Appunti di Sistemi Distribuiti

Matteo Gianello

8 gennaio 2014

Indice

1 Introduzione

A partire dalla metà degli anni '80, grazie a due innovazioni tecnologiche si fecero diversi passi avanti nell'uso dei calcolatori. La prima di queste innovazioni fu lo sviluppo di microprocessori potenti; la seconda grande innovazione fu l'invenzione delle reti di computer con l'introduzione delle **LAN** (*Local Area Network*) che consentirono a centinaia di macchine di essere connesse le une alle altre e permisero lo scambio di piccole quantità di informazioni in pochi microsecondi. Il risultato di questa innovazione tecnologica è che oggi mettere insieme una grande quantità di computer tramite una rete ad alta velocità è diventato molto semplice. Questo tipo di sistemi sono solitamente chiamate *reti di computer* o **sistemi distribuiti**.

1.1 Definizione di sistema distribuito

Esistono diverse definizioni di *Sistema distribuito* ma tutte quante sono abbastanza insoddisfacenti. Daremo ora una prima definizione che è sufficente per i nostri scopi:

Un sistema distribuito è una collezione di computer indipendenti che appare ai propri utenti come un singolo sistema coerente

Da questa definizione possiamo ricavare diversi caratteristiche di un sistema distribuito, la prima è che i sistemi distribuiti sono costituiti da componenti autonomi; la seconda è che gli utenti, siano essi persone o altri programmi, vedono il sistema come un'unica entità. Il che significa che i diversi componenti devono in qualche modo collaborare.
Quello che non viene specificato in questa definizione è il tipo di computer usati per i componenti ne come questi sono interconnessi.
Le caratteristiche più importanti dei sistemi distribuiti sono il fatto che le differenze tra i vari computer e le loro modalità di comunicazione risultano per lo più nascoste agli utenti finali. Inoltre gli utenti possono interagire con un sistema distribuito in modo *consistente* e *uniforme* ovvero indipendentemente da dove e quando avviene l'interazione.
Teoricamente i sistemi distribuiti dovrebbero essere facilmente espandibili e scalabili, inoltre, i sistemi distribuiti sono di norma sempre disponibili anche se alcune sui parti sono momentaneamente fuori uso.
Allo scopo di supportare reti eterogenee e sistemi operativi differenti alle volte si introduce uno strato software tra lo strato di applicazione e i diversi sistemi operativi, questo strato è chiamato **middleware** come mostrato in figura 1.1.

1.2 Obiettivi

La possibilità costruire sistemi distribuiti non implica che tutti i sistemi debbano essere costruiti come sistemi distribuiti. Per far si che sia utile progettare e costruire un sistema distribuito dobbiamo rispettare alcune caratteristiche. un sistema distribuito dovrebbe:

- rendere le risorse facilmente accessibili,
- nascondere il fatto che le risorse sono distribuite sulla rete,
- essere aperto,
- essere scalabile.

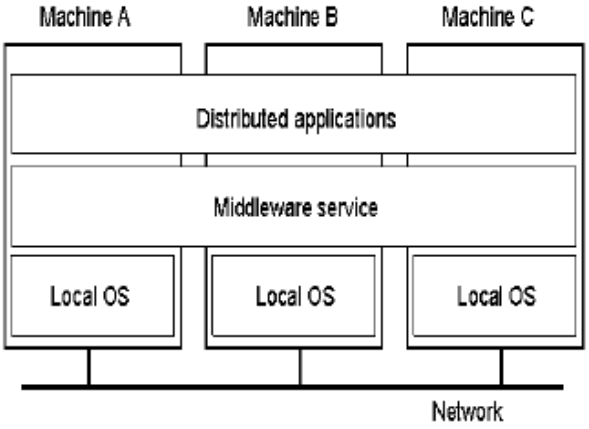

Figura 1: Schema di un middleware

1.2.1 Accessibilità delle risorse

L'obiettivo principale di un sistema distribuito è quello di rendere facile l'accesso alle risorse remote e condividerle in maniera efficiente e controllata.
Ma che cosa intendiamo per risorse? Con il termine *risorse* possiamo indicare qualsiasi cosa, alcuni esempi tipici sono stampanti, computer, dati, file, pagine web o intere reti. Le ragioni che portano a voler condividere le risorse sono molteplici, la prima è sicuramente quella economica, pensiamo ad esempio a ricercatori che condividono un supercomputer o ad una stampante condivisa in un ufficio. Inoltre, la connessione di più utenti facilita la collaborazione come avviene nei **groupware** dove gruppi di persone lavorano insieme anche stando in diverse parti del mondo.
Tutto questo incremento di connessione e collaborazione dovrebbe portare però ad una necessaria crescita anche in termini di sicurezza, anche se nella pratica attuale tale incremento nei sistemi di sicurezza non è ancora avvenuto; non è raro trovare sistemi in cui password e altre informazioni sensibili sono inviate come testo in chiaro. Altri problemi legati alla sicurezza sono l'aumento delle *junk mail* o mail di *spam* e l'invio e la raccolta di informazioni riguardanti l'utente per creare un profilo mentre è connesso.

1.2.2 Trasparenza

Uno degli obiettivi principali in un sistema distribuito è quello di nascondere che i processi e le risorse sono distribuiti. Un sistema in grado di presentarsi come un singolo computer è detto **trasparente**.
Possiamo catalogare la trasparenza in diversi tipi in quanto qesto concetto può riguardare molti aspetti di un sistema distribuito.
La **trasparenza all'accesso** riguarda le differenze nella rappresentazione dei dati e la modalità di accesso alle risorse da parte degli utenti. Ovvero si desidera nascondere le differenze nelle macchine e trovare un accordo nella rappresentazione dei dati. Un altro importante tipo di trasparenza è la **trasparenza di ubicazione** che si prefigge l'obiettivo di nascondere agli utenti la localizzazione di una risorsa. I *nomi* in questa tipo di trasparenza giocano un ruolo importante in quanto è possibile raggiungere tale trasparenza assegnando ad ogni risorsa un nome logico indipendente dalla sua locazione, un esempio di tale tecnica sono gli *URL*.
Alcuni sistemi distribuiti che consentono lo spostamento delle risorse senza compromettere la possibilità di accesso devono fornire la **trasparenza alla migrazione**. Nel caso in cui le risorse possono essere spostate *durante* l'utilizzo senza che utenti o applicazioni notino tale spostamento

si deve garantire anche la **trasparenza al riposizionamento**.
La **trasparenza alla replica** riguarda la possibilità di fornire una o più copie della stessa risorsa per aumentarne la disponibilità e migliorare le prestazioni, tutto questo nascondendo all'utente il fatto che la risorsa è replicata.
Come già detto l'obiettivo principale dei sistemi distribuiti è la condivisione di risorse, ma questa porta in alcuni casi ad avere una condivisione di tipo *competitivo* ovvero, più utenti vorrebero accedere alle stesse risorse (es. una tabella di un database) tutto ciò deve essere evitato tramite la **trasparenza alla concorrenza** che deve lasciare la risorsa in uno stato consistente. Questa consistenza può essere ottenuta tramite diverse meccanismi tra cui ad esempio il *locking* nel quale gli utenti ottengono a turno l'accesso esclusivo ad una risorsa.
Per introdurre l'ultimo tipo di concorrenza partiamo da un'altra definizione di sistema distribuito

Ne apprendi l'esistenza quando il crash di un computer di cui non hai mai sentito parlare ti impedisce di portare a termine qualunque lavoro

Questa definizione pone un altro aspetto importante della progettazione di un sistema distribuito, quello della gestione dei guasti; rendere un sistema **trasparente ai guasti** significa far si che un utente non si renda conto che una risorsa smette di funzionare correttamente. La difficoltà più grande è distinguere quando una risorsa è morta da quando è semplicemente molto lenta.
Sebbene in generale si preferisce avere sistemi trasparenti ci sono situazioni in cui nascondere completamente agli utenti la distribuzione del sistema non è una buona idea. Come nel caso si voglia contattare un servizio che sta dall'altra parte del mondo e si voglia una risposta in un tempo inferiore ai 35ms; o quando si vuole che due repliche siano sempre consistenti, nel caso di server in due continenti diversi un aggiornamento potrebbe richiedere alcuni secondi.
Il problema principale che limita però la trasparenza è la trasparenza stessa, infatti, ammettendo che la completa trasparenza di un sistema è *impossibile* è *saggio* cerca di ottenerla a tutti i costi? Rendere la distribuzione esplicita può aiutare gli utenti a capire eventuali comportamenti *anomali* del sistema.

1.2.3 Apertura

un altro obiettivo dei sistemi distribuiti è l'apertura. un sistema distribuito **aperto** è un sistema che offre servizi rispettando delle regole standar che descrivono la sintassi e la semantica dei servizi stessi.
Nei sistemi distribuiti i servizi sono descritti tramite **interfacce** per lo più utilizzando un linguaggio denominato *IDL (interface description language)* che però descrive soltanto la sintassi delle interfacce, ovvero, il nome delle funzioni e i tipi di parametri, i valori di ritorno o le possibili eccezioni sollevate. Per la descrizione di che cosa fa il servizio, invece, si usa solitamente il linguaggio naturale.
Se l'interfaccia è ben specificata un processo che ha bisogno di una determinata interfaccia può comunicare con un altro processo che implementa tale interfaccia; inoltre, consente a due processi distinti di implementare tale interfaccia in due modi completamente diversi, il che porta a due sistemi distribuiti che però operano allo stesso modo.
Una specifica però deve essere *completa* e *neutrale*, completa significa che viene specificato tutto ciò che è necessario per realizzare un'implementazione, ma ottenere la completezza è molto difficile e perciò di solito un programmatore deve aggiungere dettagli specifici dell'implementazione. Per neutrale si intende, invece, che la specifica non deve imporre dettagli sull'implementazione. Completezza e neutralità portano ad altre due importanti caratteristiche che i sistemi distribuiti dovrebbero soddisfare, **interoperabilità** che significa che due implementazioni di vendor diversi

possono collaborare e coesistere basandosi unicamente su di uno standar comune. **Portabilità** indica la possibilità di eseguire un applicazione scritta per un sistema distribuito A su di un sistema distribuito B senza dover apportare modifiche all'applicazione.
Infine un altro obiettivo che i sistemi distribuiti dovrebbero prefissarsi è che l'aggiunta o la sostituzione di componenti dovrebbe risultare facile e non influire sui componenti già presenti; questa caratteristica sta ad indicare che il sistema distribuito è **ampliabile**. Per ottenere la flessibilità in un sistema aperto è fondamentale che esso sia organizzato come un insieme di componenti piccolo e e facilmente sostituibile e adattabile. Ma questo comporta fornire le interfacce non solo dei componenti che si interfacciano direttamente con gli utenti ma anche dei componenti interni.

1.2.4 Scalabilità

La scalabilità sta diventando uno degli aspetti più importanti dei sistemi distribuiti a causa della grande diffusione di internet. Esistono diversi tipi di scalabilità la prima si ha quando un sistema è scalabile rispetto alla sua dimensione il che significa che possiamo aggiungere utenti e risorse. Un sistema può essere scalabile geograficamente quando utenti e risorse sono situati in luoghi molto lontani. Ed, infine, un sistema può essere scalabile anche dal punto di vista dell'amministrazione ovvero quando comprende molte infrastrutture indipendenti rimane comunque facile da gestire.
La scalabilità richiede di affrontare molti problemi. Prendiamo ad esempio la scalabilità rispetto alla dimensione, alcuni servizi in un sistema distribuito possono essere forniti da un unico server questo fa si che aggiungendo utenti quel particolare server diventa un collo di bottiglia per l'intero sistema. A volte l'uso di un solo server è inevitabile come nel caso della gestione di dati sensibili. Per quanto riguarda la scalabilità a livello geografico anch'essa comporta innumerevoli problemi, infatti, la maggior parte dei sistemi distribuiti che lavorano su LAN si basano sulla comunicazione **sincrona** nella quale un *client* richiede una risorsa e resta in attesa che tale risorsa sia disponibile. Questo meccanismo non è applicabile per sistemi globali nei quali la comunicazione tra due computer può richiedere anche qualceh millisecondo. In oltre si deve tener conto che la comunicazione su WAN(*wide area network*) è inaffidabile e praticamente sempre punto a punto (*pont-to-point*). Al contrario le reti locali più affidabili permettono anche il *broadcasting* ovvero l'invio simultaneo a tutte le macchine delle rete dello stesso messaggio.
Dopo aver visto i problemi di scalabilità ci chiediamo come risolvere tali problemi nei sistemi distribuiti. I problemi di scaling nei sistemi distribuiti si presentano come problemi nelle prestazioni dovuti alle limitate capacità dei server. Ad oggi esistono soltanto tre tecniche di *scaling*: nascondere le latenze, la distribuzione e la replica.
Nascondere le latenze permette di ottenere la scalabilità geografica; l'idea di base è quella di limitare il più possibile i tempi di attesa delle risposte dai servizi remoti. Alcune possibili soluzioni sono anticipare il più possibili la richiesta al server remoto e durante l'attesa della risposta svolgere qualche altra operazione; questo tipo di comunicazione è detta **comunicazione asincrona**. Quando arriva una risposta l'applicazione si interrompe e viene richiamato un gestore (*handler*) speciale per completare la richiesta sollevata. Inoltre la comunicazione asincrona è spesso utilizzata nei sistemi *batch* e nelle applicazioni *parallele*.
Un'altra soluzione è quella di eseguire un *thread* per la richiesta il quale, anche se si blocca, permette agli altri thread di proseguire.
Esiste una classe di applicazioni, però, che non può utilizzare la comunicazione asincrona, questo tipo di applicazioni sono le applicazioni *interattive* nelle quali il client dopo aver effettuato una richiesta ad un server remoto non ha niente di meglio da fare che aspettare la risposta. In questo caso l'unica cosa da fare è limitare il tempo di attesa e per fare ciò solitamente si sposta il carico

di lavoro computazionale che solitamente è svolto dal server sul client. Come ad esempio nel caso di compilazione di *form* per un accesso alla base di dati. Si può inviare al server ogni campo della form per poi aspettare dal server la conferma della correttezza dei dati, oppure, più efficiente è far controllare direttamente al client la correttezza dei dati ed inviare al server l'intera form completa.
Un'altra soluzione ai problemi di scalabilità è la **distribuzione** che comporta prendere un componente spezzarlo in parti più piccole e distribuire tali parti nel sistema. Un esempio molto noto di questo tipo di tecnica è il DNS (*domain name system.* Lo spazio dei nomi è organizzato in un albero dei *domini* divisi in zone non sovrapposte. I nomi di ogni zona sono gestiti da un unico server.
L'ultima tecnica di scalabilità è la **replicazione** che consiste nel duplicare quelle risorse che sono maggiormente richieste in modo da evitare colli di bottiglia e bilanciare il carico sulle risorse.

1.2.5 Tranelli

I sistemi distribuiti si differenziano dal software tradizionale in quanto sono i componenti sono sparsi per la rete. Il fatto di non tenere conto di questa dispersione in fase progettuale rende i sistemi inutilmente complessi. Questi errori sono dovuti a delle ipotesi (false) che ognuno fa quando progetta un'applicazione distribuita:

1. La rete è affidabile
2. La rete è sicura
3. La rete è omogenea
4. La topologia non cambia
5. La latenza è zero
6. L'ampiezza di banda è infinita
7. Il costo di trasporto è zero
8. C'è un solo amministratore

1.3 Tipi di sistemi distribuiti

Esistono diverse categorie di sistemi distribuiti

1.3.1 Sistemi di calcolo distribuiti

Una classe di sistemi distribuiti è quella utilizzata per calcoli ad alte prestazioni; questa categoria può essere divisa in due sottogruppi. Nei **sistemi di calcolo a cluster** l'hardware è composto da una serie di workstation connessi ad una rete locale ad alta velocità e in cui ogni nodo ha lo stesso sistema operativo. Nei **sistemi con tecnologia grid** invece, si intendono sistemi distribuiti costruiti come un gruppo di computer risiedenti in domini di amministrazione diversi con hardware e software differenti.

Sistemi di calcolo a cluster I sistemi di calcolo a *cluster* divennerò popolari quando il rapporto prezzo/prestazioni delle *workstation* divenne vantaggioso. In quasi tutti i casi i sistemi di calcolo a cluster sono utilizzati per per la programmazione parallela.
Un esempio di sistema a cluster molto diffuso è il **Beowolf** un sistema basato su linux in cui l'accesso al sistema avviene tramite un singolo nodo principale, che gestisce l'allocazione dei programmi sui nodi. In realtà il nodo master esegue il *middleware* necessario per l'esecuzione dei programmi e la gestione del cluster. Una parte fondamentale di questo middleware sono le librerie con il quale è sviluppato che forniscono solo funzionalità di comunicazione basate sui messaggi e non sono in grado di gestire errori, sicurezza ecc.
Un altro sistema cluster molto diffuso è il **MOSIX** che fa sembrare il cluster un singolo sistema offrendo così ai programmi la trasparenza di distribuzione.

Grid computing Al contrario dei sistemi di calcolo a cluster in cui i componenti hardware sono omogenei nei sistemi *Grid Computing* vi è un alto livello di eterogeneità sia hardware sia software.
Solitamente nella tecnologia grid le risorse di diverse aziende vengono unite per consentire la collaborazione che viene anche detta **organizzazione virtuale**.

1.3.2 Sistemi informativi basati sulle imprese

Un'altra classe di sistemi distribuiti si trova in strutture aziendali che si sono confrontate con una gran abbondanza di applicazione in rete ma per le quali l'interoperabilità non è stata facile. In alcuni casi l'integrazione riguardava solamente un server che veniva contattato da diversi client i quali confezzionavano una richiesta e la inviavano a tale server. Un'integrazione più approfondita avrebbe permesso ai client di confezzionare una richiesta diretta a molteplici server che avrebbero eseguito un **applicazione distribuita**.

Sistemi transazionali Sono sistemi incentrati su applicazioni relative a basi di dati. In pratica le operazioni sulle basi di dati sono portate a termine sotto forma di **transazioni**. In un sistema distribuito transazionale abbiamo la particolarità che all'interno di una transazione possiamo avere delle chiamate a procedura remote ottenendo così un sistema **RPC transazionale**.

1.3.3 Sistemi distribuiti pervasivi

I sistemi distribuiti visti fin qui hanno la particolarità di essere caratterizzati dalla stabilità: i nodi sono fissi ed hanno a disposizione una connessione di alta qualità. Con l'introduzione di dispositivi mobili ed embedded però abbiamo a che fare con sistemi distribuiti in cui la caratteristica principale è l'instabilità; tali sistemi sono detti **sistemi distribuiti pervasivi**.
I sistemi pervasivi hanno delle caratteristiche particolari, intanto non esiste un amministratore umano ma dopo una prima configurazione da parte del proprietario tali sistemi devono adattarsi al meglio all'ambiente circostante. Inoltre tali sistemi devono avere tre requisiti fondamentali:

- accettare cambi di contesto
- incoraggiare la composizione *ad-hoc*
- riconoscere la condivisione di default

Ovvero un dispositivo deve essere a conoscenza che il suo ambiente è in costante cambiamento e che i dispositivi che compongono il sistema saranno utilizzati in modo diverso da utenti diversi.

In questo tipo di sistemi non vi è alcun tipo di trasparenza, bensì la distribuzione dei dati dei processi e del controllo è intinseca nei sistemi ed è quindi più efficente esporla che nasconderla. Alcuni esempi di sistemi pervasivi sono:

- Sistemi domestici
- Sistemi elettronici per l'assistenza sanitaria
- Reti di sensori

2 Architetture

I sistemi distribuiti sono spesso definibili come pezzi di software sparsi su molte macchine. Al fine di dominare la loro complessità è necessario che questi sistemi siano organizzati. Un modo semplice per distinguere l'organizzazione di un sistema distribuito, è quello di distinguere l'organizzazione logica dei componenti software e la relativa realizzazione fisica.
Per *architetture software* intendiamo l'organizzazione e l'interazione dei vari componenti software; mentre l'effettiva realizzazione di un sistema distribuito richiede che i componenti software siano istanziati su macchine reali, l'architettura risultante viene detta *architettura di sistema.*Analizzeremo per prime le architetture centralizzate in cui un server implementa la maggior parte delle funzionimentre client remoti accedono al server tramite semplici mezzi di comunicazione.

2.1 Stili architetturali

Iniziamo l'analisi dalle diverse tipologie di architetture software in quanto progettazione e adozione di una adeguata architettura sono fondamentali per la riuscita e la manuntenibilità del sistema.
Introduciamo ora la nozione di *stile architetturale* che esprime in termini di componenti mezzi di comunicazione e messaggi scambiati. Un *componente* è un'unità modulare con interfacce ben definite e sostituibile nel suo ambiente.
La comunicazione tra i diversi componenti avviene tramite quello che è definito *connettore* ovvero un sistema che implementa le chiamate a procedure remote piuttosto che il passaggio di messaggi o lo streaming dei dati.
Usando componenti e connettori possiamo ottenere diverse configurazioni che a loro volta sono classificati in diversi *stili architetturali.* I più importanti stili architetturali ad oggi identificati sono:

- architettura a livelli (*layer*)
- architetture basate sugli oggetti
- architetture centrate sui dati
- architetture basate sugli eventi

Per quanto riguarda lo stile a livelli è quello più semplice nel quale i componenti sono organizzati a strati in cui un componente del livello L_i può chiamare un componente del livello L_{i-1} ma non può contattare i componenti dello stesso livello. Questo modello è uno dei più utilizzati nelle applicazioni di rete, le richieste scendono lungo la catena gerarchica mentre le risposte risalgono.
Le *architetture basate sugli oggetti* hanno un organizzazione meno rigida, ogni oggetto corrisponde ad un componente e tutti gli oggetti sono connessi tramite *chiamate a procedura remota*; questo tipo di architettura corrisponde esattamente al caso client-server ed insieme a quella a livelli costituiscono il 90% delle architetture dei sistemi distribuiti presenti oggi.
Le *architetture basate sui dati* si sviluppano attorno all'idea che i processi comunicano attraverso un *repository* comune.
Nelle *architetture basate sugli eventi* i processi comunicano essenzialmente attraverso la propagazione degli eventi, i più noti tipi di sistemi distribuiti che utilizzano la propagazione degli eventi sono i sistemi *publish/subscribe* nei quali alcuni processi pubblicano degli eventi ed è il middleware ad assicurarsi che questi eventi siano ricevuti soltanto da qui processi che si sono iscritti a quel

determinato evento.
Nel caso in cui si combinino le architetture basate sugli eventi con quelle basate sui dati si ottiene una architettura chiamata *spazio di dati condivisi* che permette ai processi di essere disaccoppiati anche nel tempo in quanto non è necessario che i processi siano attivi nell'istante in cui avviene la comunicazione.

2.2 Architetture di sistema

Abbiamo visto fino ad ora alcune scelte architetturali, vediamo ora come sono effettivamente organizzati la maggior parte dei sistemi distribuiti considerando dove sono posizionati i componenti software. La scelta di quali componenti usare e di come posizionarli porta alla realizzazione della cosiddetta *architettura di sistema.*

2.2.1 Architetture centralizzate

La prima architettura che analizzeremo è quella *centralizzata*, in quanto pensare ad un sistema in termini di *client* che richiedono dei servizi a dei *server* facilita la comprensione e la gestione dei sistemi distribuiti.
Nel modello client-server i processi sono suddivisi in due gruppi; un **server** è un processo che implementa uno specifico servizio. Un **client** è invece un processo che richiede un servizio ad un serve inviandogli una richiesta e quindi attendendo una risposta.
La comunicazione tra client e server può essere implementata per mezzo di un semplice protocollo senza connessione quando la rete sottostante è molto affidabile. In questo caso quando un client richiede un servizio invia un messaggio al server indicando il servizio richiesto e i dati di input. Quest'ultimo all'arrivo della richiesta la processa e confeziona i risultati in un altro messaggio.
L'utilizzo di un protocollo senza connessione ha il vantaggio di essere efficiente fino a quando non abbiamo perdita di pacchetti. Si potrebbe pensare di impostare il client affinché rinvii il messaggio quando non riceve alcuna rsposta, ma non si può stabilire se è stato perso il messaggio o la risposta e quindi il server ha compiuto oppure no l'operazione richiesta. In alcuni casi le richieste sono *idempotenti* ovvero possono essere ripetute senza danni.
Per risolvere il problema della perdita di messaggi, molte architetture client-server utilizzano dei protocolli affidabili orientati alla connessione.

Stratifficazioni delle applicazioni Il problema principale dell'architettura client server è che non vi è una netta distinzione tra quali sono le funzionalità del client e quelle del server; è stata così introdotta una un nuovo tipo di suddivisione delle diverse funzionalità in base che esse siano più vicine all'utente o ai dati. I tre livelli identificati sono:

- il livello dell'interfaccia utente
- il livello applicativo
- il livello dei dati

Il livello dell'interfaccia utente contiene tutto ciò che è necessario per interfacciarsi con l'utente come la gestione della grafica. Il livello applicativo di solito contiene le applicazioni. Il livello dei dati gestisce tutto ciò che concerne i dati da gestire.
Esistono diversi modi in cui questi tre livelli possono essere istanziati sull'hardware come possiamo vedere in Figura 2. La configurazione più utilizzata è quella nel quale il client implementa solo il livello dell'interfaccia utente (*thin client*) ma esistono altri sistemi una parte dal livello

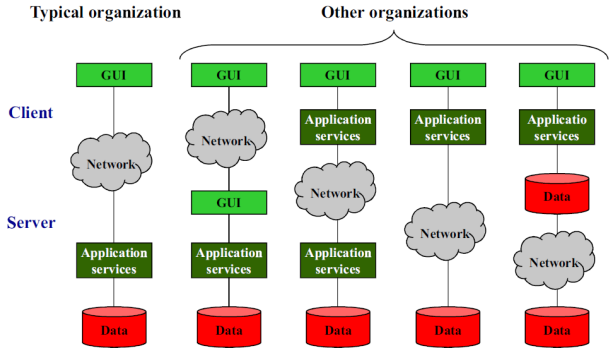

Figura 2: Esempio di suddivisione dei layer

applicativo si trova implementata nel client, in questo caso si parla di *fat client*. Per quanto riguarda il livello dei dati molte volte è gestito da meccanismi che rendono i dati **persistenti** anche quando non vi sono altre applicazioni in esecuzione. In alcuni casi molto semplici il livello dei dati consiste in un *filesystem* ma nella maggioranza dei casi si tratta di una *base di dati*

Architetture multi livello La suddivisione delle applicazioni in tre livelli logici suggerisce anche una distribuzione fisica delle applicazioni client serversu molte macchine, la distribuzione più semplice è quella su due macchine:

1. una macchina client che contiene solo i programmi dell'interfaccia utente
2. una macchina server contenete il resto dei programmi e i dati

Si possono suddividere le applicazioni anche in altri modi come visto in Figura 2.
La tendenza degli ultimi anni è quella di suddividere i diversi livelli su più macchine in modo da creare una architetture multi livello. Un esempio pratico ad esempio è quello mostrato in Figura 3 nella quale si mostra un architettura a 3 livelli.

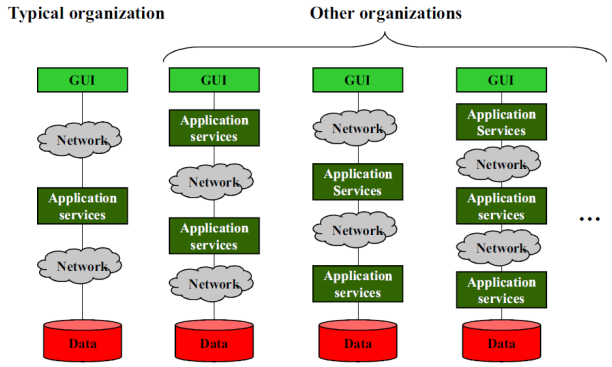

Figura 3: Esempio di suddivisione dei layer su 3 livelli

2.2.2 Architetture decentralizzate

La suddivisione delle architetture client server in livelli denota un tipo di distrinuzione detta *verticale* in quanto i componenti sono divisi su più macchine seguendo un criterio *logico*.

Questo tipo di distribuzione è utile con le architetture client-server ma nel caso in cui la distribuzione che conta è quella dei client e dei server e non delle funzioni allora si parla di *frammentazione orizzontale*. Un esempio molto conosciuto di architettura con distribuzione orizzontale sono i **sistemi peer-to-peer**.
I processi che costituiscono un sistema peer-to-peer sono tutti uguali, di conseguenza l'interazione tra processi è quasi del tutto simmetrica e i processi agiscono sia da client che da server e per questo sono anche detti **servent**.
Dato questo tipo di comportamento il problema principale delle reti *p2p* è quello di organizzare i processi in una rete *overlay* ovvero una rete nella quale i processi costituiscono i nodi e i collegamenti rappresentano i canali di comunicazione. Con questa struttura un processo non può comunicare direttamente con un altro processo arbitrario ma deve seguire i canali di comunicazione disponibili.
Esistono due tipi principali di reti *overlay* quelle strutturate e quelle non strutturate.

Architetture peer-to-peer strutturate In una rete p2p strutturata la rete overlay è costruita usando una procedura deterministica, quella più comune è l'uso di una **hash tabel distribuita** (DHT). In questo tipo di struttura ai dati viene assegnato un identificatore univoco in uno spazio molto grande (129:160 bit). Anche ai nodi del sistema viene assegnato un identificatore nello stesso spazio dei dati. Il punto cruciale di questa architettura è quello di creare uno schema efficiente e deterministico che associ univocamente la chiave di un dato con l'identificativo di un nodo basandosi su un'opportuna metrica di distanza. Inoltre, è importante che quando si effettua la ricerca di un dato sia restituito l'indirizzo del nodo al quale questo dato è associato, e ciò si ottiene *instradando* la richiesta al nodo associato.
Ad esempio nel sistema *Chord* i nodi sono organizzati logicamente ad anello, ed i dati sono organizzati assegnando i dati con identificativo k al nodo con più piccolo identificativo $id > k$; questo nodo è chiamato *successore* della chiave k ed è identificato come $succ(k)$ come mostrato in Figura 4.
La parte più importante della gestione di una rete overlay strutturata è la **gestione dell'appartenenza** da parte dei nodi. Quando un nodo vuole unirsi alla rete genera un id casuale ed effettua una ricerca di id a questo punto il sistema restituirà $succ(id)$. Alla fine per inserirsi il nuovo nodo contatterà il successore individuato dalla ricerca e il suo predecessore e si inserirà nell'anello. Inoltre, l'inserimento causa la migrazione dei dati associati ad id da $succ(id)$. L'uscita è molto semplice il nodo informa della sua dipartita il $succ(id)$ e il suo predecessore e trasferisce i dati a $succ(id)$.

Architetture peer-to-peer non strutturate I sistemi peer-to-peer non strutturati si basano su algoritmi casuali per costruire la rete *overlay*. L'idea è quella che ogni nodo mantenga una lista dei suoi vicini. Inoltre, anche i dati sono posizionati sui nodi in modo casuale, di conseguenza l'unico modo di effettuare una ricerca è inoltrare la richiesta a tutta la rete.
L'obiettivo principale di un sistema p2p non strutturato è la creazione di un **grafo disordinato**. Per fare ciò ogni nodo mantiene una lista di c vicini scelti a caso dall'insieme dei nodi *vivi*; questa vista è detta **vista parziale**. Ora supponiamo che un nodo voglia aggiungersi alla rete, esso contatta in altro nodo arbitrario dalla lista dei punti di accesso; questo punto di accesso è un normale nodo della rete che però ha un alta probabilità di essere disponibile. I meccanismi che creano la rete overlay sono detti *push* e *pull* e permettono lo scambio delle informazioni per la costruzione della rete tra i nodi. Questi meccanismi usati singolarmente possono portare alla creazione di reti overlay disconnesse, per questo si usano entrambe le tecniche.

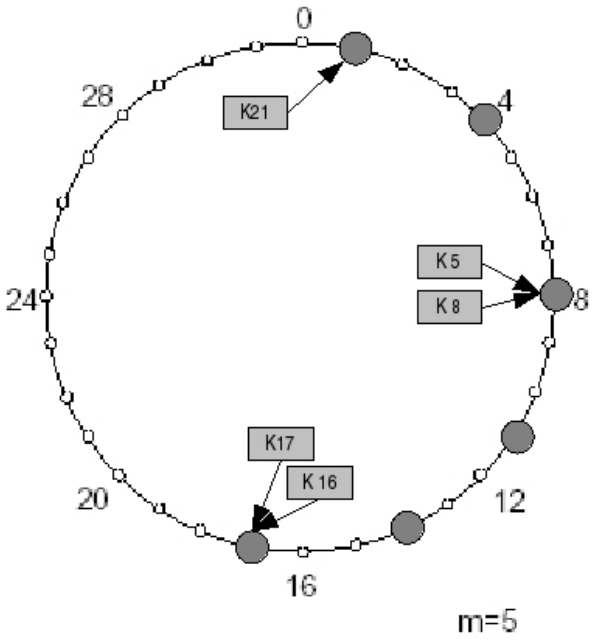

Figura 4: Esempio di sistema Chord

L'uscita dalla rete è molto semplice, infatti, visto che i nodi si scambiano periodicamente le viste parziali basta solo che il nodo lasci la rete, gli altri nodi con il passare del tempo si accorgono dell'assenza del nodo che ha lasciato la rete e lo elimina dalla sua vista parziale.

Gestione della topologia di una rete overlay Anche se i sistemi p2p strutturati e non strutturati sembrano costruire due classi completamente diverse, in realtà selezionando attentamente gli elementi scambiati nelle viste parziali è possibile costruire reti overlay con tipologie specifiche. Un esempio molto interessante sono quelle funzioni che cercano di cogliere la **vicinanza semantica** dei dati che creano reti **overlay semantiche** che permettono ricerche efficienti nei sistemi p2p non strutturati.

Superpeer Specialmente nelle reti non strutturate al crescere delle dimentsioni potrebbe diventare problematico localizzare i dati. Questo è dovuto al fatto non esiste un modo deterministico per instradare una richiesta di ricerca.
Per evitare questo problema molte reti p2p hanno proposto l'utilizzo di nodi speciali che mantengono un indice dei dati. Questo tipo di nodi sono detti **superpeer**. I superpeer sono organizzati tra loro tramite una rete p2p; si forma così una struttura ad albero. L'accesso ad un normale peer avviene attraverso l'accesso al superpeer.

2.2.3 Architetture ibride

Fino ad ora abbiamo visto le architetture centralizzate client-server e alcune architetture peer-to-peer ora vediamo come queste due tipologie di architetture possono combinarsi per dare vita ad altre architetture dette *ibride*

Sistemi edge-server I sistemi *edge-server* sono una classe molto diffusa di architetture ibride. In questa classe i server sono posti ai bordi della rete Internet ovvero tra la rete Internet e quelle

aziendali come nel caso degli **Internet service provider**. I client si connettono alla rete internet tramite un edge-server il quale fornisce i contenuti dopo aver applicato dei filtri e funzioni di transcodifica.

Sistemi distribuiti collaborativi Le strutture ibride sono largamente utilizzate nei sistemi distribuiti collaborativi dove sono necessarie velocità nell'entrare nel sistema e per questo viene utilizzato uno schema di tipo client-server. Dopo di che si usa uno schema completamente decentralizzato.
Un sistema concreto che utilizza questo meccanismo è il sistema *BitTorrent*

2.3 Architetture e middleware a confronto

Considerando le questioni architetturali viste fino ad ora ci chiediamo quale ruolo giochi il middleware visto nei capitoli precedenti.
Il middleware in realtà costituisce uno strato tra le applicazioni e le piattaforme distribuite ed il suo obiettivo è quello di fornire un certo grado di trasparenza alla distribuzione. Ciò che accade in realtà è che i middleware seguono uno specifico stile architetturale (ad oggetti come *CORBA* o ad eventi come *TIB/Rendezvous*). Avere un middleware basato su di un certo stile architetturale rende più semplice la progettazione e la realizzazione delle applicazioni.
Lo svantaggio più grande è però il fatto che un middleware può non essere ottimale per una determinata applicazione ciò porta ad avere o middleware molto grandi o a diverse versioni per una specifica classe di applicazioni.

2.3.1 Interceptor

Concettualmente un *interceptor* non è altro che un costrutto software che interrompe il normale flusso di controllo e consente ad altro codice di essere eseguito. Rendere però un interceptor generico è molto arduo e a volte averne uno con funzionalità limitate migliorerà sia la gestione del software che che il sistema distribuito nel suo complesso.
Il concetto è che un oggetto A può richiamare un metodo appartenete all'oggetto B ache se quest'ultimo risiede su una macchina diversa da A. I passi per eseguire questa chiamata remota sono:

1. All'oggetto A viene fornita un interfaccia locale esattamente uguale a quella fornita dall'oggetto B; a questo punto A richiama il metodo disponibile nell'interfaccia

2. La chiamata di A è trasformata in un'invocazione a un oggetto generico disponibile tramite un'interfaccia generale messa a disposizione dal middleware.

3. L'invocazione a questo oggetto generico viene trasformata in un messaggio inviato attraverso l'interfaccia di rete.

3 Modelling

Nel precedente capitolo abbiamo visto il perchè di alcune scelte architetturali nei sistemi distribuiti; in questo capitolo invece analizzeremo alcuni dei modelli esistenti e di come questi siano realmente utilizzati.

3.1 Architettura service oriented

Partendo dal concetto di architettura client-server si può pensare di costruire una architettura costruita interamente attorno al concetto di servizio (*service provider, service consumer, service brokers*). Un servizio rappresenta un insieme di funzionalità vagamente legate tra loro che sono messe a disposizione di una unita chiamata **fornitore di servizi** (*service provider*). Il **brokers** mantiene la descrizione dei servizi disponibili che possono essere cercati dai **consumer** che poi li richiamano quando ne hanno bisogno.
Con il termine *orchestration* si indicano l'insieme di invocazioni di determinati servizi in un flusso di lavoro per soddisfare un determinato obiettivo.

3.2 REST style

Lo stile REST (*REpresentational State Transfert*) è allo stesso tempo un buon modo di descrivere il web e un insieme di principi che definiscono come gli standard del Web dovrebbero essere utilizzati.
Gli obiettivi principali del REST includono :

- La scalabilità delle interazioni tra componenti
- Generalità delle interfacce
- Sviluppo indipendente dei componenti
- Componenti intermedi per ridurre le latenze aumentare la sicurezza ed incapsulare i componenti legacy.

Le principali caratteristiche del sistema REST sono che anch'esso è basato su un architettura client-server; le interazioni sono di tipo *stateless*, gli stati devono essere trasferiti di volta in volta dal client al server;. I dati che giungono come risposta ad una richiesta devono essere etichettati come cacheable oppure non-cacheable; ogni componente non può comunicare con se non con i layer più vicini. I client devono supportare il *code-on-demand*; ed infine, i componenti devono esporre un interfaccia uniforme.
Per quanto riguarda l'ultimo punto le interfacce dei componenti devono soddisfare quattro vincoli principali:

- Tutte le risorse devono essere identificate da un id (solitamente un *URI*) ed ogni risorsa con un id è una risorsa valida
- Manipolazione delle risorse tramite la loro rappresentazione, i diversi componenti comunicano tramite il trasferimento di rappresentazioni delle risorse in formati standard (XML) selezionati dinamicamente in base alle capacità o alle informazioni desiderate.
- Messaggi auto-descrittivi, i messaggi contengono al loro interno dei metadati che ne indicano il tipo di richiesta oppure il significato delle risposte. Questa tecnica è utilizzabile per parametrizzare le richieste

- Link ipermediali, i client cambiano il loro stato attraverso le richieste che avvengono tramite dei link ipermediali.

3.2.1 Peer-to-Peer

Come visto nel capitolo 2 nei sistemi peer-to-peer non esistono dei ruoli definiti ma tutti i componendi giocano lo stesso ruolo. Come già detto i sistemi p2p a differenza di quelli client-server permettono di scalare in modo migliore

3.3 Object oriented

Nel caso *Object Oriented* i componenti distribuiti incapsulano i dati e permettono l'accesso e la modifica solo tramite un interfaccia messa a disposizione da ogni componente. I diversi componenti interagiscono tramite RPC. Questo tipo di sistema si basa sul modello p2p ma il più delle volte è utilizzato per implementare meccanismi client-server

3.4 Data centered