Sistemi distribuiti e pervasivi

Francesco Tomaselli

5 aprile 2021

Indice

1 Introduzione ai sistemi distribuiti **2**
1.1 Definizioni . . . 2
1.2 Obiettivi . . . 2
1.3 Tipologie . . . 3
1.3.1 Computing systems . . . 3
1.3.2 Information systems . . . 4
1.3.3 Pervasive computing . . . 5

2 Architetture **5**
2.1 Architetture centralizzate . . . 5
2.1.1 Client server . . . 5
2.1.2 Event Bus . . . 6
2.2 Architetture decentralizzate . . . 6
2.2.1 Peer-to-peer . . . 6
2.3 Microservizi . . . 7

3 Modelli di comunicazione **7**
3.1 Comunicazione persistente e transiente . . . 7
3.2 Message-oriented communication . . . 8

4 Sincronizzazione **10**
4.1 Orologi fisici . . . 10
4.1.1 Metodi di sincronizzazione . . . 10
4.2 Orologi logici . . . 11
4.2.1 Algoritmo di Lamport . . . 11
4.3 Algoritmi di mutua esclusione . . . 12
4.4 Algoritmi di elezione . . . 13

1 Introduzione ai sistemi distribuiti

> *You know you have one when the crash of a computer you've never heard of stops you from getting any work done - Leslie Lamport*

1.1 Definizioni

Definizione classica Un sistema distribuito è una collezione di computer indipendenti che appaiono all'utente finale come un singolo sistema. Le macchine non hanno memoria condivisa e possono in qualche modo comunicare tra loro.

Interpretazione come middleware Da un punto di vista architetturale un sistema distribuito può essere visto come un middleware che si estende tra molte macchine, offrendo ad ognuna una qualche interfaccia.

Esempi Alcuni esempi di sistemi distribuiti possono essere:

- Un set di PC che comunicano tra loro con un processo che è assegnato dinamicamente
- Online multiplayer
- World Wide Web

1.2 Obiettivi

Gli obiettivi di un sistema distruibuito si possono ricondurre a garantire accesso delle risorse, trasparenza, apertura e scalabilità.

Trasparenza Esistono vari tipi di trasparenza da garantire in un contesto distribuito:

- *Accesso*: nasconde la rappresentazione dei dati e come gli utenti vi accedono
- *Locazione*: si nasconde dove una risorse è mantenuta
- *Migrazione*: non si mostra il fatto che una risorsa ha cambiato locazione
- *Rilocazione*: come il precedente ma quando la risorsa è in uso
- *Replicazione*: nasconde la replicazione delle risorse
- *Concorrenza*: nasconde il fatto che una risorsa è condivisa da più utenti
- *Fallimento*: non mostra fallimenti e relativi recovery del sistema

Solitamente non è possibile garantire tutti questi punti.

Apertura Un sistema distribuito aperto dovrebbe riuscire a fornire

- Interoperabilità
- Portabilità
- Possibilità di estensione

Può essere ottenuta tramite l'uso di protocolli standard, API, ...

Scalabilità La scalabilità è fondamentale in un constesto distribuito, la desiderata è che le prestazioni non peggiorino alll'estensione del sistema, bisogna perciò evitare la centralizzazione, in particolare di servizi, dati e algoritmi.
In algoritmo decentralizzato ad esempio:

- Nessuna macchina ha complete informazioni sul sistema
- Le decisioni sono prese guardando conoscenze locali
- Il fallimento di una singola macchina non pregiudica lo stato del sistema
- Non ci sono implicazioni implicite sull'esistenza di un clock globale

L'opposto di un sistema centralizzato.
Un esempio di scalabilità si trova nella divisione del DNS in zone, evitando quindi di avere un singolo name server centrale.

1.3 Tipologie

Esistono varie tipologie di sistemi distribuiti che variano a seconda degli scopi. SI definiscono i computing systems, information systems e pervasive computing.

1.3.1 Computing systems

Clusters collezione di workstations uguali o molto simili, collegate tra solo da una rete locale ad alta velocità, fanno girare lo stesso sistema operativo. Esistono due tipologie:

1. *Asimmetrico*: esiste un nodo master che coordina e controlla gli altri nodi, un esempio è Google Borg
2. *Simmetrico*: tutti i nodi hanno lo stesso software

Cloud computing Il cloud computing è un modello per fornire accesso a una rete di computer in modo conveniente e on demand, caratterizzato dalla facilità di riconfigurazione delle macchine. Alcune caratteristiche sono che:

- I nodi sono eterogenei
- Le connessioni sono eterogenee, in termini di capacità e di affidabilità
- On-demand self-service, ovvero la facile configurazione che si è discussa sopra
- La capacità di calcolo è accessibile tramite la rete, tramite meccanismi standard
- Le risorse sono condivise tra molti utenti, in modo da ottimizzare il loro utilizzo
- Elasticità rapida, ovvero le risorse sono facilmente assegnabili
- Esistono sistemi di misura, ovvero metriche che indicano l'utilizzo del sistema cloud

Esistono poi alcune categorie:

- Software as a Service: utilizzo di un software che gira in cloud
- Platform as a Service: deploy che utilizza tool del cloud provider
- Infrastructure as a Service: software che gira sfruttando l'infrastruttura cloud

Edge computing Nel corso degli anni alcuni nodi dei sistemi distruibuiti non sono più semplici computer ma possono essere sensori dispositivi che producono molti dati.

C'è bisogno di connetterli in tempo reale, ad esempio tramite una rete 5G, ed effettuare preprocessing prima di comunicare con il cloud. Si può pensare ad un solo strato intermedio tra sensori e cloud.

Fog computing Simile all'edge computing, ma potrebbero essre presenti più livelli tra i sensori e il cloud vero e proprio.

1.3.2 Information systems

Rappresentano un'altra tipologia di sistemi distruibuiti.

Database distribuiti Database distribuiti in cloud, oppure blockchain.

Transaction porocessing systems Lo scopo è quello di garantire le proprietà ACID in un contesto distribuito.

1.3.3 Pervasive computing

Un sistema distribuito pervasivo ha alcune caratteristiche che lo discostano da un sistema classico, in particolare si possono trovare nodi inconvenzionali, ad esempio smart objects vari e il principio di adattività, ovvero la capacità di adattarte il comportamento del sistema in base all'obiettivo del sistema.

Alcuni esempi possono essere l'uso di dispositivi smart in casa, oppure la guida autonoma.

2 Architetture

Un'architettura di un sistema distribuito definisce le entità del sistema, i pattern di comunicazione e come comunicano. Stabilisce inoltre il ruolo delle entità e come sono mappate sull'infrastruttura fisica.

2.1 Architetture centralizzate

Le architetture a seguire prevedono una certa centralità, nel primo caso si parla di un server che soddisfa le richieste, mentre nel secondo di un bus su cui si scrivono e leggono informazioni.

2.1.1 Client server

Architettura che prevede uno o più server e più client, il server gestisce le richieste e il client le manda. Un'interazione tipo prevede quindi la richiesta del client, l'attesa della risposta mentre il server la computa, e la ricezione. Si fa notare che esite un delay di comunicazione.

Nella progettazione di un sistema client server bisogna tenere in considerazione cosa eseguire sul server e cosa sul client.

Varianti Esistono estensioni al classico modello, si possono aggiungere ad esempio *proxy server* per fare caching, oppure avere un *multi-tier* client server, dove esiste una divisione dei nodi in base alla loro funzionalità.

Vertical e horizontal distribution Si parla di *vertical distribution* quando esiste un nodo per ogni funzionalità, offerta dal server, si definisce invece *horizontal distribution* la replica del server su più nodi, in modo da soddisfare richieste in contemporanea. Si può pensare a una combinazione di distribuzione verticale e orizzontale.

2.1.2 Event Bus

Si basa sul pattern di comunicazione *publish-subscribe*, in generale al verificarsi di un evento, un'informazione viene pubblicata sull'*event bus*, gestito da un *broker*, e quell'informazione ha un particolare argomento o topic. Esistono poi i *subscriber* che sono registrati ad un certo canale o topic, che leggono dall'event bus.

2.2 Architetture decentralizzate

Le architetture a seguire non prevedono un'entità centrale, i nodi solitamente hanno lo stesso ruolo.

2.2.1 Peer-to-peer

Tutti i nodi hanno le stesse capacità, ogni user può condividere le proprie risorse e non ci sono nodi centrali che orchestrano tutto.

Esistono varianti del modello peer-to-peer che prevedono nodi che fanno in qualche modo da server.

I problemi di questa architettura sono legati alla distribuzione delle risorse, in particolare gli obiettivi sono:

- Load balancing nell'accesso dei dati
- Avere disponibilità delle risorse, senza troppo overhead

Peer-to-peer middleware L'idea è quella di offrire in modo trasparente locazione e comunicazione delle risorse, e aggiunta e rimozione di risorse e nodi.

Overlay network Rete logica costruita sulla rete fisica, serve a garantire l'efficienza delle operazioni che deve garantire un peer-to-peer middleware.

Ad esempio deve fare routing delle richieste a livello applicativo, basato su identificatori globali.

Esistono due tipi di overlay network:

- *Strutturate*: network create in modo deterministico, utilizzano tabelle hash dsitribuite per garantire routing efficiente. Un esempio è *Chord* oppure *CAN*
- *Non strutturate*: utilizzano algoritmi probabilistici, ogni nodo ha una vista parziale della rete ed essa cambia nel tempo, in base allo scambio di informazioni tra i nodi, con algoritmi di *gossipping*. La ricerca si effettua chiedendo ai vicini, per un numero limitato di hop.

Nel primo caso un vantaggio è quello di invididuare sicuramente le risorse se presenti, inoltre esiste un numero limitato di messaggi, esiste però un costo di gestione, in aggiunta e rimozione dei nodi, vista la necessità di mantenere le strutture dati dei nodi.

Nel secondo caso invece, se un nodo sparisce dalla rete non si verificano particolari problemi, ma il numero di messaggi può essere molto alto, inoltre, essendo il numero di hop per la ricerca limitato, una risorsa presente in rete si potrebbe anche non raggiungere.

2.3 Microservizi

L'idea è quella di sppingere nella vertical distribution, separando tra loro le funzionalità da garantire. In questo modo si definiscono microservizi potenzialmente scritti in linguaggi di programmazione differenti, che comunicano tra loro tramite messaggi. Favorisce la manutenzione e l'aggiornamento delle singole parti, inoltre, alcune funzionalità possono essere replicate più di altre.

3 Modelli di comunicazione

Nei sistemi distribuiti i nodi comunicano tra loro tramite qualche mezzo, esiste una grande eterogeneità riguardo al tipo dei canali utilizzati, ma essi vengono utilizzati per scambiare messaggi.

Middleware Considerano la coda iso osi, si possono fondere sessione e presentazione in un unico livello middleware. In questo livello possono essere collocati i servizi di comunicazione di alto livello trattati a seguire.

Considerando ad esempio un client che effettua una richiesta ad un server, un middleware potrebbe gestire il messaggio mandato dal client e ad esempio fornire feedback sullo stato della richiesta, ad esempio la ricezione del server, il processing etc.

3.1 Comunicazione persistente e transiente

Un esempio intiuitivo di comunicazione persistente consiste nello scambio di lettere da post office a post office, i messaggi sono memorizzati nel transito, mentre un sistema transiente non prevede la memorizzazione dei messaggi in transito, un esempio è una chiamata telefonica.

Esistono molte varianti di queste comunicazioni, legate a quanto un client aspetta, a quando il server riceve il messaggio etc.

Modello persistente asincrono Il processo A manda un messaggio al processo B, il quale non è in esecuzione. Ad un certo punto B parte e riceve il

messaggio. Si prevede una memorizzazione intermedia. A non aspetta nessun riscontro da B, si parla quindi di comunicazione asincrona.

Modello persistente sincrono In questo caso il processo A manda una richiesta a B, aspettandosi però un riscontro.

Modello transiente asincrono In questo caso il messaggio inviato da A a B è ricevuto dal secondo solo se in esecuzione.

Modello transiente sincrono In questo caso il processo B è in esecuzione ma sta eseguendo altro. Quello che succede è che B riceve il messaggio, manda un ACK e finsice di processare quello che sta facendo. Poi inizia con la richiesta di A.

Delivery-based transiente sincrono In questo caso A si ferma fino a quando B non inizia ad elaborare.

Response-based transiente sincrono In questo caso A si ferma fino a quando B non finisce di elaborare.

3.2 Message-oriented communication

Berkeley sockets Tool di comunicazione introdotto con Berkeley Unix. Si possono considerare come comunicazione transiente.
Fanno uso di diverse primitive di sistema per creare una connessione tra due nodi:

- *Socket*: crea un endpoint. Facendo riferimento a Unix, si può pensare a una socket come un descrittore, simile ad esempio ai descrittori per standard input, output ed error. Alla creazione di una nuova socket si restituisce un descrittore sul quale scrivere e leggere
- *Bind*: si collega un indirizzo locale, ovvero una porta, a una socket
- *Listen*: segnala la disponibilità a ricevere messaggi
- *Accept*: si blocca il caller fino a quando non arriva una connessione
- *Connect*: tentativo di stabilire una connessione, c'è bisogno di ip e porta
- *Send*: invio di dati sulla socket
- *Receive*: ricezione di dati sulla socket
- *Close*: rilascio delle risorse

Queuing systems L'idea di base di questo sistema è avere delle code di messaggi, gestite da nodi intermedi detti router. Si garantisce la persistenza, ovvero, un nodo manda ad una coda, i router indirizzano il messaggio al destinatario e lo inseriscono nella loro coda. Tra le primitive del sistema si potrebbero trovare:

- *Put*: append del messaggio a una coda
- *Get*: prelevo il messaggio da una coda, primitiva bloccante
- *Poll*: controlla se una coda ha messaggi e rimuove il primo, non bloccante
- *Notify*: notifica quando un messaggio è inserito in una certa coda

In un sistema di questo tipo, il ruolo dei broker è quello di garantire la persistenza, ma anche quello di effettuare eventuali conversioni. Il secondo scenario accade quando due nodi che vogliono comunicare hanno convenzioni diverse.

Il sistema a code è preferibile quando :

- si vuole comunicazione ascincrona e persistente
- quando si può aumentare la scalabilità ammettendo ritardi di gestione delle richieste,
- quando i producer sono più veloci dei consumer
- quando si sta implementando un pattern pub-sub

Remote proceduce call L'idea è quella di nascondere la messaggistica necessaria ad effettuare una chiamata ad una procedura remota.

Non è possibile inviare al processo remoto il semplice indirizzo di memoria dei dati sul client, bensì serve mandare al processo remoto una funzione con parametri. La logica quindi si riassume in questi punti:

1. Arrivo ad un'istruzione da eseguire in remoto
2. Serializzazione dei parametri e invio nella rete del messaggio
3. Ricezione del server, deserializzazione ed esecuzione della funzione
4. Invio del risultato al client

L'implementazione di questo scambio di messaggi è offerta da uno standard, non bisogna "inventarsi" nulla al contrario della comunicazione con socket. Il passaggio da dati a messaggio e viceversa è chiamato:

- *Marshalling*: formattazione dei parametri in messaggio, prevede eventuale serializzazione, bisogna anche porre attenzione all'architettura di partenza e arrivo, potrebbero differire in termini di codifica
- *Unmarshalling*: procedure inversa di marshalling

Il binding tra client e server è trasparente, alcuni sistemi offrono la possiblità di scoprire dinamicamente quale server offre una determinata funzione.

4 Sincronizzazione

Clock interno Ogni nodo in un sistema distribuito ha un clock interno. Esistono alcuni problemi, ad esempio, considerando due eventi consecutivi su due macchine differenti, il secondo evento potrebbe avere tempo inferiore al primo, se il clock del nodo che l'ha eseguito è leggermente minore dell'altro nodo.

Questo tipo di problemi porta inevitabilmente a comportamenti inaspettati, è necessaria una sincronizzazione.

È possibile una sincronizzazione perfetta? Sostanzialmente no, non è possibile pensare di avere una sincronizzazione perfetta in un sistema distribuito, si punta ad una certa precisione.

4.1 Orologi fisici

Nel contesto degli orologi fisici, si fa riferimento al tempo UTC. Esso è calcolato osservando fenomeni naturali ed astronomici.

In particolare il tempo UTC fa riferimento al TAI, il tempo internazionale atomico, regolato aggiungendo *leap seconds* per allinearlo al tempo solare.

Clock lento e veloce Facendo riferimento all'UTC, si parla di clock perfetto se il rapporto tra clock e UTC equivale ad uno. Il clock invece è veloce se il rapporto è maggiore di uno e lento altrimenti.

Se il clock dei nodi di un sistema distribuito non è perfetto, saranno necessarie operazioni di sincronizzazione di tanto in tanto.

4.1.1 Metodi di sincronizzazione

Sincronizzazione tramite GNSS Per gli orologi fisici è possibile utilizzare il GNSS per capire la posizione attuale. Il tempo ottenuto tramite questo metodo, che sfrutta più satelliti, è preciso ai micro o anche ai nano secondi.

I satelliti hanno a bordo un orologio atomico. Il ricevitore GNSS manda un messaggio ai satelliti e, osservando il tempo di risposta e interpolando i segnali ricevuti, determina la posizione attuale e la deviazione dal tempo UTC. Ovviamente la posizione è approssimata come il tempo che si calcola.

Algoritmo di Cristian e NTP L'idea è quella di chiedere il tempo preciso ad un server, tenendo in considerazione la latenza della richiesta.

Quindi, il client manda la richiesta al server NTP, esso manda la risposta, ma ovviamente esistono ritardi. I messaggi contengono perciò timestamp, sia lato client che lato server, si stimano i ritardi di comunicazione e si calcola il tempo attuale di conseguenza. Il procedimento viene ripetuto più volte per stimare bene la latenza e il risultato è preciso ai millisecondi.

Algortimo di Berkeley Non sempre è necessario sincronizzare gli orologi di un sistema distribuito con il tempo esterno.

Esiste quindi un server che fa da *time deamon*, esso manda il suo orario a tutti i nodi del sistema, essi rispondono con la differenza del loro tempo rispetto a quella del time deamon. Il nodo deamon calcola la media degli orari del sistema e manda il risultato a tutti.

La latenza del procedimento è ignorata, si assume che esistano comunicazioni pressochè istantanee.

Inoltre, il tempo di un nodo non si riporta mai indietro, visti i possibili problemi di inconsistenza che il procedimento creerebbe. Si rallenta semplicemente il tempo fino a quando non si allinea con quello del nodo deamon.

4.2 Orologi logici

In molti casi non è necessario che i nodi siano sincronizzati rispetto ai loro clock fisici, potrebbe essere abbastanza conoscere un'ordine parziale di eventi sul sistema distribuito.

Idea Quello che serve è un modo per far concordare i nodi sulla sequenza di certi eventi, dove un evento è interno o l'invio o ricezione di un messaggio.

Si mantiene un contatore e ogni volta che accade un evento esso viene aumentato. Si nota che i contatori dei nodi potrebbero avere valori differenti.

4.2.1 Algoritmo di Lamport

L'idea dell'algoritmo è che, dati due eventi A, B l'espressione $A \to B$ indica la relazione che A è accaduto prima di B, ed essa è transitiva.

Dato $C(a)$ il valore del clock logico assegnato al momento in cui a è accaduto, l'obiettivo dell'algortimo è far valere la seguente espressione

$$A \to B \implies C(a) < C(b)$$

Logica L'aggiornamento del counter C_i per il processo P_i avviene come segue:

1. Prima di eseguire un evento, il processo P_i esegue $C_i \leftarrow C_i + 1$

2. Se il processo P_i manda un messaggio m a P_j, setta il timestamp di m con il valore del counter attuale, $ts(m) \leftarrow C_i$

3. P_j quando riceve il messaggio agggiorna il suo contatore secondo la logica $C_j \leftarrow \max(C_j, ts(m)) + 1$

Si nota che non è detto che si riesca a dare un ordine per ogni coppia di eventi, ma solo tra quelli collegati da un messaggio.

Implementazione All'interno del sistema si può pensare all'Implementazione dell'algoritmo come middleware.

Problematiche L'algoritmo può essere modificato aggiungendo un contatore, ovvero, il processo P_i assegna all'evento e, $C_i(e).i$ dove i è il suo indice.

Se due eventi nel sistema hanno lo stesso C, si riesce a disambiguare scegliendo in ordine l'evento con indice del processo minore.

Applicazioni Una possibile applicazione dell'algoritmo è quella della replica di database,

4.3 Algoritmi di mutua esclusione

È necessaria mutua esclusione quando esistono risorse comuni a cui non possono accedere più nodi in contemporanea.

Soluzione centralizzata Una soluzione semplice consiste nell'usare un nodo coordinatore che regola l'accesso alla risorsa condivisa. Si può utilizzare una coda gestita dal coordinatore.

Soluzione distribuita L'idea è che se un processo P deve utilizzare una risorsa R, costruisce un messaggio contenente il suo id, il nome della risorsa e un timestamp. Manda poi il messaggio a tutti, incluso se stesso.

Si assume che esista ordine totale degli eventi, tramite ack.

Si pensi ora a due processi 0 e 2 che vogliono accedere alla risorsa R e il processo 3 che non fa nulla.
P0 manda un messaggio con timestamp 8, P2 uno con timestamp 12. P3 risponde OK ad entrambi, mentre, P2 risponde OK a P0, non il viceversa, visto che il timestamp di P0 è minore di P2.

A questo punto P0 aggiunge nella sua coda P2 e ottiene la risorsa. Al termine manda il messaggio di OK a P2, che è l'unico processo nella sua coda che sta aspettando il messaggio per accedere alla risorsa condivisa.

Problematiche distribuite Una mancanza di risposta potrebbe essere un crash, inoltre il richiedere l'intervento di tutti i nodi ogni volta che si vuole accedere ad una risorsa non è | desiderabile.

4.4 Algoritmi di elezione

...