mokabyte.it

Il teorema CAP... in Brewer - MokaByte

Luca Vetti Tagliati

21-28 minuti

Con questo articolo, avviamo una serie dedicata al teorema di CAP, noto anche con il nome di Brewer, il quale, in sintesi, sostiene che non è possibile per un sistema software distribuito presentare simultaneamente le tre seguenti garanzie: consistenza (C), continua disponibilità (A) e tolleranza alle partizioni (P). Cominceremo, parlando di OracleCoherence, la cache distribuita, ma tratteremo anche altre implementazioni.

Introduzione

Iniziamo questo articolo con un breve ripasso del Teorema **CAP** (**Consistency**, **Availability** and **Partition tolerance**, vale a dire totale consistenza, continua disponibilità e tolleranza alle partizioni). Questo teorema è anche conosciuto come **teorema di Brewer**, dal nome di colui che per primo lo ha pensato (Eric A. Brewer, [1]). Il teorema CAP, considerato una pietra miliare anche del pensiero **NoSQL**, afferma che "è impossibile per un sistema informatico distribuito fornire simultaneamente tutte e tre le seguenti garanzie: completa coerenza dei dati, continua disponibilità e tolleranza alle partizioni" [2].

CAP è diventato uno strumento chiave sia per comprendere il comportamento di sistemi distribuiti, sia per progettare l'architettura dei sistemi che devono rispondere a requisiti non funzionali molto stringenti, quali elevate prestazioni, deployment geograficamente distribuito e continua disponibilità.

Una volta rivisto il teorema di Brewer, negli articoli successivi si proseguirà utilizzandone le direttive per analizzare le principali scelte architetturali Oracle Coherence: "a continuously available in-memory data grid across a large cluster of computers for real-time data analysis, in-memory grid computations and high-performance processing solution" (data-grid internamente gestita in memoria, sempre disponibile grazie al un elevato cluster di computer, adatta per l'analisi dei dati in tempo reale, calcoli di tipo grid eseguiti in memoria, e soluzioni ad alte prestazioni).

Questa serie di articoli evidenzia come le caratteristiche principali di Oracle Coherence sono la sua capacità di fornire la coerenza dei dati (da qui il suo nome) e l'elevata, quasi continua, disponibilità, mentre la tolleranza alla partizione ne rappresenta il punto debole. Lo scenario più pericoloso, presentato nel prossimo articolo, è il cosiddetto split-brain ("cervello diviso") che, in situazioni limite, può anche causare la perdita di dati.

Il teorema CAP

Il Teorema CAP afferma che, sebbene sia altamente desiderabile per un sistema software distribuito fornire simultaneamente totale coerenza (**Consistency**), continua disponibilità (**Availability**) e tolleranza alle partizioni (**Partition tolerance**), ciò non è possibile e quindi è necessario stabilire, di volta in volta in funzione ai requisiti, quali di queste tre garanzie sacrificare.

Il teorema è stato pensato inizialmente da Eric A. Brewer (da cui il nome di Teorema di Brewer), fondatore di Inktomi e chief scientist di Yahoo!, attualmente docente di Informatica presso UC Berkeley. Brewer presentò per la prima volta il teorema all'ACM Symposium on the Principles of Distributed Computing (PODC, Simposio sui principi della computazione distribuita, 19 luglio 2000, [1]).

Quella che inizialmente venne considerata una congettura, due anni più tardi, nel 2002, divenne a pieno titolo un teorema grazie al lavoro di Seth Gilbert e Nancy Lynch del MIT, i quali riuscirono a dimostrare formalmente che **le congetture di Brewer erano corrette**: nessun sistema informatico può assicurare consistenza, continua disponibilità e tolleranza alle partizioni allo stesso tempo.

Di seguito descriviamo brevemente la definizione delle tre garanzie.

Consistency

Un sistema è definito **completamente coerente** quando è in grado di garantire che una volta memorizzato un nuovo stato nel sistema, questo è utilizzato in ogni operazione successiva fino alla successiva modifica dello stesso. Pertanto, tutte le richieste dello stato del sistema, nell'arco di tempo che intercorre tra uno stato e quello successivo, forniscono il medesimo risultato.

Per esempio, quando si ha a che fare con una cache con un singolo nodo, a meno di errori di codifica, la cache è intrinsecamente totalmente coerente poiche' lo stato è aggiornato in un nodo e mantenuto esclusivamente nello stesso. Un singolo nodo quindi garantisce intrinsecamente la totale consistenza e anche la tolleranza alle partizioni, ma non una sufficiente

disponibilità (e tolleranza ai guasti) e, in scenari non banali, tende a presentare scarse performance. Quando invece la cache è distribuita e vi sono due o più nodi, il sistema è pienamente consistente quando tutti i nodi sono in grado di lavorare sullo stesso stato, ossia vedono gli stessi dati con gli stessi valori. Queste configurazioni permettono di aumentare la disponibilità del sistema, e come controparte si incrementa la complessità: il sistema di cache deve prevedere sofisticati meccanismi per far sì che ogni nodo sia in grado di accedere ad una specie di repository virtuale distribuito e per gestire eventuali partizioni. Sebbene non sia possibile garantire la terna CAP, è necessario implementare strategie per ridurre l'impatto dovuto al manifestarsi della garanzia trascurata (per esempio perdita di coerenza o generazione di una partizione).

Availability

Un sistema è detto **continuamente disponibile** quando è sempre in grado di soddisfare le varie richieste/erogare i propri servizi. Nel contesto di cache distribuita, come visto poc'anzi un singolo nodo non permette assolutamente di realizzare un sistema molto disponibile: è sufficiente che il nodo vada giù affinche' il sistema non sia più disponibile. Inoltre, a seconda delle configurazioni utilizzate (per esempio **write-behind**), una configurazione a singolo nodo potrebbe portare alla perdita di dati.

Qualora invece la cache distribuita funzioni su un opportuno cluster di server, lo scenario in cui un singolo nodo si renda non disponibile non generebbe alcuna perdita di informazioni: tutte le cache distribuite si occupano di mantenere nei vari nodi delle aree

di backup in cui sono memorizzati, in modo ridondante, i dati presenti negli altri nodi. Chiaramente con l'aumento del numero dei nodi aumenta anche la disponibilità del sistema.

Da tener presente che il traffico di rete tra i nodi, necessario per garantire la consistenza, è proporzionale al loro numero degli stessi. Quindi aumentando il numero di nodi oltre un certo livello di soglia si assistere a un degradamento delle performance. Inoltre, aumentando i nodi aumentano i problemi relativi alla garanzia della tolleranza alle partizioni. Come accennato poc'anzi, nel contesto delle data grid, la disponibilità non è solo una protezione da guasti ma anche dalla perdita dei dati.

Come visto, la strategia standard utilizzata per ottenere la continua disponibilità consiste nel ricorrere ad un'opportuna ridondanza che nel caso della cache implica la presenza di molteplici nodi. Ciò, oltre a richiedere opportuni meccanismi per garantire la consistenza, aumenta le problematiche relative alla tolleranza alle partizioni.

Partition tolerance

Gilbert & Lynch [2] hanno definito la **tolleranza alle partizioni** come la proprietà di un sistema di continuare a funzionare correttamente anche in presenza di una serie di fallimenti dell'infrastruttura fino a che l'interno network fallisca ("No set of failures less than total network failure is allowed to cause the system to respond incorrectly").

Per dimostrare questa tolleranza alle partizioni si consideri una configurazione (da ponderare molto attentamente nel contesto delle cache distribuite) in cui un singolo cluster preveda nodi presenti su diversi data center. Qualora, per qualsiasi motivo, si

dovesse perdere la connettività di rete tra i vari data center, oppure le performance di rete degradino improvvisamente per un arco di tempo prolungato, si genera la situazione in cui i server del cluster non sono più in grado di sincronizzare lo stato del sistema. In queste circostanze, i server in ogni centro si riorganizzano in sotto-cluster assumendo che quelli dell'altro centro non siano più raggiungibili, tagliandoli fuori.

In questo modo si finisce per generare due sotto-cluster che danno vita al classico scenario denominato del "**cervello diviso**". Il sistema continua a funzionare, i dati sono gestiti nei sotto-cluster in maniera non coordinata causando tutta una serie di problemi inclusi la perdita di dati. Come caso semplice si consideri un sistema di prenotazione. In un contesto del genere è assolutamente possibile che ogni cluster finisca per assegnare la medesima prenotazione a diversi clienti.

Le conseguenze del teorema

Il **teorema di Brewer** ha un'importanza fondamentale nella progettazione di sistemi software distribuiti: non conoscerlo nei dettagli può generare conseguenze disastrose come per esempio imbattersi nei requisiti CAP senza rendersene conto finendo per sprecare anni/budget significativi per cercare di risolvere problemi non risolvibili.

Come al solito, i requisiti non-funzionali recitano un ruolo cruciale: a bassi volumi transazionali dove sono accettabili latenze non critiche, gli effetti del teorema di Brewer tendono a non essere visibili, ne' sulle prestazioni complessive del sistema ne' sull'esperienza dell'utente. In questi contesti, anche soluzioni classiche, fondate su database, tendono ad essere sufficienti.

CAP e Web 2.0

In effetti, le prime versioni dei vari servizi Internet diventati popolarissimi con “Web2.0” non presentavano architetture particolarmente innovative/rivoluzionarie: molte erano abbastanza standard. La situazione tuttavia è cambiata drasticamente con l’enorme espansione del parco utenti. Il conseguente incremento spesso esponenziale dell’attività aumenta, ha fatto sì che tutti i limiti illustrati dal teorema si resero manifesti fino ad avere sistemi che causano errori ed, in ultima analisi, scoraggiano utenti. A chi non è capitato di imbattersi in un sito Internet, richiedere una pagina e trovarsi di fronte a un classico errore (non gestito) HTTP?. Si provi a immaginare la frustrazione e sconcerto di un utente che dopo aver immesso i dettagli di un pagamento si veda recapitato un messaggio di errore HTTP. Una (non) **esperienza utente** del genere può facilmente generare la perdita di utenti e di immagine del servizio e dell’azienda: è proprio il Web2.0 con tutti i suoi servizi che fa sì che una pessima esperienza, condivisa su uno dei social, divenga immediatamente resa nota a centinaia/migliaia/milioni di utenti generando un’enorme perdita di immagine da parte dell’azienda con conseguente perdita di business e necessità di investire massicciamente in marketing per tentare di riconquistarla. Alcune situazioni estreme sono state rese note da Amazon e Google [6]:

- Amazon: anche un “insignificante” decimo di secondo nei tempi di risposta del sito genera una riduzione delle vendite stimabile intorno all’1%;
- Google: un misero mezzo secondo di aumento della latenza causata dal traffico può generare una perdita di richieste del 20%.

Dalla tecnica alla strategia

Il teorema di Brewer evidenzia in maniera ineluttabile come la progettazione di servizi molto popolari con milioni di utenti sparsi per il mondo o anche di sistemi aziendali con centinaia di utenti globalmente distribuiti, pressati da requisiti funzionali molto stringenti, non è esclusivamente un problema tecnico, ma un aspetto cruciale per intere organizzazioni, che influenza il modo in cui vengono effettuate le scelte strategiche operative di senior management e rappresentanti del business. Sapere che non si possono avere tutte le garanzie CAP aiuta a scegliere quale è meglio sacrificare e quale impatto può avere tale scelta sul business e più in generale sull'azienda.

È appena il caso di ricordare che ogni contesto ha un proprio **pattern**. Per esempio, se si considerano servizi come Twitter e Facebook, è evidente che l'eventuale perdita di qualche messaggio non genera grandissimi problemi, mentre in un sistema di broker elettronico è fondamentale disporre di piattaforme super-veloci (**ultra-low-latency**), che non perdano assolutamente alcun messaggio e che siano sempre disponibili durante gli orari previsti anche sacrificando la totale coerenza: un broker può perdere fino a 4 milioni di dollari di introito per millisecondo qualora la piattaforma di trading elettronico dovesse essere di 5 millisecondi più lenta rispetto alla concorrenza [7].

Sistemi a consistenza "finale"

Da notare che una delle scelte più frequenti nel disegno di architetture distribuite con pressanti requisiti non funzionali, consiste nella rinuncia della coerenza totale a favore di forme più

blande, tanto che da qualche tempo si è iniziato a parlare di **eventual consistency** ("coerenza finale"). Sistemi a consistenza finale garantiscono informalmente che, qualora non vi siano nuovi aggiornamenti per un certo lasso di tempo afferenti a un determinato dato, alla fine tutti le richieste dello stesso restituiscono consistentemente il medesimo valore, risultato dell'ultimo aggiornamento [7].

Questo è il caso di **Apache Cassandra**, una delle più famose soluzioni nel gruppo dei NoSQL sviluppato inizialmente nei laboratori di Facebook per alimentare la loro funzione di ricerca della posta nella casella di arrivo. La progettazione iniziale si deve a Avinash Lakshman (uno degli autori di **Amazon's Dynamo**) e Prashant Malik. **Cassandra** è stato rilasciato come progetto open source a Google Code nel luglio 2008. Nel marzo del 2009, è diventato un progetto Apache Incubator per poi diventare qualche anno più tardi progetto di primo livello.

L'architettura di Cassandra privilegia alta disponibilità e tolleranza al partizionamento (AP) trascurando appunto la totale consistenza. Cassandra è appunto una soluzione a coerenza finale. Questi sistemi sono classificati a semantica di **BASE** (**B**asically **A**vailable, **S**oft state, **E**ventual consistency, fondamentalmente disponibili, con concetto di stato leggero e a coerenza finale) in contrasto alla tradizionale forte semantica **ACID** (**A**tomicità, **C**oerenza, **I**solamento, **D**urabilità).

Da notare che la consistenza finale è una feature (non si può proprio parlare di garanzia) un po' blanda: alla fine tutte le letture di uno stesso dato inizieranno a restituire il medesimo valore, però non vi è alcuna garanzia di quando ciò avvenga, e molto importante, dei valori restituiti durante la fase di convergenza. Poiche' le informazioni sono replicate, l'ultima versione di uno

specifico dato risiede su uno specifico nodo del cluster per un certo intervallo di tempo mentre le precedenti versioni sono ancora disponibili su altri nodi. Alla fine del processo di convergenza (che tutti i sistemi a consistenza finale devono implementare per assicurare l'evoluzione del sistema verso un medesimo stato), farà sì che alla fine tutti i nodi dispongano dell'ultima versione.

Questo processo richiede di mettere in atto sofisticate strategie per conciliare differenze tra più copie dei dati distribuiti comunemente dette di **anti-entropia**.

Compromessi del CAP

Assodato che non è possibile garantire simultaneamente completa consistenza, continua disponibilità e tolleranza alle partizioni, quando si progetta un sistema distribuito è necessario ponderare attentamente quale soluzione di compromesso accettare tra le seguenti coppie possibili: **AC**, **AP** e **CP**, come illustrato in figura 1.

Figura 1 – Rappresentazione grafica del Teorema CAP e dei possibili compromessi fra le diverse garanzie.

Consistency/Availability

CA: elevata coerenza/alta disponibilità. Si tratta del compromesso tipicamente offerto da **RDBMS** e da soluzioni quali **Oracle Coherence**, **Vertica**, **Aster**, etc. I dati sono mantenuti in **modo coerente** in tutti i nodi del cluster, a patto che ovviamente tali nodi siano disponibili. È sempre possibile leggere e/o scrivere su qualsiasi nodo ed essere sicuri che il nuovo dato sia propagato a

tutti i nodi del cluster. Non vi è quindi la possibilità che alcuni nodi abbiano delle versioni dei dati non aggiornate. Tuttavia, la totale coerenza può incidere sulle performance (latenza) e sulla scalabilità. Inoltre, si possono avere problemi qualora si venisse a formare una partizione tra i nodi. In situazioni limite è possibile che la presenza di partizioni generi un disallineamento dei dati di non facile recupero. Pertanto, è necessario predisporre delle strategie che sia in grado di individuare eventuali partizione della rete. Qualora queste siano individuate, il database deve interrompere tutte le partizioni tranne una e procedere ad un'eventuale risoluzione dei conflitti.

Consistency/Partition tolerance

CP: elevata coerenza/tolleranza alle partizioni. Questo compromesso è quello preferito da soluzioni quali **MongoDB**, **HBase**, **BigTable**, **Terrastore**, **Redis** e altri. I dati sono mantenuti in maniera coerente in tutti i nodi del cluster, e viene garantita la tolleranza partizione (ciò **evita che dati possano desincronizzarsi**). Tuttavia si possono avere problemi di disponibilità perche' il sistema diviene non disponibile quando un nodo va giù. Da notare che quasi tutte le soluzioni prevedono una configurazione in cui **un nodo** agisce come **master** (questo è anche il caso di Coherence come mostreremo nei successivi articoli) e gli altri come **slave**. Inoltre, i sistemi prevedono particolari procedure per "eleggere" un nuovo master qualora quello in carica divenga non raggiungibile.

Availability/Partition tolerance

AP: continua disponibilità/tolleranza alle partizioni. Questo

compromesso è stato prescelto da soluzioni quali **Apache Cassandra**, **CouchDB**, **DynamoDB** e **Riak**. I nodi restano on-line anche nelle situazioni in cui non possono comunicare tra loro. È poi compito del processo di risincronizzazione dei dati risolvere eventuali conflitti una volta che la partizione è risolta (per questo sistemi che forniscono le garanzie AP sono detti anche “disponibili alla fine”). Chiaramente non è possibile avere garanzia che tutti i nodi abbiano gli stessi dati con gli stessi valori durante la partizione e la relativa risoluzione. Soluzioni AP tengono a presentare **migliori prestazioni** in termini di latency e a scalare in modo più lineare.

Una nota imporante è la seguente: quasi tutte le soluzioni prevedono un **tuning** della modalità operativa; in pratica, il programmatore può decidere su quale compromesso operare. Per esempio, **MongoDB** è tradizionalmente **CP** (opzione **safe** impostata a **true**) ma, opportunamente “tarato”, può funzionare anche in modalità **AP**.

Conclusioni

Con questo articolo abbiamo avviato la miniserie dedicata al teorema CAP e alla sua implementazione in Oracle Coherence: data-grid internamente gestita in memoria, sempre disponibile grazie a un elevato cluster di computer, adatta per l’analisi dei dati in tempo reale, calcoli di tipo grid eseguiti in memoria, e soluzioni ad alte prestazioni.

In particolare, in questo primo articolo ci siamo focalizzati sul teorema di Brewer o CAP. Come visto, questo sostiene che non è possibile per un sistema software distribuito presentare simultaneamente le tre seguenti garanzie: consistenza, continua

disponibilità e tolleranza alle partizioni. La progettazione di sistemi distribuiti gravati da pressanti NFR, richiede di conoscere intimamente il teorema CAP e di comprendere vantaggi e svantaggi di ciascun compromesso.

Da tenere presente che in presenza di sistemi distribuiti con NFR non particolarmente pressanti, è spesso possibile (dopo un'attenta analisi) ignorare il teorema di CAP. In questi contesti, gli effetti del teorema CAP non sono visibili. Non è un caso che la maggior parte dei servizi che poi hanno dato vita a soluzioni NoSQL avevano avviato i lavori con architetture abbastanza tradizionali basate su database relazionali tradizionali. Questo è il caso di Amazon Dynamo, Google BigTable, LinkedIn Voldemort, Twitter FlockDB, Facebook Cassandra, Yahoo! PNUTS, giusto per citarne alcuni tra i più famosi.

Tuttavia con l'incremento spesso esponenziale degli utenti e quindi delle transazioni, si trovarono ben presto nell'impossibilità di riuscire a fornire servizi, basati su grandi volumi, con la latenza richiesta, mantenendo nel frattempo elevata disponibilità e consistenza. Molte di queste aziende iniziarono ad attuare le classiche strategie di ottimizzazione: incremento dell'hardware (aggiunta di server sempre più potenti con maggiore RAM) fino al limite in cui le performance iniziavano a peggiorare; denormalizzazione della base dati; aggiunta di cache; analisi passo passo del codice; e così via. Sebbene alcune di queste tecniche riuscirono a migliorare la situazione, nessuna riuscì a risolverla.

Pertanto ben presto i vari architetti si resero conto che era necessario ricorrere a soluzioni che riuscivano a fornire i servizi nel tempo previsto su larghe base dati correndo qualche rischio, come per esempio una coerenza ritardata, dati mantenuti

essenzialmente in memoria, etc. Avevano intuito l'esistenza del teorema CAP più o meno consciamente e avevano iniziato a costruire prodotti che poi saranno etichettati come NoSQL.

Riferimenti

[1] Eric A. Brewer, "ACM Symposium on the Principles of Distributed Computing"

http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[2] Nancy Lynch and Seth Gilbert, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59.

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.6951&rep=rep1&type=pdf

[3] Julian Browne, "Brewer's CAP Theorem – The kool aid Amazon and Ebay have been drinking"

http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

[4] Oracle and Tangosol

http://www.oracle.com/us/corporate/acquisitions/tangosol/index.html

[5] Luca Vetti Tagliati, "Component based architectures for eXtreme Transaction Processing", SEKE 2008 PROCEEDINGS, July 1, 2008

www.ksi.edu/seke/Proceedings/seke/SEKE2008_Proceedings.pdf

[6] Todd Hoff, "Latency is Everywhere and it Costs You Sales – How to Crush it", 25 luglio 2009

http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it

[7] Werner Hans Peter Vogels, “Eventually consistent”. Communications of the ACM 52: 40. 2009