente si no
- Al salir de la sección crítica, ha de pasar el token

Representación gráfica



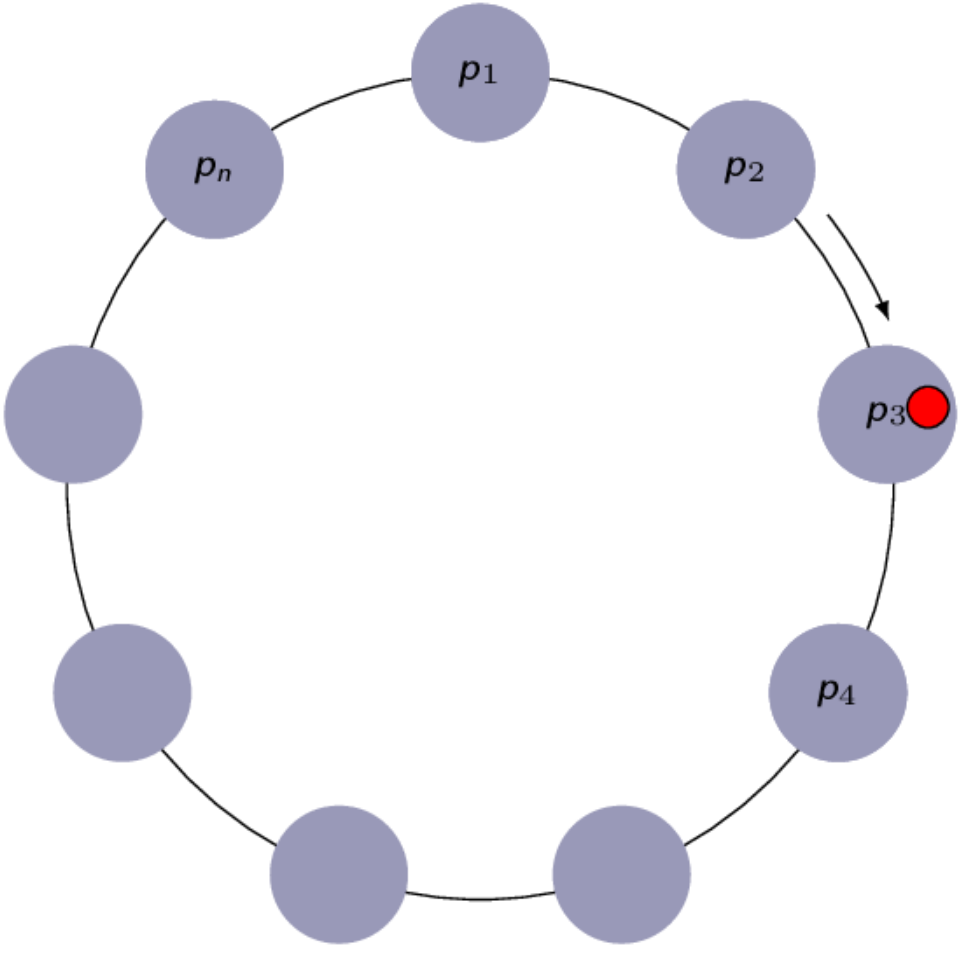
El "token" es un mensaje que va de un nodo al siguiente
Supongamos que $p4$ quiere entrar en su sección crítica

Representación gráfica



$p2$ recibe el token.
Ya que no quiere entrar en su sección crítica, lo pasa.

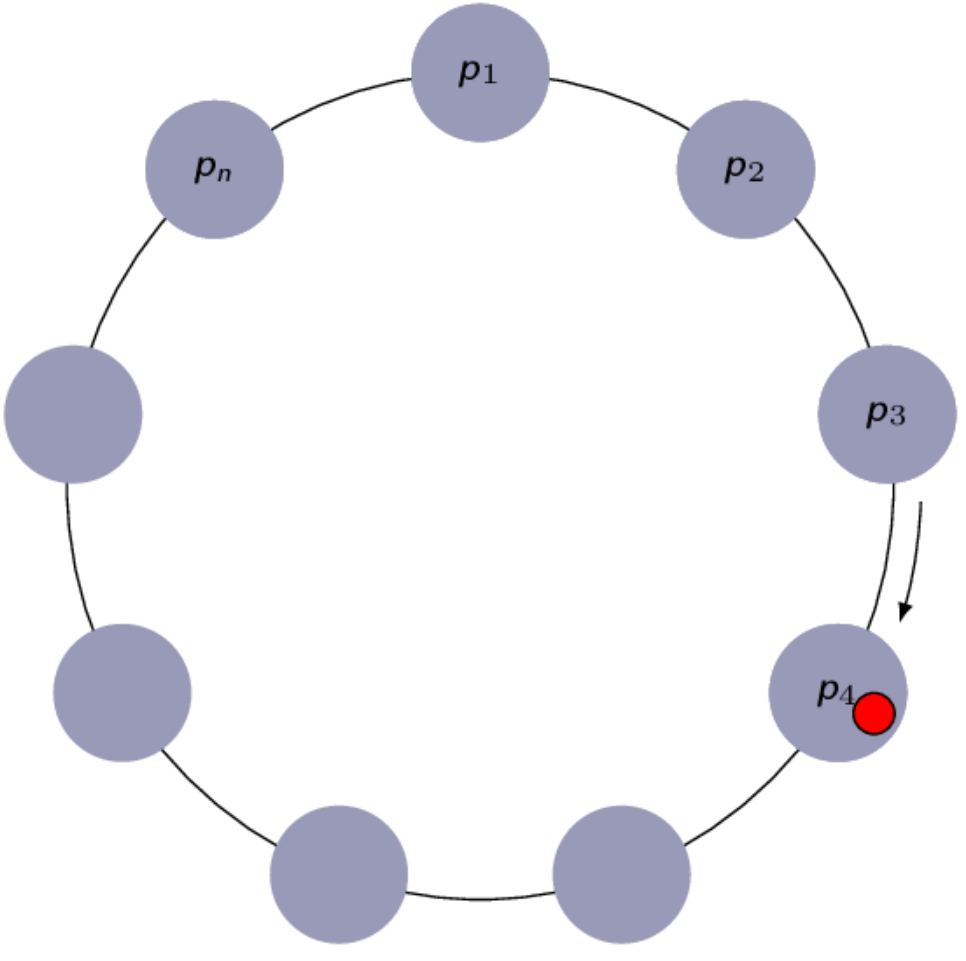
Representación gráfica



Supongamos que $p3$ quiere ahora entrar en su sección crítica.

Ha recibido el token, de modo que no lo pasa. $p4$ debe seguir esperando

Representación gráfica



Cuando $p3$ sale de su sección crítica, pasa el token.

Ahora $p4$ puede entrar.

Ventajas

- Garantiza las propiedades 1 y 2
- Pero no la 3
- Muy sencillo de implementar. No requiere proceso extra

Inconvenientes

- Consume ancho de banda continuamente
- Si el testigo se pierde, debe ser regenerado
- No es fácil descubrir cuándo se pierde
- Si un proceso del anillo muere repentinamente, debe ser detectado por el anterior para "saltárselo" (pero esto implica que cada nodo conoce a todos los demás)

Consenso

El problema

Varios nodos de un sistema distribuído deben tomar una decisión, y es necesario que todos tomen la misma decisión.

Ejemplo: Cada proceso controla un motor de una nave espacial.

Deben decidir si apagarse todos o seguir encendidos todos.

- Uno propone una acción
- Debe asegurarse que la acción propuesta es comunicada a todos
- Cuando se haya alcanzado esa garantía, se efectúa la acción

Un ejemplo más mundano

- Una base de datos tiene varias réplicas (para mejor tolerancia a fallos)
- Un cliente conecta con una cualquiera de las réplicas y le pide guardar un dato. Este debe ser comunicado a todas las réplicas, para que todas lo guarden.
- Imagina que una de las réplicas encuentra un problema y no puede guardar el dato.
- El sistema distribuido debe funcionar "como un todo". O bien todas las réplicas guardan el dato, o ninguna lo hace.

El consenso debe alcanzarse para poder informar al usuario si el dato ha sido guardado o no.

Consenso: el verdadero problema

Los **nodos** pueden fallar

- Uno podría dejar de funcionar (parar)
- Quizás pueda reiniciarse y volver a funcionar, pero "se ha perdido" cosas que han pasado en el sistema.
- O peor aún, enviar mensajes equivocados o contradictorios (fallo *bizantino*)

La **red** puede fallar

- ¿Datos corruptos o fuera de orden? (TCP se ocupa de esto)
- Partición de la red (nodos inalcanzables)
- Es difícil (¿imposible?) distinguir esa situación de la de "nodo caído"

Modelos de sistema

Ya que hay muchos posibles tipos de fallo y sistemas, debemos ser más precisos:

- ¿Qué fallos pretendemos detectar?
  (*crash-fail*, particiones, fallos *bizantinos*)
- ¿Cómo son las comunicaciones en nuestro sistema?
  (existe garantía de tiempo de respuesta o no)
- ¿Qué tipo de consistencia buscamos?
  (fuerte o eventual)

Un algoritmo de consenso dado funcionará para una elección concreta de los tres puntos anteriores.

Tipos de fallo

Nodos:

- ***Crash-fail*.** Supone que una vez que un nodo falla, deja de responder y de hacer operaciones hasta que se reinicie.
  Ej: *segfault*, apagado repentino

- **Fallos *bizantinos*.** Uno nodo que falla no deja de funcionar, sino que pasa a comportarse erráticamente, y puede seguir enviando mensajes, quizás contradictorios.

  Estrategia: tratar de detectar y parar

Red:

- **Particiones.** La red pierde alguno de sus enlaces, y se generan dos o más partes *aisladas*. Hay comunicación dentro de cada isla, pero no de una isla a otra.
  Más tarde la partición *se sana*

Tipo de comunicación

- **Sistemas síncronos.** Existen relojes sincronizados. La red, por diseño, tiene un tiempo máximo de respuesta. Si la respuesta no llega en ese tiempo es seguro asumir que ya no llegará (este tipo de redes no es realista)

- **Sistemas asíncronos.** No hay reloj global. Los relojes no están perfectamente sincronizados. La red no tiene acotado el tiempo de respuesta. Ejemplo: Internet

Tipo de consistencia buscada

- **Fuerte**. Las operaciones se ven en el mismo orden desde todos los nodos.
  El sistema se comporta "como un solo nodo".
  Si un cliente solicita una modificación en el sistema, cuando el cliente es respondido afirmativamente, cualquier otro cliente vería esa modificación (aún si consulta otro nodo).

- **Eventual**. Las operaciones no se ven necesariamente en el mismo orden en todos los nodos, pero llevan a la larga al mismo resultado.
  Aún si un cliente ha solicitado un cambio y ha sido respondido afirmativamente, otros clientes podrían tardar en ver el cambio. No se comporta "como uno solo"

¿Qué es razonable esperar?

Lo deseable no siempre es posible.

Ideal. Sistema asíncrono, fallos arbitrarios (bizantinos) y consistencia fuerte.

Es imposible.

Resultado FLP

Fischer-Lynch-Paterson (1985)

> En un sistema asíncrono **no existe algoritmo** que garantice el consenso (ni siquiera eventual) ante el fallo de un solo nodo (ni siquiera *crash-fail*)

Implicaciones:

- Resultado de gran importancia teórica
- Pero que no se tiene en cuenta en la práctica, ya que **sí existen algoritmos** que pueden alcanzar consenso con alta probabilidad (si bien no 100%)

Conclusión (Teorema CAP)

Parece que es imposible lograr a la vez los tres objetivos siguientes (CAP):

- **C**onsistencia fuerte
- **A**vailability (que el sistema, globalmente, no se detenga ante fallos, sino que pueda continuar con los nodos que aún funcionan)
- **P**articiones (que las tolere)

Si queremos garantizar la **C**onsistencia fuerte, cuando un nodo falla (*crash-fail*) y los demás lo detecten, deberían parar a la espera de que el que falló rearranque.

Si se detecta una **P**artición, deberían parar todos a la espera de que la red se reunifique. Pero eso impide la **A**vailability.

Teorema CAP gráficamente

Debe elegirse CA, CP o AP.
En la práctica, hay que elegir entre CP o AP (pues P siempre está presente)

Casos prácticos y soluciones conocidas

- Red asíncrona. Fallos *crash-fail*. Particiones. Consistencia eventual. SOLUCIÓN: Paxos, RAFT
- Red asíncrona. Fallos bizantinos. Particiones. Consistencia eventual o imposible. SOLUCIÓN: votación ("Los generales bizantinos")
- Red asíncrona. Fallos bizantinos. Particiones. Uso de criptografía. Consistencia eventual. SOLUCIÓN: BlockChain (bitcoin)

Sólo veremos, bastante por encima, los casos de "Los generales bizantinos" y "Paxos".

Coordinación ante fallos bizantinos

El problema de los dos generales

Es la mínima expresión de un problema de consenso.

- Dos generales están separados, sitiando una ciudad.
- Deben decidir si al día siguiente atacar o no.
- Deben decidir lo mismo (si atacara uno solo, perdería)
- Se comunican a través de un mensajero, que podría no llegar a destino (ser capturado)

¿Qué protocolo podría idearse para resolverlo?

¿Quizás un *three way handshaking* como el que usa TCP? 🤔

— Atacaré si me respondes que tú también
— Y yo atacaré si tú me respondes a éste
— Atacaremos!

El problema de los dos generales no tiene solución

Supongamos ahora otro modelo de fallo:

- La red no falla (los mensajes no se pierden, acaban llegando)
- Pero los nodos pueden fallar y emitir mensajes arbitrarios y contradictorios.

El problema se denomina ahora **Los Generales Bizantinos**

- ¿Cómo haríamos un protocolo para soportar fallos bizantinos?
- ¿Cuántos fallos bizantinos (nodos) simultáneos pueden soportarse?
- ¿Merece la pena hacerlo?

Los generales bizantinos

Problema propuesto y resuelto por Leslie Lamport [1982], que dio nombre al tipo de fallo.

- Hay un comandante y $N-1$ tenientes (total: N procesos)
- El comandante cursa una orden ("ataque" o "retirada") a sus $N-1$ tenientes
- Cada teniente comunica a los demás lo que le ha dicho el comandante (para redundancia)
- Puede haber un *traidor*:
  - Si el comandante es traidor, puede dar órdenes contradictorias
  - Si el teniente es traidor puede comunicar al revés la orden del comandante

La solución del problema debe cumplir:

- **Terminación**: Pasado un tiempo, los tenientes deben tomar una decisión
- **Acuerdo**: Todos los tenientes correctos habrán tomado la misma decisión
- **Integridad**: Si el comandante era correcto, la decisión tomada debe coincidir con la del comandante

¿Tiene solución?

- Lamport demostró que para $N = 3$ no hay solución con que sólo haya un traidor.
- En general, si k procesos son defectuosos (*traidores*), debe haber al menos $2k + 1$ que no lo sean.
- En caso de que haya $2k + 1$ no defectuosos, existe un algoritmo que garantiza el consenso entre ellos, pero requiere k rondas de mensajes, y un total de N^k mensajes transmitidos.
- Las rondas pueden reducirse usando criptografía.

En general implementar un sistema tolerante a fallos bizantinos es demasiado costoso y no suele merecer la pena.

Blockchain

- **Blockchain** [Nakamoto, 2008] es una estructura de datos distribuida, criptográficamente segura y replicada,
- Tolerante a fallos bizantinos y particiones,
- **Resuelve** en la práctica (aunque no en la teoría) el problema de la coordinación,
- pero es **muy costosa** computacionalmente y requiere incentivos económicos en los nodos participantes.

Usada en Bitcoin y en otras criptodivisas.

Coordinación ante fallos *crash*

Paxos, RAFT

¿Qué buscan estos algoritmos?

Que un conjunto de nodos separados,

- que pueden fallar y reiniciarse (*crash-fail*),
- unidos por una red que puede fallar y reconectarse (*partition*),

se pongan de acuerdo en una secuencia de operaciones (un *log*).

De modo que todos ellos al ejecutar operaciones del log lo hagan en el mismo orden y por tanto alcancen el mismo estado.

Paxos, RAFT

- **Paxos** [Lamport, 1998] es el más antiguo y el más estudiado. Tiene merecida fama de ser muy difícil de comprender y de implementar.
- **RAFT** [Ongaro, 2013] es una alternativa más sencilla de comprender que logra los mismos objetivos.

Paxos

En realidad Paxos intenta resolver antes un problema más simple: que varios nodos tomen la misma decisión.

Ejemplo práctico. Quedar para cenar.

- Un grupo de amigos tiene que decidir un sitio donde cenar.
- Su único medio de comunicación es con SMS, punto a punto (no hay Whatsapp de grupo)
- El SMS va fatal (los mensajes pueden llegar arbitrariamente tarde, o no llegar nunca). Quien envía un mensaje no sabe si fue recibido o no.
- Los teléfonos van fatal (se quedan sin cobertura, sin batería, se reinician)
- Pero cada teléfono tiene copia (que sobrevive al reinicio) de los mensajes que recibió.

Objetivo:

- La mayoría del grupo ha de tomar una decisión en tiempo finito, y la decisión debe ser la misma para todos (nadie debe tomar una decisión diferente)

Paxos: solución

El algoritmo parte de esta idea:

- En una primera fase hay que elegir un **líder**
  - Cualquiera se convierte en candidato, emitiendo al resto un mensaje que diga "Quiero ser lider"
  - Los demás le responden "OK, propón tú"
  - Al recibir una **mayoría** de OK, el candidato se convierte en líder (los demás pasan a ser seguidores)
- El *líder* **propone** un sitio para cenar, enviando a todos los demás: "Propongo cenar en X".
- Los seguidores **aceptan** emitiendo un mensaje a todos los demás: "Acepto cenar en X"
- Cuando cualquier nodo recibe una **mayoría** de "Acepto cenar en X", sabrá que esa es la decisión final.

> Las *mayorías* son la mitad + 1, lo que exige número **impar** de nodos

Paxos: ¿y si algo va mal?

La solución anterior funciona bien si las comunicaciones no fallan y todos los nodos están siempre funcionando. Pero...

- ¿Y si el candidato se apaga tras haber propuesto su candidatura?
- ¿Y si el líder se apaga tras haber hecho la propuesta?
- ¿Y si hay una partición?

La solución es añadir a cada nodo un tiempo de espera, y si transcurre ese tiempo sin recibir mensajes del líder, se autopropone nuevo candidato.

- ¿Y si se autoproponen varios a la vez?
- ¿Y cómo sabe el nuevo candidato si el anterior ya había propuesto algo?
- ¿Y si la propuesta ya había sido aceptada por varios nodos?
- ¿Y si el anterior líder "resucita" e intenta proseguir?

Paxos: la verdadera solución

Para que funcione bien incluso ante los diferentes fallos enumerados el algoritmo **se complica mucho**.

- Cada mensaje debe incluir un "número de ronda" R
- El número lo elige el candidato, y debe ser mayor a cualquier número de ronda previo.
- El número de ronda se tiene en cuenta para saber si los mensajes vienen de un líder que ya ha sido sustituído por otro más nuevo.
- Cuando un nodo ya había emitido un "Cenaré en X" y luego recibe una candidatura nueva, en lugar de responderle "OK, propón tú", responde "OK, pero ya me había comprometido a X en la ronda R"
- El nuevo líder debe proponer entonces la X que corresponda a la R más alta.

Paxos tiene fama de difícil

> Paxos es difícil de comprender y de implementar, pero está matemáticamente demostrado que es correcto.

- **1989**: Lamport escribe el artículo pero no lo publica (lo explica en conferencias)
- **1990**: Lo envía a una revista
- **1998**: Se publica finalmente con el título "*The Part-Time Parliament*"
- **2001**: publica una explicación "*Paxos made simple*", igualmente compleja y difícil de comprender. Deja "al lector" demasiados detalles de implementación.
- **2007**: ingenerios de Google publican "*Paxos made live*", completando muchos de esos detalles y posibilitando implementaciones reales.
- **2012**: El artículo es premiado por ACM por su relevancia.

Raft

- Raft simplifica el algoritmo PAXOS
- Y lo generaliza, ya que permite obtener consenso no sólo en una decisión, sino en una *serie de ellas* (Paxos requería una variante llamada Multipaxos para ésto)

Raft tiene recursos web que explican claramente su funcionamiento, incluyendo un simulador.

https://raft.github.io/