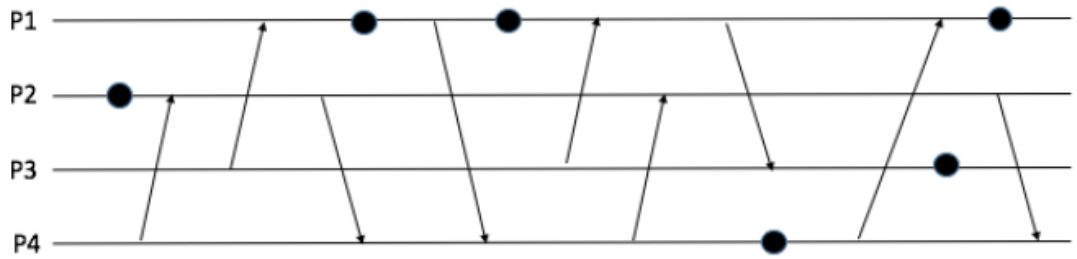
Sistemas distribuidos

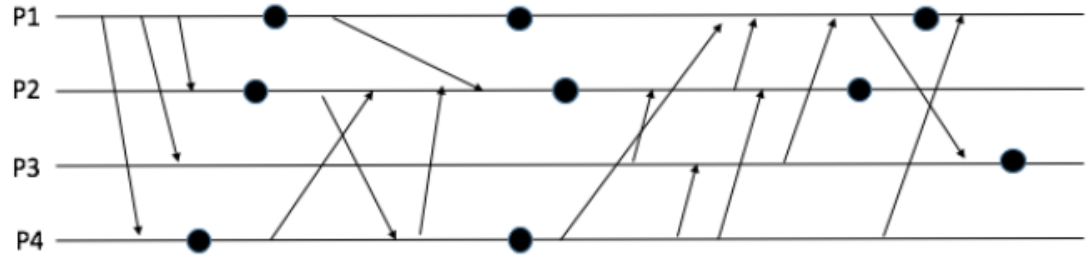
PEC 2

1. En los siguientes casos, donde tanto las flechas como los puntos indican el progreso del reloj en cada proceso, responde a las preguntas:

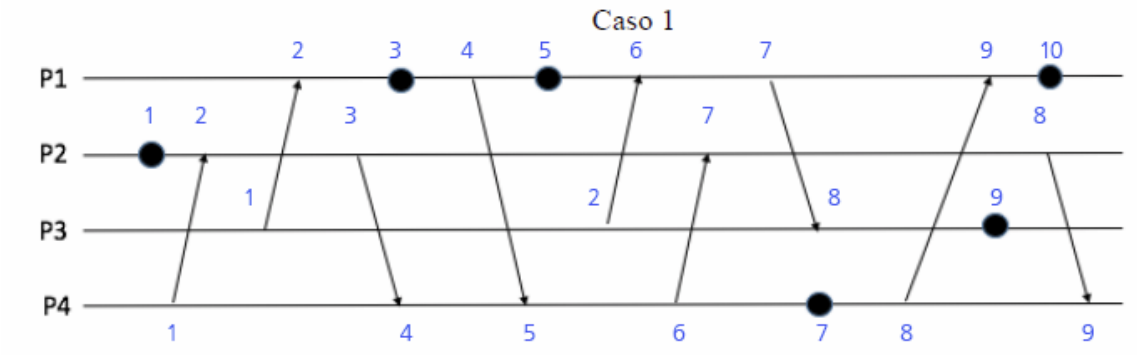
Caso 1



Caso 2

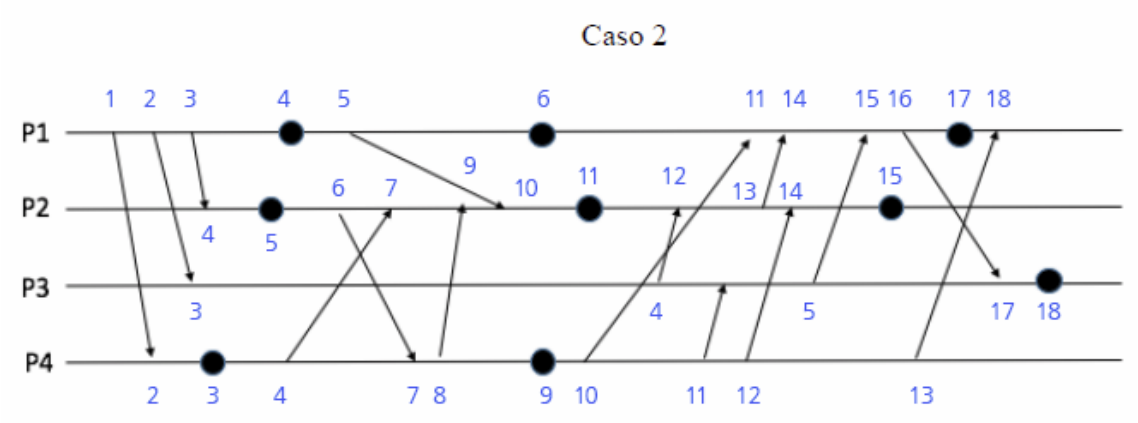


A. Con relojes Lamport. Todos los relojes comienzan en cero. Etiqueta ambos diagramas (caso 1 y caso 2) con los valores del reloj de Lamport. Muestra el estado final de los relojes.



Estado final de los relojes de Lamport del Caso 1:

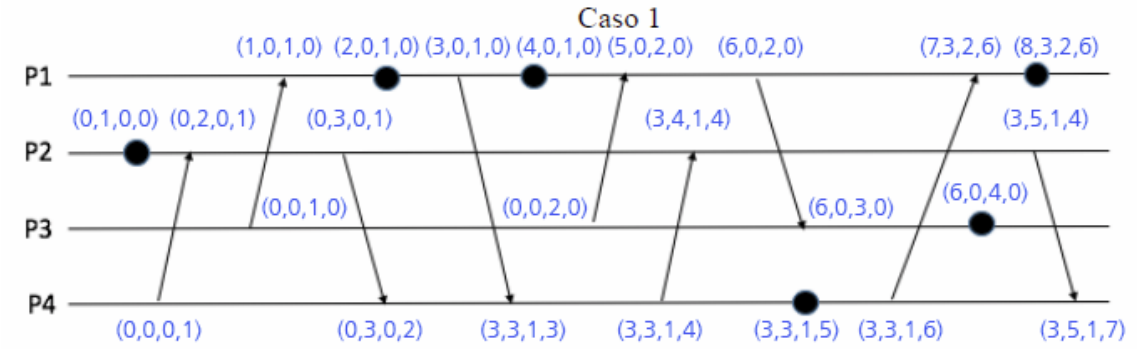
P1 = 10
P2 = 8
P3 = 9
P4 = 9



Estado final de los relojes de Lamport del Caso 2:

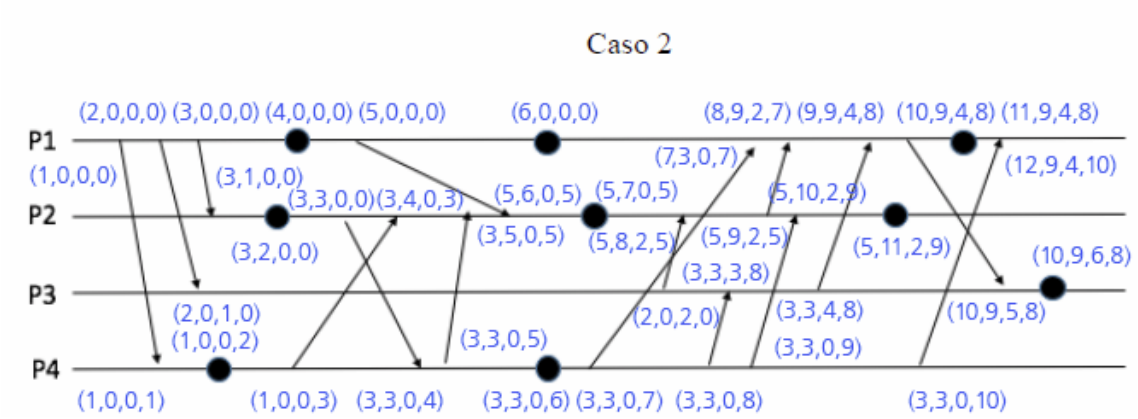
P1 = 18
P2 = 15
P3 = 18
P4 = 13

B. Con relojes vectoriales. Todos los relojes comienzan en (0,0,0,0). Etiqueta ambos diagramas (caso 1 y caso 2) con los valores del reloj vectorial. Muestra el estado final de los relojes.



Caso 1 Vectorial:

P1 = (8, 3, 2, 6)
P2 = (3, 5, 1, 4)
P3 = (6, 0, 4, 0)
P4 = (3, 5, 1, 7)



Caso 2 Vectorial:

P1 = (12, 9, 4, 10)
P2 = (5, 11, 2, 9)
P3 = (10, 9, 6, 8)
P4 = (3, 3, 0, 10)

C. Explica la aportación clave de los relojes vectoriales.

Dos eventos, por ejemplo A y B, tienen una relación de causalidad entre ellos si A sucede antes que B, es decir, A → B.

El signo → se lee como “happened before” (ocurrió antes) y denota que un evento se ha producido antes que otro.

Cuando se usan relojes lógicos (o relojes Lamport), se tiene en cuenta una sola marca de tiempo en cada evento, por lo que estos relojes nos permiten saber si un evento ocurrió antes que otro pero no nos permiten saber con certeza la causalidad entre dichos eventos. Esto significa que aunque el tiempo asociado a A sea menor que el tiempo asociado a B, es decir, A < B, los relojes de Lamport no nos permiten asegurar que A haya ocurrido antes que B, es decir, A → B. Lamport no nos permite saber si los eventos son o no concurrentes.

Los relojes vectoriales, por otra parte, almacenan varias marcas de tiempo dentro de un vector. En este vector, cada elemento es un número que corresponde a un evento y se conoce la posición que cada evento ocupa dentro del vector, por lo que para averiguar si un evento ha sucedido antes que otro o si dos eventos son concurrentes, basta con comparar las marcas de tiempo de cada elemento y su posición. De esta forma, se puede establecer el orden en el que se han producido los eventos contenidos en el vector.

En resumen, los relojes vectoriales nos permiten saber si un evento A ocurrió antes que un evento B y si ambos eventos fueron concurrentes o no, ya que tienen en cuenta las marcas de tiempo de todos los eventos del sistema y estas nos permiten conocer el orden exacto en el que ocurrieron dichos eventos.

2. Compara brevemente el protocolo “two-phase commit” con “three phase commit”.

Two-phase commit protocol es un protocolo diseñado para permitir que todos los hosts que forman un sistema distribuido lleguen a un consenso para hacer *commit* de una transacción.

En el inicio del proceso, no hay comunicación entre el coordinador y los participantes más allá de los participantes informando al coordinador de que se incorporan a la transacción.

En la primera fase, el coordinador pregunta a todos los hosts participantes si están dispuestos a comprometerse para hacer *commit* (*canCommit?*).

En la segunda fase, el coordinador les dice a los participantes que confirmen (*doCommit*) o aborten (*doAbort*) la transacción.

Si uno o más participantes solicitan *doAbort*, el coordinador informa a todos los participantes inmediatamente y la transacción es abortada.

Si todos los participantes votan hacer *commit* (*doCommit*), este se realiza siempre y cuando no se encuentre un conflicto, en cuyo caso el *commit* se cancela y el coordinador envía un mensaje a todos los participantes para que aborten la transacción. Si no hay conflictos, se realiza el *commit* y luego los participantes envían un mensaje de confirmación al coordinador (*haveComitted*).

El *Three-phase commit protocol*, por otro lado, añade una fase intermedia (*preCommit*) en la que el coordinador envía un mensaje a todos los hosts preguntando si se va a hacer *commit*. Si el coordinador recibe una respuesta afirmativa de todos los hosts (*acknowledgement*), se hace *commit*.

Este protocolo soluciona un problema presente en el protocolo de 2 fases: evita que el proceso se cancele si el coordinador falla. En el *two-phase commit protocol*, si el coordinador falla, se crea un período de incerteza en el que algunos participantes nunca realizarían o abortarían sus transacciones. En el protocolo de 3 fases, en cambio, cada participante almacena la decisión de forma local, así que si el coordinador cae, la ejecución del *commit* no se ve afectada.

Como contrapartida, el protocolo de 3 fases consume más recursos y más tiempo que el de 2 fases ya que tiene que mandar más mensajes.

Otra de las diferencias entre ambos protocolos es que en el *three-phase commit protocol*, el coordinador puede cancelar unilateralmente el *commit* o designar un nuevo coordinador mientras que en el *two-phase commit protocol*, en cambio, el coordinador solo puede cancelar la transacción si uno o varios participantes envían un mensaje *doAbort*.

a) Indica el rendimiento de los dos protocolos cuando tenemos 1 coordinador y 4 participantes y todos los participantes hacen “commit”

En el caso de usar *Two-phase commit protocol,* si todo va bien - eso es, el coordinador, los participantes y la comunicación entre ellos no fallan - el protocolo de dos fases, que en este caso involucra a 4 participantes, se puede completar con 4 mensajes de *canCommit* y sus 4 respuestas, seguidas de 4 mensajes *doCommit*.

El costo en mensajes es proporcional a 3N, en este caso, 3 * 4 = 12 mensajes, y el costo en tiempo es de 3 rondas de mensajes. Recordemos que los mensajes *haveCommitted* no se cuentan en el costo.

En el caso de usar *Three-phase commit protocol*, se añaden el paso *preCommit* y sus *acknowledgements*, por lo que el coste en mensajes es proporcional a 5N.

Al haber 4 participantes involucrados, el costo en mensajes en *Three-phase commit protocol* es 20 (5 * 4) y el costo en tiempo es de 5 rondas de mensajes.

b) Indica el rendimiento de los dos protocolos cuando tenemos 1 coordinador y 4 participantes y dos participantes hacen “abort”.

En *Two-phase commit protocol*, el coordinador envía una solicitud *canCommit?* a los participantes. Cuando un participante recibe un *canCommit?* puede responder Sí o No. En este caso, dos participantes votan sí y otros dos votan no (hacen “abort”).

En la segunda fase, el coordinador recoge los votos. Como hay al menos un voto No, el coordinador cancela la transacción y envía *doAbort* a los participantes que votaron Sí. Los participantes que votaron Sí reciben esta solicitud y abortan el proceso de *commit*.

En total, hay 4 mensajes de *canCommit?*, 4 respuestas y 2 mensajes de *doAbort* enviados por el coordinador hacia los dos participantes que estaban preparados para hacer *commit*. Los 2 participantes que abortaron ya no reciben este mensaje, por lo tanto, hay 10 mensajes en total enviados en 3 rondas.

En el caso de usar *Three-phase commit protocol*, el coordinador envía 4 mensajes *canCommit?* a lo que 2 participantes responden *doCommit* y 2 participantes responden *doAbort*. Por tanto, se usan 4 mensajes *canCommit?*, 4 respuestas (2 mensajes *doAbort* más 2 mensajes *doCommit*) y 2 respuestas *doAbort* del coordinador a los participantes que respondieron *doCommit*. Esto hace un total de 10 mensajes en 3 rondas de mensajes, igual que ocurre en *Two-phase commit protocol*. Como se aborta el *commit*, el proceso no llega al *preCommit*, de ahí que se mantenga en 3 rondas.

3. Explica brevemente 4 algoritmos diferentes para la exclusión mutua. Analice su escalabilidad y tolerancia a fallas. ¿Cómo podemos solucionar, si existe, el problema de escalabilidad en estos algoritmos?

Multicast

El algoritmo multicast se basa en multidifusión desde un host hacia el resto de host. Cuando todos han respondido al mensaje, se permite utilizar el recurso. Si un host está ocupado, solo se liberará el recurso cuando este host conteste.

Este algoritmo usa un alto porcentaje de ancho de banda ya que requiere del envío de muchos mensajes. Por otra parte, es el algoritmo más rápido.

Tolerancia a fallos: multicast puede adaptase para tolerar el fallo de un host.

Escalabilidad: una arquitectura multicast podría escalar horizontalmente añadiendo nuevos host al sistema.

Servidor central

El algoritmo centralizado se cimienta en un servidor central que permite el acceso a recursos compartidos accesibles por otros host. Cuando un cliente quiere usar un recurso, pide permiso al servidor central y este le otorga un testigo. Cuando el cliente ha terminado de usar el recurso, devuelve el testigo al servidor central y este queda disponible para que lo use otro cliente.

Si otro cliente pide el testigo al servidor central mientras el cliente anterior aun lo está usando, el servidor central lo pondrá en cola y le entregará el testigo en el momento en el que el cliente anterior termine de usarlo.

Tolerancia a fallos: el servidor central puede soportar la caída de un cliente sin riesgo, pero si cae él, cae el sistema entero.

Escalabilidad: en caso de que el servidor central se quede sin recursos, podría escalar verticalmente asignándosele más recursos (RAM, CPU).

Maekawa

El algoritmo Maekawa, cuyo nombre se debe a Mamoru Maekawa - quien observó que para que un proceso entre en la sección crítica no es necesario que todos los procesos de una categoría pareja a la suya le permitan acceso - es análogo al algoritmo de relojes lógicos (multicast), pero reduce el número de mensajes. El procesador elegido es aquel que obtiene la mitad más 1 votos.

Tolerancia a fallos: Maekawa tolera que algunos host caigan siempre que no estén en un conjunto de votantes requeridos.

Escalabilidad: puede escalar horizontalmente añadiendo más host al sistema.

Anillo

El algoritmo de anillo realiza la comunicación en una única dirección. De este modo, cada proceso solo se puede comunicar con el siguiente. De esta manera, cada proceso sabe cuál es su turno para ejecutarse.

Como cada proceso tiene un turno preestablecido, se hace un uso elevado de ancho de banda, además de penalizar el tiempo de acceso a los recursos.

Tolerancia a fallos: si el proceso falla, se bloquea el anillo y cae el sistema.

Escalabilidad: se pueden añadir más host al anillo (escalabilidad horizontal).

El problema de la exclusión mutua es que cada sección crítica está asociada con una sesión. Las secciones críticas que pertenecen a la misma sesión se pueden ejecutar simultáneamente, mientras que las secciones críticas que pertenecen a diferentes sesiones deben ejecutarse en serie.

Todos los algoritmos existentes que tratan de resolver este problema tienen una alta complejidad espacial de Ω (n) o una alta complejidad de paso de Ω (n) donde n denota el número de procesos en el sistema.

Para solucionar el problema de escalabilidad en la exclusión mutua, podemos usar el algoritmo GME (*Group Mutual Exclusion*), inspirado en la construcción universal de Herlihy para derivar una implementación de un objeto concurrente que se pueda realizar en cola a partir de su especificación secuencial utilizando objetos de consenso.

El estado del objeto concurrente se representa utilizando (i) su estado inicial y (ii) la secuencia de operaciones que se han aplicado al objeto hasta el momento. Los dos se mantienen utilizando una lista enlazada individual en la que el primer nodo representa el estado inicial y los nodos restantes representan las operaciones. Para realizar una operación, un proceso primero crea un nuevo nodo y lo inicializa con todos los detalles relevantes de la operación (tipo, argumentos de entrada, etc.). Luego intenta agregar el nodo al final de la lista. Para gestionar los conflictos en caso de que varios procesos intenten añadir su propio nodo a la lista, se utiliza un objeto de consenso para determinar cuál de estos varios nodos se elige para añadir a la lista.

Específicamente, cada nodo almacena un objeto consensuado y el objeto consensuado del último nodo actual se usa para decidir su sucesor (es decir, la siguiente operación que se aplicará al objeto). Un proceso cuyo nodo no está seleccionado simplemente vuelve a intentarlo.

Fuente:

https://drops.dagstuhl.de/opus/volltexte/2018/9838/pdf/LIPIcs-DISC-2018-49.pdf

4. Compara los siguientes conceptos:

a) Transacciones en bases de datos: ACID y BASE

ACID y BASE son dos modelos diferenciados de transacción de base de datos.

Las siglas ACID significan *Atomicity* (atomicidad), *Consistency* (consistencia, *Isolation* (aislamiento), *Durability* (durabilidad).

Atomicidad significa que se realizan todas las tareas dentro de una misma transacción o no se realiza ninguna de ellas. Dicho de otro modo, si un elemento falla, la transacción entera falla.

Consistencia significa que la base de datos debe permanecer siempre en un estado consistente, tanto al principio como al final de una transacción.

Aislamiento quiere decir que ninguna transacción tiene acceso a otra transacción. Cada transacción es independiente.

Durabilidad significa que una vez se haya completado una transacción, esta persistirá como completa y sobrevivirá a cualquier problema del sistema.

Hay que tener presente que mientras se accede a una base de datos y se modifica un registro, ACID bloquea esa parte de la base de datos, con lo que consigue coherencia en la escritura así como consistencia en la base de datos.

Un ejemplo de uso de ACID lo encontramos en los sistemas bancarios, donde se utiliza casi exclusivamente bases de datos con modelo ACID. Esto ocurre porque las transferencias de dinero dependen de la atomicidad de ACID. Si una transacción queda interrumpida y no se elimina inmediatamente puede causar serios problemas a los balances bancarios, como retirarse dinero de una cuenta y nunca depositarse en la cuenta de destino.

Por otro lado, BASE está ligado a las bases de datos NoSQL y ofrece:

Básicamente disponible: el sistema garantiza la disponibilidad de los datos. Las bases de datos replican los datos en todos los nodos del clúster de BBDD, con lo que se consigue una alta disponibilidad de los mismos.

Estado blando: BASE delega la responsabilidad de la consistencia al equipo de desarrollo, de modo que se podría dar inconsistencia de datos.

Consistencia eventual: una vez la base de datos deja de recibir información, el sistema se vuelve finalmente consistente. Una vez los datos son consistentes, estos se propagan a todas las otras bases de datos y se continúa recibiendo entradas de datos (escritura a la base de datos).

En definitiva, el modelo BASE nos ofrece una mayor rapidez en la entrada de datos a una BBDD, ya que no se efectúan bloqueos de escritura. Por este motivo, BASE es elegido por los proyectos que requieren un mayor volumen de

introducción de datos en un corto espacio de tiempo. ACID, por su parte, es elegido para aquellos proyectos que requieren un especial énfasis en la consistencia y coherencia de los datos.

b) Replicación activa y replicación pasiva

En la replicación activa de tolerancia a fallos (ARM o *active replication model*), las peticiones se envían a todas las réplicas. Los procesos frontales envían mensajes *multicast* a todas ellas y cada gestor los procesa de forma independiente, pero de igual forma que el resto.

Un sistema de replicación activa puede soportar la caída de alguna réplica y continuar trabajando con el resto. Además, la replicación activa permite reducir la latencia al permitir el acceso a varias réplicas. Como contrapartida, el hecho de que las actualizaciones se procesen en todas las réplicas conlleva un alto coste de tiempo y recursos.

En la replicación pasiva de tolerancia a fallos (PRM o *passive replication model*) se usa un nodo primario al que el cliente envía la petición y un conjunto de nodos secundarios a los que el nodo primario envía la actualización de manera síncrona. Cuando los nodos secundarios han actualizado su copia, envían un mensaje de ACK al nodo primario. En la replicación pasiva, los nodos secundarios no manejan concurrencia; esta es gestionada por el nodo primario. Si este falla, un nodo secundario pasa a ser gestor primario.

La replicación activa es la opción preferible cuando se trata de sistemas distribuidos que requieren una respuesta rápida y que deban manejar fallos de tipo bizantino, pero por contra, no alcanza la linealizabilidad *(linearizability)* que se obtiene mediante la replicación activa.

c) Control de concurrencia: Wikipedia y Google Docs

Wikipedia permite la edición concurrente de artículos mediante un control de concurrencia optimista ya que permite a varios usuarios editar un mismo artículo sin bloquearlo.

No obstante, si más de un usuario edita el mismo artículo al mismo tiempo, Wikipedia sólo acepta los cambios introducidos por el usuario que ha empezado antes la edición. El siguiente usuario que guarde cambios, verá un mensaje titulado “conflicto de edición” y se le propondrá una fusión entre su edición y los otros cambios realizados en el mismo periodo de tiempo.

Google Docs ofrece edición de documentos *on-line* en tiempo real por parte de varios usuarios. Usa un control de concurrencia optimista porque permite a varios usuarios editar el mismo fichero sin bloquearlo. Si se producen cambios en un mismo fichero por parte de más de un usuario al mismo tiempo, solo se guarda la última edición.

5. Discute el problema de los generales bizantinos en base a la información presentada en Wikipedia:

https://en.wikipedia.org/wiki/Byzantine_fault

Basándote en el contenido del libro de Coulouris, busca y propón al menos una información que podrías agregar a esta página web de Wikipedia para mejorar su contenido real.

El problema de los generales bizantinos (también llamado fallo bizantino) es una condición que se da mayoritariamente en sistemas informáticos distribuidos y se basa en la idea de que los miembros del sistema pueden fallar pero, aunque varios miembros fallen, el sistema pueda llegar a un consenso sobre una decisión planteada.

Este término toma su nombre de una alegoría, la cual reza:

Dos ejércitos, ubicados en distintos lugares, tienen como misión asediar una misma ciudad. Estos ejércitos, liderados por dos generales leales, requieren comunicarse entre sí para ponerse de acuerdo a la hora de realizar el ataque.

Los generales de cada ejército se comunican a través de un mensajero, quien se encarga de llevar cada mensaje (atacar o retirarse) de un ejército a otro.

Uno de los problemas que pueden surgir en este escenario se da por la presencia de generales traidores, los cuales pueden enviar un voto de retirada a los generales que están a favor de la retirada y un voto de ataque al resto. Aquellos que recibieron un voto de retirada se retirarán, mientras que el resto atacará, lo cual puede hacer que los atacantes no logren el éxito.

Otro problema implica a los mensajeros, pues los generales, que están separados físicamente, deben enviar sus votos a través de mensajeros, los cuales pueden no entregar los votos o falsificarlos.

Al trasladar estos hechos a términos informáticos, podemos visualizar la figura del general como un proceso que inicia la petición, podemos ver los mensajes como las peticiones enviadas a otros procesos y al traidor como a un proceso defectuoso, siendo los demás procesos leales o correctos.

La solución al problema pasa por mandar una petición, tras la cual todos los procesos correctos tienen que tomar una decisión con respecto a esta. Lo importante es que todos los procesos se pongan de acuerdo, más que el valor elegido por los participantes.

En 1978 se llevó a cabo un proyecto para averiguar el número mínimo de ordenadores defectuosos, denotado como n, que podían llegar a frustrar el consenso para tomar una decisión en una red de ordenadores.

Se demostró que se necesitan un mínimo de $3 \cdot n + 1$ ordenadores para que eso suceda, por lo que se ideó un protocolo de mensajería $3 \cdot n + 1$ de dos rondas.

Añadir a Wikipedia (1):

Se podría añadir al artículo un apartado con tipos de fallos arbitrarios que puede sufrir el envío de mensajes en una red de sistemas distribuidos:

Class of failure	Affects	Description
Fal-Stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-Omission	Process	A process completes a send operation but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Aribitrary (Byzantine)	Process	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times or commit omissions; a process may stop or take an incorrect step.

Añadir a Wikipedia (2):

También se podría hablar sobre los requerimientos de un problema bizantino:

- Termination: Eventually each correct process sets its decision variable.

- Agreement: The decision value of all correct processes is the same: if p_i & p_j are correct and have entered the decided state: $d_i = d_j (i, j = 1,2, \dots, N)$.

- Integrity: If the commander is correct, then all correct processes decide on the value that the commander proposed.

Añadir a Wikipedia (3):

Imposibilidad de llegar a un consenso en sistemas asíncronos:

There is no guaranteed solution in an asynchronous system to the Byzantine generals problem, to interactive consistency or to totally ordered and reliable multicast. The algorithms assume that message exchanges take place in rounds, and that processes are entitled to time out and assume that a faulty process has not sent them a message within the round, because the maximum delay has been exceeded. Fischer proved that no algorithm can guarantee to reach consensus in an asynchronous system, even with 1 process crash failure.

Fuente: DISTRIBUTED SYSTEMS. Concepts and Design. G. Coulouris.