CAP dodici anni dopo:

Come sono state stabilite le "regole"

Cambiato

Eric Birraio

34-43 minuti

Questo articolo è apparso per la prima volta su Computer rivista e vi è offerto da InfoQ e IEEE Computer Society.

Il teorema CAP afferma che qualsiasi sistema di dati condivisi in rete può avere solo due delle tre proprietà desiderabili. Tuttavia, gestendo esplicitamente le partizioni, i progettisti possono ottimizzare la coerenza e la disponibilità, ottenendo così un compromesso tra tutte e tre.

Nel decennio trascorso dalla sua introduzione, i progettisti

e i ricercatori hanno utilizzato (e talvolta abusato) il teorema CAP come motivo per esplorare un'ampia varietà di nuovi sistemi distribuiti.

Anche il movimento NoSQL lo ha applicato come argomentazione contro i database tradizionali.

Contenuto sponsorizzato correlato

Il teorema CAP afferma che qualsiasi sistema di dati condivisi in rete può avere al massimo due delle tre proprietà desiderabili:

- coerenza (C) equivalente ad avere una singola copia aggiornata dei dati;
- elevata disponibilità (A) di tali dati (per gli aggiornamenti); e
- tolleranza alle partizioni di rete (P).

Questa espressione di CAP ha raggiunto il suo scopo, che era quello di aprire le menti dei progettisti a una gamma più ampia di sistemi e compromessi; infatti, nell'ultimo decennio è emersa una vasta gamma di nuovi sistemi, nonché un ampio dibattito sui meriti relativi di coerenza e disponibilità.

La formulazione "2 su 3" è sempre stata fuorviante

perché tendeva a semplificare eccessivamente le tensioni tra le proprietà. Ora, tali sfumature contano. CAP proibisce solo una piccola parte dello spazio di progettazione: perfetta disponibilità e coerenza in presenza di partizioni, che sono rare.

Sebbene i progettisti debbano ancora scegliere tra coerenza e disponibilità quando sono presenti partizioni, esiste un'incredibile gamma di flessibilità per la gestione delle partizioni e il ripristino da esse. L'obiettivo CAP moderno dovrebbe essere quello di massimizzare le combinazioni di coerenza e disponibilità che hanno senso per l'applicazione specifica. Tale approccio incorpora piani per il funzionamento durante una partizione e per il ripristino successivo, aiutando così i progettisti a pensare a CAP oltre i suoi limiti storicamente percepiti.

Perché "2 su 3" è fuorviante

Il modo più semplice per comprendere la CAP è pensare a due nodi su lati opposti di una partizione. Consentire ad almeno un nodo di aggiornare lo stato causerà l'incoerenza dei nodi, quindi

rinunciando a C. Allo stesso modo, se la scelta è quella di preservare la coerenza, un lato della partizione deve comportarsi come se non fosse disponibile, rinunciando quindi ad A. Solo quando i nodi comunicano è possibile preservare sia la coerenza che la disponibilità, rinunciando così a P. La convinzione generale è che per i sistemi ad area estesa, i progettisti non possano rinunciare a P e quindi hanno una scelta difficile tra

C e A. In un certo senso, il movimento NoSQL riguarda la creazione di scelte che si concentrano prima sulla disponibilità e poi sulla coerenza; i database che aderiscono alle proprietà ACID (atomicità, coerenza, isolamento e durata) fanno l'opposto. La barra laterale "ACID, BASE e CAP" spiega questa differenza in modo più dettagliato.

In effetti, questa discussione esatta ha portato al teorema CAP. A metà degli anni '90, i miei colleghi e io stavamo costruendo una varietà di sistemi wide-area basati su cluster (essenzialmente i primi cloud computing), tra cui motori di ricerca, proxy cache e sistemi di distribuzione dei contenuti.1 A causa sia degli obiettivi di fatturato che delle specifiche contrattuali, la disponibilità del sistema era a un

premium, quindi ci siamo ritrovati a scegliere regolarmente di ottimizzare la disponibilità tramite strategie come l'impiego di cache o la registrazione di aggiornamenti per una successiva riconciliazione. Sebbene queste strategie abbiano effettivamente aumentato la disponibilità, il guadagno è avvenuto a scapito di una minore coerenz

La prima versione di questo argomento coerenza-disponibilità è stata pubblicata con il titolo ACID contro BASE,2 che all'epoca non è stata ben accolta, principalmente perché le persone amano le proprietà ACID ed sono restie a rinunciarvi.

Lo scopo del teorema CAP era giustificare la necessità di esplorare uno spazio di progettazione più ampio, da cui la formulazione "2 su 3". Il teorema apparve per la prima volta nell'autunno del 1998. Fu pubblicato nel 19993 e nel discorso principale al Simposio del 2000 sui principi del calcolo distribuito4, che portò alla sua dimostrazione.

Come spiega la barra laterale "CAP Confusion", la visione "2 su 3" è fuorviante su più fronti. Innanzitutto, poiché le partizioni sono rare, c'è poca ragione di rinunciare a C o A quando il sistema non è partizionato. In secondo luogo, la scelta tra C e

A può verificarsi molte volte all'interno dello stesso sistema a una granularità molto fine; non solo i sottosistemi possono fare scelte diverse, ma la scelta può cambiare in base all'operazione o persino ai dati specifici o all'utente coinvolto. Infine, tutte e tre le proprietà sono più continue del binario. La disponibilità è ovviamente continua dallo 0 al 100 percento, ma ci sono anche molti livelli di coerenza e persino le partizioni hanno sfumature, incluso il disaccordo all'interno del sistema sull'esistenza o meno di una partizione.

Esplorare queste sfumature richiede di spingere il modo tradizionale di gestire le partizioni, che è la sfida fondamentale. Poiché le partizioni sono rare, CAP dovrebbe consentire C e A perfetti la maggior parte delle volte, ma quando le partizioni sono presenti o percepite, è necessaria una strategia che rilevi le partizioni e ne tenga conto in modo esplicito. Questa strategia dovrebbe avere tre passaggi: rilevare le partizioni, immettere una modalità di partizione esplicita che possa limitare alcune operazioni e avviare un processo di ripristino per ripristinare la coerenza e compensare gli errori commessi durante una par

Acido, base e tappo

ACID e BASE rappresentano due filosofie di progettazione agli estremi opposti dello spettro di coerenza-disponibilità. Le proprietà ACID si concentrano sulla coerenza e sono l'approccio tradizionale dei database. I miei colleghi e io abbiamo creato BASE alla fine degli anni '90 per catturare gli approcci di progettazione emergenti per l'alta disponibilità e per rendere espliciti sia la scelta che lo spettro. I moderni sistemi wide-area su larga scala, incluso il cloud, utilizzano un mix di entrambi gli approcci.

Sebbene entrambi i termini siano più mnemonici che precisi, l'acronimo BASE (il secondo) è un po' più scomodo: Basically Available, Soft state, Eventually consistent. Soft state ed eventual consistency sono tecniche che funzionano bene in presenza di partizioni e quindi promuovono la disponibilità.

La relazione tra CAP e ACID è più complessa e spesso fraintesa, in parte perché la C e la A in ACID rappresentano

concetti diversi rispetto alle stesse lettere in CAP e in parte perché la scelta della disponibilità influisce solo su alcune delle garanzie ACID. Le quattro proprietà ACID sono:

Atomicità (A). Tutti i sistemi traggono vantaggio dalle operazioni atomiche. Quando l'attenzione è rivolta alla disponibilità, entrambi i lati di una partizione dovrebbero comunque utilizzare operazioni atomiche. Inoltre, le operazioni atomiche di livello superiore (il tipo che ACID implica) in realtà semplificano il ripristino.

Coerenza (C). In ACID, la C significa che una transazione conserva tutte le regole del database, come le chiavi univoche. Al contrario, la C in CAP si riferisce solo alla coerenza a copia singola, un sottoinsieme rigoroso della coerenza ACID. La coerenza ACID non può essere mantenuta anche tra

partitions.partition recovery dovrà ripristinare la coerenza ACID. Più in generale, mantenere le invarianti durante le partizioni potrebbe essere impossibile, quindi è necessario riflettere attentamente su quali operazioni non consentire e su come ripristinare le invarianti durante il recovery.

Isolamento (I). L'isolamento è al centro della PAC

teorema: se il sistema richiede l'isolamento ACID, può operare al massimo su un lato durante una partizione. La serializzabilità richiede la comunicazione in generale e quindi fallisce tra le partizioni. Definizioni più deboli di correttezza sono praticabili tra le partizioni tramite compensazione durante il ripristino della partizione.

Durabilità (D). Come per l'atomicità, non c'è motivo di rinunciare alla durabilità, anche se lo sviluppatore potrebbe scegliere di evitarne l'utilizzo tramite soft state (nello stile di BASE) a causa del suo costo. Un punto sottile è che, durante il ripristino della partizione, è possibile invertire le operazioni durevoli che hanno violato inconsapevolmente un invariante durante l'operazione. Tuttavia, al momento del ripristino, data una cronologia durevole da entrambe le parti, tali operazioni possono essere rilevate e corrette. In generale, l'esecuzione di transazioni ACID su ciascun lato di una partizione semplifica il ripristino e abilita un framework per compensare le transazioni che possono essere utilizzate per il ripristino da una

Connessione Cap-latenza

Nella sua interpretazione classica, il teorema CAP ignora la latenza, sebbene nella pratica latenza e partizioni siano strettamente correlate. Operativamente, l'essenza del CAP si realizza durante un timeout, un periodo in cui il programma deve prendere una decisione fondamentale:

decisione di partizione:

- annullare l'operazione e quindi ridurre la disponibilità,
- *oppure* procedere con l'operazione e quindi rischiare l'incoerenza.

Ritentare la comunicazione per ottenere coerenza, ad esempio tramite Paxos o un commit in due fasi, ritarda solo la decisione. A un certo punto il programma deve prendere la decisione; ritentare la comunicazione indefinitamente è in sostanza scegliere C invece di A.

Quindi, pragmaticamente, una partizione è un limite temporale per la comunicazione. Non riuscire a raggiungere la coerenza entro il limite temporale implica una partizione e quindi una scelta tra C e A per questa operazione. Questi concetti catturano il problema di progettazione principale per quanto riguarda la latenza: sono due

le parti vanno avanti senza comunicare?

Questa visione pragmatica dà origine a diverse conseguenze importanti. La prima è che non esiste una nozione globale di partizione, poiché alcuni nodi potrebbero rilevarne una e altri no. La seconda conseguenza è che i nodi possono rilevare una partizione ed entrare in *modalità partizione, una* parte centrale dell'ottimizzazione di C e A.

Infine, questa visione significa che i progettisti possono impostare intenzionalmente limiti temporali in base ai tempi di risposta target; i sistemi con limiti più rigidi probabilmente entreranno in modalità di partizione più spesso e in momenti in cui la rete è semplicemente lenta e non effettivamente partizionata.

A volte ha senso rinunciare al C forte per evitare l'elevata latenza del mantenimento della coerenza su un'ampia area. Il sistema PNUTS di Yahoo incorre nell'incoerenza mantenendo copie remote in modo asincrono.5 Tuttavia, rende la copia master locale, il che riduce la latenza. Questa strategia funziona bene nella pratica perché i dati di un singolo utente sono naturalmente partizionati

in base alla posizione (normale) dell'utente.

L'ideale sarebbe che il data master di ogni utente fosse nelle vicinanze.

Facebook usa la strategia opposta:6 la copia master è sempre in una posizione, quindi un utente remoto in genere ha una copia più vicina ma potenzialmente obsoleta. Tuttavia, quando gli utenti aggiornano le loro pagine, l'aggiornamento va direttamente alla copia master, così come tutte le letture dell'utente per un breve periodo, nonostante una latenza più elevata. Dopo 20 secondi, il traffico dell'utente torna alla copia più vicina, che a quel punto dovrebbe riflettere l'aggiornamento.

Confusione sul tappo

Aspetti del teorema CAP sono spesso fraintesi, in particolare l'ambito di disponibilità e coerenza, il che può portare a risultati indesiderati. Se gli utenti non riescono a raggiungere il servizio, non c'è scelta tra C e A, tranne quando parte del servizio viene eseguito sul client. Questa eccezione, comunemente nota come operazione disconnessa o modalità offline,7 sta diventando sempre più importante. Alcuni

Le funzionalità HTML5, in particolare l'archiviazione persistente sul client, semplificano le operazioni disconnesse in futuro. Questi sistemi normalmente scelgono A invece di C e quindi devono recuperare da partizioni lunghe.

L'ambito di coerenza riflette l'idea che, entro un certo limite, lo stato è coerente, ma al di fuori di quel limite tutte le scommesse sono spente. Ad esempio, all'interno di una partizione primaria, è possibile garantire coerenza e disponibilità complete, mentre al di fuori della partizione, il servizio non è disponibile. Paxos e multicast atomico

I sistemi solitamente corrispondono a questo scenario.8 In Google, la partizione primaria risiede solitamente all'interno di un data center; tuttavia, Paxos viene utilizzato su un'area estesa per garantire un consenso globale, come in Chubby,9 e un archivio durevole ad alta disponibilità, come in Megastore.10

Sottoinsiemi indipendenti e autoconsistenti possono progredire mentre sono partizionati, sebbene non sia possibile garantire invarianti globali. Ad esempio, con lo sharding, in cui i progettisti pre-partizionano i dati tra i nodi, è altamente probabile

che ogni frammento può fare qualche progresso durante una partizione. Al contrario, se lo stato rilevante è diviso in una partizione o sono necessarie invarianti globali, allora nella migliore delle ipotesi solo un lato può fare progressi e nella peggiore delle ipotesi nessun progresso è possibile.

Ha senso scegliere coerenza e disponibilità (CA) come "2 su 3"? Come alcuni ricercatori sottolineano correttamente, non è chiaro cosa significhi esattamente rinunciare a P.11,12 Un progettista può scegliere di non avere partizioni? Se la scelta è CA, e poi c'è una partizione, la scelta deve tornare a C o A. È meglio pensarci in termini probabilistici: scegliere CA dovrebbe significare che la probabilità di una partizione è molto inferiore a quella di altri guasti sistemici, come disastri o guasti multipli simultanei.

Una tale visione ha senso perché i sistemi reali perdono sia C che A in alcuni set di guasti, quindi tutte e tre le proprietà sono una questione di grado. In pratica, la maggior parte dei gruppi presuppone che un data center (singolo sito) non abbia partizioni al suo interno e quindi progetta per CA all'interno di un singolo sito; tali progetti, inclusi i database tradizionali, sono i pre-

CAP predefinito. Tuttavia, sebbene le partizioni siano meno probabili all'interno di un datacenter, sono effettivamente possibili, il che rende problematico un obiettivo CA. Infine, data l'elevata latenza su un'area estesa, è relativamente comune rinunciare alla perfetta coerenza su un'area estesa in nome di prestazioni migliori.

Un altro aspetto della confusione CAP è il costo nascosto della perdita di coerenza, che è la necessità di conoscere le invarianti del sistema. La sottile bellezza di un sistema coerente è che le invarianti tendono a valere anche quando il progettista non sa cosa sono. Di conseguenza, un'ampia gamma di invarianti ragionevoli funzionerà benissimo. Al contrario, quando i progettisti scelgono A, che richiede il ripristino delle invarianti dopo una partizione, devono essere espliciti su tutte le invarianti, il che è sia impegnativo che soggetto a errori. In sostanza, questo è lo stesso problema di aggiornamenti simultanei che rende il multithreading più difficile della programmazione sequenziale.

Gestione delle partizioni

La sfida per i progettisti è quella di mitigare gli effetti di una partizione su coerenza e disponibilità. L'idea chiave è quella di gestire le partizioni in modo molto esplicito, includendo non solo il rilevamento, ma anche uno specifico processo di ripristino e un piano per tutte le invarianti che potrebbero essere violate durante una partizione. Questo approccio di gestione ha tre fasi:

(Clicca sull'immagine per ingrandirla)

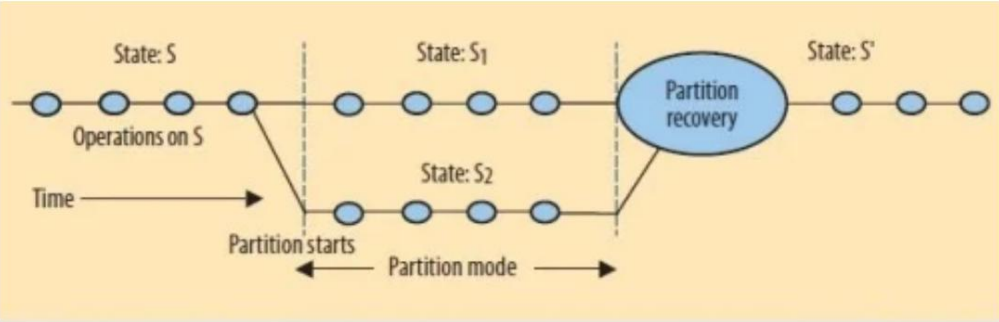

Figure 1. The state starts out consistent and remains so until a partition starts. To stay available, both sides enter partition mode and continue to execute operations, creating concurrent states S_1 and S_2, which are inconsistent. When the partition ends, the truth becomes clear and partition recovery 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.

- rilevare l'inizio di una partizione,
- immettere una modalità di partizione esplicita che potrebbe limitare alcune operazioni e
- avviare il ripristino della partizione quando la comunicazione viene ripristinata.

L'ultimo passaggio mira a ripristinare la coerenza e compensare gli errori commessi dal programma

mentre il sistema era partizionato.

La figura 1 mostra l'evoluzione di una partizione. Il normale funzionamento è una sequenza di operazioni atomiche, e quindi le partizioni iniziano sempre tra le operazioni. Una volta che il sistema scade, rileva una partizione e il lato che rileva entra in modalità partizione. Se una partizione esiste effettivamente, entrambi i lati entrano in questa modalità, ma sono possibili partizioni unilaterali. In tali casi, l'altro lato comunica come necessario e

o questa parte risponde correttamente o non è stata richiesta alcuna comunicazione; in entrambi i casi, le operazioni rimangono coerenti. Tuttavia, poiché la parte di rilevamento potrebbe avere operazioni incoerenti, deve entrare in modalità partizione. I sistemi che utilizzano un quorum sono un esempio di questa partizione unilaterale. Una parte avrà un quorum e potrà procedere, ma l'altra no. I sistemi che supportano operazioni disconnesse hanno chiaramente una nozione di modalità partizione, così come alcuni sistemi multicast atomici, come JGroups di Java.

Una volta che il sistema entra in modalità partizione, due

sono possibili strategie. La prima è quella di limitare alcune operazioni, riducendo così la disponibilità. La seconda è quella di registrare informazioni extra circa operazioni che saranno utili durante il ripristino della partizione. Continuare a tentare la comunicazione consentirà al sistema di discernere quando la partizione termina.

Quali operazioni bisogna effettuare?

La decisione su quali operazioni limitare dipende principalmente dalle invarianti che il sistema deve mantenere. Dato un set di invarianti, il progettista deve decidere se mantenere o meno una particolare invariante durante la modalità di partizione o rischiare di violarla con l'intento di ripristinarla durante il ripristino. Ad esempio, per l'invariante che le chiavi in una tabella sono univoche, i progettisti in genere decidono di rischiare quell'invariante e consentire chiavi duplicate durante una partizione.

Le chiavi duplicate sono facili da rilevare durante il ripristino e, supponendo che possano essere unite, il progettista può facilmente ripristinare la

invariante.

Per un invariante che deve essere mantenuto durante una partizione, tuttavia, il progettista deve proibire o modificare le operazioni che potrebbero violarla. (In generale, non c'è modo di sapere se l'operazione violerà effettivamente l'invariante, poiché lo stato dell'altro lato non è conoscibile.) Gli eventi esternalizzati, come l'addebito su una carta di credito, spesso funzionano in questo modo. In questo caso, la strategia è quella di registrare l'intento ed eseguirlo dopo il recupero. Tali transazioni sono in genere parte di un flusso di lavoro più ampio che ha uno stato di elaborazione degli ordini esplicito e ci sono pochi svantaggi nel ritardare l'operazione fino alla fine della partizione. Il progettista rinuncia ad A in un modo che gli utenti non vedono. Gli utenti sanno solo di aver effettuato un ordine e che il sistema lo eseguirà in seguito.

Più in generale, la modalità di partizione dà origine a una sfida fondamentale per l'interfaccia utente, che consiste nel comunicare che le attività sono in corso ma non completate. I ricercatori hanno esplorato questo problema in dettaglio per le

operazione, che è semplicemente una lunga partizione. L'applicazione del calendario di Bayou, ad esempio, mostra dati potenzialmente incoerenti (provvisori) voci in un colore diverso.13 Tali notifiche sono regolarmente visibili sia nelle applicazioni di flusso di lavoro, come il commercio con notifiche e-mail, sia nei servizi cloud con modalità offline, come Google Docs.

Uno dei motivi per cui è opportuno concentrarsi sulle operazioni atomiche esplicite, anziché solo sulle letture e sulle scritture, è che è molto più facile analizzare l'impatto delle operazioni di livello superiore sugli invarianti. In sostanza, il progettista deve creare una tabella che esamina il prodotto vettoriale di tutte le operazioni e di tutti gli invarianti e decidere per ogni voce se tale operazione potrebbe violare l'invariante. In tal caso, il progettista deve decidere se proibire, ritardare o modificare l'operazione. In pratica, queste decisioni possono anche dipendere dallo stato noto, dagli argomenti o da entrambi. Ad esempio, nei sistemi con un nodo home per determinati dati, 5 operazioni possono in genere procedere sul nodo home ma non su altri nodi.

Il modo migliore per tracciare la cronologia delle operazioni su entrambi i lati è usare i vettori di versione, che catturano le dipendenze causali tra le operazioni. Gli elementi del vettore sono una coppia (nodo, tempo logico), con una voce per ogni nodo che ha aggiornato l'oggetto e il tempo del suo ultimo aggiornamento. Date due versioni di un oggetto, A e B, A è più recente di B se, per ogni nodo in comune nei loro vettori, i tempi di A sono maggiori o uguali a quelli di B e almeno uno dei tempi di A è maggiore.

Se è impossibile ordinare i vettori, allora gli aggiornamenti sono stati simultanei e probabilmente incoerenti. Quindi, data la cronologia dei vettori di versione di entrambe le parti, il sistema può facilmente dire quali operazioni sono già in un ordine noto e quali eseguite contemporaneamente.

Lavori recenti14 hanno dimostrato che questo tipo di coerenza causale è il miglior risultato possibile in generale se il progettista sceglie di concentrarsi sulla disponibilità.

Recupero partizione

A un certo punto, la comunicazione riprende e la partizione termina. Durante la partizione, ogni lato era disponibile e quindi stava facendo progressi, ma la partizione ha ritardato alcune operazioni e violato alcune invarianti. A questo punto, il sistema conosce lo stato e la cronologia di entrambi i lati perché ha tenuto un registro accurato durante la modalità di partizione. Lo stato è meno utile della cronologia, da cui il sistema può dedurre quali operazioni hanno effettivamente violato le invarianti e quali risultati sono stati esternalizzati, incluse le risposte inviate all'utente. Il progettista deve risolvere due problemi difficili durante il ripristino:

- lo stato su entrambi i lati deve diventare coerente, e
- deve essere prevista una compensazione per gli errori commessi durante la modalità di partizione.

In genere è più facile correggere lo stato attuale partendo dallo stato al momento della partizione e riportando in avanti entrambi i set di operazioni in qualche modo, mantenendo uno stato coerente lungo il percorso. Bayou lo ha fatto esplicitamente riportando indietro il database a un

tempo corretto e riproducendo l'intero set di operazioni in un ordine deterministico ben definito in modo che tutti i nodi raggiungano lo stesso

stato.15 Allo stesso modo, i sistemi di controllo del codice sorgente come il Concurrent Versioning System (CVS) partono da un punto coerente condiviso e proseguono gli aggiornamenti per unire i rami.

La maggior parte dei sistemi non riesce sempre a risolvere i conflitti. Ad esempio, CVS occasionalmente presenta conflitti che l'utente deve risolvere manualmente e i sistemi wiki con modalità offline in genere lasciano conflitti nel documento risultante che richiedono una modifica manuale.16 Al contrario, alcuni sistemi possono sempre unire i conflitti

scegliendo determinate operazioni. Un caso emblematico è la modifica del testo in Google Docs,17 che limita le operazioni all'applicazione di uno stile e all'aggiunta o all'eliminazione di testo. Pertanto, sebbene il problema generale della risoluzione dei conflitti non sia risolvibile, in pratica i progettisti possono scegliere di limitare l'uso di determinate operazioni durante il partizionamento in modo che il sistema possa unire automaticamente lo stato durante il ripristino. Ritardare le operazioni rischiose è

un'implementazione relativamente semplice di questa strategia.

L'utilizzo di operazioni commutative è l'approccio più vicino a un framework generale per la convergenza automatica dello stato. Il sistema concatena i log, li ordina in un certo ordine e poi li esegue. La commutatività implica la capacità di riorganizzare le operazioni in un ordine globale coerente preferito. Sfortunatamente, utilizzare solo operazioni commutative è più difficile di quanto sembri; ad esempio, l'addizione è commutativa, ma l'addizione con un controllo dei limiti non lo è (un saldo zero, ad esempio).

Un recente lavoro di Marc Shapiro e colleghi dell'INRIA18,19 ha notevolmente migliorato l'uso delle operazioni commutative per la convergenza dello stato. Il team ha sviluppato tipi di dati replicati commutativi (CRDT), una classe di strutture dati che convergono in modo dimostrabile dopo una partizione e descrive come utilizzarli strutture a

- assicurarsi che tutte le operazioni durante una partizione siano commutative, *o*

- rappresentare i valori su un reticolo e garantire che tutte le operazioni durante una partizione siano monotonicamente crescenti rispetto a tale reticolo.

Quest'ultimo approccio converge lo stato spostandosi al massimo dei valori di ogni lato. È una formalizzazione e un miglioramento di ciò che Amazon fa con il suo carrello della spesa:20 dopo una partizione, il valore convergente è l'unione dei due carrelli, con l'unione che è un'operazione di insieme monotona. La conseguenza di questa scelta è che gli elementi eliminati potrebbero riapparire.

Tuttavia, i CRDT possono anche implementare set tolleranti alle partizioni che aggiungono ed eliminano elementi. L'essenza di questo approccio è quella di mantenere due set: uno per gli elementi aggiunti e uno per quelli eliminati, con la differenza che rappresenta l'appartenenza al set. Ogni set semplificato converge, e così fa anche la differenza. A un certo punto, il sistema può ripulire le cose semplicemente rimuovendo gli elementi eliminati da entrambi i set. Tuttavia, tale pulizia è generalmente possibile solo finché il sistema non è partizionato. In altre parole, il progettista deve proibire o posticipare alcune operazioni

durante una partizione, ma queste sono operazioni di pulizia che non limitano la disponibilità percepita. Quindi, implementando lo stato tramite CRDT, un progettista può scegliere A e comunque garantire che lo stato converga automaticamente dopo una partizione.

Compensare gli errori

Oltre al calcolo dello stato post-partizione, c'è il problema un po' più arduo di correggere gli errori commessi durante il partizionamento. Il tracciamento e la limitazione delle operazioni in modalità partizione garantiscono la conoscenza di quali invarianti potrebbero essere stati violati, il che a sua volta consente al progettista di creare una strategia di ripristino per ciascuna di tali invarianti. In genere, il sistema scopre la violazione durante il ripristino e deve implementare qualsiasi correzione in quel momento.

Esistono vari modi per risolvere gli invarianti, tra cui modi banali come "l'ultimo scrittore vince" (che ignora alcuni aggiornamenti), approcci più intelligenti che uniscono le operazioni e l'escalation umana. Un esempio di quest'ultimo è l'aereo

overbooking: salire a bordo dell'aereo è in un certo senso un recupero di partizione con l'invariante che ci devono essere almeno tanti posti quanti sono i passeggeri. Se ci sono troppi passeggeri, alcuni perderanno i loro posti, e idealmente il servizio clienti risarcirà quei passeggeri in

in qualche modo.

Anche l'esempio dell'aereo mostra un errore esternalizzato: se la compagnia aerea non avesse detto che il passeggero avesse un posto, risolvere il problema sarebbe stato molto più facile. Questo è un altro motivo per ritardare le operazioni rischiose: al momento del recupero, la verità è nota. L'idea di compensazione è davvero al centro della correzione di tali errori; i progettisti devono creare operazioni di compensazione che ripristinino un invariante e più ampiamente correggano un errore esternalizzato.

Tecnicamente, i CRDT consentono solo invarianti verificabili localmente, una limitazione che rende compensazione non necessaria ma che in un certo senso riduce la potenza dell'approccio. Tuttavia, una soluzione che utilizza CRDT per la convergenza di stato potrebbe consentire la violazione temporanea di un globale

invariante, converge lo stato dopo la partizione e quindi esegue le compensazioni necessarie.

Recuperare da errori esternalizzati richiede in genere un po' di storia sugli output esternalizzati. Si consideri lo scenario di "chiamata" in stato di ebbrezza, in cui una persona non ricorda di aver fatto varie telefonate mentre era ubriaca la notte precedente. Lo stato di quella persona alla luce del giorno potrebbe essere sano, ma il registro mostra comunque un elenco di chiamate, alcune delle quali potrebbero essere state degli errori. Le chiamate sono gli effetti esterni di

lo stato della persona (ubriachezza). Poiché la persona non è riuscita a ricordare le chiamate, potrebbe essere difficile compensare qualsiasi problema che ha causato.

In un contesto di macchina, un computer potrebbe eseguire ordini due volte durante una partizione. Se il sistema riesce a distinguere due ordini intenzionali da due ordini duplicati, può annullare uno dei duplicati. Se esternalizzata, una strategia di compensazione sarebbe quella di generare automaticamente un'e-mail al cliente spiegando che il sistema ha eseguito accidentalmente l'ordine due volte ma che

l'errore è stato corretto e per allegare un buono sconto per uno sconto sul prossimo ordine. Senza la cronologia corretta, tuttavia, l'onere di individuare l'errore ricade sul cliente.

Alcuni ricercatori hanno formalmente esplorato le transazioni di compensazione come un modo per gestire le transazioni di lunga durata.21,22 Le transazioni di lunga durata affrontano una variazione della decisione di partizione: è meglio mantenere i blocchi per un lungo periodo per garantire la coerenza o rilasciarli in anticipo ed esporre i dati non impegnati ad altre transazioni ma consentire una maggiore concorrenza? Un esempio tipico è il tentativo di aggiornare tutti i record dei dipendenti come una singola transazione. La serializzazione di questa transazione nel modo normale blocca tutti i record e impedisce la concorrenza. Le transazioni di compensazione adottano un approccio diverso, suddividendo la transazione di grandi dimensioni in una saga, che consiste in più sottotransazioni, ciascuna delle quali esegue il commit lungo il percorso. Pertanto, per annullare la transazione più grande, il sistema deve annullare ogni sottotransazione già impegnata emettendo una nuova transazione che corregge i suoi effetti: la transazion

In generale, l'obiettivo è evitare di interrompere altre transazioni che hanno utilizzato dati committati in modo errato (nessun annullamento a cascata). La correttezza di questo approccio non dipende dalla serializzabilità o dall'isolamento, ma piuttosto dall'effetto netto della sequenza di transazioni sullo stato e sugli output. Vale a dire, dopo le compensazioni, il database finisce essenzialmente in un posto equivalente a dove sarebbe stato se le sottotransazioni non fossero mai state eseguite? L'equivalenza deve includere azioni esternalizzate; ad esempio, rimborsare un acquisto duplicato non è la stessa cosa che non addebitare quel cliente in primo luogo, ma è presumibilmente equivalente.

La stessa idea vale per il recupero delle partizioni. Un fornitore di servizi o prodotti non può sempre annullare direttamente gli errori, ma mira ad ammetterli e a intraprendere nuove azioni compensative. Il modo migliore per applicare queste idee al recupero delle partizioni è un problema aperto. La barra laterale "Problemi di compensazione in uno sportello bancomat" descrive alcune delle preoccupazioni in una sola applicazione zona.

I progettisti di sistemi non dovrebbero sacrificare ciecamente

coerenza o disponibilità quando esistono partizioni. Utilizzando l'approccio proposto, possono ottimizzare entrambe le proprietà tramite un'attenta gestione degli invarianti durante le partizioni. Man mano che tecniche più recenti, come i vettori di versione e i CRDT, si spostano in framework che ne semplificano l'uso, questo tipo di ottimizzazione dovrebbe diffondersi maggiormente. Tuttavia, a differenza delle transazioni ACID, questo approccio richiede un'implementazione più ponderata rispetto alle strategie passate e le soluzioni migliori dipenderanno in larga misura dai dettagli sugli invarianti e sulle operazioni del servizio.

Problemi di compensazione in uno sportello bancomat

Nella progettazione di uno sportello bancomat (ATM), la coerenza elevata sembrerebbe essere la scelta logica, ma in pratica A prevale su C.
Il motivo è abbastanza semplice: una maggiore disponibilità significa maggiori entrate.
Indipendentemente da ciò, la progettazione di ATM funge da buon contesto per esaminare alcune delle sfide co

compensando le violazioni invarianti durante una partizione.

Le operazioni essenziali dell'ATM sono deposito, prelievo e controllo del saldo. L'invariante fondamentale è che il saldo deve essere pari o superiore a zero. Poiché solo withdrawal può violare l'invariante, necessiterà di un trattamento speciale, ma le altre due operazioni possono sempre essere eseguite.

Il progettista del sistema ATM potrebbe decidere di vietare i prelievi durante una suddivisione, poiché è impossibile conoscere il saldo effettivo in quel momento, ma ciò ne comprometterebbe la disponibilità. Invece, utilizzando la modalità stand-in (modalità partizione), gli sportelli bancomat moderni limitano il prelievo netto a la maggior parte k, dove k potrebbe essere $200. Al di sotto di questo limite, i prelievi funzionano completamente; quando il saldo raggiunge il limite, il sistema nega i prelievi. Quindi, l'ATM sceglie un limite sofisticato sulla disponibilità che consente i prelievi ma limita il rischio.

Quando la partizione termina, deve esserci un modo per ripristinare la coerenza e compensare gli errori commessi durante la

sistema è stato partizionato. Ripristinare lo stato è facile perché le operazioni sono commutative, ma la compensazione può assumere diverse forme. Un saldo finale inferiore a zero viola l'invariante. Nel

Nel caso normale, il bancomat ha erogato il denaro, il che ha causato l'errore esterno.

La banca compensa addebitando una commissione e aspettandosi il rimborso. Dato che il rischio è limitato, il problema non è grave. Tuttavia, supponiamo che il saldo fosse sotto zero a un certo punto durante la partizione (all'insaputa dell'ATM), ma che un deposito successivo lo abbia riportato su. In questo caso, la banca potrebbe comunque addebitare una commissione di scoperto retroattivamente, oppure potrebbe ignorare la violazione, poiché il cliente ha già effettuato il pagamento necessario.

In generale, a causa dei ritardi di comunicazione, il sistema bancario non dipende dalla coerenza per la correttezza, ma piuttosto dall'audit e dalla compensazione. Un altro esempio di questo è il "check kiting", in cui un cliente preleva denaro da più filiali prima di poter comunicare e poi scappa. Lo scoperto

essere catturato più tardi, forse portando a

risarcimento sotto forma di azione legale.

Ringraziamenti

Ringrazio Mike Dahlin, Hank Korth, Marc Shapiro, Justin Sheehy, Amin Vahdat, Ben Zhao e i volontari dell'IEEE Computer Society per i loro utili commenti su questo lavoro.

Informazioni sull'autore

Eric Brewer è professore di informatica presso l'Università della California, Berkeley, e vicepresidente delle infrastrutture presso Google. I suoi interessi di ricerca includono cloud computing, server scalabili, reti di sensori e tecnologia per le regioni in via di sviluppo. Ha anche contribuito a creare USA.gov, il portale ufficiale del governo federale. Brewer ha conseguito un dottorato di ricerca in ingegneria elettrica e informatica presso il MIT. È membro del National Academy of Engineering. Contattatelo a brewer@cs.berkeley.edu

Computer, la pubblicazione di punta della IEEE Computer Society, pubblica articoli peer-reviewed molto acclamati, scritti per e da professionisti che rappresentano l'intero spettro della tecnologia informatica, dall'hardware al software e dalla ricerca attuale alle nuove applicazioni. Offrendo più sostanza tecnica rispetto alle riviste di settore e più idee pratiche rispetto alle riviste di ricerca.

Computer fornisce informazioni utili applicabili agli ambienti di lavoro quotidiani.

Riferimenti

1. E. Brewer, "Lezioni dai servizi su larga scala", *IEEE Internet Computing,* luglio/agosto 2001, pp. 46-55.
2. A. Fox et al., "Servizi di rete scalabili basati su cluster", Proc. 16th ACM Symp. *Principi dei sistemi operativi* (SOSP 97), ACM, 1997, pp. 78-91.
3. A. Fox e EA Brewer, "Raccolto, resa e

Sistemi scalabili tolleranti", *Atti 7° workshop Hot Topics in Operating Systems* (HotOS 99), IEEE CS, 1999, pp. 174-178.

4. E. Brewer, "Verso sistemi distribuiti robusti", *Proc. 19th Ann. ACM Symp.Principles of Distributed Computing* (PODC 00), ACM, 2000, pp. 7-10; on-line risorsa.

5. B. Cooper et al., "PNUTS: piattaforma di distribuzione dati ospitata di Yahoo!", *Proc. VLDB Endowment* (VLDB 08), ACM, 2008, pp. 1277-1288.

6. J. Sobel, "Scaling Out", *Facebook Engineering Notes,* 20 agosto 2008; on-line risorsa.

7. J. Kistler e M. Satyanarayanan, "Operazione disconnessa nel file system Coda" *ACM Trans. Computer Systems,* febbraio. 1992, pagg. 3-25.

8. K. Birman, Q. Huang e D. Freedman, "Superare la 'D' in CAP: usare Isis2 per creare servizi cloud reattivi a livello locale", *Computer,* febbraio 2011, pp. 50-58.

9. M. Burrows, "Il servizio Chubby Lock per

Sistemi distribuiti debolmente accoppiati", *Proc. Symp. Progettazione e implementazione di sistemi operativi* (OSDI 06), Usenix, 2006, pp. 335-350.

10. J. Baker et al., "Megastore: fornire storage scalabile e altamente disponibile per servizi interattivi", *Atti 5a Conferenza biennale. Ricerca sui sistemi di dati innovativi* (CIDR 11), ACM, 2011, pp. 223-234.

11. D. Abadi, "Problemi con CAP e il poco noto sistema NoSQL di Yahoo", *DBMS Musings,* blog, 23 aprile 2010; risorsa online.

12. C. Hale, "You Can't Sacrifice Partition Tolerance", 7 ottobre 2010; risorsa on-line.

13. WK Edwards et al., "Progettazione e implementazione di applicazioni collaborative asincrone con Bayou", *Proc. 10th Ann. ACM Symp. Software e tecnologia dell'interfaccia utente* (UIST 97), ACM, 1999, pp. 119-128.

14. P. Mahajan, L. Alvisi e M. Dahlin, *Coerenza, disponibilità e convergenza,* rapporto tecnico UTCS TR-11-22, Univ. del Texas ad Austin, 2011.

15. DB Terry et al., "Gestione degli aggiornamenti

Conflitti in Bayou, un sistema di archiviazione replicato debolmente connesso," *Proc. 15th ACM Symp. Principi dei sistemi operativi* (SOSP 95), ACM, 1995, pp. 172-182.

16. B. Du e EA Brewer, "DTWiki: un wiki tollerante alla disconnessione e all'intermittenza", Atti 17a Conferenza internazionale World Wide Web (WWW 08), ACM, 2008, pp. 945-952.

17. Blog "Le novità del nuovo Google Docs: risoluzione dei conflitti".

18. M. Shapiro et al., "Tipi di dati replicati senza conflitti", *Proc. 13a Conferenza internazionale. Stabilizzazione, sicurezza e protezione dei sistemi distribuiti* (SSS 11), ACM, 2011, pp. 386-400.

19. M. Shapiro et al., "Convergent and Commutative Replicated Data Types", *Bollettino dell'EATCS,* n. 104, giugno 2011, pp. 67-88.

20. G. DeCandia et al., "Dynamo: archivio chiave-valore altamente disponibile di Amazon", *Proc. 21st ACM SIGOPS Symp. Principi dei sistemi operativi* (SOSP 07), ACM, 2007, pp. 205-220.

21. H. Garcia-Molina e K. Salem, "SAGAS", *Proc. ACM SIGMOD Int'l Conf. Gestione di*

Dati (SIGMOD 87), ACM, 1987, pp. 249-259.

22. H. Korth, E. Levy e A. Silberschatz, "Un Approccio formale al recupero mediante compensazione Transazioni", *Proc. VLDB Endowment* (VLDB 90), ACM, 1990, pagg. 95-106