CAP Twelve Years Later: How “Rules” Have Changed

Brewer, E. (2009) CAP Twelve Years Later: How “Rules” Have Changed. IEEE Computer, Vol.45, No. 2, pp. 23-29

Introducción

- Eric Brewer
- Preliminares:
  - ACID
  - Teorema CAP
  - BASE

Introducción

Eric Brewer:

- BSc en Ingeniería Eléctrica y Ciencias de la Computación por Berkeley
- MSc y PhD. en Ingeniería Eléctrica y Ciencias de la Computación por MIT
- Profesor emérito de Ciencias de la Computación en Berkeley
- Vicepresidente de infraestructura de Google
- Investigación en Sistemas Operativos y Cómputo Distribuido
- Conocido por formular el teorema CAP

Preliminares: ACID

Transacción: Secuencia de operaciones en base de datos

ACID construido por Andreas Reuter y Theo Härder. Constituye propiedades de las transacciones para garantizar validez del dato a pesar de errores, fallas de energía y otros problemas.

A **Atomicidad (A):** La transacción ejecuta completamente o no se ejecuta

C **Consistencia (C):** La base de datos debe ser consistente antes y después de la transacción

I **Aislamiento (I):** Múltiples transacciones pueden ocurrir independientemente sin interferencia

D **Durabilidad (D):** Los cambios de una transacción exitosa ocurren incluso si con falla en sistema

Preliminares: Teorema CAP

El **Teorema CAP** enuncia que es imposible para un almacenamiento de datos distribuido proveer simultáneamente más de dos de los siguientes tres garantías:

C

Consistencia: Todos los usuarios ven los mismos datos todo el tiempo

A

Disponibilidad: El sistema continúa funcionando aún con fallas en nodos

p

Tolerancia a Particiónes: Sistema continúa funcionando aún si la comunicación entre nodos falla

Cuando hay falla en la red, entonces debemos decidir:

- Cancelar la operación y decrecer la A para asegurar C
- Proceder con la operación y entonces garantizar A arriesgando C

Notar que la C y A de CAP es distinta a la de ACID

Ejemplo:

Dos nodos en lados opuestos de una partición:

- Permitir que un nodo actualice su estado lleva a una inconsistencia -> **No C**
- Si se desea consistencia, entonces una parte de la partición debe mostrarse no disponible -> **No A**
- Solo cuando los nodos se comunican, permite A y C, pero implica -> **No P**

Preliminares: BASE

En química BASE es lo contrario a ACID, que ayuda a recordar el acrónimo.

B

A

Básicamente Disponible: Las operaciones de lectura y escritura básica son disponibles lo más posible (usando todos los nodos del cluster), pero ninguna consistencia es garantizada.

S

Estado Suave (Soft State): Sin la garantía de consistencia, después de un tiempo solo podemos tener alguna probabilidad de conocer el estado, ya que aún no ha convergido

E

Consistencia Eventual: SI el sistema sigue funcionando y esperamos el tiempo suficiente después de un conjunto de entrada, eventualmente podremos conocer el estado de la base de datos, por lo que cualquier lectura posterior será consistente

Planteamiento del problema

- Formulación 2 de 3
- ACID vs BASE
- CAP - latency connection

Formulación 2 de 3

El teorema CAP enuncia que cualquier sistema de datos compartidos en red pueden tener dos de tres propiedades deseables: Consistencia, Disponibilidad y Tolerancia a Particiones.

Esta formulación es engañosa, ya que simplifica la relación entre C-A-P (más detalles adelante)

Para esto se propone un “CAP moderno” que:

- Maximice las combinaciones de C y A (no solo de ellas)
- Incorpore un plan para operar en caso de Partición
- Incorporar un plan para recuperación
- Considere las necesidades de la aplicación

De esta manera se busca tener no solo 2 de 3 si no el continuo entre C-A-P

¿Por qué es engañosa la formulación 2 de 3?

La mayor parte del tiempo no hay particiones, por que podemos tener A y C la mayor parte del tiempo.

- Las particiones no son comunes: Hay pocas razones para perder A o C cuando el sistema no está particionado.
- La selección entre C y A puede ocurrir diferente en el mismo sistema:
  - La decisión puede cambiar acorde a la operación, usuario o datos.
- Las propiedades son más continuas que binarias:
  - Hay varios niveles de A y C
  - Las particiones tienen matices, incluyendo el decidir cuando existen.

1)
Detectar Particiones

2)
Iniciar un modo partición que limite ciertas operaciones

3)
Iniciar el proceso para recuperar C y compensar los errores consecuencia de la partición

ACID vs BASE

ACID y BASE representan dos filosofías opuestas en el espectro de C-A

- **ACID:** Se centra en C y el acercamiento tradicional de las BD.
- **BASE:** Se centra en la A y en realizar explícita la elección y el espectro.
  - Estado Suave y Consistencia Eventual trabajan bien en presencia de particiones
  - Como consecuencia se promueve A

CAP vs ACID

Elegir la disponibilidad solo afecta alguna de las garantias de ACID:

Atomicidad: No afecta

Consistencia: No puede ser mantenida entre particiones. Se tiene que restaurar cuando exista el proceso de recuperación.

Aislamiento: Si se requiere aislamiento, sólo puede operar durante un lado de la partición. Requiere comunicación

Durabilidad: No afecta

CAP - latency connection

CAP ignora latencia, aunque en practica latencia y particiones están relacionadas.

Durante un timeout, el programa debe tomar la decisión de decidir la partición:

1. Cancelar la operación y decrecer A
2. Proceder con la operación y arriesgar C

Reintentar la comunicación para alcanzar consistencias (Paxos, TPC), solo retrasa la decisión. En algún punto el programa debe tomar la decisión de escoger C o A

Es importante que:

- No existe un concepto global de particiones, ya que unos los detectan otros no
- Los nodos pueden detectar la partición y entrar en “modo partición” para optimizar C o A

Como consecuencia, se puede decidir la partición adecuando el tiempo de acuerdo a la red.

¿Qué hacer cuando ocurren particiones?

- Manejando Particiones
- ¿Qué operaciones deben proceder?
- Compensando por los errores

Manejando particiones

El reto es mitigar los efectos en C y A durante una P. La idea es manejar las particiones explícitamente, incluyendo un proceso de recuperación y un plan para todos los invariantes que ahora no se cumplen debido a la partición:

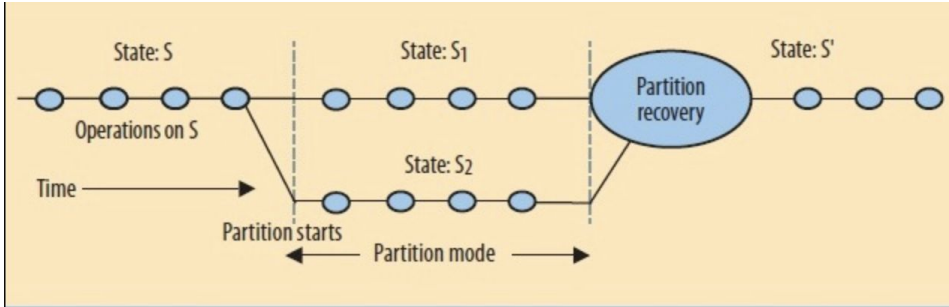

Figure 1. The state starts out consistent and remains so until a partition starts. To stay available, both sides enter partition mode and continue to execute operations, creating concurrent states S_1 and S_2, which are inconsistent. When the partition ends, the truth becomes clear and partition recovery starts. During recovery, the system merges S_1 and S_2 into a consistent state S' and also compensates for any mistakes made during the partition.

¿Qué operaciones deben proceder?

Problema: Determinar qué invariantes se deben mantener en caso de partición.

Ejemplo de invariante que NO se mantiene: Llaves únicas en base de datos.

Se permiten duplicados de llaves durante una partición, ya que es fácil identificar las llaves duplicadas durante el proceso de recuperación, por lo que la invariante puede ser fácilmente restaurada.

Ejemplo de invariante que SÍ se mantiene: Cargo a una tarjeta bancaria.

Para que la invariante no sea modificada durante la partición, se deben prohibir o modificar aquellas operaciones que puedan corromperla. La operación de validación de un cargo a una tarjeta bancaria se suspende hasta que se haya restablecido el sistema.

Comunicar al usuario que tareas están en progreso pero no completas

Recuperación de la partición

Se sabe que después de una partición, algunas operaciones fueron suspendidas y algunas invariantes se han corrompido, por lo que es posible recuperar la consistencia del sistema.

Para recuperar la consistencia se pueden tomar varios enfoques:

- Un enfoque es volver a ejecutar el conjunto de operaciones en orden determinista de manera que todos los nodos alcancen el mismo estado. (vector : nodo-tiempo lógico)
- Determinar ciertas operaciones que podrán ser ejecutadas durante la partición y después de ésta se hará una unión de los estados.
- Uso de operaciones conmutativas o estructuras de datos replicadas conmutativamente.

Bayou storage system

Offline

amazon

Carrito de compra

Compensando por los errores

Ejemplo: ATM

Tres operaciones:

- Depósito
- Retiro
- Consulta de saldo

Invariante principal: **el balance de la cuenta debe ser mayor o igual a cero.**

El retiro es la única operación que podría violar la invariante, por lo que se podría decidir suspender esta operación en caso de una partición, pero esta decisión comprometería la disponibilidad del sistema.

Solución:

Limitar el monto de los retiros a una cantidad k. Para casos en que el balance resultó menor a cero: se notifica el cargo de una cuota o se recalcula el saldo en el próximo depósito.

Compensando por los errores

Las invariantes que han sido corrompidas típicamente se detectan durante la etapa de recuperación y deben ser restauradas.

Existen diferentes maneras de recuperar las invariantes:

- Soluciones triviales como “last writer wins”
- Enfoques de solución más interesantes que implican la unión de operaciones.

Ejemplo: Airplane booking

Recuperación de errores externalizados: **drunk dialing scenario**