Sistemi Distribuiti

Leonardo Valente

March 14, 2022

Contents

1 Introduzione

1.1 Che cosa è un sistema distribuito?

Ci possono essere diverse definizioni di "Sistema Distributo".

- Definiamo un sistema distribuito come un insieme di componenti Hardware e Software localizzati in una rete di computer che comunicano e coordinano le loro azioni solo passandosi messaggi.
- Un "Sistema Distribuito" è un insieme di elementi di computazione autonomi che appaiono all'utente come un singolo sistema coerente.

1.2 Caratteristiche

Gli "elementi di computazione autonoma" di un SD, anche chiamati "Nodi" sono i device hardware oppure i processi software. Il fatto un SD debba sembrare un singolo sistema all'utente implica il fatto che tra i nodi ci deve essere un sistema di collaborazione.
Quindi ogni nodo in quanto autonomo avrà la sua singola nozione di tempo. Non esiste un clock globable per tutti i nodi.
Questo porta quindi a problemi di sincronizzazione e di coordinamento.

La parola chiave resta sempre: **Trasparenza**

E' inevitabile il fatto che in qualsiasi momento solo parte del sistema fallirà. Nascondere questi fallimenti e il loro recupero è molto spesso difficile e in generale impossibile da nascondere.

Gestione della memoria?

- Non c'è memoria condivisa.
- Comunicazione via scambio di messaggi.
- Ogni componente conosce solo il proprio stato e può sondare lo stato degli altri

Gestione dell' esecuzione?

- Ogni componente è autonomo.
- Il coordinamento delle attività è importante per il funzionamento di un sistema formato da più componenti.

Gestione dell tempo?

- Non c'è un clock globale.
- Non c'è possibilità di scheduling globale.

Tipi di fallimenti

- Fallimenti indipendenti dei singoli nodi.
- Non c'è fallimento globale.

1.3 Gruppi

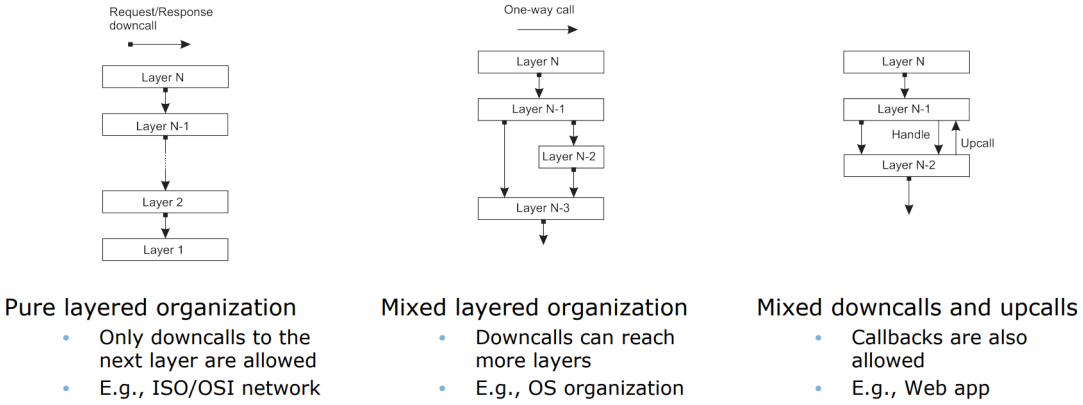
I gruppi possono essere **aperti** (tutti i nodi possono partecipare) o **chiusi** (solo membri selezionati possono partecipare).

2 Architetture Software

Una architettura software definisce la struttura del sistema, le interfacce tra i componenti e i pattern di interazione.
Ci possono essere diversi stili di architettura per un SD.

- **Architetture a strati** (layered)
  Ho un livello superiore che nasconde il lavoro effettuato dal livello sottostante.
- **Architetture a livelli**
  Applicazioni client server.
- **Architetture basate sugli oggetti**
- **Architetture basate su eventi**
  Applicazioni web dinamiche basate su callback (AJAX).

2.1 Tipi di layer



- Il primo caso è molto semplice. Abbiamo a disposizione N livelli di layer, ognuno che comunica solo ed esclusivamente con il livello sottostante.
- Il secondo caso invece potrebbe risultare un po più complicato. Per spiegarlo usiamo un semplice esempio: supponiamo che il nostro programma effettui una operazione di divisione per 0 (zero). Ovviamente sappiamo che ciò non è possibile, quindi passerà il compito di risolvere questa eccezione all'exception handler e successivamente tornerà a fare quello che doveva fare. Invece se effettuiamo una normale operazione che non genera eccezzioni ovviamente non dovrà fare la parte dell'exception handler.
- Per il terzo caso basti immaginare a come funziona AJAX o più in generale il funzionamento di una web app. Quindi chiamate al server con eventuale risposta e aggiornamento della UI.

2.2 Diversi tipi di Sistemi Distribuiti

E' importante differenziare i 3 tipi principali di SD.

- DOS (Distributed Operating System)
- NOS (Network Operating System)
- Middleware

Distributed Operating System

L'utente non è a conoscenza della molteplicità delle macchine che compongono il sistema.
I dati possono essere spostanti in modo intero o parziale, così come le operazioni di computazione.
Un processo può essere migrato interamente o in modo parziale su diversi siti, facendo così avremo un effetto di **Load balancing**, che ci permette di distribuire il carico di lavoro su più macchine, e di **Compuatation speedup** (i sottoprocessi possono essere eseguiti concorrentemente su più siti). Però il processo potrebbe aver bisogno di un determinato hardware (**Hardware preference**) oppure di un determinato software (**Software preference**). Avendo la possibilità di migrare il processo questo può essere possibile. Infine il processo può essere eseguito in modo remoto, invece che trasferire i dati in locale.

Network Operating System

L'utente è a conoscenza della molteplicità delle macchine che compongono il sistema.
NOS permette operazioni esplicite di comunicazione: **Socket**. (Comunicazione diretta tra processi.)
L'accesso alle risorse sulle varie macchine è effettuata in modo esplicito tramite:

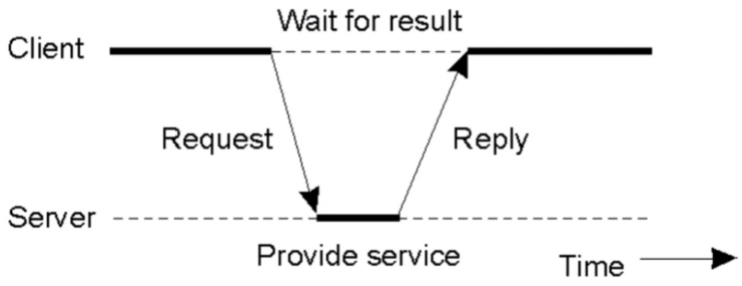
- Remote logging (telnet, ssh)
- Remote desktop
- FTP

Middleware

Il compito del middleware è quello di implementare i servizi per renderli trasparenti all'applicazione.

- Definisce e offre un modello di comunicazione che nasconde i dettagli dei messaggi passati.
- Definisce e offre un servizio automatico per il salvataggio dei dati (su file system o DB).
- Definisce e offre un modello persistente per garantire consistenza su operazioni di lettura e scrittura (di solito su DB).
- Definisce e offre modelli di protezione nell'accesso ai dati e servizi.

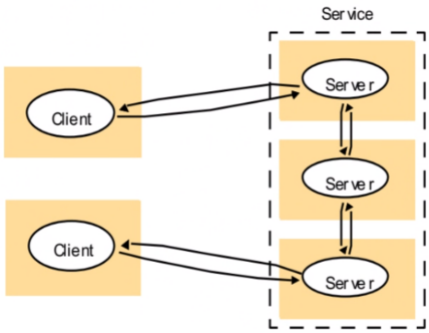
3 Il Modello Client-Server



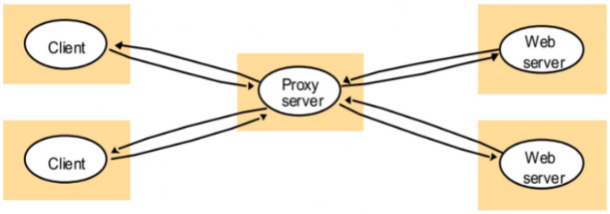
Di base questa architettura prevede che un **client** acceda ad un **server** con una richiesta e che il server risponda con un risultato.
Come possiamo notare dall'immagine, chiaramenta la richiesta con annessa risposta non è immediata, dovrà trascorrere un determinato quanto di tempo affinchè il server riesca a soddisfare la richiesta del client.

Ci possono essere diversi tipi di modelli.

- **Accesso a Server multipli**.
  Il client accede ad un server che a sua volta può accedere ad un altro server.



- **Accesso via Proxy**
  Il *Proxy* è un tipo di server che funge da intermediario per le richieste da parte di client alla ricerca di risorse su altri server.
  Il Server Proxy è molto utile per fornire l'anonimato durante la navigazione.



3.1 Problemi Fondamentali

In generale, ogni Sistema Distribuito deve aver a che fare con 4 problemi fondamentali:

- **Namig** (Identificare la controparte)
  Chi è la mia controparte? Dobbiamo assegnare dei nomi
- **Access point** (Accedere alla controparte)
  Come posso accedere ad una risorsa remota o un processo?
- **Protocol** (Comunicazione)
  Come posso scambiare messaggi? Bisogna mettersi d'accordo sul formato
- **Still an open issue** (Comunicazione)
  Come posso capire il contenuto di un messaggio?

3.2 Trasparenza

Il concetto fondamentale alla base di un buon Sistema Distribuito è, appunto, la trasparenza. Per cui, per esempio, il come un particolare dato venga rappresentato e la metodologia di accesso a quel dato sono *trasparenti* all'utente, non li vede. Così come anche la *location*. Non sappiamo dove quel dato risiede fisicamente.

Però, avere un **grado** di trasparenza troppo elevato potrebbe risultare eccessivo. Per esempio:

- Alcune *latenze di comunicazione* non possono essere nascoste.
- Nascondere i fallimenti del sistema e dei nodi è talvolta **impossibile**.
- Esporre le distribuzioni potrebbe essere alcune volte una buona pratica. Se un server non dovesse rispondere ad una chiamata per troppo tempo, bisogna riportare il fallimento per dare un feedback all'utente di cosa sta accadendo e di agire di conseguenza.
- Inoltre, la trasparenza completa ha un costo elevato (e.g. Mantenere le repliche di tutti i dati esattamente "up-to-date" con il master)

3.2.1 Information Hiding

L'Information Hiding è il principio che sta alla base dell'*Ingegneria del Software*.
E' importante fare una distinzione tra:

- **cosa** un servizio o un sistema mette a disposizione
  definisce l'*Application Programming Interface* (API) dei componenti o del sistema.
- **come** un servizio è stato implementato e distribuito
  definisce come il tool adatto per quel specifico problema.

Queste interfacce però devono essere sviluppate seguendo certi criteri, quindi devono mantenere una struttura che segua i principi prestabiliti, deve essere completa (mettere a disposizione tutto quello che serve) e neutrale.

3.3 Il concetto di Protocollo

Per poter capire le richieste e formulare le risposte i due processi devono concordare un **protocollo**.
I protocolli definiscono il **formato**, l'**ordine** di invio e di ricezione dei messaggi, il **tipo dei dati** e le **azioni** da eseguire quando si riceve un messaggio.
Alcuni esempi di protocolli sono:

- HTTP - HyperText Transfer Protocol
- FTP - File Transfer Protocol
- SMTP - Simple Mail Transfer Protocol

4 Le Socket

I processi sono i programmi in esecuzione, che sono aree di memoria gestite dal Sistema Operativo. Ogni processo comunica attraverso dei **canali**, che controllano i *flussi di dati in* ***entrata*** *e* ***uscita***.
Dall'esterno, ogni canale è identificato da un numero intero detto **porta**.

Le **socket** sono dei particolari canali per la comunicazione tra *processi* che non condividono memoria.
Per potersi connettere o inviare dati ad un processo A, un processo B deve conoscere la macchina (*host*) che esegue A e la porta a cui A è connesso. (*Well known port (?)*)

4.1 Servizi di trasporto di dati

TCP

Il servizio *TCP* è un protocollo **orientato alla connessione**, ovvero che il *client* invia al *server* una richiesta di connessione, e il server risponde di conseguenza.
TCP è famoso per il suo **trasporto affidabile**, tra la comunicazione tra processi.
Possiede anche un **controllo di flusso** (il mittente rallenta per non sommergere il ricevente) e un **controllo della congestione** (il mittente rallenta quando la rete è sovraccaricata).

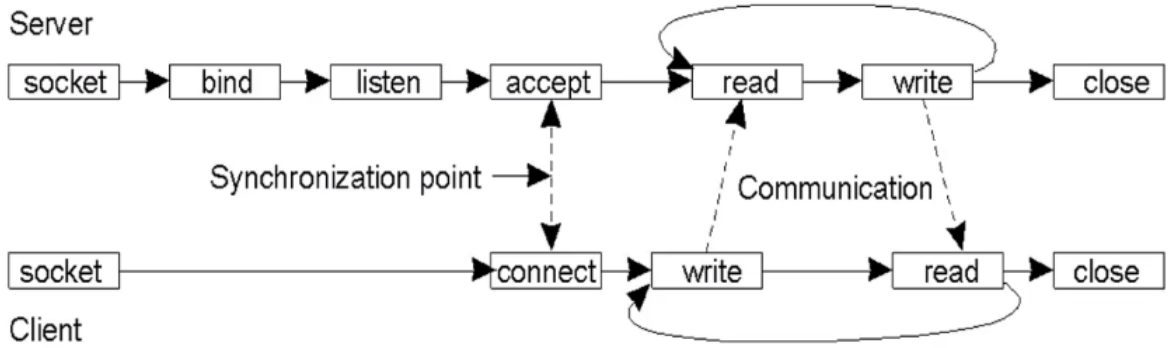
UDP

UDP è un protocollo che **non** garantisce la ricezione del messaggio, e in generale **non** possiede tutte le funzionalità di sicurezza del TCP.
Quindi, *perchè esiste UDP?*
UDP può essere molto utile per le applicazioni che tollerano perdite parziali (ad esempio un servizio di video streaming, anche se perdiamo qualche frame non ci accorgiamo nemmeno) a vantaggio delle prestazioni.

Le socket non sono altro che delle API per accedere ai servizi di trasporto TCP e UDP.

4.2 Comunicazione via socket

La comunicazione TCP/IP avviene attraverso **flussi di byte** dopo una **connessione esplicita**, tramite normali *System Call* di read/write.



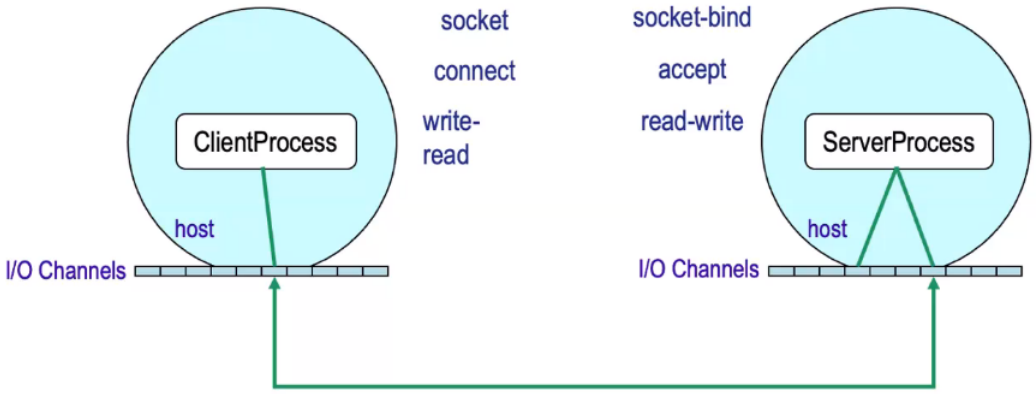
- **Socket**: viene creata un nuovo canale socket.
- **Bind**: viene associato un *local address (porta)* alla socket per l'identificazione del processo
- **Listen**: si mette in ascolto di nuove connessioni
- **Connect**: il client invia una richiesta di connessione al server
- **Accept**: accetta la connessione del client.
  Crea una nuova porta per gestire la **comunicazione** (dedicata).
- **Write**: invia dei dati attraverso il canale
- **Read**: riceve dei dati attraverso il canale
- **Close**: chiude la connessione

La **comunicazione** è un ciclo continuo di *read* e *write* tra il client e il server fino a che non viene chiusa la connessione.
Le socket trasportano degli *stream*, quindi non esiste il concetto di messaggio, è appunto un flusso continuo di dati.
Però la lettura e la scrittura avvengono per un numero arbitrario di byte.

Connect e **Accept** "bloccano" il sistema (rispettivamente il client e il server) fino a che non viene stabilita una connessione (vengono messi in attesa).

Perchè non viene creata solo una porta per la comunicazione ma ne vengono create molteplici?

Per gestire e identificare le varie connessioni che sono state create tra i client che si sono collegati al server.



4.3 Progettare una Applicazione con le Socket

- **Client**: L'architettura è più semplice di quella di un server:
  di solito è una applicazione che usa una socket anzichè un altro canale I/O
- **Server**: L'architettura prevede che:
  - venga creata una socket con una porta nota per accettare le richieste di connessione
  - entri in un ciclo infinito in cui alternare:
    * attesa/accettazione di una richiesta di connessione da un client
    * ciclo lettura-esecuzione
    * chiusura connessione

5 Architettura dei Server

I server possono essere:

- **Iterativi**: soddisfano una richiesta alla volta
- **Concorrenti a processo singolo**: simulano la presenza di un server dedicato
- **Concorrenti multi-processo**: creano server dedicati
- **Concorrenti multi-thread**: creano thread dedicati

5.1 Server Iterativi

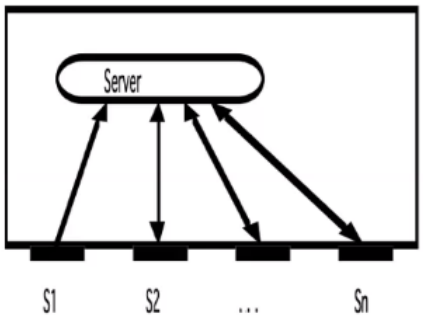
Al momento di una richiesta di connessione, il server crea una socket temporanea per stabilire una connessione diretta con il client.
Eventuali ultieriori richieste, verranno accodate alla porta nota per essere soddisfatte in seguito.

Questa architettura ha il **vantaggio** che è molto semplice da progettare, **però** viene servito un client alla volta, mettendo appunto in attesa gli altri.

5.2 Server Concorrenti

Un server concorrente può gestire più connessioni client.

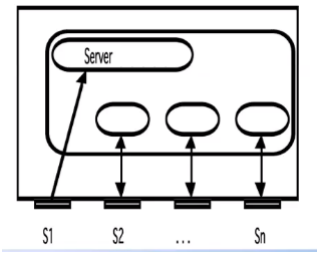
- **Monoprocesso**: Esistono delle funzioni (*select in C*) che vanno a selezionare i canali pronti all'uso.



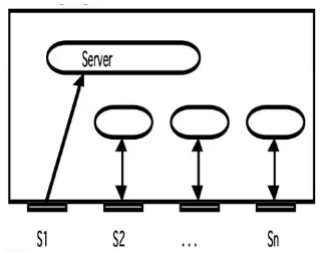
S1 è la socket per accettare le richieste di connessione, le altre sono le connessioni individuali.

Nei server monoprocesso, gli utenti condividono lo stesso spazio di lavoro, quindi adatto per le applicazioni cooperative che prevedono la modifica dello stato

- **Multi-thread**:
  Ho un processo singolo, e all'interno dei piccoli sotto processi (Thread) che simulano un processo a se



- **Multi-processo**: Ho degli effettivi processi clone, quindi veri e propri server "fisici", non simulati.



Nei server multiprocesso, ogni utente ha uno spazio di lavoro autonomo, quindi adatto per le applicazioni che non modificano lo stato del server, oppure per quelle applicazioni che prevedono una modifica solamente al proprio spazio di lavoro dedicato.

Per quanto riguarda la funzione di *select*, essa permette di gestire in modo non bloccante i diversi canali di I/O.
(Una istruzione *bloccante* è quando il sistema attende la conclusione dell'operazione richiesta prima di restituire il controllo al chiamante)