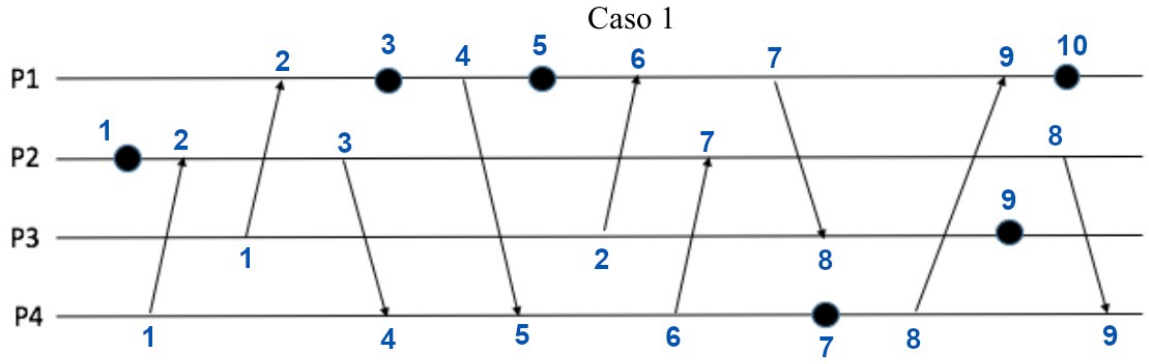
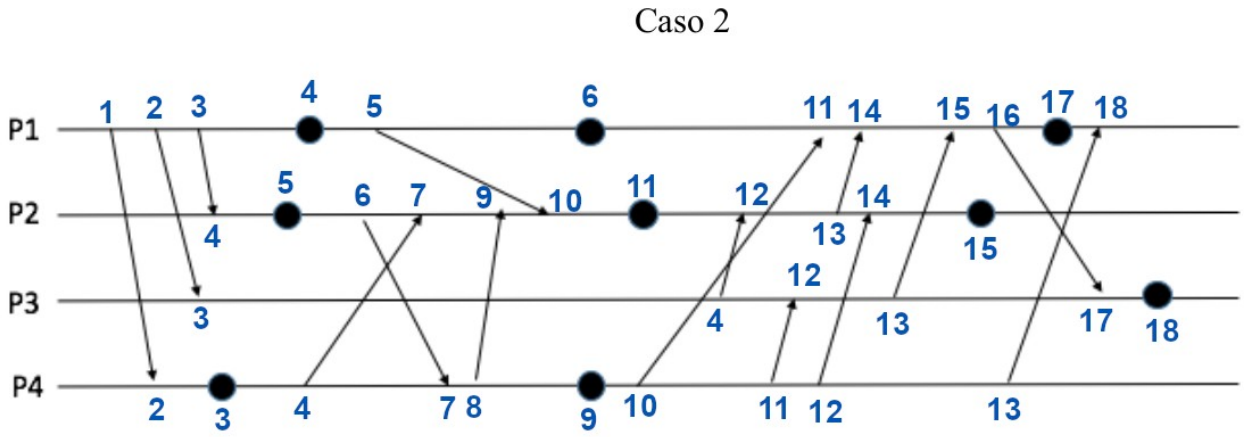
PEC2. Sistemas distribuidos. Kepa Sarasola Bengoetxea

1.1.A: relojes de Lamport

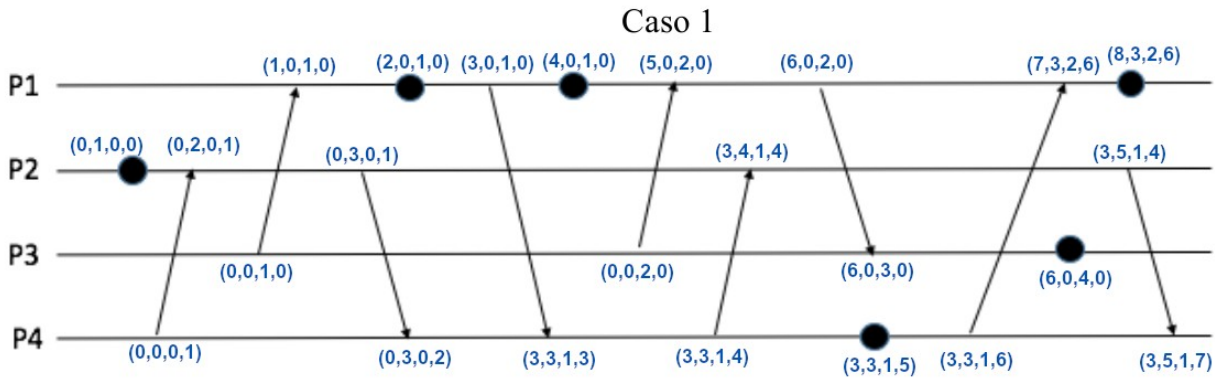


Caso 1, relojes de Lamport: Estado final P1=10, P2=8, P3=9, P4=9



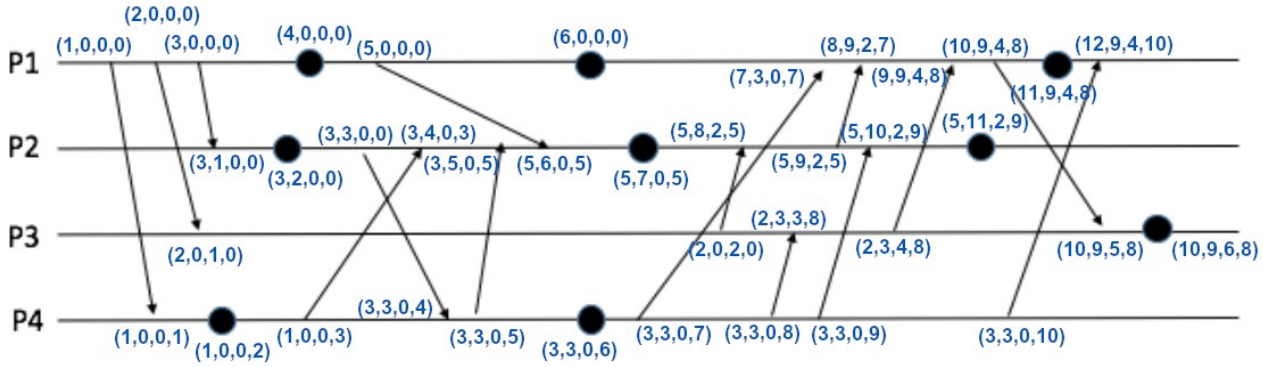
Caso 2, relojes de Lamport: Estado final P1=18, P2=15, P3=18, P4=13

1.1.B: relojes vectoriales



Caso 1, relojes vectoriales: Estado final P1=(8,3,2,6), P2=(3,5,1,4), P3=(6,0,4,0), P4=(3,5,1,7)

Caso 2



Caso 2, relojes vectoriales: Estado final P1=(12,9,4,10), P2=(5,11,2,9), P3=(10,9,6,8), P4=(3,3,0,10)

1.1.C: Explica la aportación clave de los relojes vectoriales.

Los relojes vectoriales asocian un valor vectorial V(x) a cada envío o recepción de un mensaje (evento x), que sucede en un sistema distribuido y permiten saber si un evento ocurre antes que otro y si dos eventos son concurrentes entre sí. Cada nodo mantiene un vector con *n* marcas temporales, siendo *n* el número de nodos, de tal forma que el nodo conoce el número de eventos que han ocurrido en su propio sistema y a su vez puede conocer un número mínimo de eventos que han ocurrido en otros nodos del sistema.

Los relojes vectoriales tienen las siguientes características:

- Decimos que V(a) < V(b), si cada una de las componentes de V(a) es menor o igual que la respectiva componente de V(b) y además debe de haber al menos una de ellas que sea estrictamente menor.
- Si a→b, es decir si a ocurre antes que b, podemos decir entonces que el reloj vectorial de a es menor que el de b, V(a)<V(b).
- Lo mismo ocurre al contrario, es decir si V(a)<V(b) entonces podemos decir que a→b, es decir, que a ocurre antes que b.
- Si no se cumple lo anterior podemos decir que ambos eventos son concurrentes.

La aportación con de los relojes vectoriales, con respecto a los relojes lógicos como los de Lamport es que:

- En el caso de los relojes lógicos como el de Lamport un nodo no tiene ningún conocimiento acerca de los eventos ocurridos en otros nodos del sistema, mientras que esto mismo es una característica de los relojes vectoriales.
- En el caso de los relojes lógicos, el algoritmo garantiza que si a→b los relojes cumplen la misma condición es decir C(a)<C(b), pero esto no ocurre al contrario, es decir si C(a)<C(b) no se puede concluir que a→b.
- Esto no ocurre en los relojes vectoriales, ya que se cumple que si a→b entonces V(a)<V(b), pero también se cumple que si V(a)<V(b) entonces a→b

2.1. Compara el protocolo “two-phase commit” con “three phase commit”:

Tanto el protocolo “two-phase commit” como el protocolo “three phase commit” son protocolos de consenso distribuido, que permiten a todos los nodos de un sistema distribuido llegar a un acuerdo para realizar una transacción. Estos protocolos se suelen utilizar en bases de datos distribuidas.

El protocolo de 2 fases o “two-phase commit”, se desarrolla durante dos fases:

- **La fase de preparación**, en esta fase el coordinador de la operación se pone en contacto con todos los nodos participantes preparándoles para la siguiente fase. Los nodos contestan si aceptan o no (abortan) la operación, pero esta respuesta supone un compromiso por parte del nodo una vez iniciada la siguiente fase, por lo que es necesario que el nodo se asegure de que cuenta con los recursos suficientes para poder realizar la transacción, en el caso de que esto no sea así, es decir en el caso de que no pueda afrontar la transacción el nodo abortará esta.

- **La fase de commit,** en esta fase, el coordinador registra las respuestas, informando a todos los participantes del resultado de la fase anterior y si esta es positiva confirmando la transacción. En caso pues de que sea positiva, los participantes también confirman la transacción, dándose por permanente.

El protocolo de 3 fases añade una **fase preCommit** en la que el coordinador notifica a todos los nodos la decisión tomada sobre si se puede realizar el *commit* o no, es decir realiza una petición de *preCommit* o *doAbort*, según el caso. Los participantes una vez informados realizan un acuse de recibo de la resolución. Una vez recibidos los acuses de recibo por parte del coordinador, la fase *commit* se pone en marcha.

El protocolo de 3 fases “three phase commit”, resuelve el problema de los posibles fallos del coordinador de la operación y los posibles bloqueos que pueden darse en los nodos en consecuencia. El protocolo en 2 fases es bloqueante con respecto a los nodos, ya que estos se encuentran a la espera de recibir los mensajes del coordinador y si este falla por cualquier motivo pueden quedarse en este estado, bloqueados. Además la principal ventaja del protocolo “three phase commit”, es que cada nodo está informado del desarrollo de la operación y no depende del coordinador para conocer las decisiones que cada nodo ha tomado, por lo que ante una caída del nodo coordinador o de cualquier otro nodo, otro nodo puede sustituir a este y llevar a cabo la operación con éxito.

Como contrapartida el protocolo de 3 fases es más costoso y complejo en cuanto al aumento del número de mensajes y el de rondas necesarias en un caso sin fallos.

2.2.C: a) todos los participantes hacen “commit”

“Two-phase commit”

1. Fase de preparación. El coordinador enviará 4 mensajes, uno a cada uno de los nodos participantes, para que preguntarles si están preparados para el commit. Si no hay problemas, estos responderán cada uno con un mensaje afirmativo, por lo tanto se enviarán otros 4 mensajes.

2. Fase de commit. El coordinador enviará 4 mensajes para realizar el Commit, los mensajes posteriores de los nodos una vez realizados el commit, informando de su realización y consecuencias no se tienen en cuenta.

   Por lo tanto los mensajes serán 12 y el tiempo necesario será el de 3 rondas de mensajes

“Three phase commit”

1. Fase de preparación. Como en el protocolo anterior, el coordinador enviará 4 mensajes, uno a cada uno de los nodos participantes, para que preguntarles si están preparados para el commit. Si no hay problemas, estos responderán cada uno con un mensaje afirmativo, por lo tanto se realizarán otros 4 mensajes.
2. Fase preCommit. El coordinador enviará 4 mensajes para notificar la confirmación del Commit, y los nodos transmitirán cada uno un mensaje con el acuse de recibo.
3. Fase Commit. El coordinador enviará 4 mensajes para realizar el Commit, los mensajes posteriores de los nodos una vez realizados el commit, informando de su realización y consecuencias no se tienen en cuenta.

Por lo tanto los mensajes serán 20 y el tiempo necesario será el de 5 rondas de mensajes.

b) dos participantes hacen “abort”.

“Two-phase commit”

1. Fase de preparación. El coordinador enviará 4 mensajes, uno a cada uno de los nodos participantes, para que preguntarles si están preparados para el commit. Dos de ellos responderan afirmativamente y los otros 2 responderán abort.
2. Fase de commit. El coordinador enviará 2 mensajes para abortar a los dos participantes que han respondido afirmativamente, los otros dos participantes no recibirán ningún mensaje.

Por lo tanto los mensajes serán 10 y el tiempo necesario será el de 3 rondas de mensajes

“Three phase commit”

1. Fase de preparación. Como en el protocolo anterior, el coordinador enviará 4 mensajes, uno a cada uno de los nodos participantes, para que preguntarles si están preparados para el commit. Si no hay problemas, estos responderán cada uno con un mensaje afirmativo, por lo tanto se realizarán otros 4 mensajes.
2. Fase preCommit. El coordinador enviará 2 mensajes para abortar a los dos participantes que han respondido afirmativamente, los otros dos participantes no recibirán ningún mensaje.
3. Fase Commit. No se pasa a esta fase

Por lo tanto los mensajes serán 10 y el tiempo necesario será el de 3 rondas de mensajes

3. Explica brevemente 4 algoritmos diferentes para la exclusión mutua:

La exclusión mutua es una herramienta que permite compartir un recurso o un conjunto de recursos por parte de los procesos del sistema de manera que los datos comprometidos en esos procesos mantengan su integridad. Los requisitos esenciales para la exclusión mutua son:

- ME1 (seguridad): como máximo un proceso puede estar ejecutándose en la sección crítica en un mismo instante.
- ME2 (supervivencia): las solicitudes para entrar y salir de la sección crítica deben de ser concedidas con éxito, que no existan deadlocks ni inanición.
- ME3(orden): si una solicitud para entrar a la sección crítica ocurrió antes que otra, la entrada en la sección crítica ha de darse en ese orden.

Algoritmo del servidor central
Usar un servidor que conceda los permisos para entrar en la sección crítica (SC) es la manera más sencilla de lograr la exclusión mutua. El proceso es el siguiente, un proceso que necesita acceder a una SC realiza una petición al servidor y espera a que este le conteste. Esta solicitud se responderá de manera instantánea si el acceso a la SC está libre, pero en caso contrario se integrará en una cola de solicitudes que el servidor responderá en función de su antigüedad.

Este sistema tiene el problema de que no satisface la condición ME3(debido a los posibles delays en la transmisión de mensajes), además de que puede convertirse en un cuello de botella, ya que cada proceso que necesita acceder a la SC el servidor han de gestionarse por lo menos dos mensajes, el de la solicitud y el de la concesión por parte del servidor, además el proceso que abandona la SC debe de enviar otro mensaje al servidor notificando que ha abandonado la sección, este último mensaje también implica un retraso en el sistema del servidor.

Algoritmo basado en un anillo
Esta manera de gestionar el acceso a la SC es bastante simple, igual que la anterior, la diferencia estriba en que en este caso no se usa un servidor, sino un anillo lógico. La idea es que el permiso de acceso (token) se pasa de un proceso a otro en una sola dirección, es decir, si un proceso no necesita entrar en la SC cuando recibe el token, se lo pasa a su vecino, en caso de que requiera entrar en la SC, espera a recibir el token, lo retiene mientras trabaja en al SC y cuando va a salir de esta envía el token a su vecino.

Este algoritmo satisface las dos primeras condiciones ME1 y ME2, pero no se puede asegurar que se cumpla ME3. Otro de los inconvenientes de éste algoritmo es que consume ancho de banda de manera continua, exceptuando el momento en el que un proceso está dentro de una SC, ya que los procesos envían mensajes al rededor de un anillo, incluso cuando hay un proceso que requiere la entrada a la SC.

Algoritmo que usa multidifusión y relojes lógicos (Ricart y Agrawala)
Este algoritmo se basa en *multicast* o *multidifusión*, la idea es que los procesos que requieren la entrada en una SC envíen un mensaje de petición mediante multidifusión al resto y puedan entrar en ella sólo cuando el resto de procesos lo hayan respondido. Si un proceso envía una solicitud de entrada y el estado de todos los demás procesos (y su

respuesta) es de LIBERADO, el solicitante obtendrá la entrada. Si algún proceso está en el estado de RETENIDA (dentro de la SC), entonces este proceso no responderá hasta que salga de la SC, por lo que el solicitante tendrá que esperar. Las solicitudes se ordenan por medio de las marcas de tiempo (relojes lógicos) de Lamport. Este algoritmo cumple las condiciones M1 y M3.

La ventaja de este algoritmo es que su retraso de sincronización es sólo el tiempo de transmisión de un mensaje, mientras que en el resto de algoritmos se incurre en un retraso de sincronización por mensajes de ida y vuelta.

Algoritmo de votación de Maekawa
Maekawa observó que para que un proceso pudiese entrar en una SC no era necesario que los demás procesos del mismo tipo le permitieran el acceso. Los procesos necesitan obtener el permiso de un subconjunto de sus compañeros, siempre y cuando estos subconjuntos se solapen. La idea es que un proceso necesita un número de votos de otros procesos similares a él para poder entrar en una SC, la única restricción es que se debe de asegurar la condición ME1, es decir que solamente un proceso puede entrar de manera simultanea en la SC.

Este algorimo cumple con las condiciones ME1 y ME3 y en cuanto al rendimiento podemos decir que la espera para el solicitante es igual que en el caso del algoritmo de multidifusión, pero el retraso por la sincronización aumenta.

Escalabilidad
Para evaluar la escalabilidad nos basaremos en 3 factores, el ancho de banda, el retraso con el cliente y el retraso en la sincronización:

- Algoritmo del servidor central:
  - Es el menos ancho de banda consume, con 3 mensajes se accede a la SC.
  - El retraso es el tiempo que tarde la respuesta del servidor central.
  - El retraso en la sincronización es de 1 tiempo.
  - Pueden producirse cuellos de botella.
- Algoritmo basado en un anillo:
  - El ancho de banda se usa de manera continua, aunque no se entre en la SC.
  - El retraso del cliente va de 1 a N mensajes por N procesos que haya.
  - El retraso en la sincronización es de N-1 mensajes.
- Algoritmo que usa multidifusión:
  - Sólo se usa el ancho de banda cuando se entra en la SC, 2*(N-1) mensajes.
  - El retraso con el clientes es de 2n-2 mensajes.
  - El retraso de la sincronización es de 1 tiempo.
- Algoritmo de votación:
  - El ancho de banda es 2*raíz(n) para los mensajes de entrada y raíz(n) para los de salida
  - El retraso del cliente es de 2n-2 mensajes
  - El retraso en la sincronización es de 2 tiempos

Tolerancia a fallos
Respecto de la tolerancia a fallos de los algoritmos que hemos visto:
- Ninguno de los algoritmos tolera la perdida de mensajes.

- El algoritmo basado en anillo no puede tolerar un fallo por caída de ningún proceso individual.
- El algoritmo de Maekawa puede tolerar que algunos procesos se caigan, siempre que estos no participen en el proceso de votación.
- El algoritmo de servidor central puede tolerar un fallo de un proceso cliente, sin riesgo, pero si falla el servidor cae el sistema completo.
- El algoritmo multicast puede adaptarse para tolerar la caída de un proceso haciendo que este conceda todas las peticiones.

4.a. Transacciones en bases de datos: ACID y BASE:

Se denomina transacción a una única operación lógica de negocio. ACID y BASE son dos filosofias de diseño en cuanto a bases de datos que están en extremos opuestos en cuanto a consistencia-disponibilidad. Mientras que los principios ACID se centran en la consistencia de las bases de datos, las indicaciones de la filosofía BASE se centran en aprovechar las ventajas en cuanto a disponibilidad de los sistemas distribuidos.

Los principios ACID son un grupo de 4 propiedades que dicen que las transacciones en las bases de datos deben de ser realizadas de forma confiable:

- **A**tomicidad: requiere que cada transacción se realice completamente o no se realice, es decir si alguna operación de la transacción falla, falla toda la operación.
- **C**onsistencia: esta propiedad asegura que cualquier transacción finalizada parte de un estado válido de la base de datos y termina con una situación igualmente válida. Es decir, cualquier dato que se escriba en la base de datos ha de cumplir las propias normas definidas en ella mediante por ejemplo, constraints, tiggers, etc.
- **I**solation (aislamiento): la ejecución concurrente de las transacciones tiene que asegurar que no interfieran unas con otras, o lo que es lo mismo que el resultado ha de ser igual que si las transacciones se hubieran realizado una detrás de otra.
- **D**urabilidad: significa que una vez confirmada una transacción (commit), quedará persistida incluso en situaciones de fallo general o catástrofe.

Todas estas propiedades son muy interesantes y parecen indispensables, pero son incompatibles con la disponibilidad y el rendimiento de sistemas muy grandes.

BASE surge como una alternativa a ACID, veamos el significado de este acrónimo:

- Disponibilidad **Bá**sica: el sistema estará disponible todo el tiempo posible.
- Estado **S**oft (Suave): el sistema puede cambiar, o pueden efectuarse cambios en el sistema en cualquier momento sin necesidad de que exista una entrada de datos en ese momento.
- Consitencia **E**ventual: puede ser que cuando se de una entrada de datos en el sistema, este no verifique la consistencia de la transacción.

La diferencia entre los dos conceptos es que en vez de requerir consistencia después de cada transacción, es suficiente que la base de datos esté eventualmente en un estado consistente, pudiendo usar para ello datos obsoletos y/o respuestas aproximadas. En principio parece que el sistema ACID es más fácilmente controlable y más fácil de desarrollar que una aproximación BASE, pero el teorema de CAP de Brewer deja claro esta es la única opción si se quiere escalar el sistema.

Según el teorema de CAP de Eric Brewer, si se desea consistencia, disponibilidad y tolerancia a fallos parciales en nodos (partición de la red), se ha de conformar con dos de estas tres características. El teorema no afirma que esta decisión haya que tomarse en todo momento, sino que hay que dejar el sistema abierto a que esto pueda suceder, y si pueda ocurrir que en algún momento haya que elegir entre consistencia y disponibilidad.

4.b. Replicación activa y replicación pasiva:

Podemos definir replicación como el conjunto de técnicas que usa el mantenimiento de copias de un recurso con el objetivo de proporcionar un mayor rendimiento, disponibilidad y escalabiliad a un sistema. En cuanto a los objetos replicados, estos han de cumplir con la propiedad de lineabilidad, es decir, se ha de cumplir que si para cualquier ejecución existe algún entrelazamiento de operaciones, estas han de satisfacer que:

- La secuencia entrelazada cumple la especificación de una única copia correcta de los objetos.
- El orden de las operaciones del entrelazamiento es consistente con los tiempos reales en los que ocurrieron las operaciones en tiempo real.

Replicación Activa:
La petición realizada por un cliente se envía a todas las replicas del sistema, las cuales tratan esta de manera concurrente, el cliente se actualiza con la primera respuesta que recibe y descarta las siguientes. Es por esto que cada proceso realizado es determinista es decir, dada una solicitud, todos los nodos producirán la misma secuencia de respuesta y devolverán el mismo estado.

La replicación activa permite reducir la latencia al permitir el acceso a varias replicas de las solicitudes del cliente, pero hay que tener en cuenta que el coste de que las actualizaciones se procesen en todos los nodos es muy alto. Este sistema no logra cumplir el principio de lineabilidad.

Replicación Pasiva:
El sistema se compone de un nodo primario y un conjunto de nodos secundarios en los que se realiza la copia del nodo primario. Cuando el nodo primario recibe una petición del cliente, este la procesa y se encarga de actualizar el estado en los servidores secundarios enviando la actualización a estos, estos una vez realizada la actualización envían un mensaje de ACK al primario. Estos servidores secundarios ejercen una labor de *backup*, y no han de manejar concurrencia, ya que esta es manejada por el servidor primario.

En este modelo es necesario gestionar las altas y las bajas de manera consistente, ya que si un nodo primario falla, este ha de ser sustituido por uno de los secundarios. Pero por otro lado hay que decir que también es un sistema costoso ya que incluye un número mayor de rondas de comunicación entre los servidores y además al tener que sustituir al servidor principal en caso de fallo, todos los servidores secundarios, mantienen todos los datos del servidor principal. La replicación pasiva cumple los principios de lineabilidad.

4.c. Control de concurrencia: Wikipedia y Google Docs:

El control de concurrencia asegura que se generen resultados correctos para operaciones concurrentes. Existen dos modelos de control de concurrencia. El modelo optimista no

utiliza el bloqueo de recursos y permite que varios usuarios intenten actualizar el mismo recurso sin informar a estos de que otros usuarios tratan de actualizar el registro. Los cambios de registro se validan cuando se confirma el registro (commit). Este modelo es óptimo cuando se espera que las actualizaciones simultaneas no sean frecuentes.

Por otra parte el modelo pesimista, impide actualizaciones simultáneas de los registros, ya que cuando un usuario empieza a modificar un registro, este se bloquea, de tal manera que el resto de usuarios no pueden realizar cambios sobre el mismo. Este es un enfoque útil cuando se pueden retardar las actualizaciones posteriores hasta que se puedan realizar las anteriores.

Wikipedia
Wikipedia utiliza un modelo de control de concurrencia optimista, ya que cuando más de un usuario deciden realizar un cambio sobre el mismo registro, wikipedia guarda el primer commit realizado sobre el registro y trata de adecuar los siguiente cambios sobre el primero. En caso de conflicto los editores son notificados y es su responsabilidad solucionar el conflicto que en todo caso deberá resolverse de manera manual entre los diferentes editores.

Google Docs
En Google Docs, que tambien utiliza un modelo optimista, todos los usuarios que tienen permisos sobre un documento pueden realizar actualizaciones de manera continua sobre ellos, los cambios se van viendo en la pantalla. La técnica que subyace sobre este modo de funcionamiento es “Operation transformation” que se basa en mantener replicas del mismo documento, con el objetivo de no tener bloqueos. La actualización del documento se basa siempre en el último cambio que se ha guardado.

5. Generales bizantinos, Wikipedia.

El problema de los generales bizantinos es una analogía a un problema que puede darse en un sistema distribuido, que no es más que un conjunto de sistemas informáticos con un objetivo común. Ante la posibilidad de fallos en el sistema, los nodos han de llegar a un consenso sobre cómo actuar en cada caso, para lo que deben de establecer un conjunto de reglas y llegar a un acuerdo sobre como implementar estas.

Académicamente, el problema de los generales bizantinos se fija en 1982, cuando Lamport, Shostak y Pease publican su articulo (https://people.eecs.berkeley.edu/~luca/cs174/byzantine.pdf). La analogía utilizada para explicar el problema es la siguiente: varios generales preparan una batalla sobre una ciudad enemiga, estos generales sólo pueden comunicarse vía mensajero y entre ellos han de acordar un plan común atacar o retirarse. En este caso se plantea que existe un número indeterminado de traidores que intentará desbaratar las comunicaciones. Para resolver el problema, los generales leales necesitan llegar a un acuerdo de manera segura (consenso) y llevar a cabo el plan elegido (coordinación). Este problema se aplica en realidad a cualquier grupo distribuido de nodos que necesite lograr comunicaciones confiables.

Lo que en la analogía se denomina traición, en un sistema informático real pude identificarse como fallos de software, mal funcionamiento del hardware o un ataque

malintencionado, es decir de manera indistinta pueden ser provocados por un mal funcionamiento o por ataques foráneos al sistema, por lo que se pueden denominar como fallos inevitables en general.

Para responder a este problema los antes mencionados Lamport, Shostak y Pease plantean ciertas soluciones:

- Demuestran que si n es el número de generales en total y t es el número de traidores, existen soluciones al problema solamente cuando se cumple que n>3t+1 y la comunicación es sincrónica.
- La segunda solución se basa en la firmas digitales de los mensajes envíados, pero los sistemas basados en esta solución no son los mejores para sistemas con un nivel de seguridad crítico.

Más adelante se presentaron soluciones más avanzadas (Castro, Liskov 1999), como el PBFT, BFT, etc. una implementación muy conocida de esta última es el bitcoin, que funciona en paralelo para generar una cadena de bloques con prueba de trabajo permitiendo al sistema superar estos fallos y lograr una visión global coherente del sistema.

Propuesta de información para agregar al artículo
Las soluciones planteadas en el documento de Wikipedia, como en la bibliografía proporcionan una solución al problema bizantino solamente cuando se asegura que la comunicación del sistema se realiza de manera sincrónica, ya que se asume que los intercambios de mensajes se realizan en las rondas, quedando fuera de estas rondas los mensajes que debido al retardo han superado un tiempo máximo.

En un sistema asíncrono, los procesos pueden responder a los mensajes en momentos arbitrarios, por lo que queda claro que tal y como Fisher (1985) demostró no es posible solucionar este problema en sistemas asíncronos.

Aún así existen ciertas técnicas para trabajar con nodos o procesos fallidos en sistemas asíncronos:

- Enmascaramiento de fallos: esta técnica consiste en evitar en todos los casos un resultado fallido de un proceso, reiniciándose este en caso de que sea necesario y devolviendo un resultado optimo en todos los casos. Es decir el sistema se comporta como si no hubiera existido el fallo, se trata como un resultado correcto, pero que ha tardado mucho tiempo en dar el resultado.
- Utilizando detectores de fallos, que se encargan de detectar los procesos que no responden (por consecuencia de un fallo o porque están tardando demasiado) del proceso de toma de decisiones. Esta técnica convierte en la práctica un sistema asíncrono en un sistema sincrónico. Existe la posibilidad de establecer de manera flexible estos detectores para dotar al sistema de mayor flexibilidad.
- Usando el azar, esta técnica depende de que podamos considerar el fallo como un adversario que pueden explotar las características del sistema para frustrar los intentos de llegar a un consenso. La aplicación de la técnica consiste en la introducción de procesos con un comportamiento aleatorio, de modo que el adversario no pueda realizar su estrategia de manera efectiva.