

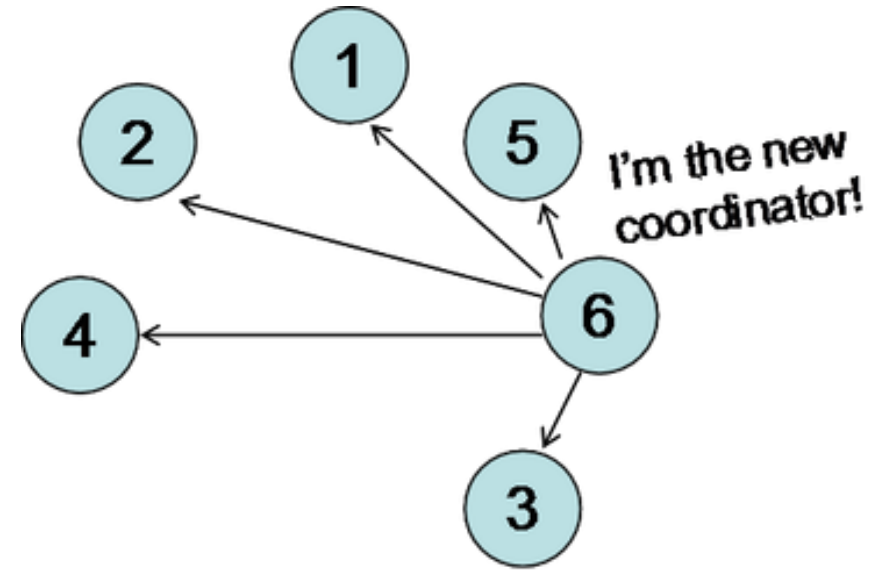
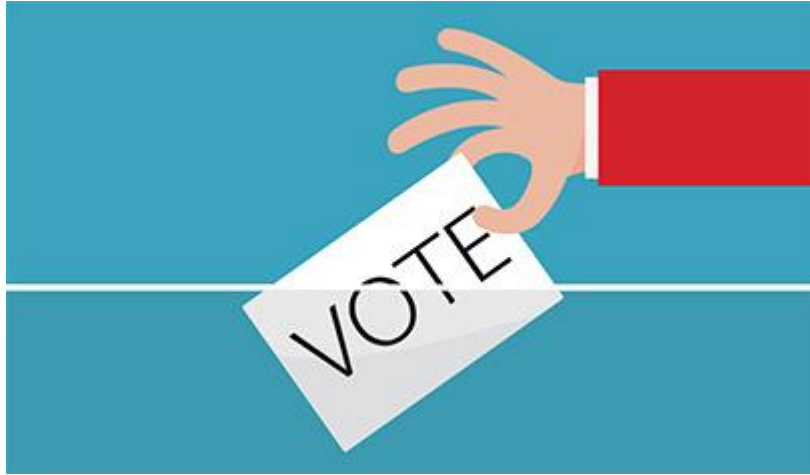
Algoritmos de elección y consenso

SISTEMAS DISTRIBUIDOS E INTERNET

Vicente García Díaz
garciavicente@uniovi.es

Objetivos

1. Algoritmos de elección
2. Algoritmos de consenso



Algoritmo Bully

Algoritmo de elección

¿Cuál es la idea?

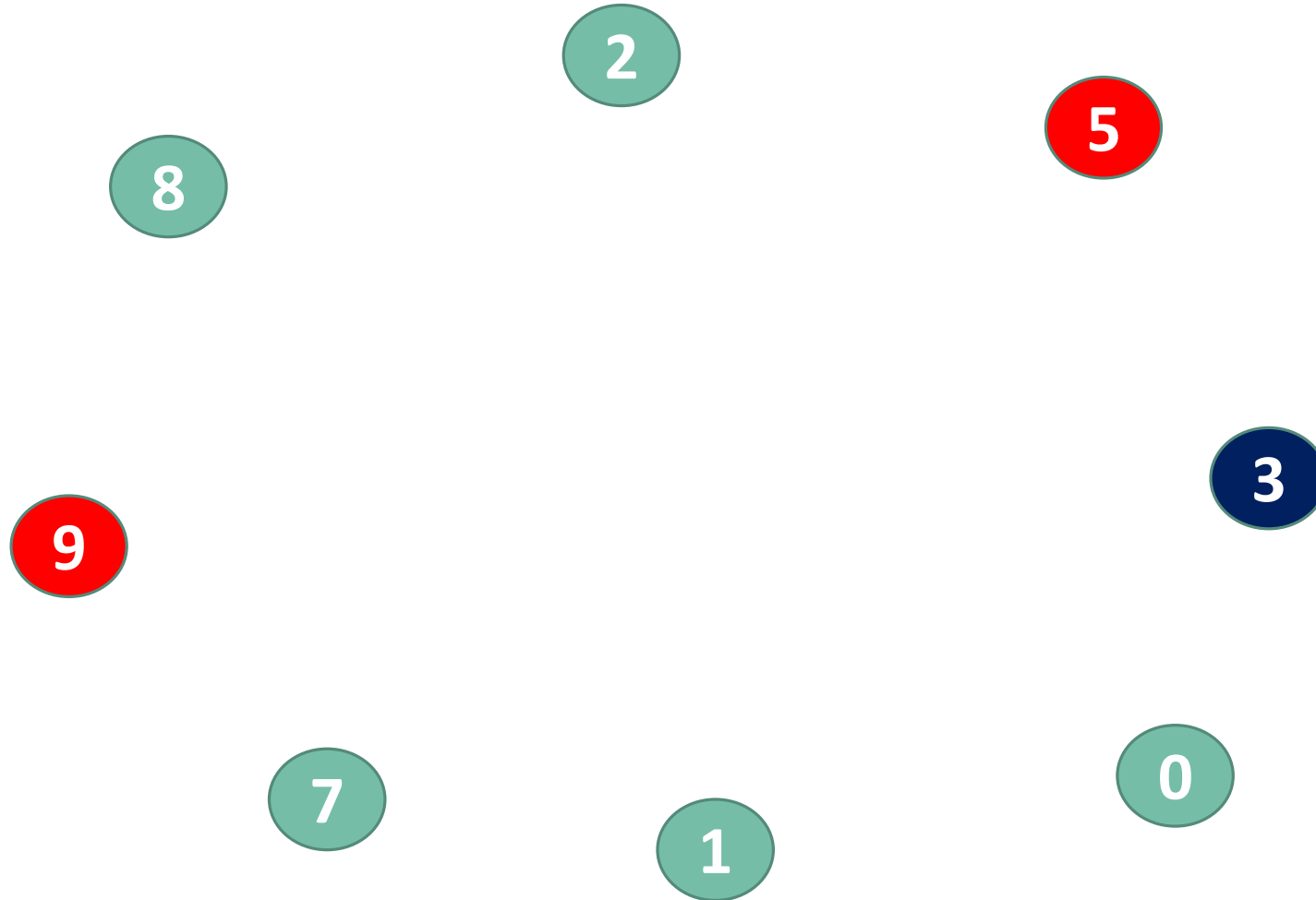
- Elegir un líder en un grupo de nodos distribuidos
 - Para ello se elegirá aquel con un ID más “grande”
- Consideraciones
 - Cada proceso tiene un ID único
 - Todos los procesos saben el ID de los demás nodos
 - No necesariamente, podría saberse únicamente qué nodos son más importantes pero no en qué medida
 - El envío de mensajes es confiable
 - El sistema es síncrono

¿Cuándo necesitamos un líder?

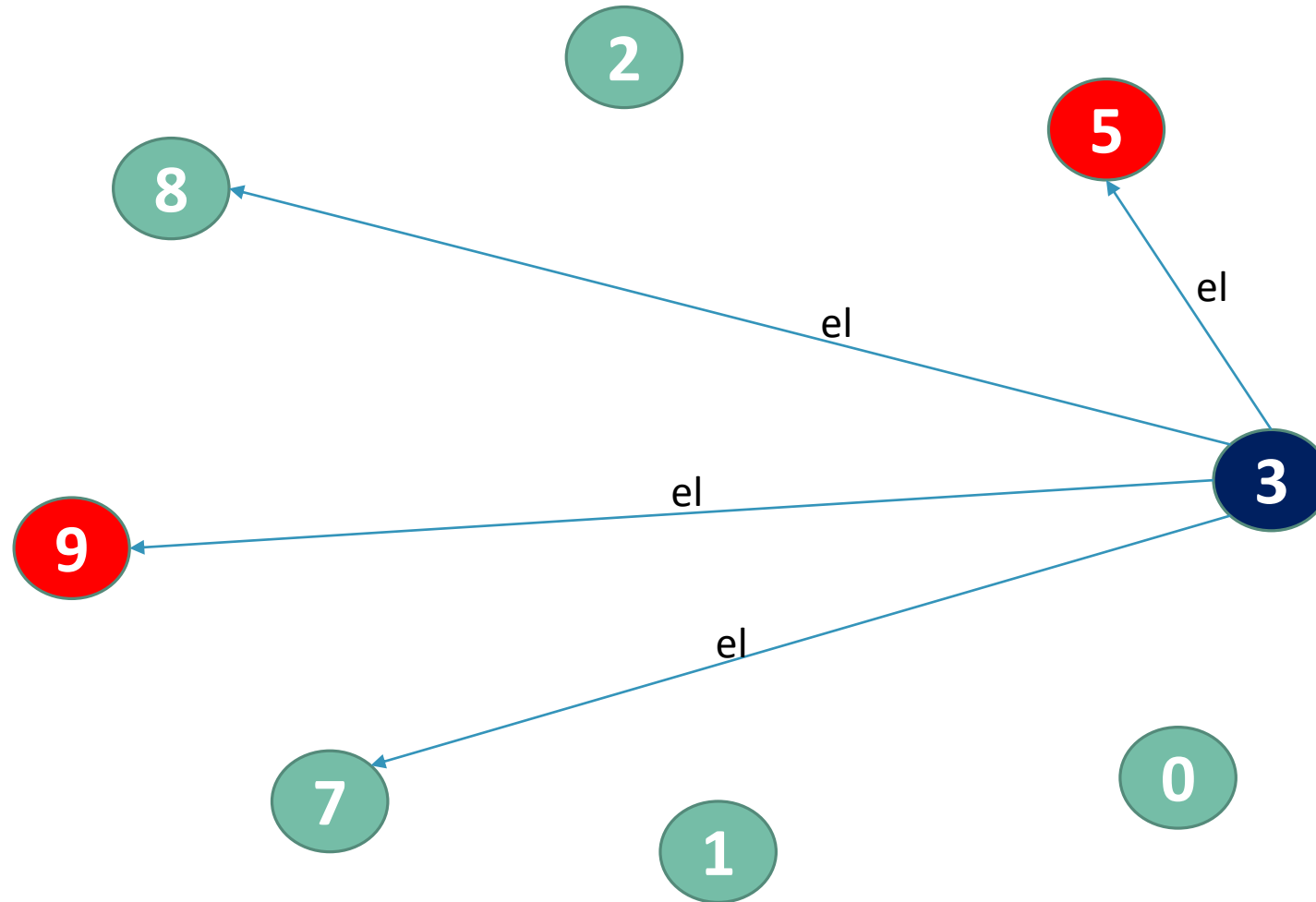
1. Cuando el sistema se inicializa
2. Cuando un nodo nota que el líder se ha caído
3. Cuando nodo caído vuelve a estar operativo

Un sistema con nodos independientes

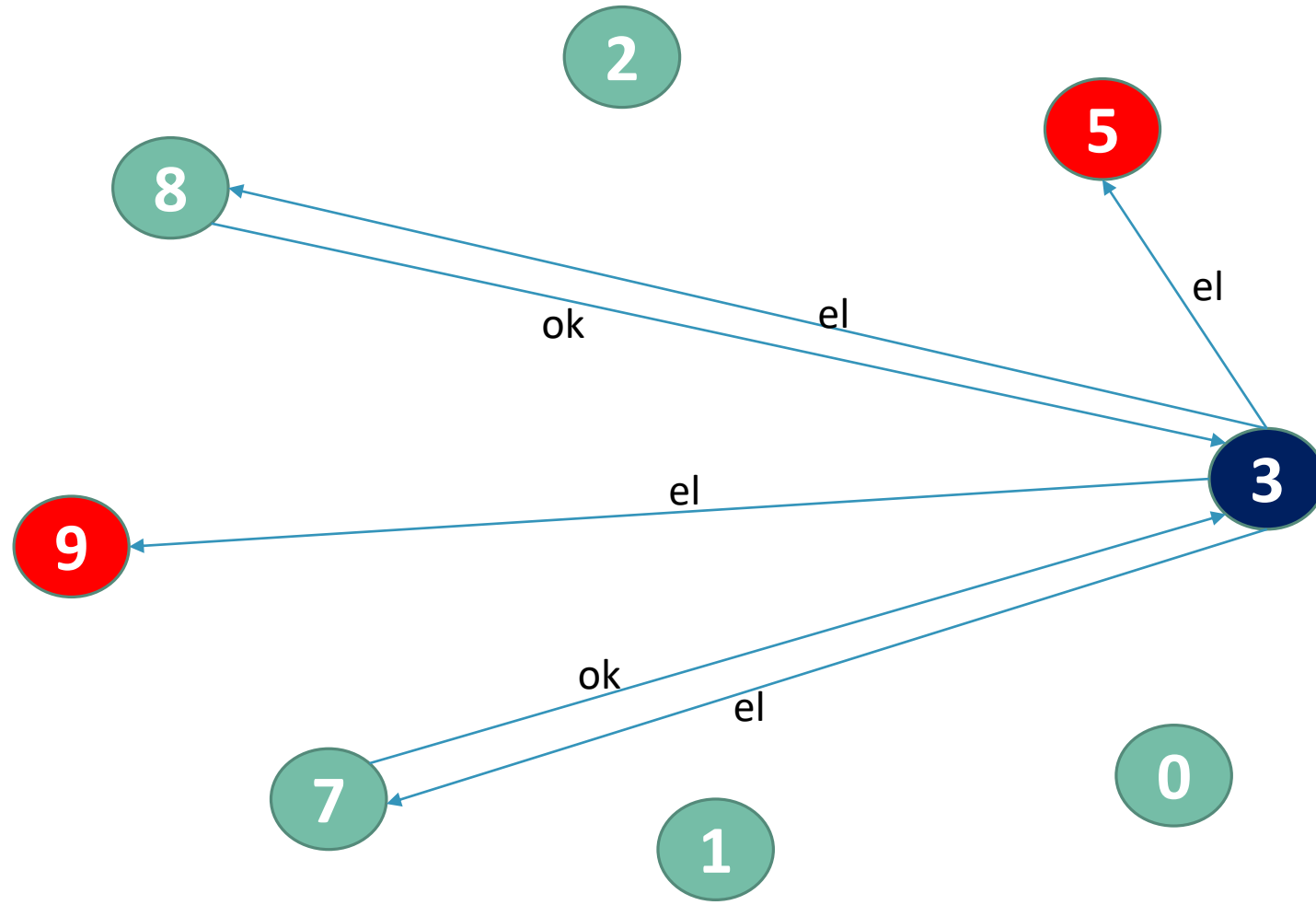
2 fases



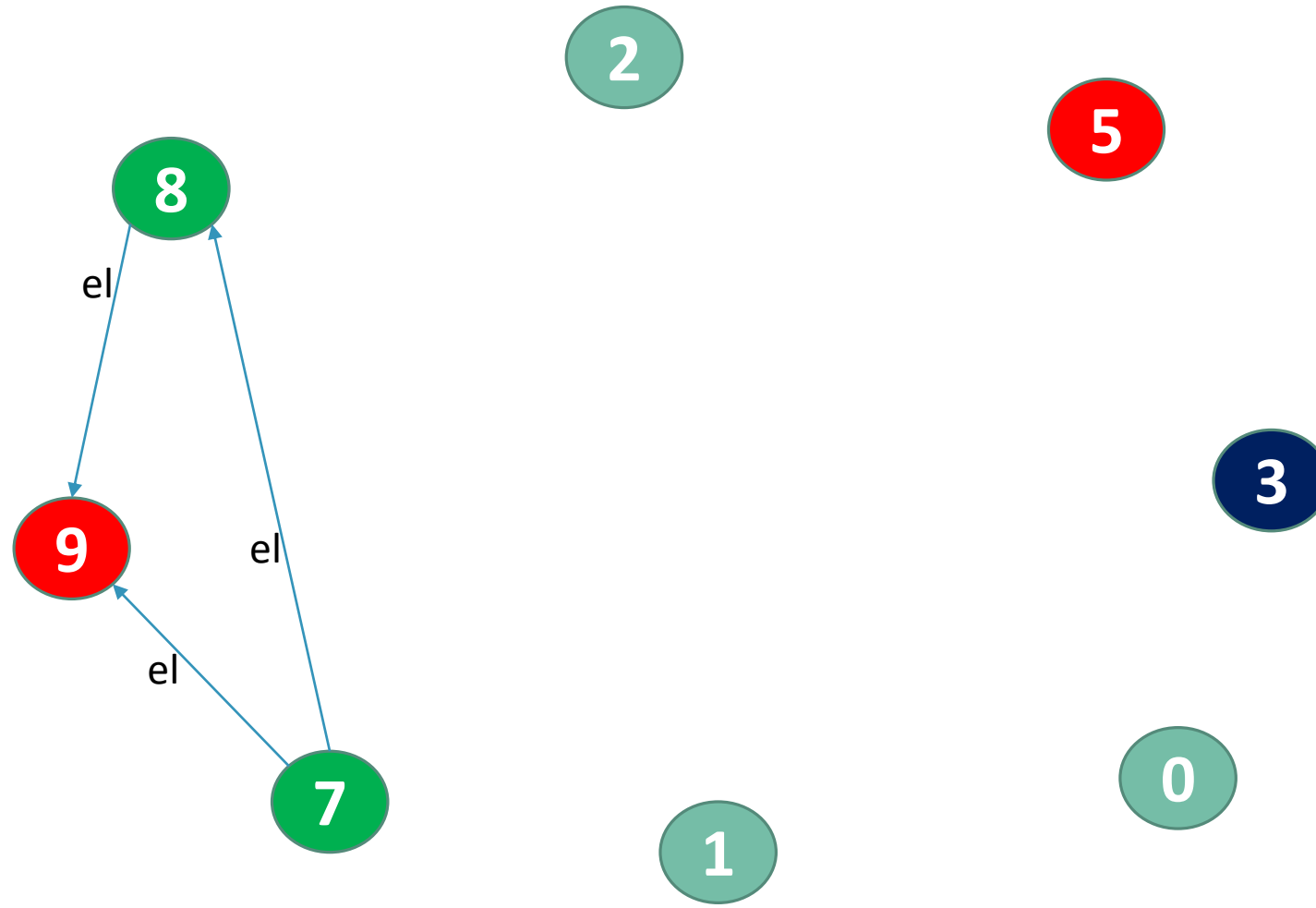
Se envían mensajes “elección”



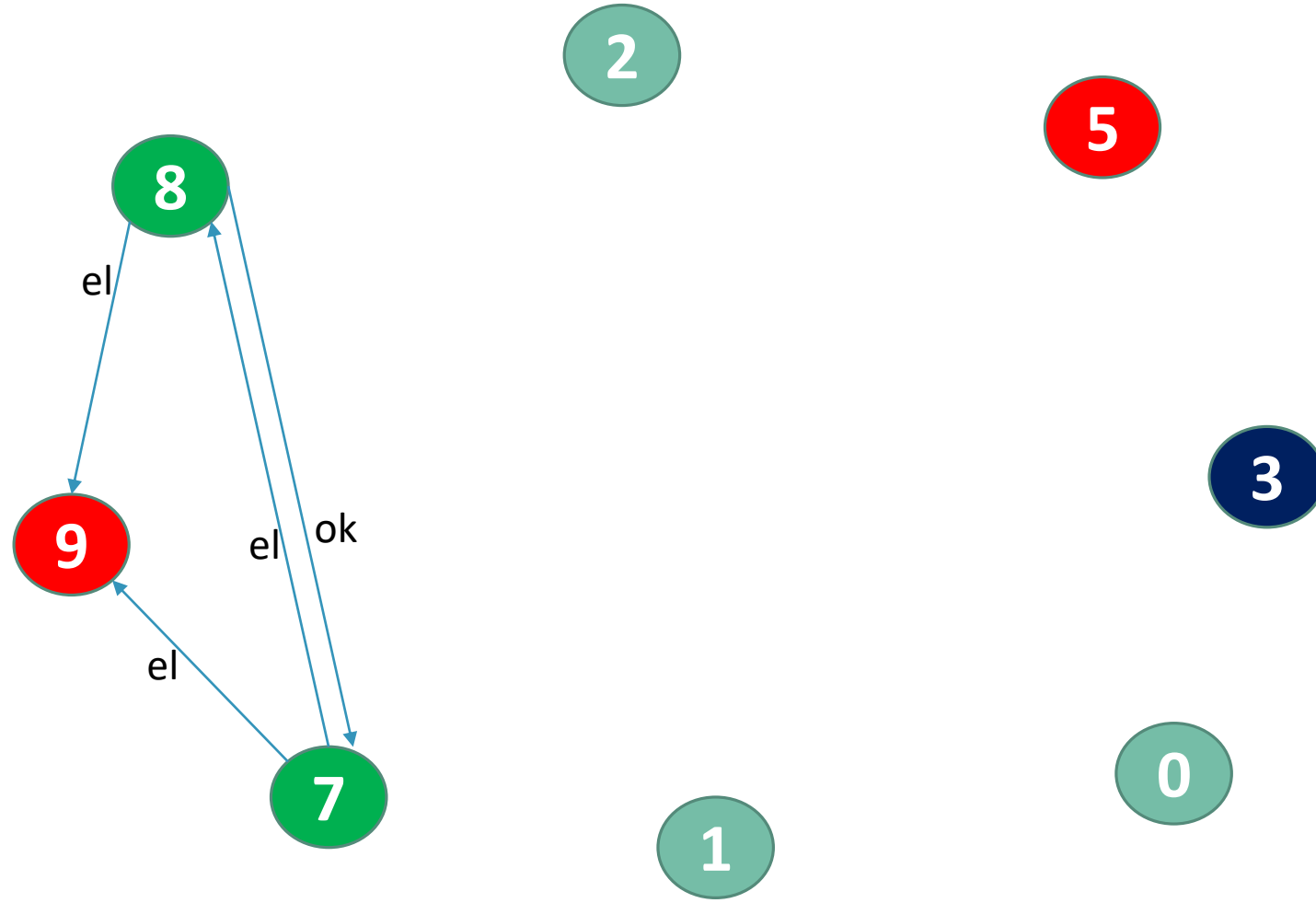
Respuestas



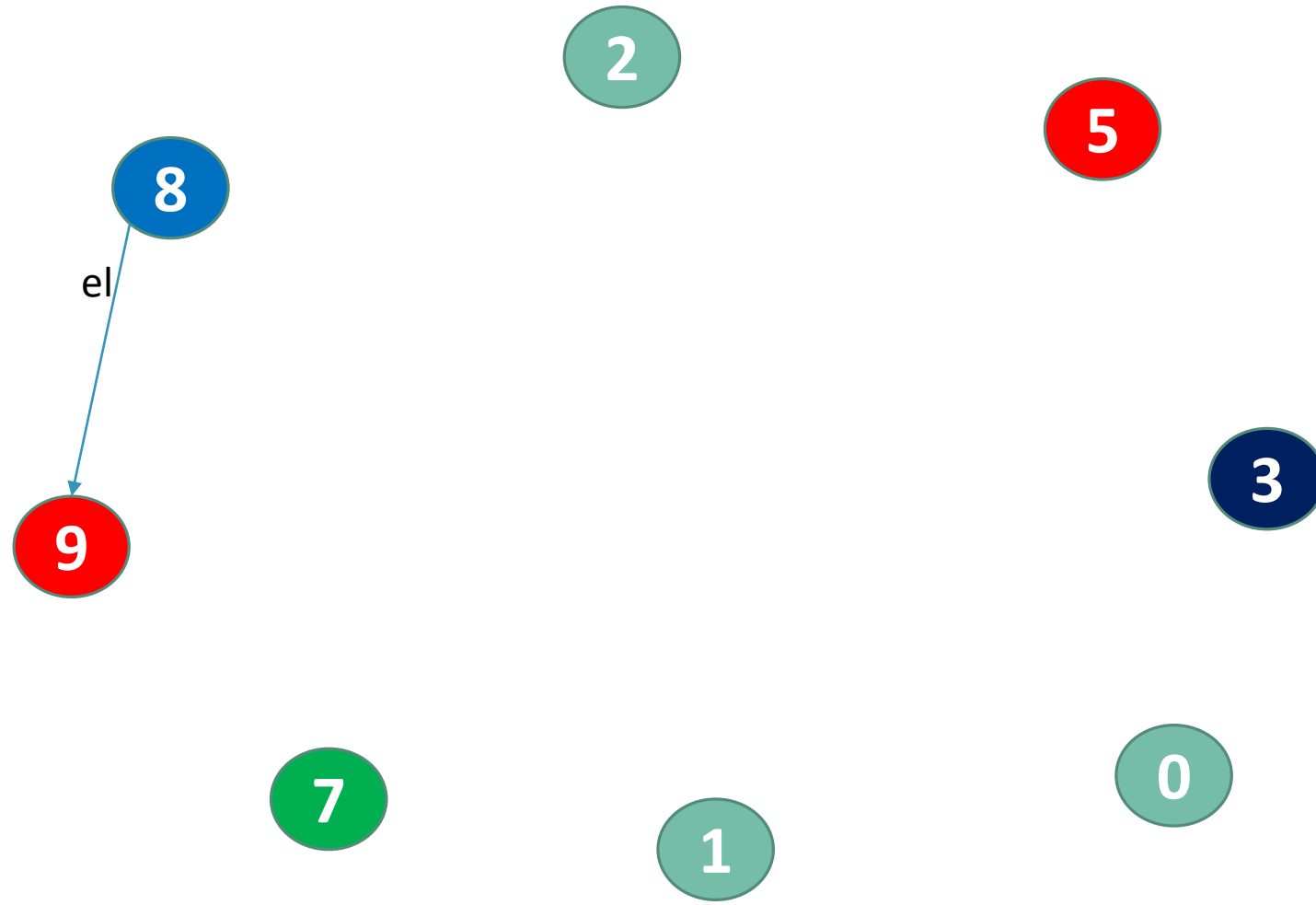
Se re-comienza el proceso elección



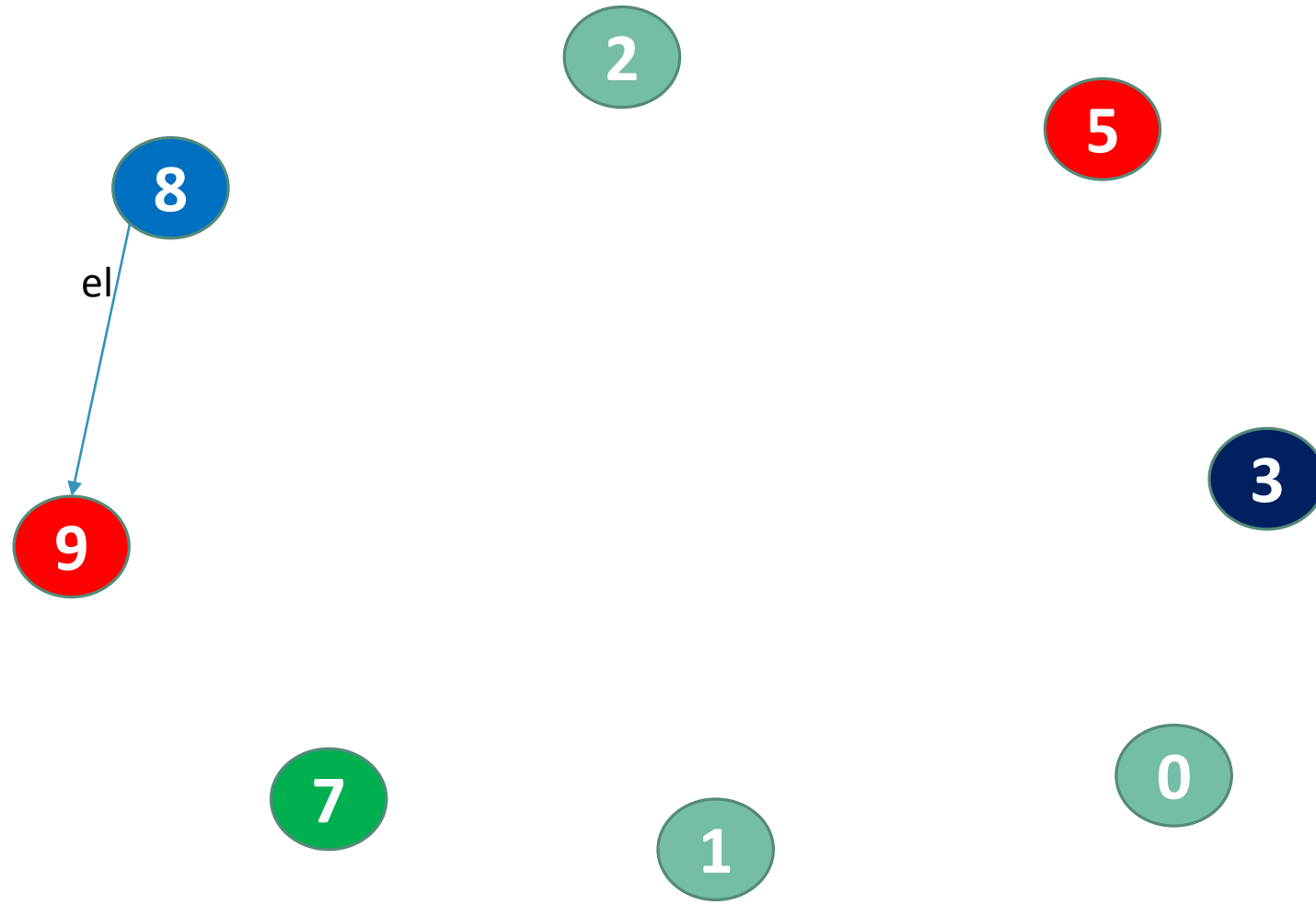
Respuestas



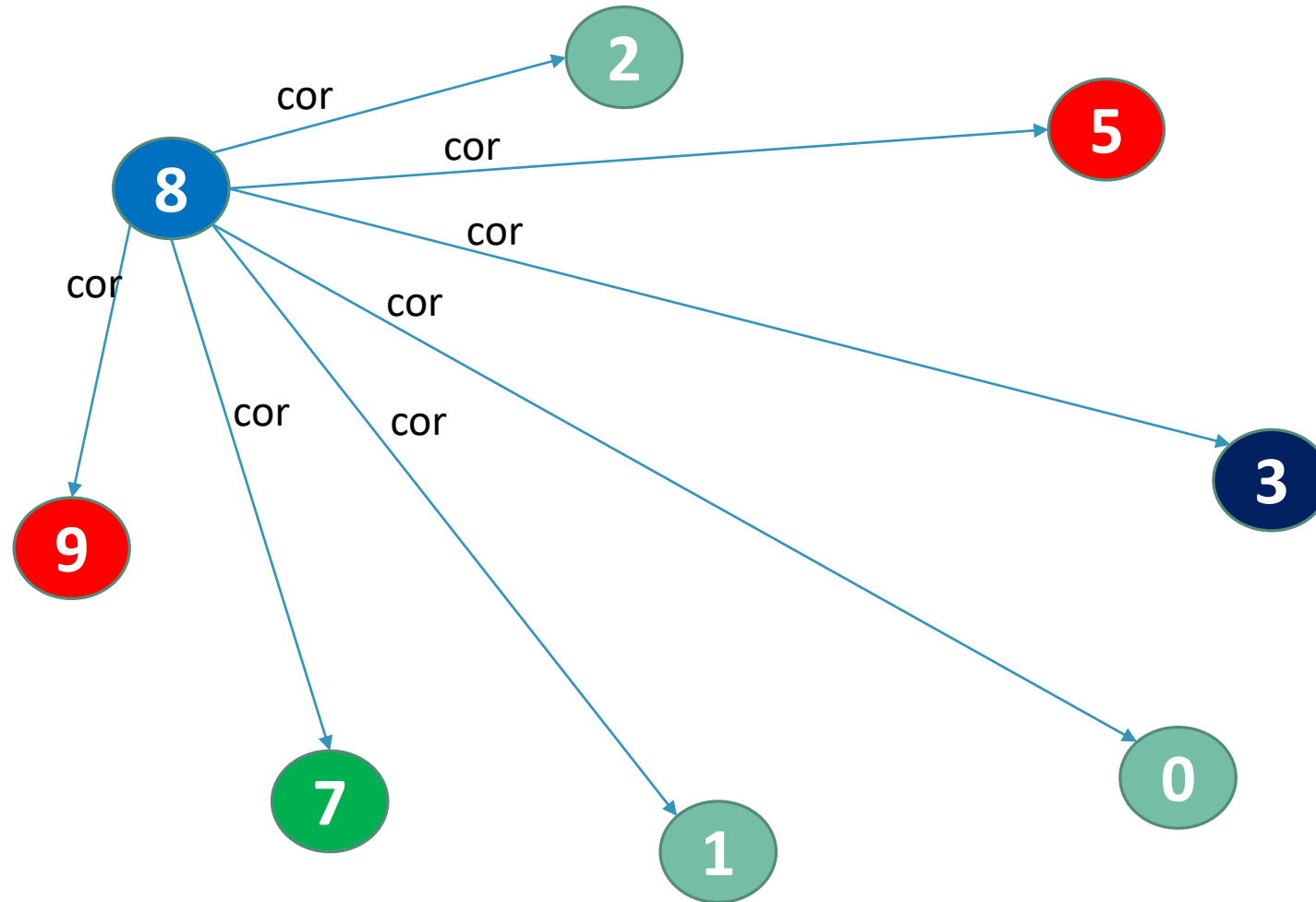
Se re-comienza el proceso elección

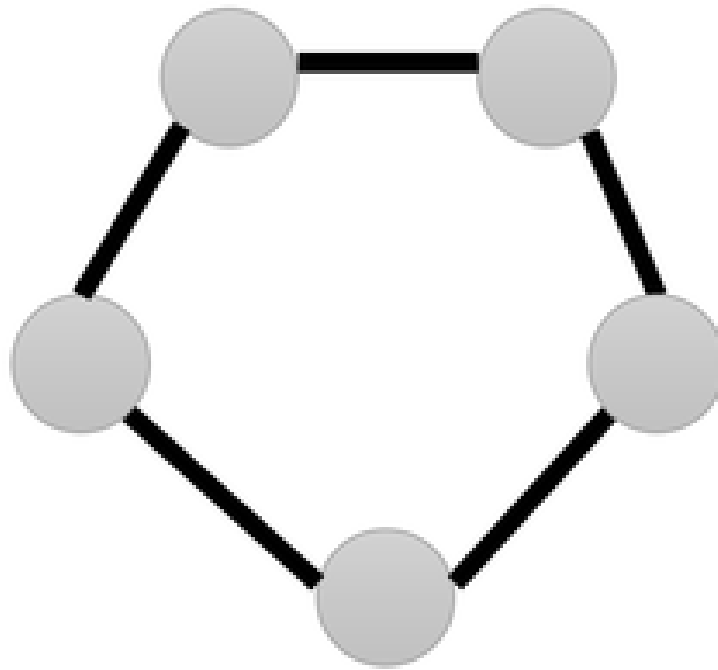


Se re-comienza el proceso elección



Mensaje “coordinador”





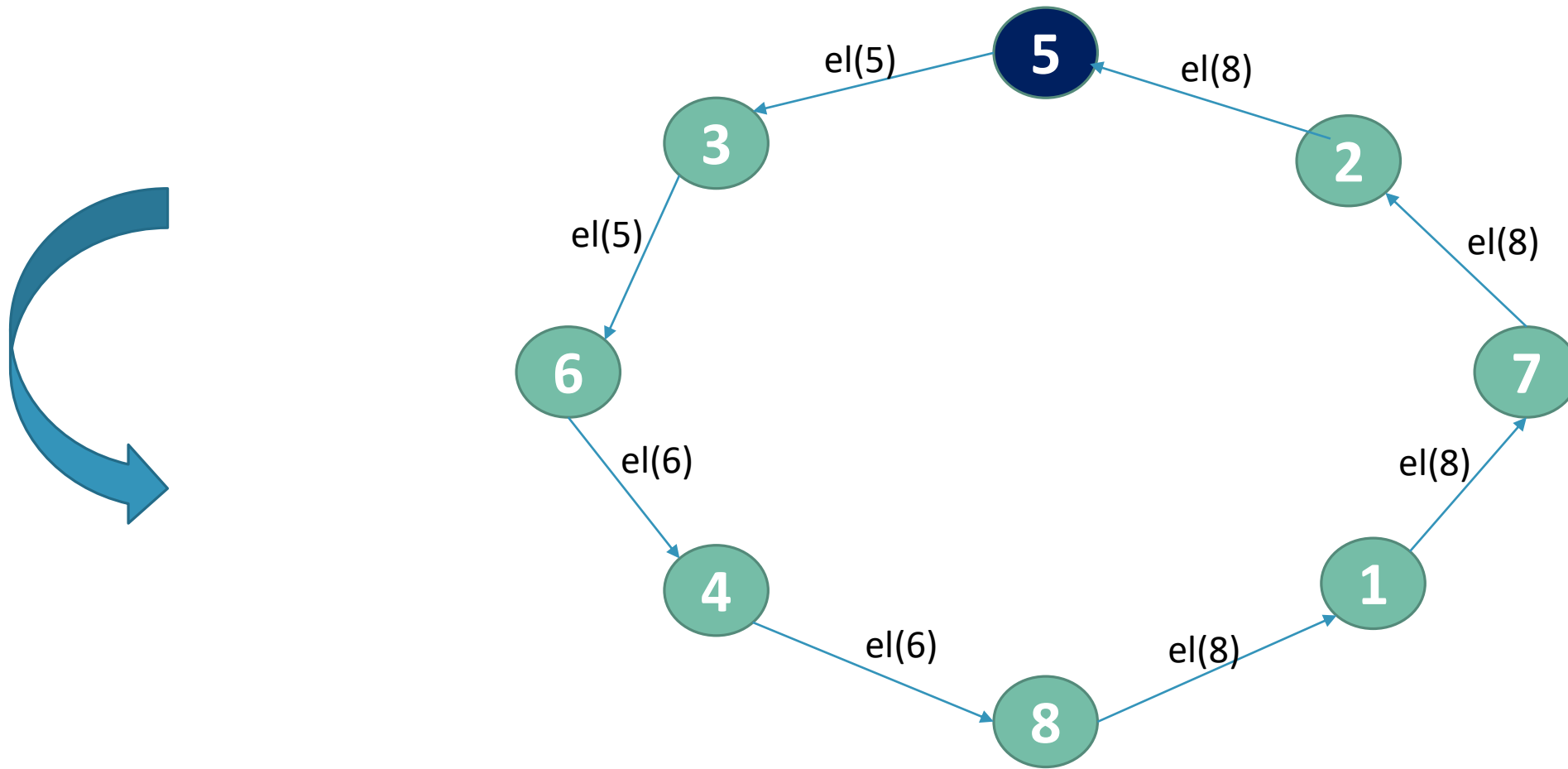
Algoritmo Ring Leader Election

Algoritmo de elección

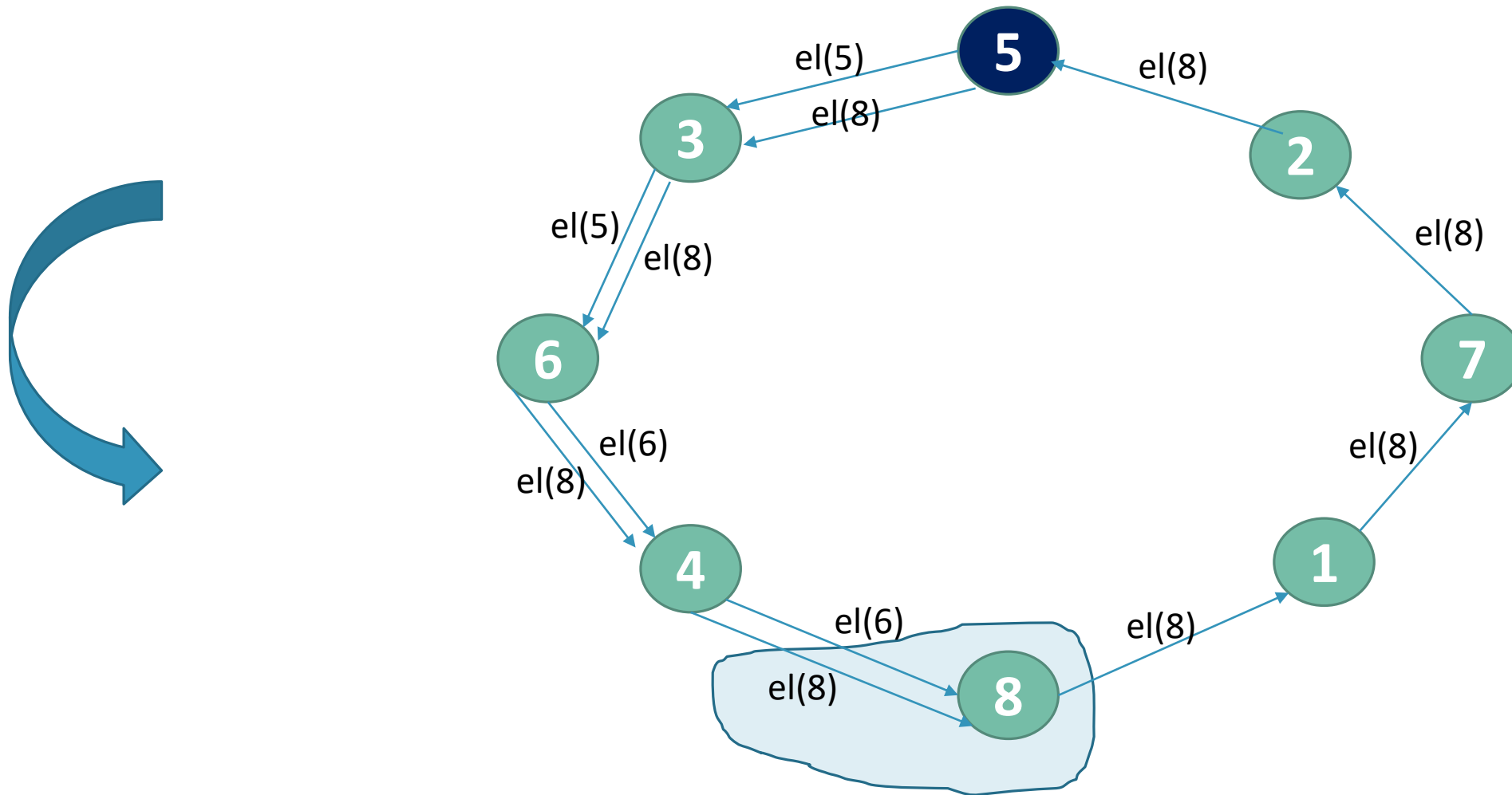
¿Cuál es la idea?

- Elegir un líder en un grupo de nodos distribuidos
 - Para ello se elegirá aquel con un ID más “grande”
- Consideraciones
 - Cada proceso tiene un ID único
 - ~~◦ Todos los procesos saben el ID de los demás nodos~~
 - El envío de mensajes es confiable
 - El sistema es síncrono
 - **Hay una dirección de comunicación**

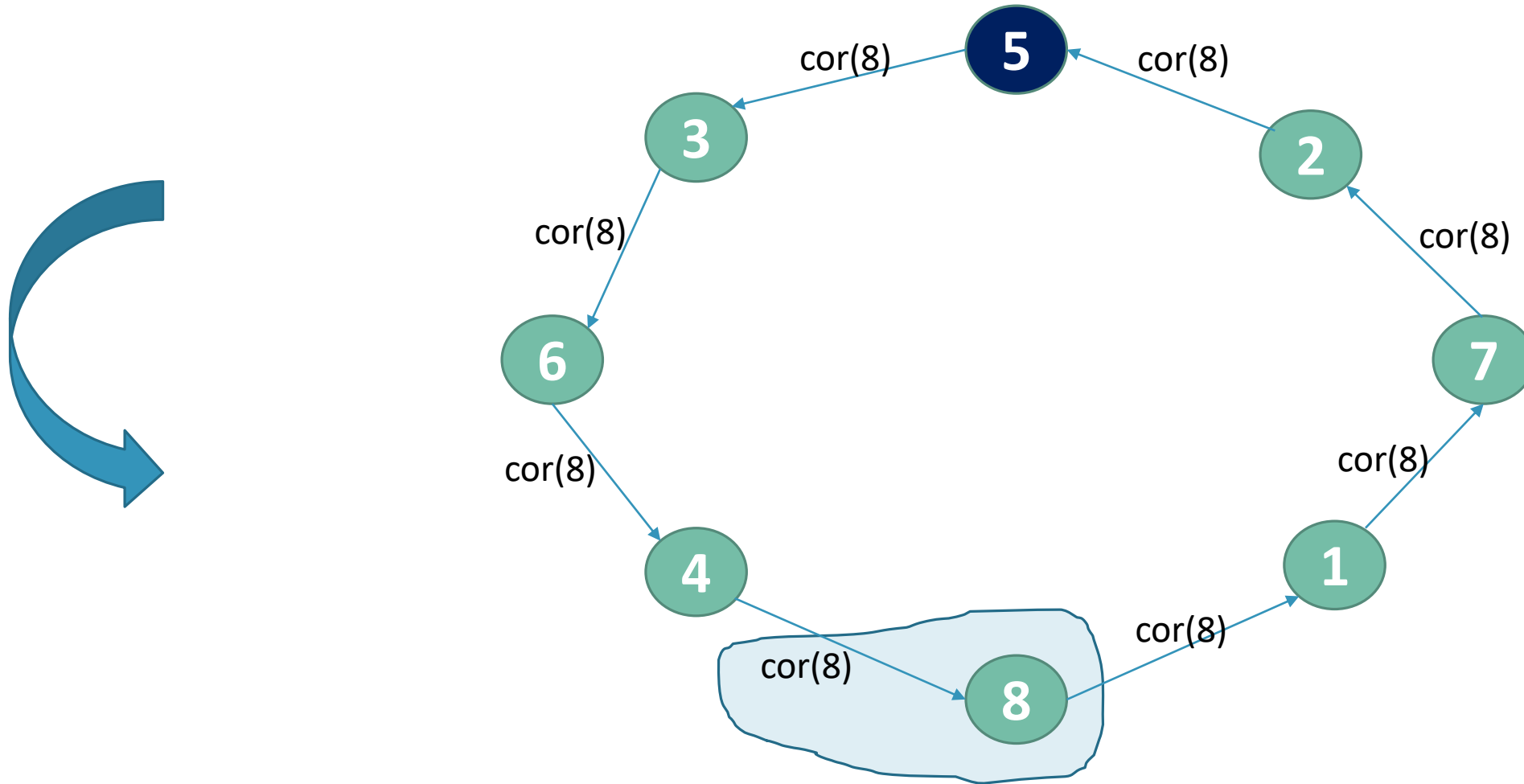
Un sistema con nodos independientes

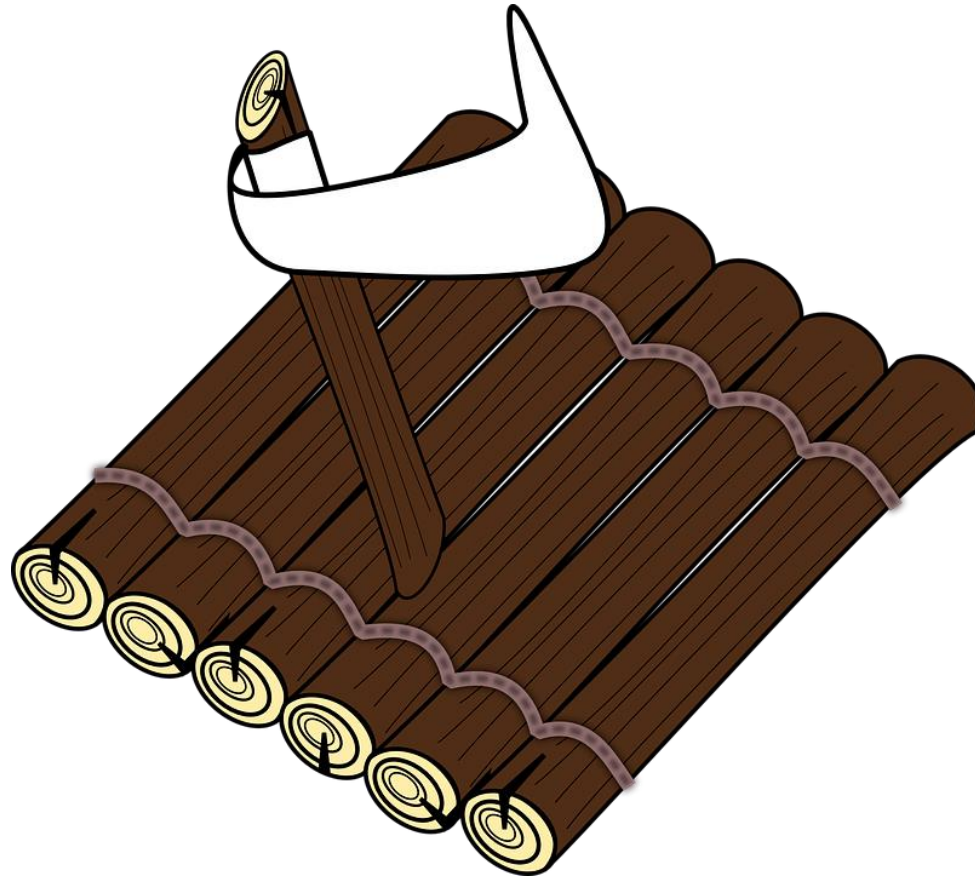


Se sigue el ciclo



Mensaje “coordinador”



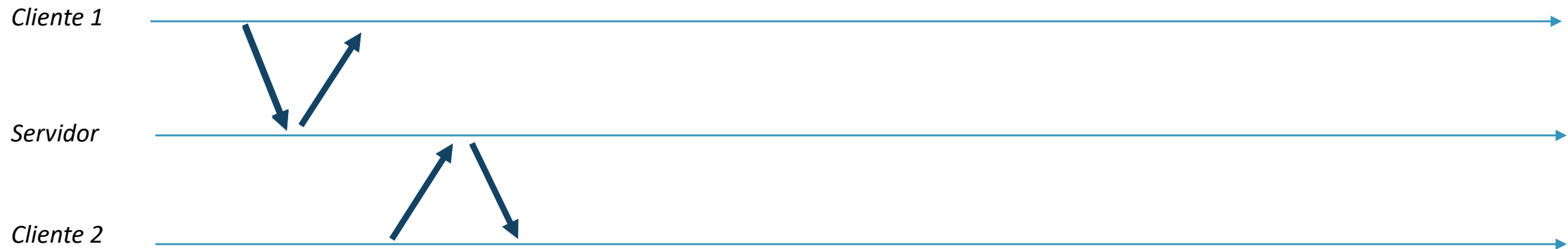


Algoritmo Raft

Algoritmo de consenso

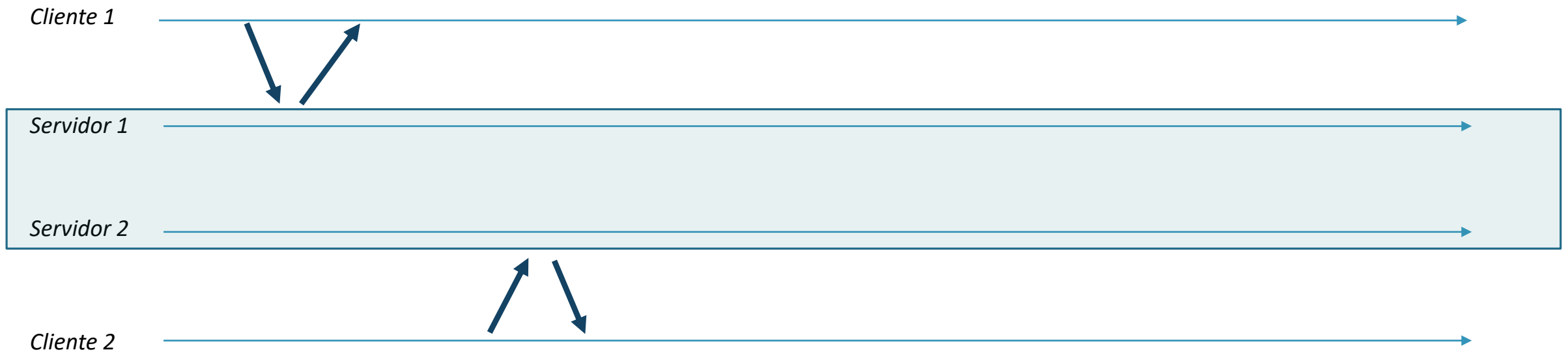
Queremos un sistema confiable

- Un servidor que gestiona cuentas bancarias
- El servidor, ¿es confiable?



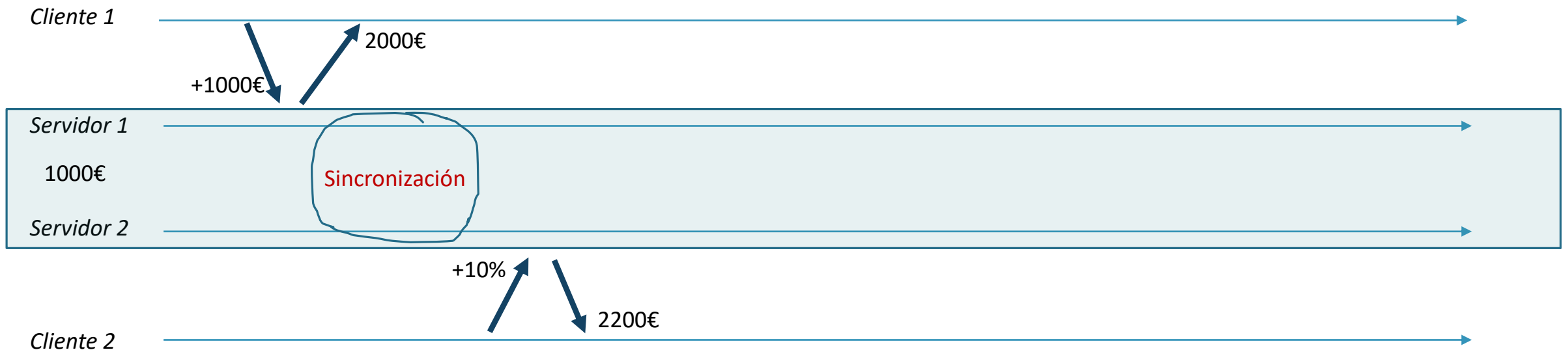
Replicación de componentes

- Varios servidores que gestionan cuentas bancarias
- Solución típica



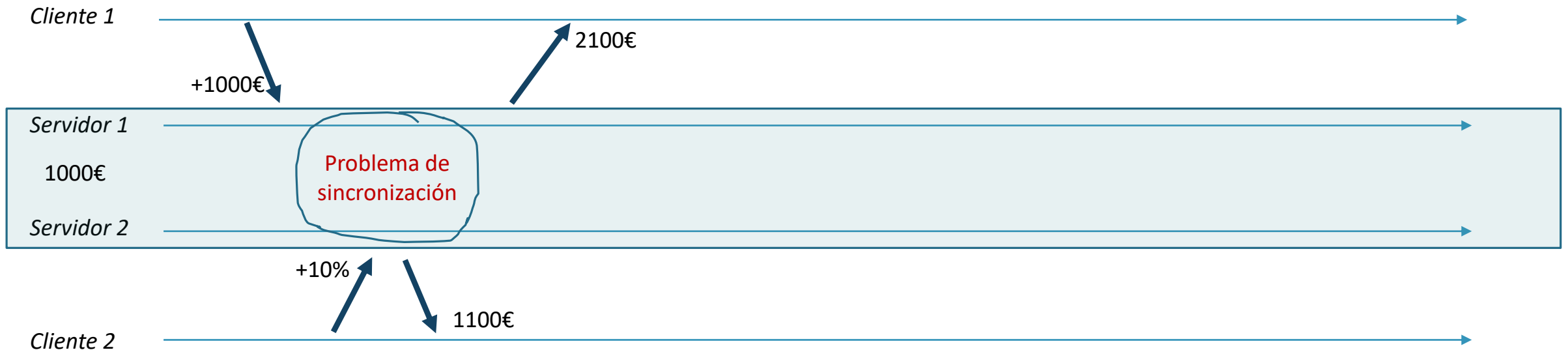
Funcionamiento

- Varios servidores que gestionan cuentas bancarias
- Si hay sincronización, todo perfecto



Problemática

- Varios servidores que gestionan cuentas bancarias
- Ambos servidores han de ponerse de acuerdo (**consenso**) sobre en qué servidor se ejecutarán las órdenes llegantes de los clientes



Dos formas de alcanzar consenso

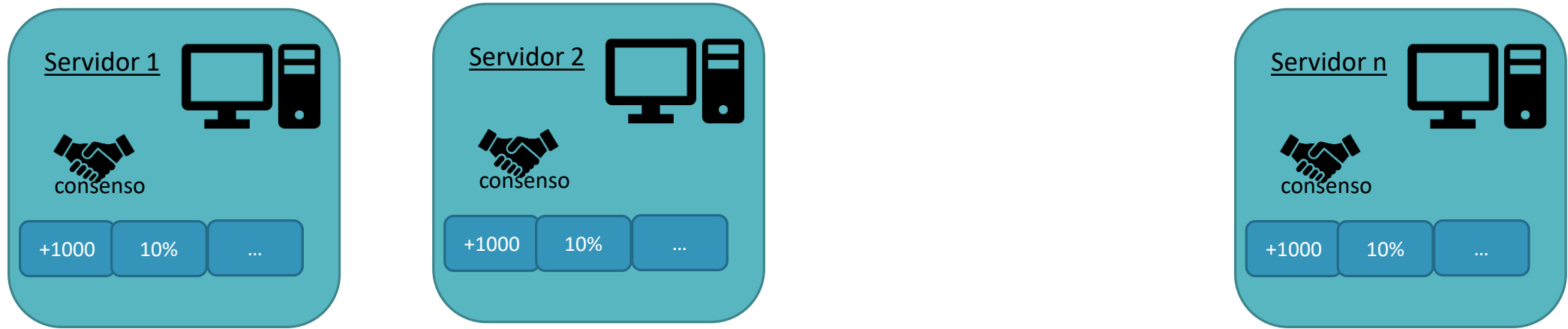
■ Simétrica

- No hay un nodo líder
- Todos los nodos tienen el mismo rol
- Los clientes pueden acceder a cualquiera de ellos

■ Asimétrica

- Hay un nodo líder (puede cambiar)
- Los clientes comunican con el líder

Replicación del log



- Todos los servidores deberán manejar el mismo log
- Si más de la mitad de los servidores están en funcionamiento, no habrá problema

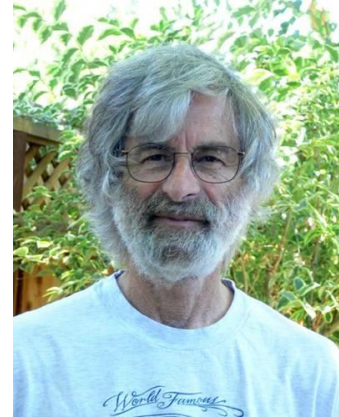
¿Qué opciones tenemos?

■ Algoritmo Paxos

- Desarrollado por Leslie Lamport
 - “The Part-Time Parliament” en *ACM Transactions on Computer Systems* 16, 2 (May 1998)
 - <http://dl.acm.org/citation.cfm?id=279229>

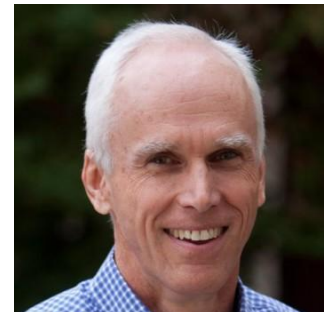
“The dirty little secret of the NSDI (Networked Systems Design and Implementation) is that at most five people really, truly understand every part of Paxos...”

“There are significant gaps between the description of the Paxos algorithm and the needs of a real world system...”



■ Algoritmo Raft

- Desarrollado por Diego Ongaro y John Ousterhout
 - “In search of an Understandable Consensus Algorithm” en *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*
 - <https://ramcloud.stanford.edu/wiki/download/attachments/11370504/raft.pdf>

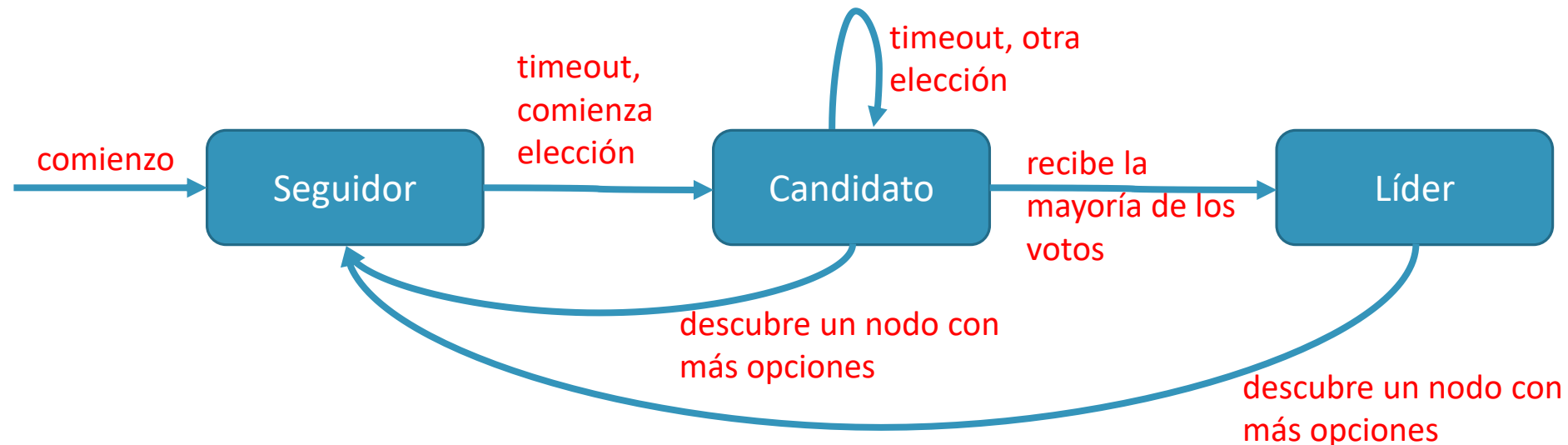


Principales acciones

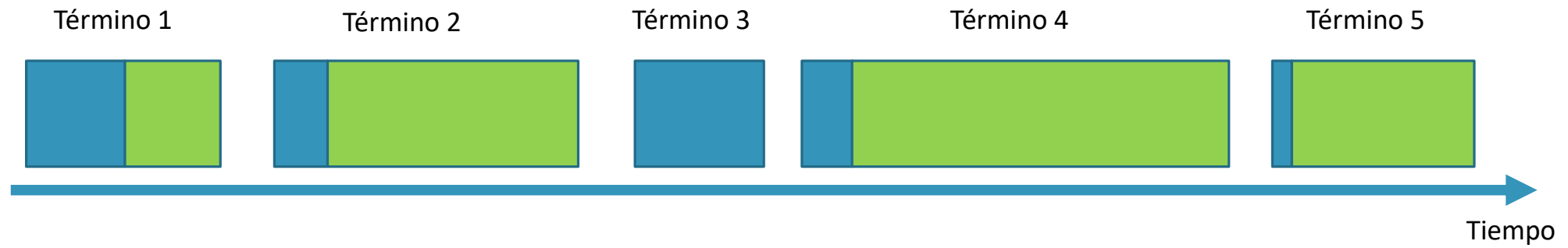
1. Elección de un líder
2. Funcionamiento “normal”
3. Seguridad y consistencia después de un cambio de líder
4. Neutralización de líderes viejos
5. Interacción con los clientes
6. Configuración de cambios

Diferentes posibles estados

- En cada instante de tiempo los nodos pueden estar en uno de estos estados
 - Líder
 - Seguidor
 - Candidato
- En funcionamiento “normal” hay un líder y $n - 1$ seguidores



Tiempo y Términos (periodos)



- Todos los nodos guardan el número de término actual
- Tienen dos partes:
 - Elección
 - Funcionamiento normal una vez elegido el líder
- Aunque a veces sólo hay elecciones

1 - Elecciones - Heartbeats y timeouts

- Todos los nodos comienzan como seguidores
 - Los seguidores esperan recibir mensajes del líder o de los candidatos para serlo
- El líder tiene que enviar periódicamente **latidos** (mensaje **AppendEntries**)
- Hay un **timeout (electionTimeout)** que si se supera se asume que el líder ha fallado
 - ...y se comienza una nueva elección
 - Entre 100 y 500ms

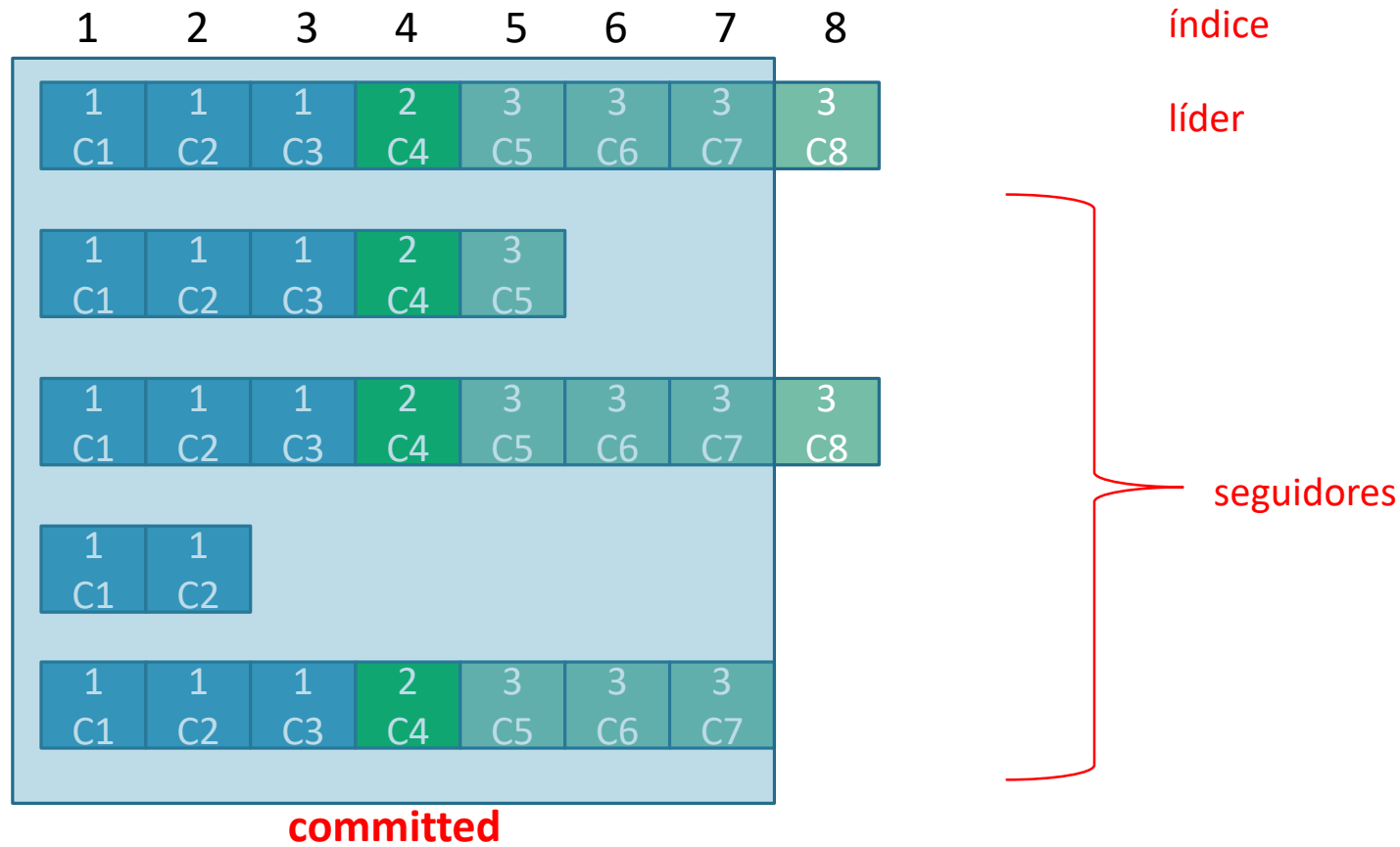
1 – Elecciones - Proceso

- El nodo:
 - Incrementa el número de término
 - Cambia al estado “candidato”
 - Se vota a si mismo
- Envía un mensaje (**RequestVote**) a todos los demás:
 1. Recibe la mayoría de votos de los demás
 - Cambia al estado “líder”
 - Envía mensajes (**AppendEntries**) a los demás nodos
 2. Recibe un mensaje desde un nodo líder
 - Cambia al estado “seguidor”
 3. No hay un acuerdo
 - Incrementa el nº de término y vuelve a la elección

1 – Elecciones – Seguridad y viveza

- Seguridad (**safety**) → Solo un nodo ganará en cada término
 - Cada nodo sólo responderá a una de las solicitudes de voto en cada término
 - En cada término puede ganar un nodo diferente
- Viveza (**liveness**) → Algún nodo candidato ganará antes o después
 - Cada nodo elegirá en cada término sus timeouts de forma aleatoria entre $[T, 2T]$
 - T es el **electionTimeout**
 - Funciona muy bien si algún nodo tiene un $T \ll$ *que los demás nodos*

2 – Funcionamiento – Estructura del log



- Cada nodo tiene su propia copia del log (persistente)
- Cuando una entrada es **committed**, se garantiza que se ejecutará en todos los nodos

2 – Funcionamiento – Proceso

1. El cliente envía un mensaje al líder
 2. El líder introduce el comando en una nueva entrada del registro
 3. El líder envía mensajes **AppendEntries** a los seguidores
 4. El líder espera por las respuestas hasta que la entrada sea **committed**
 - El líder notifica con mensajes **AppendEntries** a los seguidores sobre que la entrada es **committed**
 - El líder puede ejecutar el comando del log y dar respuesta al cliente
 - Los seguidores también pasarán a ejecutar el comando
-
- Si hay nodos que fallan o lentos el líder seguirá tratando de enviarle mensajes
 - Para ejecutar un comando sólo hace falta que la mayoría lo tenga en su registro

2 – Funcionamiento – Consistencia


- El algoritmo intenta en todo momento mantener todos los logs lo más idénticos posible
- Dos reglas:
 - Si dos entradas tienen el mismo índice y término
 - Deben tener el mismo comando
 - Todas las anteriores también
 - Si una entrada es **committed**
 - Todas las anteriores también

1	2	3	4	5	6
1 C1	1 C2	1 C3	2 C4	3 C5	3 C6
1 C1	1 C2	1 C3	2 C4	3 C5	4 C6'

2 – Funcionamiento – Comprobación consistencia

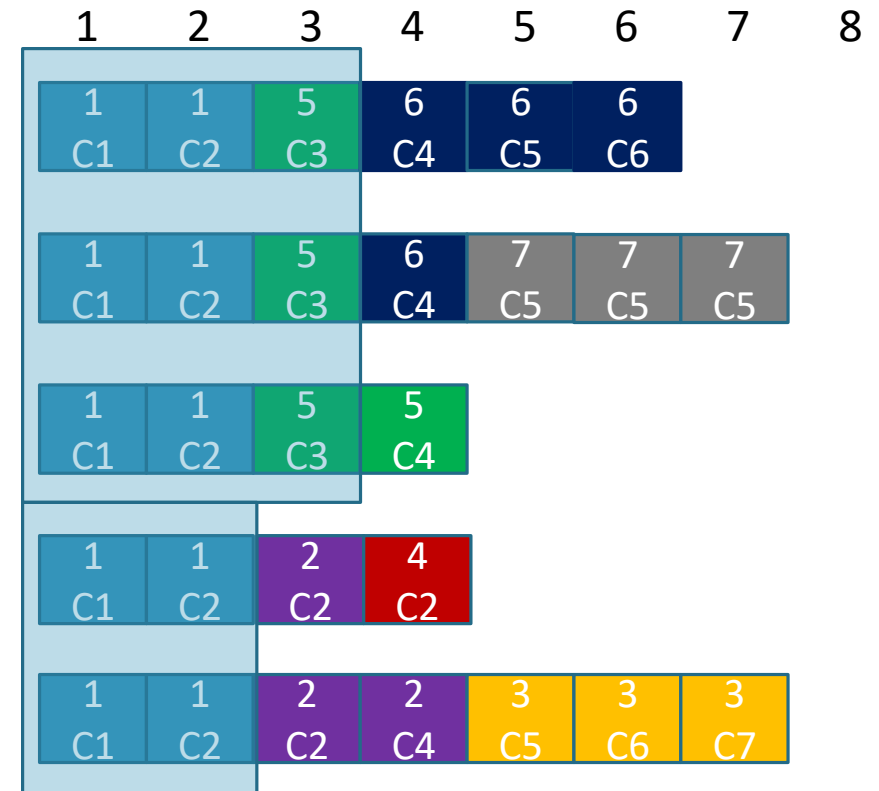
- Cada mensaje **AppendEntries** de log contiene:
 - El comando
 - El índice precedente del líder
 - El término precedente del líder
- Deben coincidir
 - De ese modo se asegura la consistencia y que **ambos logs son idénticos**

1	2	3	4	5
1 C1	1 C2	1 C3	2 C4	3 C5
1 C1	1 C2	1 C3	2 C4	3 C5

1	2	3	4	5
1 C1	1 C2	1 C3	2 C4	3 C5
1 C1	1 C2	1 C3	1 C4'	

3 – Cambio de líder - Idea

- Cuando un líder se “estrena”:
 - El líder anterior podría haber dejado el log a “medias”
 - Sin embargo, el nuevo no hace nada especial
- Varios cambios de líderes seguidos podrían provocar un “lío” en el log
- La opción a seguir es que el log del líder siempre es el log “bueno”
 - Sólo interesan los registros **committed**



3 – Cambio de líder – Seguridad

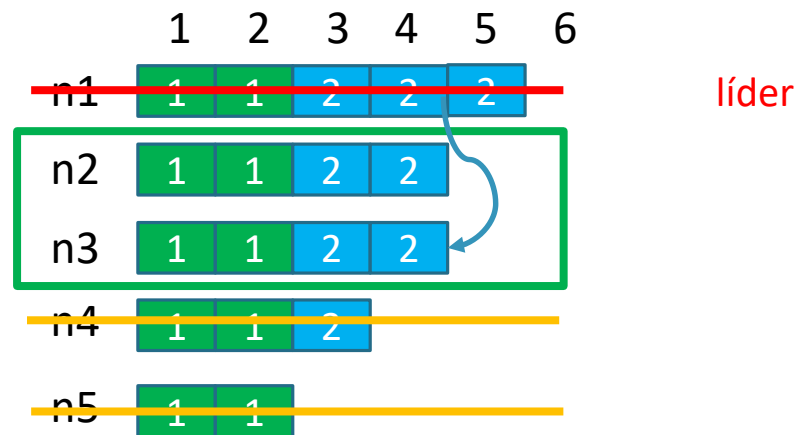
- Dijimos que:
 - El líder tiene el log “bueno”
 - Todos los demás nodos acabarán teniendo el mismo log que el líder
- Pero también dijimos que una vez que una entrada era **committed**, eso ya no se cambiaba más ¿¿??

Si un líder ha decidido que una entrada es **committed**, esa entrada **TIENE** que estar presente en los logs de todos los líderes futuros

- Necesitamos nuevas reglas

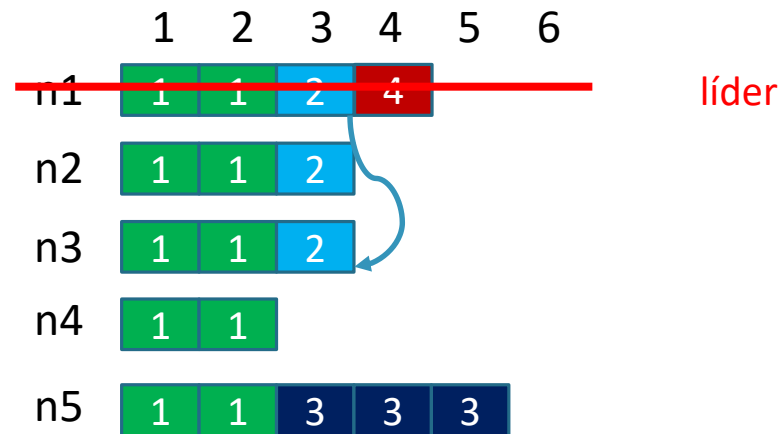
3 – Cambio de líder – Cuál es el mejor líder

- Los candidatos tienen que incluir el índice y el término de su última entrada
- Un nodo n NO votará a un candidato c si se cumple una de estas condiciones:
 - $ULTIMO_TERMINO_n > ULTIMO_TERMINO_c$
 - $(ULTIMO_TERMINO_n == ULTIMO_TERMINO_c) \ \&\& \ (ULTIMO_INDICE_n > ULTIMO_INDICE_c)$
- Ejemplo cuando el líder se cae justo después de saber que la entrada es **committed**



3 – Cambio de líder –Cuál es el mejor líder (II)

- Ejemplo cuando el líder se cae justo después de saber que la entrada de un término anterior es **committed**
 - El líder para el término 2 replicó la entrada sólo en dos máquinas antes de caerse
 - El líder para el término 3 no fue capaz de replicar sus entradas antes de caerse
 - El líder para el término 4 intenta hacer que los demás logs sean como el suyo
- ¿Qué ocurre si se cae ahora el líder?

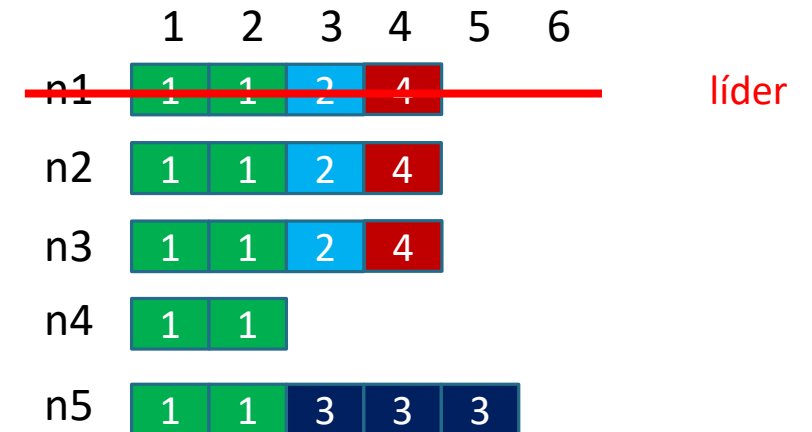


- Se intenta hacer **commit** de una entrada no segura. El sistema podría corromperse

3 – Cambio de líder – Cuando es committed

- Una entrada será **committed** si:
 - Está guardada en la mayoría de los nodos
 - **Al menos una entrada del término del líder también deberá estar almacenada en la mayoría de los nodos**
- ¿Qué ocurre si se cae ahora el líder?
 - El nodo 5 no podría ser el siguiente líder
 - Las entradas de los índices 3 y 4 son seguras ahora

La combinación de las nuevas reglas de elección y de commitment hacen al algoritmo seguro



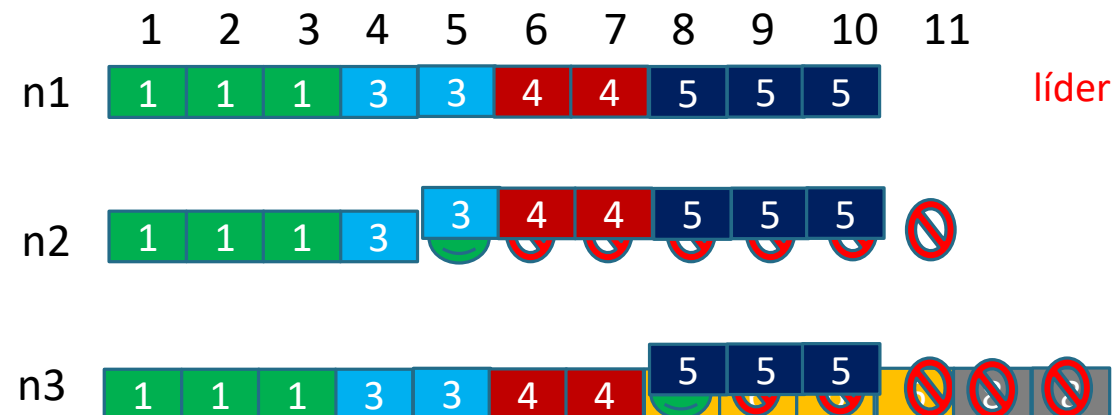
3 – Cambio de líder – Unificar los logs

- Puede haber dos tipos de inconsistencias

	1	2	3	4	5	6	7	8	9	10	11	12
n1	1	1	1	4	4	5	5	6	6	6		líder
n2	1	1	1	4	4	5	5	6				
n3	1	1	1	4	4	5	5					
n4	1	1	1	4	4							
n5	1	1	1	4	4	5	5	6	7	7	7	
n6	1	1	1	3	3	3	5	6	6	8		
n7	1	1	1	4	4	5	5	6	6	6		

3 – Cambio de líder – Reparación de logs

- El líder debe:
 - Borrar entradas “extrañas”
 - Rellenar entradas faltantes
- El líder guarda un **nextIndex** para cada seguidor
 - Es el índice de la siguiente entrada a enviar al seguidor
 - Lo envía junto con su índice previo y su término
 - Cuando la comprobación de la consistencia falla, se decrementa en una unidad



4 – Neutralización de líderes viejos

- Un líder puede desconectarse solo por un tiempo
 - Otro nodo será nombrado nuevo líder
 - Si el viejo líder vuelve a conectarse, no sabrá que ya no es el líder
- La forma de neutralizarlos es a través del **número de término**
 - Todos los mensajes llevan el número de término
- Cuando un nodo recibe un mensaje con un término posterior al suyo, ese nodo siempre se pondrá en el estado “seguidor”
 - Si el que tiene el término menor es el supuesto líder entonces ese mensaje se rechazará

5 – Interacción con clientes - Proceso

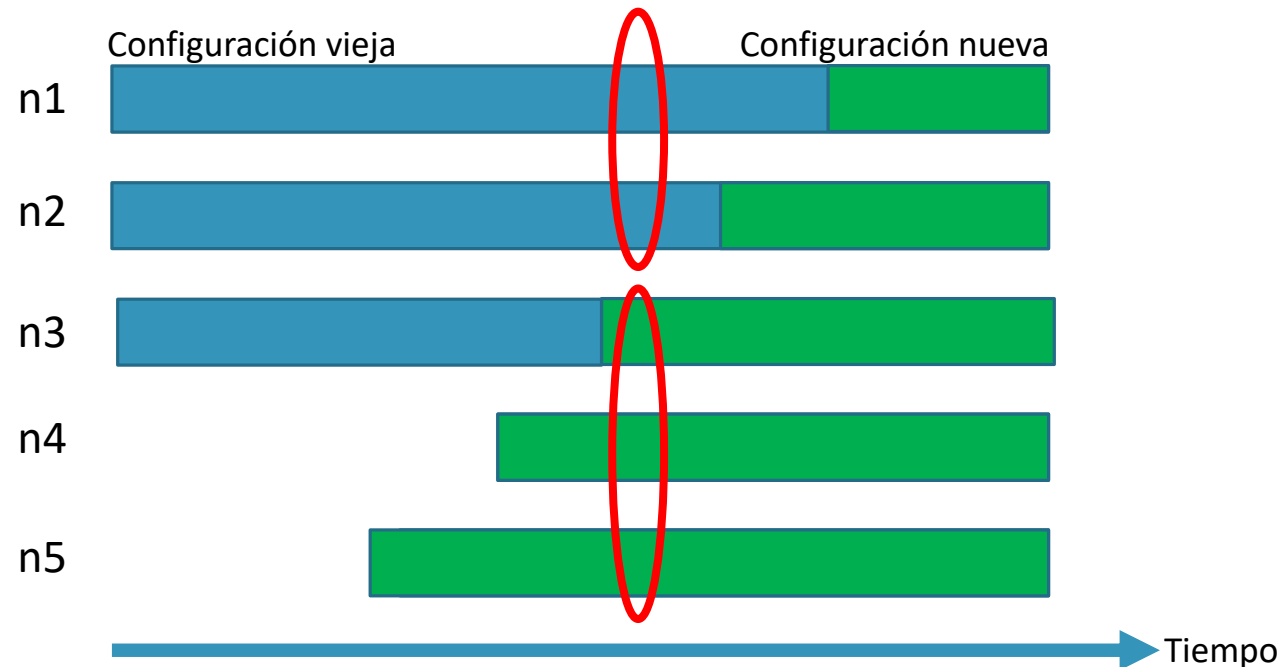
- El cliente envía mensajes al líder
 - Si no sabe quien es el líder puede enviarlo a cualquiera de los nodos
 - Si el nodo que recibe la comunicación no es el líder, este redireccionará al cliente
- El cliente no tendrá respuesta hasta que el comando haya sido registrado, **committed** y ejecutado por el nodo líder
- Si la solicitud no recibe respuesta en un **timeout** dado
 - El cliente volverá a enviar el comando a otro nodo
 - El nodo redireccionará al cliente con el nuevo líder
 - El nuevo líder intentará llevar a cabo el comando

5 – Interacción con clientes - Duplicación

- ¿Qué pasa si el líder se cae después de ejecutar el comando pero antes de responder al usuario?
 - Estaremos en riesgo de que el comando se ejecute dos veces
- Para solventarlo el cliente incluye un **ID único en cada comando**
 - Ese ID también va incluido en cada entrada del log
 - Antes de aceptar un comando, el líder comprueba en el log si ya hay otro comando con ese ID

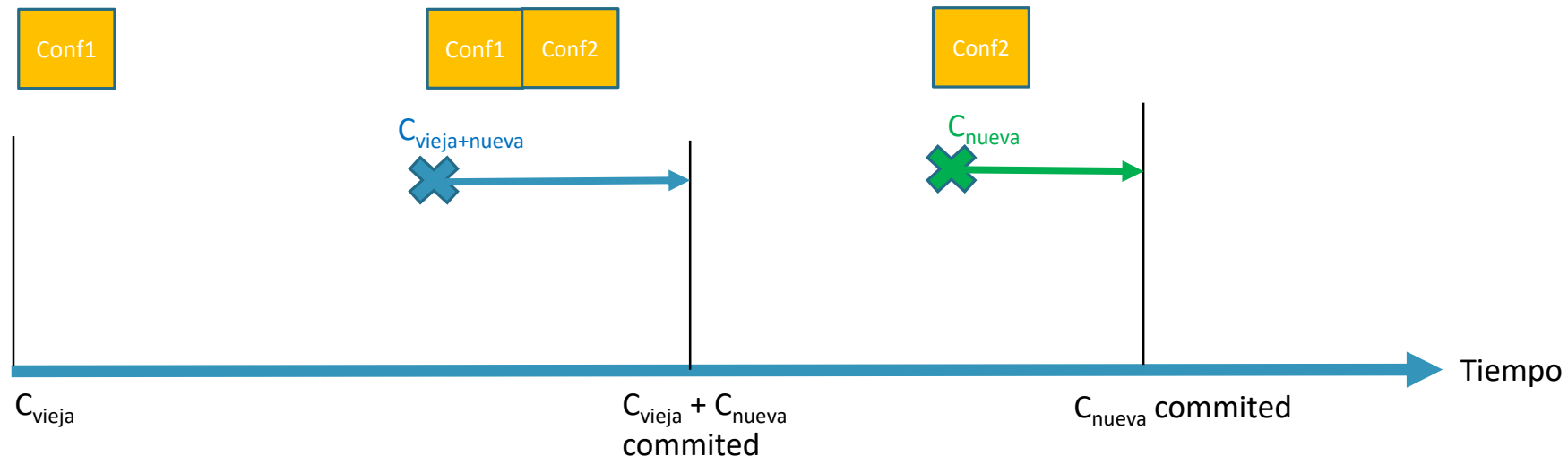
6 – Cambios de configuración - Idea

- Debemos permitir cambiar de configuración
 - ID y dirección de cada nodo
- Es muy peligroso cambiar directamente de una configuración a otra



6 – Cambios de configuración - Proceso

- No debería haber al mismo tiempo más nodos nuevos que viejos en el sistema
- Los cambios en la configuración se propagan como cualquier otra entrada en el log
- Utilizamos dos pasos para cambiar la configuración
 - Hay una fase intermedia llamada **joint consensus**
 - Necesita la mayoría de las configuraciones viejas y nuevas para funcionar de forma separada



Algunas implementaciones

<https://raft.github.io/#implementations>

Lenguaje	Librería	URL
Java	Barge	https://github.com/mgodave/barge
JavaScript	Raft.js	https://github.com/kanaka/raft.js
Python	Py-raft	https://github.com/kurin/py-raft
C	CRaft	https://github.com/sargon/CRaft
C++	LogCabin	https://github.com/logcabin/logcabin
Ruby	Floss	https://github.com/celluloid/floss
NodeJS	Skiff	https://github.com/pgte/skiff-algorithm
C#	RaftCore	https://github.com/AzoresCabbage/RaftCore
Scala	Ckite	https://github.com/pablosmedina/ckite
Clojure	Saebyn/raft	https://github.com/saebyn/raft
Go	Hashicorp/raft	https://github.com/hashicorp/raft
Erlang	Rafter	https://github.com/andrewjstone/rafter
Hashkell	Kontiki	https://github.com/NicolasT/kontiki

Recursos

- Descripción, simulador y publicaciones

- <https://raft.github.io/>

- Descripción interactiva

- <http://thesecretlivesofdata.com/raft/>

