ALTRI PARADIGMI DI PROGRAMMAZIONE

A.A. 2020/2021

Laurea triennale in Informatica

18: Stato Distribuito

STATO DISTRIBUITO

Abbiamo giustificato la necessità di realizzare un sistema distribuito con la ricerca di:

- affidabilità
- suddivisione del carico
- distribuzione dei risultati

Le stesse caratteristiche possono essere desiderabili per un insieme di dati:

- disponibili continuamente, anche in presenza di guasti
- quantità superiore a quella gestibile da una sola macchina
- accessibilità da parte di più posizioni geografiche

Un insieme di dati gestito da un sistema distribuito è disponibile, accessibile e coerente solo nella misura in cui i singoli nodi riescono a rimanere allineati, coordinati e concordi fra loro.

RFC 1034-1035

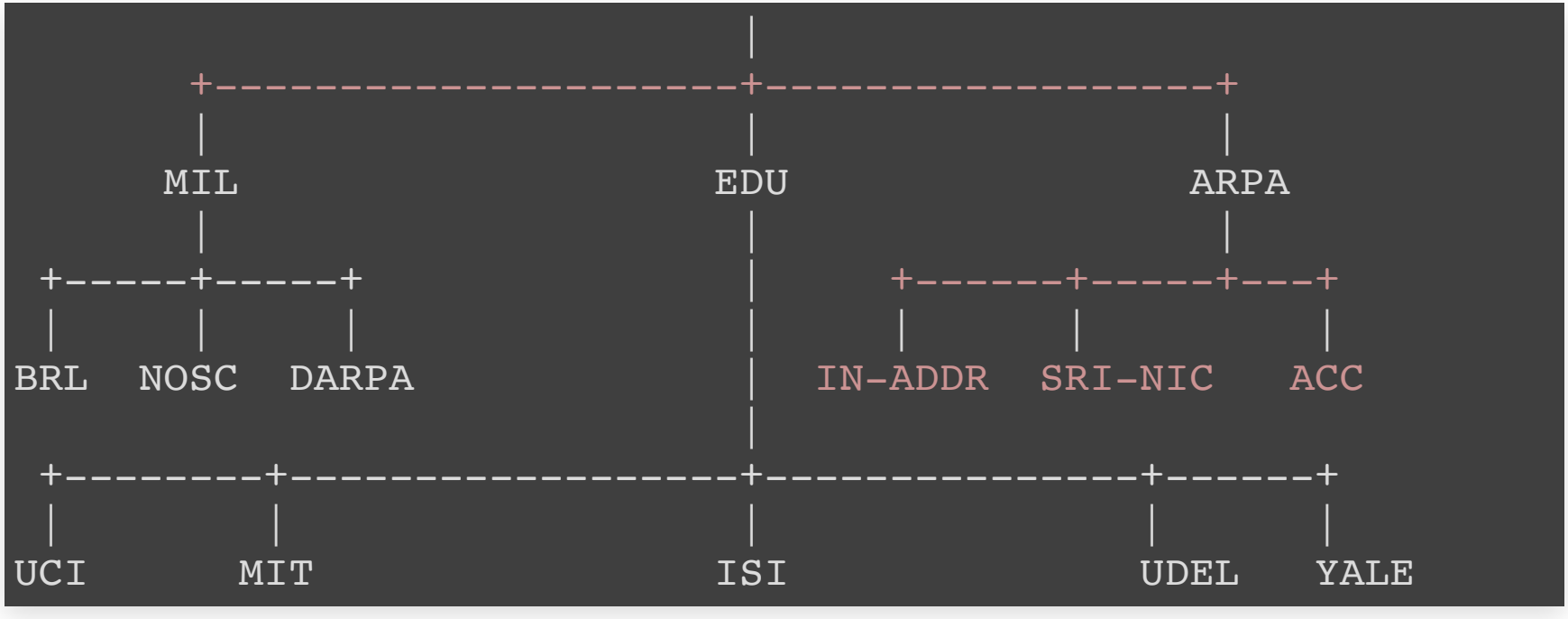
"Host name to address mappings were maintained by the Network Information Center (NIC) in a single file (HOSTS.TXT) which was FTPed by all hosts [RFC-952, RFC-953]."

"Explosive growth in the number of hosts didn't bode well for the future."

"The network population was also changing in character."

"Local organizations were administering their own names and addresses, but had to wait for the NIC to change HOSTS.TXT to make changes visible to the Internet at large. Organizations also wanted some local structure on the name space."

"A design using a distributed database and generalized resources was described in [RFC-882, RFC-883]. Based on experience with several implementations, the system evolved into the scheme described in this memo."


MIL
EDU
ARPA
BRL
NOSC
DARPA
IN-ADDR
SRI-NIC
ACC
UCI
MIT
ISI
UDEL
YALE

Nel Domain Name System ogni nodo è responsabile (autorevole) di un ramo di un'albero degli indirizzi.

Un client chiede la risoluzione di un nome al DNS cui appartiene o più vicino, e questo eventualmente propaga la richiesta se non possiede la risposta.

Il requisito principale del Domain Name System è la suddivisione delle responsabilità.

Quindi, la distribuzione dello stato è effettivamente implementata attraverso la suddivisione gerarchica dello stato stesso, senza condivisione fra i nodi.

CONSENSO

Se il requisito principale è la resistenza ai fallimenti o le prestazioni particolarmente elevate (in spazio o velocità) non abbiamo altra scelta che replicare lo stato in più di un nodo, in modo da avere che:

- un guasto non porti a perdita di dati
- aggiungendo nodi aumenti la capacità di risposta del sistema

Ma avendo più nodi che contengono lo stesso dato, nasce il problema del **consenso**, ovvero di garantire che tutti i nodi del sistema contengano la stessa versione di un dato.

Il problema è stato denominato da Leslie Lamport come il problema dei *Generali Bizantini*

Diversi generali di un esercito Bizantino assediano una città. Devono raggiungere un consenso unanime su di una decisione: attaccare o ritirarsi.

Un attacco parziale è molto più svantaggioso di un attacco coordinato o di una ritirata completa.

Possono comunicare solo scambiandosi messaggi l'uno con l'altro.

Tuttavia, non si sono garanzie sulla loro lealtà: alcuni generali possono tradire, inviando messaggi contraddittori o non rispondendo ai messaggi ricevuti.

Il problema modella il caso in cui un nodo di un sistema distribuito si comporta in modo diverso ad ogni risposta.

Di fatto, non è conoscibile dall'esterno se il suo comportamento è corretto o se il nodo è guasto.

Lo strumento per affrontare la soluzione di tale problema è un *algoritmo di consenso*.

PAXOS

L'algoritmo PAXOS è basato su di un protocollo a tre fasi che garantisce il l'assenza di blocchi nel caso di un guasto singolo.

I possibili comportamenti dei nodi sono modellati con delle macchine a stati per garantire la copertura di tutti i casi.

La correttezza dell'algoritmo è dimostrata formalmente, rendendolo al momento della pubblicazione (1989) uno dei primi algoritmi del genere dotato di una prova di funzionamento.

Nonostante alcuni lavori precedenti molti simili, è considerato il primo algoritmo di consenso distribuito pubblicato per l'eleganza del modello proposto.

Nell'algoritmo PAXOS i nodi assumono dei precisi ruoli:

- leader
- votanti
- ascoltatori
- proponente
- client

Il Proponente inoltra la richiesta del Client ai Votanti per raggiungere un Quorum. Una volta che un sufficiente numero di votanti appartiene ad un Quorum, la richiesta è stata accolta.

Gli Ascoltatori, ricevendo messaggi dai Votanti, forniscono ridondanza addizionale in caso di guasti.

Il Leader coordina il protocollo ed è in grado di fermarlo in caso di eccessivi fallimenti in attesa che possa essere ripresa la normale attività.

Una variante dell'algoritmo è in grado di fornire garanzie di correttezza anche in presenza di Guasti Bizantini.

RAFT

Raft è un algoritmo di consenso scritto con l'obiettivo di essere più comprensibile di PAXOS

https://raft.github.io/

RAFT suddivide il problema del consenso in sottoproblemi per rendere il modello più semplice e l'implementazione più agevole.

Il concetto principale è la replicazione di una macchina a stati.

Il ruolo del leader e la sua elezione sono concetti più enfatizzati che in PAXOS, e garantiscono la correttezza dell'algoritmo

Sono disponibili (ed utilizzate in produzione in alcuni database distribuiti) implementazioni in Java, Scala, e altri linguaggi.

CAP THEOREM

Tramite il consenso, possiamo ottenere che il sistema distribuito abbia uno stato coerente, e che quindi ogni nodo possa rispondere allo stesso modo alla richiesta di un dato.

Ma quali sono i limiti ed i compromessi che dobbiamo prevedere di dover fare in caso di guasti?

CAP THEOREM

Il teorema **CAP** definisce tre caratteristiche di un database distribuito:

	Nome	Descrizione
C	Consistenza	Ogni lettura riceve o il valore più recente o un errore
A	Disponibilità (*Availability*)	Ogni richiesta riceve una risposta valida (ma non necessariamente l'ultimo valore)

	Nome	Descrizione
P	Tolleranza alla separazione (*Partition tolerance*)	Il sistema funziona anche se la rete fallisce per un insieme di nodi, cioè viene *partizionata*

Il teorema afferma che solo *due* delle proprietà CAP possono essere garantite contemporaneamente.

Siccome ogni rete può fallire, in pratica il teorema afferma che:

in caso di partizione di rete un sistema può essere o consistente o disponibile, ma non entrambe le cose.

Vale a dire che un database distribuito può essere costruito in due modi; in caso di partizione di rete

- la consistenza viene garantita ma alcune richieste non possono essere soddisfatte e vanno in errore
- tutte le richieste ritornano un valore, ma alcune potrebbero non ritornare l'ultimo valore scritto

Il teorema è stato enunciato da Eric Brewer nel 1998 prima come principio, poi come congettura, ed infine dimostrato nel 2002.

Attenzione: la C di CAP non è la C di ACID.

Una estensione del teorema, *PACELC*, considera anche il caso dell'operatività normale:

- in caso di (P) Partizione, si deve scegliere fra (A) disponibilità e (C) consistenza
- (E) altrimenti, fra (L) latenza e (C) consistenza

La maggior parte dei sistemi NoSQL si orienta verso PA/EL.

I database relazionali tipicamente sono invece PC/EC.

CRDT

Finora abbiamo lavorato nell'ipotesi che il sistema distribuito conservi dati che vengono scritti da una sola fonte o che solo occasionalmente arrivino da più di una direzione.

In alcuni sistemi distribuiti invece è normale che ogni nodo abbia nuove informazioni da fornire e che queste vadano riconciliate e riunite con quelle prodotte indipendentemente da un altro nodo:

- sistemi di telepresenza/chat
- editor collaborativi
- eventi in real-time con molti partecipanti

In questi sistemi l'esigenza non è quella di raggiungere un consenso su di un dato.

Al contrario, ogni nodo produce nuove versioni dello stesso dato che ogni altro nodo deve riunire a quelle prodotte localmente.

Ci sono in letteratura due soluzioni a questo problema:

- Operational Transformation
- *Conflict-Free Replicated Data-Type*

OPERATIONAL TRANSFORMATION

L'approccio delle OT è il seguente:

- ogni nodo produce delle modifiche al documento in corso di elaborazione
- le modifiche sono propagate agli altri nodi
- ogni nodo trasforma le modifiche ricevute in modo da applicarle al suo stato del documento

Sebbene abbia alcune implementazioni di successo (*Google Wave*, *Google Docs*), le OT non hanno preso piede: l'approccio sembra mancare di generalità e si rivela molto complesso da implementare.

Una più recente soluzione a questo tipo di problema è una classe di strutture dati dette *Conflict-Free Replicated Data-Type*.

Una CRDT può essere replicata su più nodi di un sistema, modificata indipendentemente, ma fornisce la garanzia che esiste un modo di riconciliare tutte le possibili modifiche risolvendo ogni possibile conflitto.

Esempio: un flag booleano che possa solo passare da falso a vero è una banale CRDT. Anche se più nodi effettuano la modifica, il risultato può essere tranquillamente replicato senza che si siano problemi a riunire le differenti versioni.

Diverse strutture dati con queste caratteristiche sono note ed implementate:

- Grow-only Counter
- Positive-Negative Counter
- Grow-only Set
- 2-Phase Set
- Last-Write-Wins Set

Progetti che usano implementano CRDT:

- Akka (modulo Akka Data)
- Riak
- Redis Enterprise
- Facebook Apollo

CALM THEOREM

Un risultato molto recente permette di affrontare il problema del consenso distribuito in modo positivo, in contrasto con l'affermazione negativa del teorema CAP.

Nel paper "*Keeping CALM: When Distributed Consistency is Easy*", pubblicato a gennaio 2019, Joseph M. Hellerstein e Peter Alvaro propongono un risultato che caratterizza i programmi che possono mantenere la consistenza anche in presenza di partizionamento della rete.

CALM sta per "Consistency As Logical Monotonicity"

Gli autori affermano che i programmi che mantengono le proprietà CP nel teorema CAP sono esattamente i programmi che possono essere espressi termini di "*logica monotona*"

Un programma è "*logicamente monotono*" se all'aumentare della dimensione del problema il risultato non cambia.

Esempio: un sistema distribuito che individua un deadlock al suo interno è *logicamente monotono*.

Una volta individuato il deadlock, anche con più nodi la soluzione non cambia.

Esempio: un sistema distribuito di Garbage Collection non è *logicamente monotono*.

Aggiungendo nodi al sistema, è possibile che all'interno di uno di essi ci sia un riferimento ad un oggetto che era stato classificato come non più in uso. Il risultato del sistema quindi **cambia** con l'aggiunta di nodi.

Il CALM Theorem è così un risultato costruttivo: individua una classe precisa di programmi che forniscono una garanzia. Non solo, individua anche uno strumento, la *logica monotona*, che può essere usato per costruire tali programmi.

Il problema è che tale classe di programmi non è molto popolata, e molti problemi interessanti non sono esprimibili con tali strumenti.

Non a caso, i CRDT svolgono un ruolo importante nel CALM Theorem, arrivando addirittura a caratterizzare le diverse garanzie di consistenza che si possono ottenere da un sistema.

APPROFONDIMENTI

PERSPECTIVES ON THE CAP THEOREM

(Gilbert, Lynch 2012)

`papers/l18/Brewer2.pdf`

THE BYZANTINE GENERALS PROBLEM

(Lamport, Shostak, Pease 1982)

`papers/118/byz.pdf`

THE PART-TIME PARLIAMENT

(Lamport 1998)

`papers/118/lamport-paxos.pdf`

PAXOS MADE SIMPLE

(Lamport 2001)

`papers/l18/paxos-simple.pdf`

IN SEARCH OF AN UNDERSTANDABLE CONSENSUS ALGORITHM (EXT. VER.)

(Ongaro, Ousterhout 2014)

`papers/l18/raft.pdf`

A COMPREHENSIVE STUDY OF CONVERGENT AND COMMUTATIVE REPLICATED DATA TYPES

(Shapiro, Preguiça, Baquero, Zawirski 2011)

`papers/118/techreport.pdf`

WHAT HAPPENED TO DPLS?

https://pwlconf.org/2017/heather-miller/

Heather Miller è professore ordinario di Computer Science alla Carnegie Mellon University.

In questo talk ripercorre una breve storia dei linguaggi per la programmazione distribuita, cercando il motivo del successo di Java e C++.

Il testo di riferimento del suo corso di programmazione distribuita è su GitHub:

https://github.com/heathermiller/dist-prog-book