

# Sistemi distribuiti e pervasivi

Francesco Tomaselli

27 marzo 2021

## Indice

<b>1</b>	<b>Introduzione ai sistemi distribuiti</b>	<b>2</b>
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
<b>2</b>	<b>Architetture</b>	<b>5</b>
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
<b>3</b>	<b>Modelli di comunicazione</b>	<b>7</b>
3.1	Comunicazione persistente e transiente . . . . .	7
3.2	Message-oriented communication . . . . .	8

# 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 distribuito 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 risorsa 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 contesto distribuito, la desiderata è che le prestazioni non peggiorino all'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 loro 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 distribuiti 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 essere presenti più livelli tra i sensori e il cloud vero e proprio.

### 1.3.2 Information systems

Rappresentano un'altra tipologia di sistemi distribuiti.

**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 esiste 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 distribuite 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 *gossiping*. La ricerca si effettua chiedendo ai vicini, per un numero limitato di hop.

Nel primo caso un vantaggio è quello di individuare 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 spingere 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 intuitivo 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 finisce 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*: preleva 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 asincrona 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 procedure 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 possibilità di scoprire dinamicamente quale server offre una determinata funzione.