Relazione su articolo “There Is No Getting Around It: You Are Building a Distributed System” di Mark Cavage

da *Communications of the ACM*, Vol. 56 No. 6, Pages 63-70

scritta da Andrea Mantoni

per il corso di Ingegneria del Software 2014/2015

Introduzione

La tendenza attuale mostra un passaggio per le applicazioni web dai sistemi centralizzati tipici degli anni '80 e '90 ai sistemi distribuiti.

I motivi di questo cambiamento sono molteplici:

1. aumento della complessità dei software realizzati sproporzionale all'aumento delle performance dei singoli elaboratori (vedi caduta della legge di Moore);
2. diffusione del web e del numero degli accessi remoti (di nuovo, problema di performance insufficienti);
3. diminuzione del costo dei singoli elaboratori;
4. aumento della velocità delle reti;
5. aumento della concorrenza e sofisticazione della clientela sul web che richiede una elevata qualità dei servizi forniti (elevata disponibilità, affidabilità, bassa latenza, etc.).

I sistemi distribuiti tuttavia presentano diversi problemi mai risolti definitivamente:

- maggiori difficoltà di progettazione e debugging rispetto ai sistemi centralizzati;
- **Consistenza, Disponibilità e Tolleranza alle partizioni** sono in concorrenza tra loro (**CAP theorem**). Ciò porta spesso a delle scelte progettuali che privilegiano una caratteristica piuttosto che un'altra.

La maggior parte dei sistemi distribuiti realizzati attualmente sfrutta il **cloud computing**.

Il cloud computing è disponibile in diverse modalità:

1. **cloud computing privato (intra-organizzazione)**
2. **cloud computing pubblico (inter-organizzazione)**, fornito da terze parti dette **cloud service providers** (ad es. Amazon Web Services, Windows Azure, Google Compute Engine)
3. **cloud computing ibrido** (in parte privato e in parte pubblico), mediante l'uso di standard come OpenStack.

Nei progetti con maggiori restrizioni di budget la scelta più comune è la 2.

Il motivo è che i cloud providers richiedono il pagamento solo delle risorse effettivamente utilizzate, mentre un sistema privato avrebbe dei costi maggiori e resterebbe inutilizzato per molto tempo.

Come è noto infatti, il traffico di rete presenta caratteristiche “bursty” (si concentra in periodi di tempo relativamente brevi), ed è fondamentale per le attività di e-commerce essere disponibili quando il mercato lo richiede (time-to-market).

D'altro canto è anti-economico mantenere un sistema che è utilizzato al massimo delle capacità solo per brevi periodi.

Nonostante il cloud computing pubblico sembrerebbe molto vantaggioso, presenta diversi problemi spesso non considerati:

- i cloud provider mirano ovviamente a massimizzare l'utilizzo dei propri sistemi per ragioni economiche. Ciò fa sì che le applicazioni virtualizzate girino spesso in ambienti saturi, con una elevata probabilità di errori temporanei e diminuzioni di performance difficili da simulare in ambiente di sviluppo (**brownouts**);
- non tutti i parametri del servizio sono negoziabili con il cloud provider;
- problemi di sicurezza dovuti alla gestione esterna di dati sensibili;
- difficoltà per il monitoraggio della qualità del servizio.

Ciò porta spesso a dei compromessi costi/qualità non sempre soddisfacenti.

La situazione è del tutto identica anche per le offerte di **PaaS (Platform As A Service)** come Google App Engine, dove lo sviluppatore non ha la possibilità di gestire gli host virtuali (VMs) direttamente, ma è forzato ad utilizzare una infrastruttura preesistente.

Design di architetture per il cloud

In generale il focus nella progettazione di sistemi distribuiti deve essere posto nell'analisi dei requisiti non-funzionali indicati con la sigla **SLAs (Service-Level Agreements): disponibilità, latenza, throughput, consistenza e durabilità**.

Anche se potrebbe sembrare ovvia, la prima domanda da porsi è se l'uso di un sistema distribuito è effettivamente necessario.

La risposta viene spesso dai SLAs, ma anche i requisiti funzionali possono fornire degli indicatori. Ad esempio, volendo realizzare un sistema di ridimensionamento immagini remoto, è possibile fare una stima delle risorse hardware necessarie, e valutare rapidamente se un sistema centralizzato potrebbe essere sufficiente.

(la risposta è spesso negativa)

Una volta stabilita l'effettiva necessità di creare un sistema distribuito, il passo seguente consiste nella scomposizione del sistema in servizi.

La **SOA (Service-Oriented Architecture)** è infatti l'architettura più usata, ed è supportata da diversi framework (ad es. Java EE).

Sommerville suggerisce che i servizi dovrebbero essere il più possibile **indipendenti, riusabili e state-less**.

Ad es., per un sistema di ridimensionamento immagini una possibile decomposizione in servizi potrebbe essere la seguente:

1. API per la comunicazione con il cliente;
2. coda di messaggi distribuita (per raccogliere tutte le richieste e servirle in ordine);
3. servizio di gestione utenze (identity-management system);
4. servizio di aggregazione dati di utilizzo (usage aggregation fleet) (per fatturazione e a scopi statistici)

Per ottimizzare il design dei vari servizi, l'autore dell'articolo suggerisce di stimare vari parametri per ognuno di essi:

- **Distribuzione geografica:** il sistema dovrà operare su aree geografiche distinte o su una sola?
- **Segregazione dati:** necessario isolamento tra i vari clienti? (**single o multi-tenancy?**)

- **SLAs**: requisiti non-funzionali come disponibilità, latenza, throughput, consistenza e durabilità;
- **Sicurezza**: identità, autenticazione, autorizzazione, audit (IAAA), confidenzialità e riservatezza dati
- **Controllo utilizzo**: necessità di monitorare l'utilizzo del servizio strategica ai fini del business?
- **Deployment e gestione delle configurazioni**: problemi relativi all'aggiornamento del sistema.

In questo modo diventa possibile ottimizzare il design dei vari servizi in base alle effettive necessità dell'applicazione.

Ad es. nel caso del sistema di ridimensionamento immagini, possiamo osservare che l'API del cliente (1) dovrà essere ottimizzata per fornire una bassa latenza ed una elevata disponibilità. Per ottenere questo sarà necessario predisporre più di un server per l'API e un meccanismo di **bilanciamento del carico (load balancing)**. Per quest'ultimo esistono molte soluzioni ben sperimentate come il round-robin DNS.

Per quanto riguarda il servizio di coda messaggi (2), la maggior parte delle implementazioni preesistenti privilegiano la consistenza piuttosto che la disponibilità (vedi teorema di CAP). Questa scelta è la migliore nella maggior parte degli scenari, anche se in alcuni potrebbe non esserlo.

Nel nostro caso, poiché miriamo ad ottenere una disponibilità vicina al 99,9% sarà necessario predisporre un meccanismo di **failover automatico**.

Il failover automatico consente di ripristinare immediatamente l'operatività del servizio a seguito di guasti delle singole macchine del sistema. Uno degli algoritmi più usati è quello di **Paxos**.

Anche per il servizio di gestione utenze (3) bisogna massimizzare la disponibilità e minimizzare la latenza. L'autore suggerisce di utilizzare un sistema di caching read-only (data plane) in quanto nell'uso tipico il sistema elaborerà molte più richieste di autenticazione che di modifica dati (management plane).

Poiché la sincronizzazione tra i due planes non è istanea, questa scelta ha come conseguenza una diminuzione della consistenza (sempre per il teorema CAP).

Infine, il servizio di aggregazione dati (4) ha delle SLA molto diverse: disponibilità e consistenza non sono importanti come nei casi precedenti. Per quanto riguarda la durabilità, può essere ripartita a due livelli:

i singoli API server effettuano l'aggregazione localmente e solo quando viene superata una certa soglia inviano un blocco di dati ad uno staging system.

Lo staging system esegue dei batch jobs periodicamente (in maniera asincrona rispetto all'arrivo di nuovi blocchi di dati) ed immagazzina i risultati permanentemente.

Sviluppi futuri e considerazioni personali

Come si è visto affidarsi ad un solo cloud provider pubblico può essere fonte di problemi. D'altro canto per una piccola azienda allestire un cloud privato può non essere conveniente. Un cloud ibrido potrebbe essere la soluzione, ma attualmente non esistono standard per la comunicazione inter-cloud molto diffusi.

Una soluzione alternativa sicuramente praticabile è sfruttare l'indipendenza dei singoli servizi e pianificare il loro deployment su diversi cloud provider con caratteristiche e costi diversi, ottenendo così un migliore compromesso costi/qualità (**cloud mashups**).

Ciò richiede inevitabilmente una progettazione delle applicazioni mediante la SOA, e che l'attività di analisi metta in evidenza i SLAs più critici per ogni servizio.

Un altro importante parametro da considerare è la dimensione delle risorse hardware richieste per ogni servizio.

Ad es. nel caso del ridimensionamento immagini, l'API per la comunicazione con il cliente e l'identity-management system potrebbero essere tenuti su un cloud privato (o su un cloud con elevata QoS). Questi servizi infatti richiedono relativamente poche risorse, ma hanno elevati requisiti di disponibilità e sicurezza.

I servizi che invece presentano una grande variabilità delle risorse richieste sono più adatti per l'installazione su cloud pubblici (nell'esempio il servizio di coda messaggi ed i server di elaborazione immagini sono i candidati migliori).

Un'altra interessante possibilità è offerta dal rebinding a runtime dei servizi.

Nel nostro esempio, si potrebbero installare più copie dello stesso servizio di coda messaggi su diversi clouds. Quindi, se si rileva che uno di questi è troppo saturo o presenta problemi, si potrebbe bindarne automaticamente un altro.

L'unico problema in questo caso è legato al monitoraggio dei cloud esterni, perché non esistono standard affermati (diversi cloud provider offrono diverse soluzioni di monitoraggio).

Di fatti questa strategia è usata soprattutto con i cloud privati che sono più facili da monitorare (**cloud bursting**).

In conclusione, anche se il cloud computing offre diversi vantaggi economici, pone nuovi problemi e mantiene quelli già presenti con i sistemi distribuiti tradizionali.

Un altro sviluppo che tenta di risolvere questi problemi sono le **distributed virtual machines (DVM, o virtual machines clusters)**.

Queste sono in grado di distribuire automaticamente il carico su più host e creare l'illusione di un sistema centralizzato, semplificando così il processo di sviluppo.

La loro diffusione è però ancora limitata e sono ancora oggetto di studio.