

SISTEMAS OPERATIVOS
LOS PROBLEMAS DE SINCRONIZACIÓN Y CONSENSO
EN SISTEMAS DISTRIBUIDOS

El avance de tecnologías de cómputo ha permitido una gran expansión de los sistemas de información; en cuestión de disponibilidad, conectividad, variedad de sistemas, de comunicaciones, etcétera.

Estos avances han permitido que surjan instalaciones de cómputo donde se pueden desplegar aplicaciones paralelas para conseguir procesamiento distribuido de tareas.

A los sistemas con estas características se les conoce como **sistemas distribuidos**.

¿Qué son?

Una colección de computadoras independientes que aparecen ante los usuarios del sistema como una única computadora (Tanenbaum, 1996).

¿Para qué?

Estos sistemas ofrecen múltiples ventajas en economía, velocidad, datos compartidos, dispositivos compartidos, rendimiento, entre otras.

Ventajas

Confiabilidad: La distribución de recursos y datos reduce la dependencia de un punto de falla.

Economía: Mejor relación rendimiento / precio.

Velocidad: Mayor poder de cómputo que en una computadora centralizada.

Rendimiento: Debido a que los nodos pueden escalarse fácilmente de forma vertical y horizontal, este aumento de forma considerable.

Datos y Dispositivos Compartidos: Los usuarios tienen acceso a una Base de Datos y a un Dispositivo costoso en común.

Desventajas

Complejidad de Desarrollo: El diseño, la implementación y el mantenimiento de sistemas distribuidos son generalmente más complejos que los sistemas centralizados.

Redes: Suelen ser frecuentes los problemas de transmisión de datos cuando se trata de grandes volúmenes de información.

Seguridad: La seguridad en sistemas distribuidos es un desafío, ya que la comunicación entre nodos puede ser vulnerable a amenazas como ataques de red.

Tolerancia a fallos: Las fallas operativas y de componentes aún son frecuentes.

Esta última desventaja es la que nos ocupará principalmente en esta presentación, pues revisaremos distintos casos de una de las situaciones que más pueden atraer problemas en los sistemas distribuidos; la sincronización.

La sincronización se vuelve más compleja en sistemas distribuidos que en sistemas centralizados, por razones como:

- La información se distribuye en varios dispositivos.

- Los procesos toman decisiones con base en información local.

- Se debe evitar incluso un punto único de falla.

- No existe un *reloj* o fuente de tiempo global.

Exclusión mutua

En los sistemas distribuidos se presentan situaciones donde hay recursos compartidos que, por integridad del sistema, no deben ser utilizados por más de un proceso al mismo tiempo.

Como lo vimos en clase, los sistemas que comparten memoria pueden valerse de semáforos o monitores para garantizar el uso de la región crítica de manera exclusiva. No obstante, en los sistemas distribuidos los procesos ya no comparten la memoria de manera física, por lo que se requiere idear nuevos algoritmos que permitan la sincronización y garanticen la exclusión mutua.

Un algoritmo de exclusión mutua debe cumplir con tres requisitos básicos:

Evitar la inanición.

Garantizar la seguridad de los datos.

Llevar un cierto orden de los procesos.

A continuación, se presentan tres ejemplos de algoritmos de exclusión para sistemas distribuidos.

Algoritmo de servidor centralizado.

Algoritmo distribuido.

Algoritmo de anillo de token.

Algoritmo de Servidor Centralizado (Tanenbaum & Van Steen, 2008)

Este algoritmo consiste en simular un solo procesador para realizar la exclusión. El algoritmo nombra a un proceso como *coordinador*, que es el que se encarga de dar acceso a la región crítica. Los procesos que quieran acceder al recurso compartido debe enviar una petición al coordinador, el cual brindará acceso dependiendo de si ya se encuentra otro proceso en la región crítica o no.

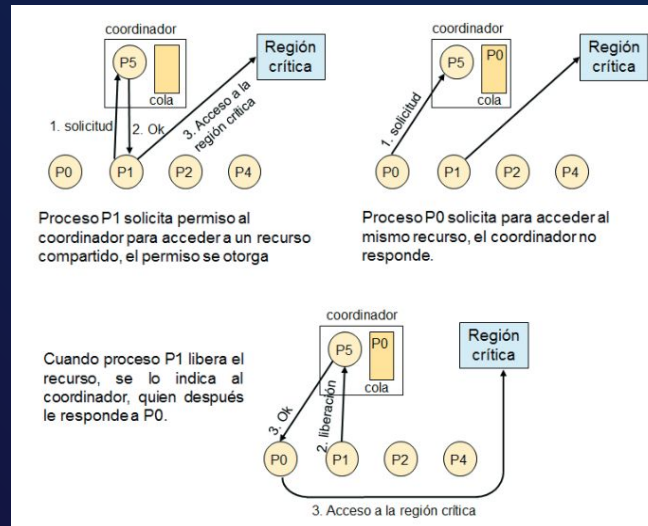


Imagen:
Sistemas Distribuidos, UAM.

Algoritmo Distribuido (Ricart & Agrawala)

Basado en un consenso distribuido y requiere de un orden para todos los eventos del sistema.

Cuando un proceso quiere acceder, construye un mensaje formado por su nombre, número de proceso y hora de la petición, posteriormente este mensaje se envía a todos los procesos, incluyéndose a sí mismo, con lo cual se considera que la comunicación es confiable.

Cuando un proceso recibe el mensaje puede suceder:

- Si el proceso no está en la región crítica y no quiere entrar, envía un *OK*.
- Si el proceso se encuentra en la región crítica, no responde y se coloca la solicitud en cola.
- Si no está en la región crítica pero desea entrar, se comparan los tiempos y el primero en hacer la solicitud gana. El proceso solicitante espera el permiso de los otros, y entra a la región, si no, se bloquea. Cuando sale de la región envía un *OK* y saca a los procesos de su cola.

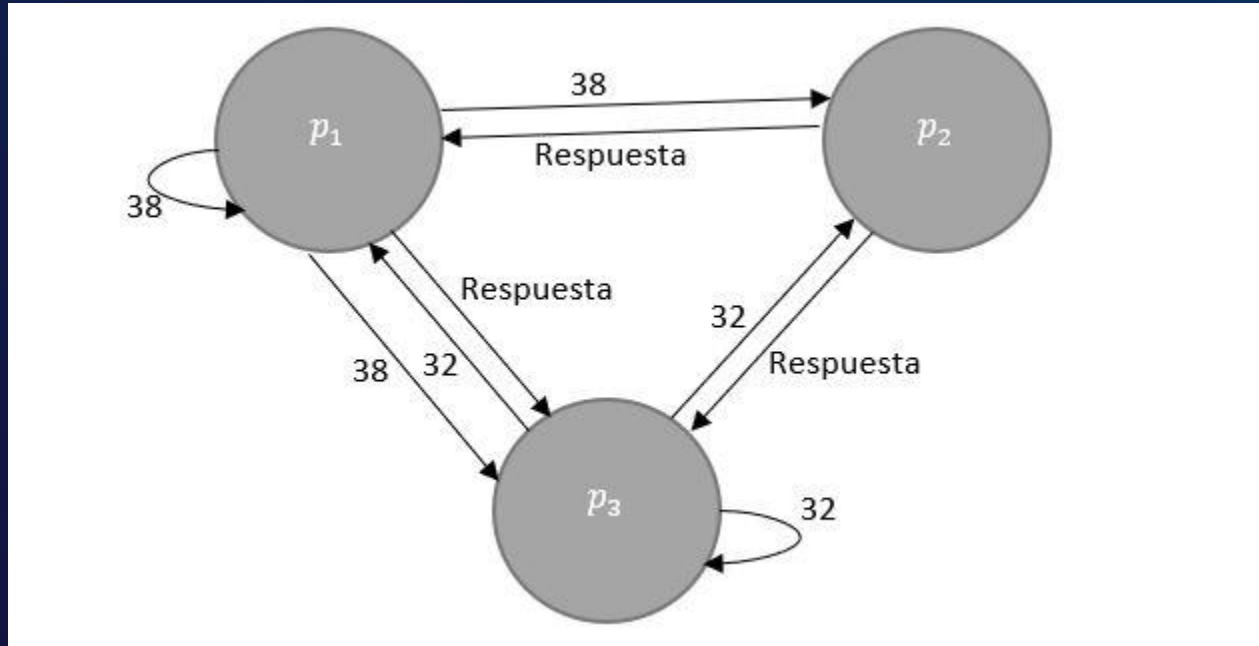
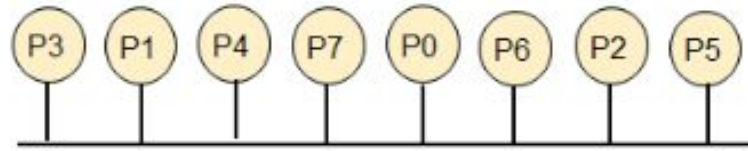


Imagen: Wikimedia.

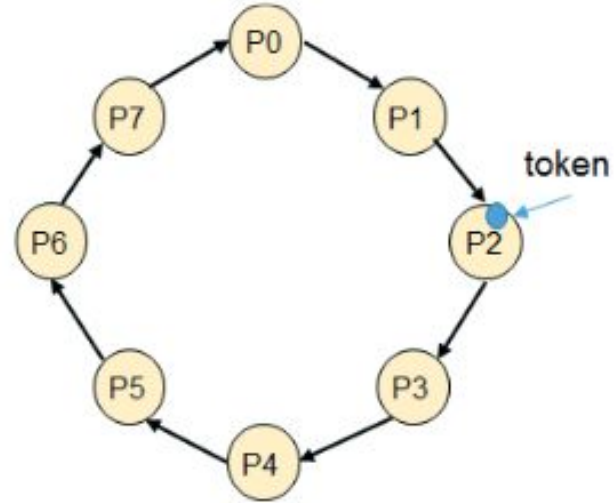
Algoritmo de anillo de token (Lann, 1977)

Se forma con las siguientes premisas:

- Los procesos pueden estar desordenados.
- Todos los equipos conectados forman un anillo lógico.
- Cada proceso es un nodo del anillo y se le asigna una posición en el anillo.
- Cada nodo del anillo debe de saber cual es la dirección de su vecino.



Procesos desordenados en una red



Anillo lógico con token

Imagen: Sistemas Distribuidos, UAM.

Hemos visto que existen distintas formas de entender e implementar la sincronización entre procesos dentro del cómputo distribuido, y que algunas de estas implementaciones logran el objetivo con cierto éxito. Sin embargo, esto no es suficiente para garantizar que un sistema distribuido puede operar correctamente, pues existe otro problema considerado por expertos en el área como la mayor problemática del cómputo distribuido; el consenso.

El problema del consenso consiste en conseguir que, aún en presencia de procesadores *erróneos*, los procesadores *correctos* consigan ponerse de acuerdo con respecto al valor de un dato común.

Este concepto nos lleva a un particular problema, considerado un clásico del cómputo distribuido: *El problema de los generales bizantinos*.

El problema de los generales bizantinos

Planteado originalmente por Lamport y Shostak en 1982, describe lo siguiente:

Un grupo de generales sitia una ciudad y deben ponerse de acuerdo a través de un plan de ataque, si deciden atacar o retirarse. Los generales se comunican con otros generales.

Un comandante da las órdenes, mientras los otros deben decidir si atacar o retirarse.

El no llegar a un consenso resultaría en una operación no coordinada, por lo tanto fallida y *catastrófica* para los bizantinos. El problema reside en el hecho de que uno de los generales puede ser un traidor, lo cual puede interpretarse de las siguientes formas:

Los mensajes pueden no llegar (las comunicaciones no son confiables).

Un general traidor puede mentir (un nodo puede fallar de manera impredecible).

Entonces se plantea la pregunta, ¿Pueden los generales leales llegar a un consenso sobre si deciden atacar o retirarse?

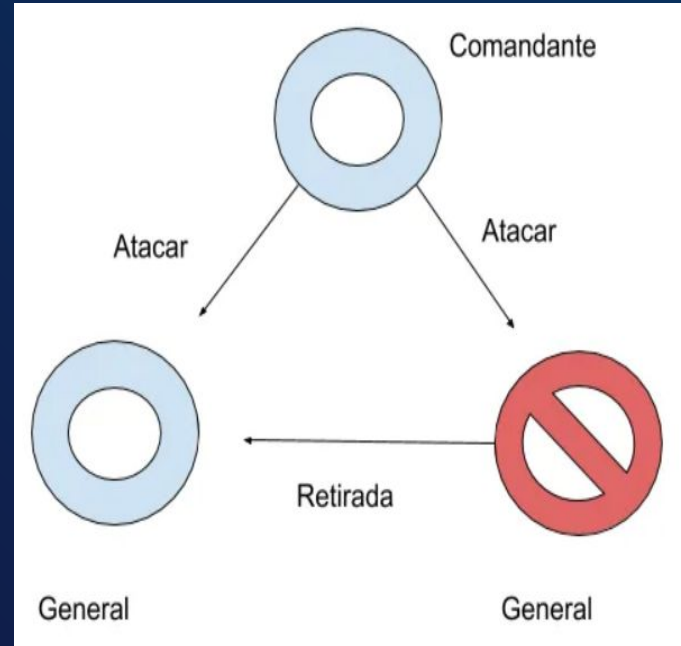
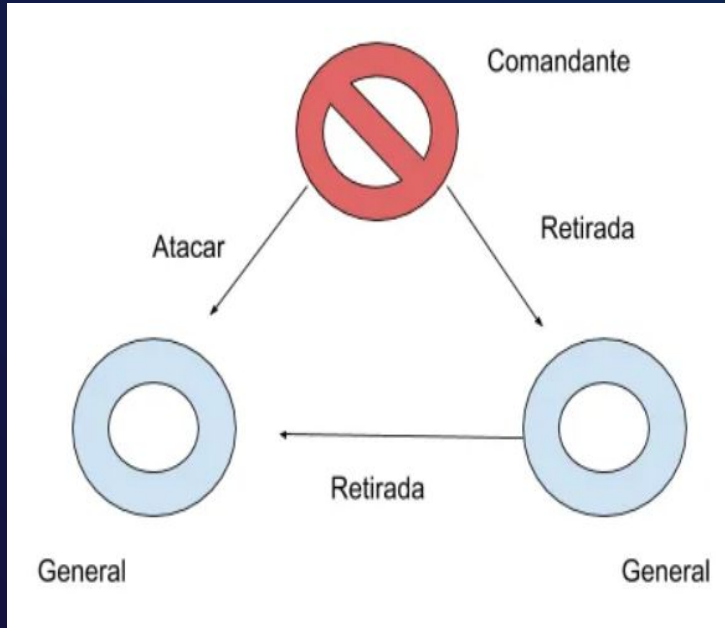
Para responder esta pregunta, podemos analizar distintos casos del problema.

Primer caso

Hay tres generales: un comandante y dos tenientes.

En el subcaso donde el comandante es el traidor, comunicará mensajes opuestos a cada uno de los tenientes. Los tenientes se comunican entre sí, con lo cual consiguen determinar únicamente que el comandante es un traidor, más no consiguen llegar a una decisión consensuada.

En el subcaso donde uno de los tenientes es el traidor, el comandante envía el mismo mensaje a ambos tenientes, y al momento de intercambiar la información entre ellos, el traidor cambia el mensaje recibido, ocasionando que tampoco se llegue a un consenso.

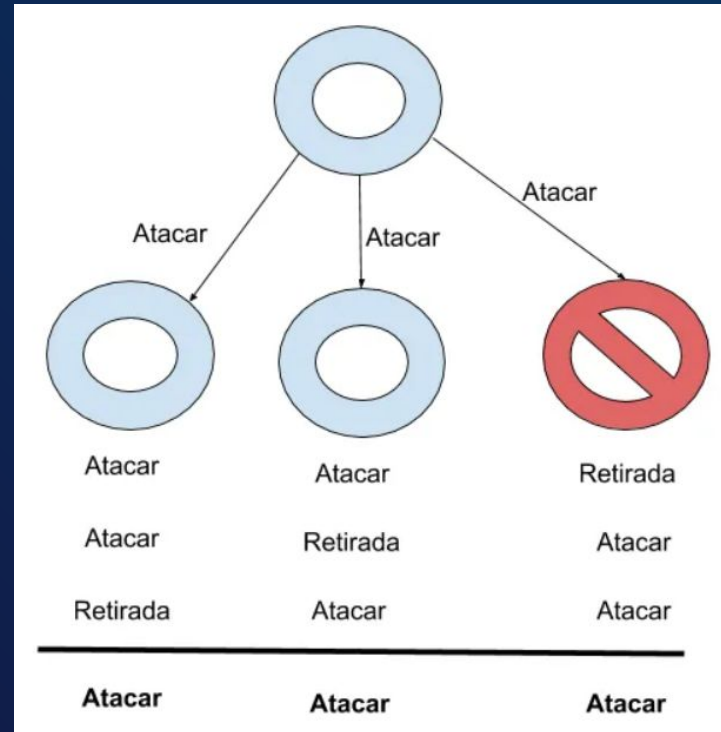
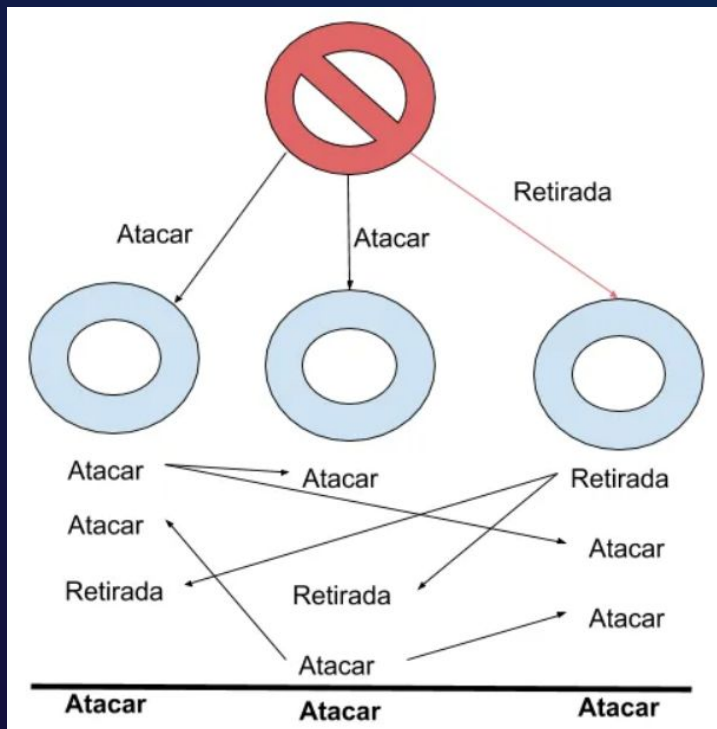


Segundo caso

Hay cuatro generales: un comandante y tres tenientes.

En el subcaso donde el comandante es el traidor, comunicará mensajes diferentes a los tenientes. Debido a que son tres tenientes y solo hay dos diferentes mensajes posibles, los tenientes intercambian información entre ellos y se forma el consenso en base al mensaje que haya recibido la mayoría.

En el subcaso donde uno de los tenientes es el traidor, se presenta un escenario parecido al subcaso anterior: el comandante envía el mismo mensaje a los tenientes, y aunque el traidor cambie el mensaje al momento de intercambiarlo con los demás, los otros dos poseen el mismo mensaje y la decisión se toma según esta mayoría.



Posibles soluciones

En 1982 Lamport (et al) demostraron que, en un sistema con k procesos defectuosos, el consenso se puede lograr sólo si están presentes $2k+1$ procesos correctos, para un total de $3k+1$ procesos.

Esto es, por ejemplo:

Si se tiene 1 proceso *traidor*, se necesita tener por lo menos 3 procesos *leales* para poder lograr el consenso, como se revisó en el segundo caso.

Si se tiene 1 proceso *traidor*, no es suficiente tener 2 procesos *leales*, pues no se cumple con la regla descrita, como se revisó en el primer caso.

Con 2 procesos *traidores*, necesitaríamos 5 procesos leales; con 3 procesos *traidores* necesitaríamos 7 procesos *leales*, y así sucesivamente

A lo largo del tiempo, se han formulado otras aproximaciones que pretenden resolver el problema, o por lo menos, reducir al mínimo posible la posibilidad de que el sistema se vea afectado por una falta de consenso.

Una de ellas es establecer un valor por defecto. En el contexto del problema, sería como que todos los generales tengan previamente definido que deben decidir retirarse en caso de que no se haya llegado a un consenso para atacar, o que se presente algún conflicto. En los sistemas distribuidos permitiría generalizar un mensaje en caso de que haya información conflictiva.

Otra aproximación es aumentar la cantidad de mensajes entre los generales, de manera que se pueda llegar a hacer una mejor confirmación de la información recibida, o incluso poder detectar quién es el traidor; poder detectar cuál es el proceso *erróneo*.

Ejemplo de implementación.

```
import random

class General:
    def __init__(self, nombre, es_traidor=False):
        self.nombre = nombre
        self.es_traidor = es_traidor

    def enviar_mensaje(self, mensaje):
        # Simula el envío de un mensaje, puede perderse o ser alterado
        if random.random() < 0.3:
            return None # El mensaje se perdió
        else:
            return mensaje

    def recibir_mensaje(self, remitente, mensaje):
        # Simula la recepción de un mensaje
        if remitente.es_traidor and random.random() < 0.5:
            # El general traidor puede alterar el mensaje
            return "Mensaje alterado"
        else:
            return mensaje

def consenso_bizantino(generales, umbral_decision):
    # Cada general propone una decisión
    decisiones = [random.choice(["Atacar", "Retirarse"]) for _ in generales]

    # Simula el proceso de consenso
    for general in generales:
        # Envía su decisión a los otros generales
        mensajes = [general.enviar_mensaje(decision) for decision in decisiones]

        # Recibe las decisiones de los otros generales
        decisiones_recibidas = [general.recibir_mensaje(remitente, mensaje) for remitente, mensaje in zip(generales, mensajes)]

        # Verifica el consenso
        decisiones_coherentes = [decision for decision in decisiones_recibidas if decision is not None and decision == decisiones[0]]

        # Si el umbral de consenso se alcanza, toma la decisión final
        if len(decisiones_coherentes) >= umbral_decision:
            return decisiones[0]

    return None # No se alcanzó el consenso
```

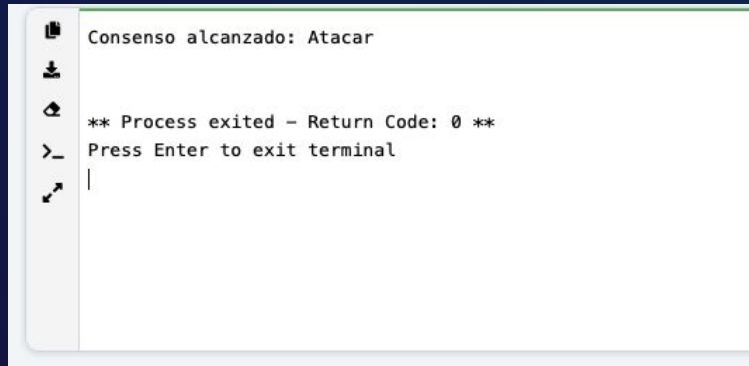
```
# Crear generales
general_A = General("A")
general_B = General("B", es_traidor=True)
general_C = General("C")

generales = [general_A, general_B, general_C]

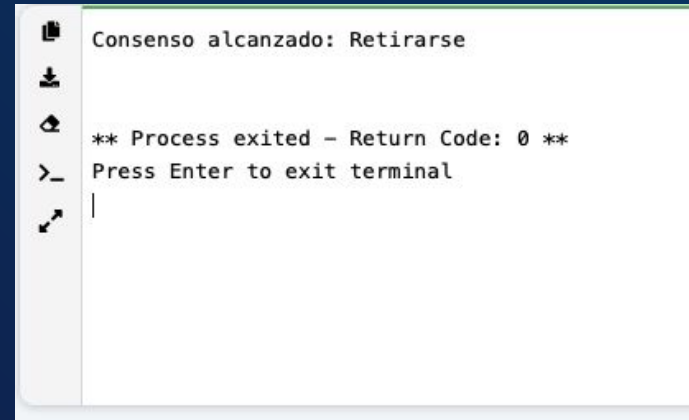
# Intentar alcanzar un consenso con un umbral del 2/3
consenso = consenso_bizantino(generales, len(generales) * 2 // 3)

# Resultado
if consenso is not None:
    print(f"Consenso alcanzado: {consenso}")
else:
    print("No se pudo alcanzar el consenso")
```

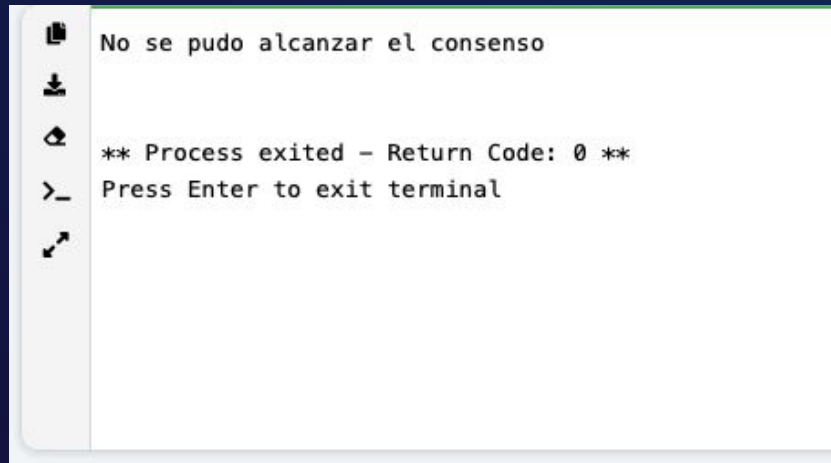
Resultados del Programa



```
Consenso alcanzado: Atacar
** Process exited - Return Code: 0 **
Press Enter to exit terminal
```



```
Consenso alcanzado: Retirarse
** Process exited - Return Code: 0 **
Press Enter to exit terminal
```



```
No se pudo alcanzar el consenso
** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Conclusión

Los sistemas de cómputo distribuidos ofrecen una gran cantidad de ventajas en aplicaciones de sistemas informáticos contemporáneos, al ser un concepto “novedoso” (con respecto al cómputo centralizado). Sin embargo, cada que surge un nuevo paradigma y una nueva manera de entender los sistemas, vienen ligados de ciertos problemas y desventajas que, en caso de no atenderse oportunamente, pueden llegar a opacar las ventajas que nos ofrecía en un principio.

Los aspectos de sincronización y consenso son los que más comúnmente pueden producir estos problemas, y por tanto obstaculizar el éxito de esta forma de manejar sistemas de cómputo.

Comprender los conflictos que estos aspectos pueden ocasionar, así como conocer y analizar las posibles soluciones aplicables, nos permiten aprovechar al máximo el potencial del cómputo distribuido en una gran variedad de aplicaciones.

Referencias

López Fuentes, F. (2015). *Sistemas Distribuidos [Digital]*. Universidad Autónoma Metropolitana.
http://dccd.cua.uam.mx/libros/archivos/03IXStream_sistemas_distribuidos.pdf

Rodríguez, A. (2019, 17 septiembre). ¿Conoces el problema de los generales bizantinos? Aquí te lo explico. Medium. Recuperado 15 de noviembre de 2023, de
<https://adr-rod87.medium.com/conoces-el-problema-de-los-generales-bizantinos-aqui%C3%AD-te-lo-explico-710d9b3da72>

Ezpeleta, J., & Álvarez, P. (2011, 23 agosto). Lección 12: Algoritmos de consenso. Universidad de Zaragoza. Recuperado 15 de noviembre de 2023, de
<https://webdiis.unizar.es/asignaturas/pscd/lib/exe/fetch.php?media=contenidos:leccion12.pdf>