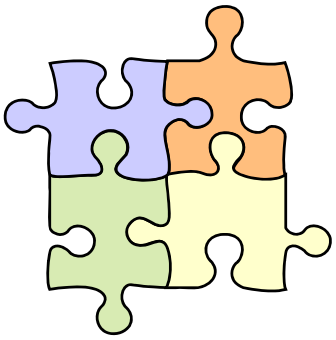
Luca Cabibbo



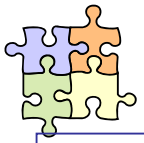
# Architetture Software

# Architetture dei sistemi distribuiti

**Dispensa PA 2**

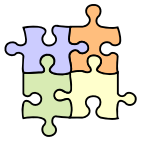
ottobre 2008





## - Fonti

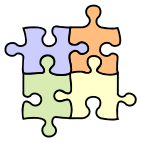
- [SSA] Chapter 11, Using Styles and Patterns
- [POSA] Pattern-Oriented Software Architecture – A System of Patterns
- [POSA4] Pattern-Oriented Software Architecture – A Pattern Language for Distributed Computing
- [Somm/7e] Capitolo 12, Architetture dei sistemi distribuiti
- [CDK/4e] Distributed Systems – Concepts and Design



# - Obiettivi e argomenti

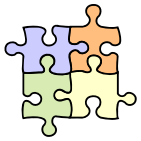
- Obiettivi
  - conoscere alcuni stili architetturali fondamentali per sistemi distribuiti
- Argomenti
  - introduzione
  - tecnologie che sostengono la distribuzione
  - architetture client/server
  - architetture a oggetti distribuiti
  - proxy [POSA]
  - Broker [POSA]
  - architetture peer-to-peer
  - discussione





# * Introduzione

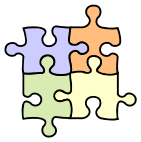
- Tutti i grandi sistemi informatici sono ora sistemi distribuiti
- Alcune possibili definizioni – un **sistema distribuito** è
  - un sistema in cui l'elaborazione delle informazioni è distribuita su più calcolatori – anziché centralizzata su una singola macchina
  - un sistema di elaborazione in cui un numero di componenti coopera comunicando in rete [POSA4]
  - un sistema in cui i componenti hardware o software posizionati in calcolatori collegati in rete comunicano e coordinano le proprie azioni solo tramite lo scambio di messaggi [CDK/4e]
  - un sistema in cui il fallimento di un calcolatore di cui nemmeno conosci l'esistenza può rendere inutilizzabile il tuo calcolatore [Lamport]



# Benefici della distribuzione

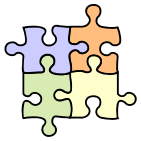
- Connettività e collaborazione
  - possibilità di condividere risorse hardware, software – tra cui dati
- Economicità
  - i sistemi distribuiti offrono spesso un miglior rapporto prezzo/prestazioni che i sistemi centralizzati basati su mainframe
- Prestazioni e scalabilità
  - la possibilità di aggiungere risorse fornisce la capacità di migliorare le prestazioni e sostenere un carico che aumenta





# Benefici della distribuzione

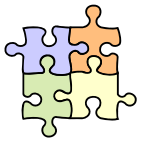
- Tolleranza ai guasti
  - grazie alla possibilità di replicare risorse
- Apertura
  - l'uso di protocolli standard aperti favorisce l'interoperabilità di hardware e software di fornitori diversi
- Sistemi inerentemente distribuiti
  - alcune applicazioni sono inerentemente distribuite – non sono possibili opzioni diverse



# Svantaggi legati alla distribuzione

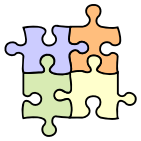
- ☹ Complessità
  - i sistemi distribuiti sono più complessi di quelli centralizzati
  - più difficile capirne e valutarne le qualità
- ☹ Sicurezza
  - l'accessibilità in rete pone il problema della sicurezza
- ☹ Non prevedibilità
  - la risposta del sistema dipende dal carico del sistema e della rete, che possono cambiare rapidamente
- ☹ Gestibilità
  - è necessario uno sforzo maggiore per la gestione del sistema operativo e delle applicazioni





# Svantaggi legati alla distribuzione

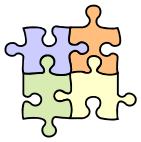
- ☹ Complessità accidentale
  - introdotta dall'uso di strumenti di sviluppo non opportuni
- ☹ Metodi e tecniche non adeguati
  - i metodi di analisi e progettazione più diffusi fanno riferimento allo sviluppo di applicazioni mono-processo, mono-thread
  - i metodi di analisi e progettazione per sistemi distribuiti sono più complessi e meno diffusi
- ☹ Continua re-invenzione e riscoperta di concetti e soluzioni
  - l'industria del software ha una lunga storia di ri-creazione di soluzioni (spesso incompatibili con le precedenti) di problemi che sono stati già risolti
  - questo è vero anche nel campo dei sistemi distribuiti



# Sfida posta dalla distribuzione

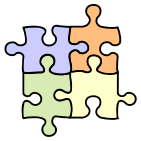
- La sfida posta dai sistemi distribuita
  - ottenere i possibili benefici
  - minimizzando gli svantaggi





# Parentesi: mainframe

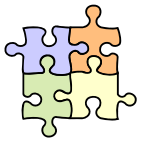
- Le moderne tecnologie per mainframe, anche basate sull'uso della virtualizzazione, si propongono di ottenere i benefici dei sistemi distribuiti, cercando allo stesso tempo di fornire soluzioni per ridurre i possibili svantaggi
  - ad esempio, un mainframe può essere utilizzato per la consolidazione di più server
    - un singolo server fisico può ospitare N server virtuali
    - utilizzate tecniche/tattiche ad hoc per prestazioni, scalabilità, disponibilità, sicurezza, ...
    - gestione più semplice – risparmio energetico – in generale, un TCO inferiore
  - è una tecnologia adatta per sistemi molto grandi e complessi
- In ogni caso, l'organizzazione dei server virtuali può essere basata sugli stili architetturali per sistemi distribuiti



# * Tecnologie che sostengono la distribuzione

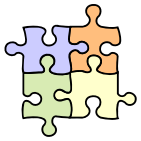
- Nel corso del tempo sono stati definiti diversi strumenti di supporto allo sviluppo di sistemi distribuiti
  - programmazione ad hoc
    - memoria condivisa, pipe, socket
    - strumenti di basso livello, difficili da usare
      - problemi, ad es., di portabilità
  - comunicazione strutturata
    - viene alzato il livello di astrazione della comunicazione
    - ad es., RPC
      - rende possibile l'invocazione remota di procedure – maschera i messaggi scambiati
    - non maschera altri aspetti della comunicazione – ad es., locazione dei componenti, eterogeneità, ...
  - middleware





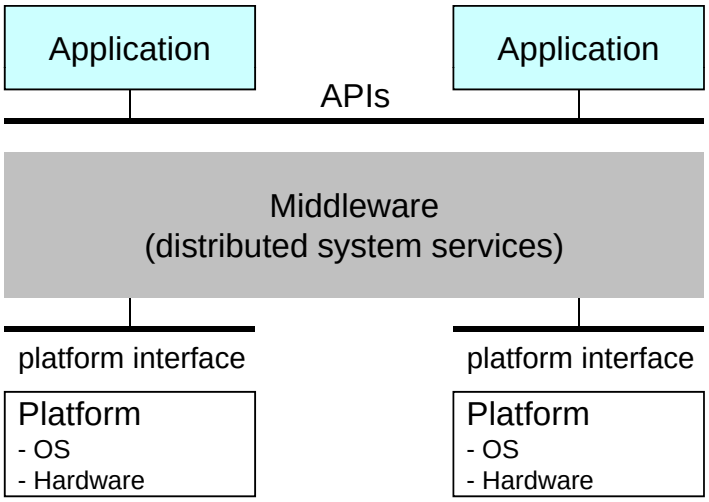
# Middleware

- Per **middleware** si intende
  - una classe di tecnologie software sviluppate per aiutare gli sviluppatori nella gestione della complessità e della eterogeneità presenti nei sistemi distribuiti
  - uno strato software "in mezzo"
    - sopra al sistema operativo, ma sotto i programmi applicativi
    - fornisce un'astrazione di programmazione distribuita – un modello computazionale uniforme
    - per mascherare alcune eterogeneità degli elementi sottostanti – reti, hardware, sistemi operativi, linguaggi di programmazione, ...
- L'evoluzione nello sviluppo dei sistemi distribuiti e l'evoluzione degli strumenti di middleware sono mutuamente correlate

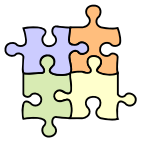


# Middleware

- Un **servizio di middleware** è un servizio general-purpose che si colloca tra piattaforme e applicazioni [Bernstein]

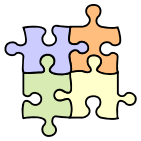




# Middleware

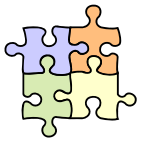
- Varie famiglie di strumenti di middleware
  - middleware per oggetti distribuiti
    - evoluzione di RPC – i componenti distribuiti sono considerati oggetti – con identità, interfaccia, incapsulamento
    - non sostiene la gestione della configurazione degli oggetti distribuiti
  - middleware message-oriented
    - basato sullo scambio asincrono di messaggi – e non su protocolli sincroni di richiesta/risposta
    - possono offrire elevata flessibilità ed affidabilità



# Middleware

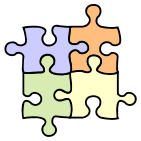
- Varie famiglie di strumenti di middleware
  - middleware per componenti
    - evoluzione del middleware per oggetti distributi
    - possibile sia la comunicazione sincrona che quella asincrona
    - i componenti vivono in contenitori (application server) in grado di gestire la configurazione e la distribuzione dei componenti, e fornire ad essi funzionalità di supporto
  - middleware orientato ai servizi
    - enfasi sull'interoperabilità tra componenti eterogenei, sulla base di protocolli standard aperti ed universalmente accettati
    - generalità dei meccanismi di comunicazione – sia sincroni che asincroni
    - flessibilità nell'organizzazione dei suoi elementi (servizi)
  - ... in continua evoluzione ...





# Uso efficace del middleware

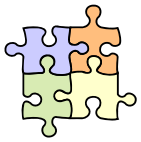
- Se utilizzato in modo opportuno, il middleware affronta e risolve diverse problematiche significative nello sviluppo dei sistemi distribuiti
  - in questo modo, consente di concentrarsi sullo sviluppo della logica applicativa – e non sui dettagli della comunicazione e della piattaforma hw/sw utilizzata
- Tuttavia, per aumentare effettivamente la produttività, il middleware scelto deve essere utilizzato in modo corretto
  - è necessaria una buona comprensione del paradigma di comunicazione implementato dal middleware, nonché della sua struttura e dei suoi principi di funzionamento
  - anche la decisione del middleware utilizzato richiede delle considerazioni e delle decisioni esplicite



## - Di che cosa parleremo

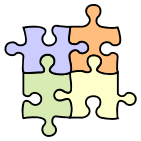
- Nel seguito di questa dispensa sono descritti
  - alcuni stili architetturali fondamentali per sistemi distribuiti
    - architetture client/server
    - architetture a oggetti distribuiti
    - architetture peer-to-peer
  - alcuni pattern utilizzati nella realizzazione di infrastrutture di middleware
    - proxy
    - broker
- Altri stili architetturali per sistemi distribuiti sono descritti in ulteriori dispense





## * Architetture client/server

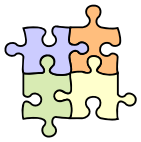
- Lo stile architetturale **client/server** organizza un sistema come
  - un insieme di *servizi*
    - ciascun servizio è caratterizzato da un'interfaccia – definisce protocollo e formato dei messaggi scambiati
  - un insieme di *server*
    - i server sono processi che erogano servizi
  - un insieme di *client*
    - i client sono processi fruitori di servizi
  - connessione tramite protocolli e porte
  - i client sono attivi – i server reattivi
  - un server può essere acceduto concorrentemente da molti client



# Architetture client/server

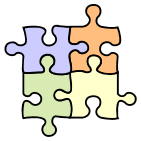
- Esempi di uso
  - molti servizi di Internet, accesso alle basi di dati, ...
- Conseguenze – in prima approssimazione
  - ☺ condivisione di risorse, centralizzazione di elaborazione complessa o sensibile, ...
  - ☹ overhead della comunicazione, ...





# Architetture client/server

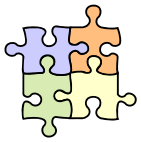
- Nelle architetture client/server gli elementi client e server sono componenti – processi logici
  - l'architettura client-server viene comunemente adottata nel contesto della *vista funzionale* o della *vista della concorrenza*
- Tuttavia, è spesso utile/necessario descrivere la particolare modalità di applicazione dello stile client/server anche con riferimento alla *vista di deployment*
  - processi diversi possono essere allocati su processori/ calcolatori diversi
  - è anche possibile che processi diversi siano allocati sullo stesso processore/calcolatore



# Architetture client/server e livelli

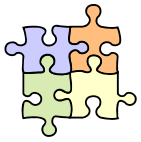
- Esistono diversi tipi di architetture client/server (C/S)
  - le architetture C/S sono normalmente organizzate a livelli
  - un **livello** (*tier*) corrisponde ad un nodo o gruppo di nodi di calcolo su cui è distribuito il sistema
    - punto di vista del deployment
  - il sistema è organizzato come una sequenza di livelli
    - ciascun livello funge da server per i suoi chiamanti – nel livello precedente
    - ciascun livello funge da client per il livello successivo
  - i livelli sono comunemente organizzati in base al tipo di servizio (responsabilità) che forniscono
- Si tratta di un'interpretazione particolare dell'architettura a strati
  - in cui gli strati corrispondono all'allocazione di server (processi) su nodi fisici



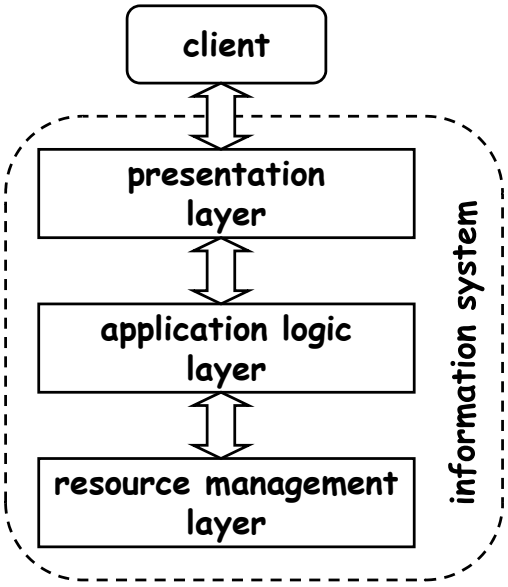


# Livelli e strati

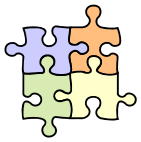
- Lo stile client-server a livelli è spesso combinato con lo stile a strati – nel senso che
  - nella vista funzionale, il sistema adotta un'architettura a strati
    - gli strati sono organizzati in base al livello di astrazione
  - nella vista di deployment, il sistema adotta un'architettura a livelli
    - i livelli sono organizzati in base al tipo di servizio (responsabilità) che forniscono
  - ciascun livello è spesso internamente organizzato a strati
- E' possibile/utile fare una discussione in relazione alle possibili corrispondenze tra livelli e strati
  - assumiamo che il sistema debba gestire tre tipi principali di responsabilità – (1) presentazione, (2) logica applicativa e (3) accesso alle risorse/ai dati



# Architettura a strati

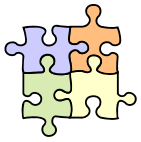




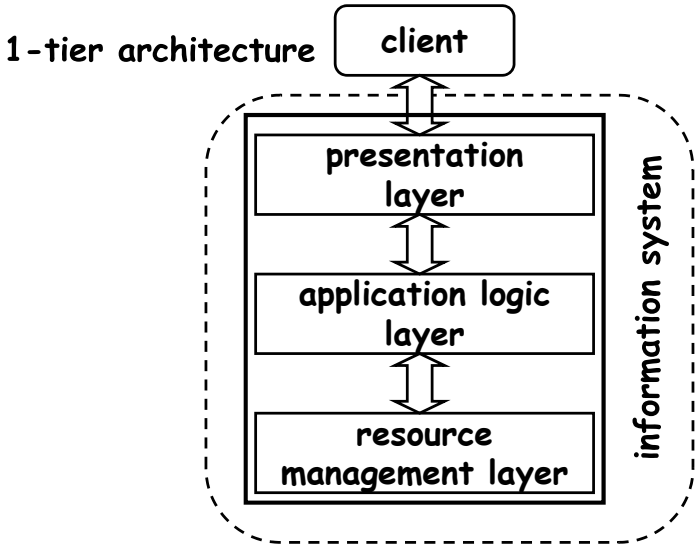
# Architetture (C/S) a livelli

- Architettura a un livello – non è client/server – fino agli anni ‘80
  - architettura monolitica – mainframe + terminali “stupidi”
  - il caso di molti legacy system
- Architetture client/server a due livelli – anni ‘80
  - modello thin-client
  - modello fat-client
- Architetture client/server a tre livelli – anni ‘90
- Architetture client/server ad N livelli

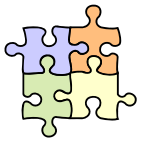


# - Architettura a un livello

- *Architettura a un livello* – non è distribuita – non è client/server
  - il livello server è tipicamente realizzato con un mainframe
  - il client non costituisce un livello – è un "terminale stupido"
  - stato dell'arte prima dell'avvento dei sistemi distribuiti

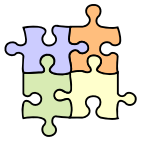




# Architettura a un livello

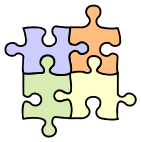
- Conseguenze
  - ☺ possibile combinare gli strati a piacimento – per ottimizzare le prestazioni
  - ☺ non c'è un overhead dovuto alla comunicazione ed ai cambiamenti di contesto
  - ☺ non ci sono problemi di compabilità/interoperabilità – perché si usa una singola tecnologia
  - ☺ costo basso di gestione dei client
  - ☹ soluzione monolitica, proprietaria, non aperta
  - ☹ difficoltà di manutenzione – soprattutto dei legacy system
- Architettura rilevante oggi solo nel contesto della gestione di legacy system
  - meno rilevante per lo sviluppo di nuovi sistemi



# - Architetture C/S a due livelli

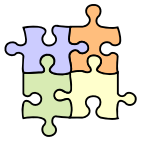
- *Architettura client-server a due livelli* (anni ‘80)
  - le responsabilità sono distribuite su due livelli
    - un livello server
    - un livello client
- Possibili due varianti principali
  - modello thin-client
    - server – logica applicativa e gestione dei dati
    - client – esecuzione del software di presentazione
  - modello fat-client
    - server – gestione dei dati
    - client – presentazione e logica applicativa



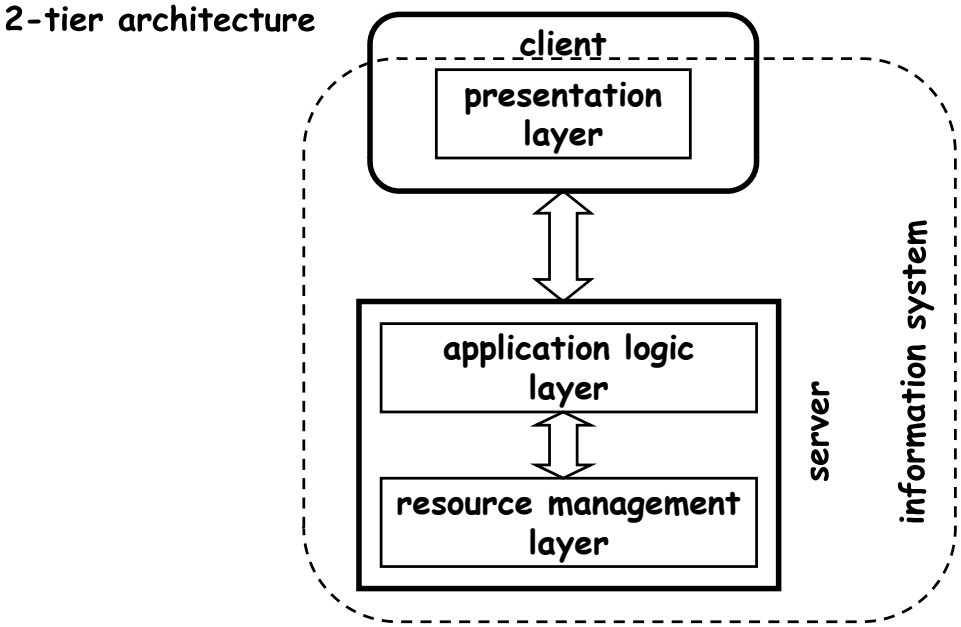


# Architettura C/S a due livelli thin-client

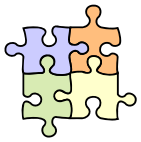
- *Modello thin-client*
  - il livello server è responsabile della logica applicativa e della gestione dei dati
  - il livello client è responsabile dell’esecuzione del software di presentazione
  - alcune accezioni
    - il livello server è basato su mainframe – i client sono usati inizialmente per l’emulazione di “terminali stupidi” – dapprima come UI a caratteri, poi come GUI
    - il server è un DBMS che offre funzionalità implementate come procedure SQL – client realizzati mediante 4GL



# Architettura C/S a due livelli thin-client

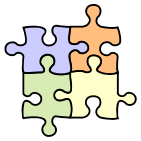




# Architettura C/S a due livelli thin-client

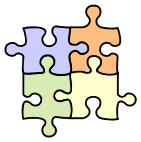
- Conseguenze
  - ☺ il modello thin-client costituisce la soluzione più semplice quando si vuole far migrare un legacy system verso un'architettura client/server
  - ☺ adatto quando il client ha potenza di calcolo limitata – in particolare, ai primi PC
  - ☺ client facile da portare, installare e mantenere
  - ☺ possibili più client – di tipo diverso
  - ☺ possibili buone prestazioni – perché logica applicativa ed accesso ai dati non sono separati
- Conseguenze
  - ☹ carico concentrato sul server – e sulla rete
  - ☹ l'eventuale potenza di calcolo del calcolatore client non è utilizzata



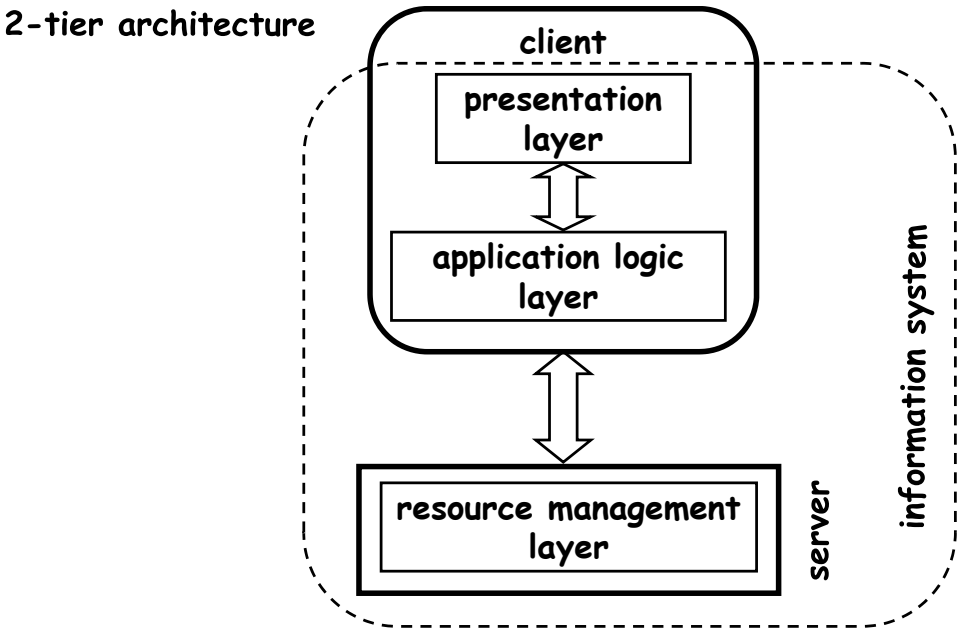
# Architettura C/S a due livelli fat-client

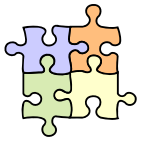
- *Modello fat-client*
  - il livello server è responsabile solo della gestione dei dati
    - ad es., il server è un server che gestisce transazioni di basi di dati
  - il software sul livello client implementa sia la logica applicativa che la presentazione





# Architettura C/S a due livelli fat-client

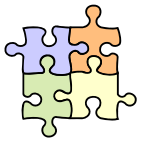




# Architettura C/S a due livelli fat-client

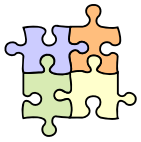
- Conseguenze
  - ☺ il modello fat-client è adatto allo sviluppo di nuovi sistemi C/S – in cui si vogliono sfruttare le capacità di calcolo del client
  - ☺ il modello fat-client può distribuire il carico di elaborazione in modo più efficace del modello thin-client
  - ☹ richiede client sufficientemente potenti
  - ☹ la gestione del sistema è più complessa – rispetto al modello thin client
    - ad es., nuove versioni della logica applicativa devono essere installate su tutti i client





# Architetture C/S a due livelli

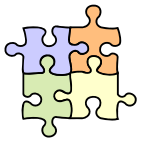
- Altre conseguenze – per tutte le architetture C/S a due livelli
  - ☹ le architetture client/server a due livelli sono in genere poco scalabili
    - un singolo server può servire solo un numero limitato di client – infatti, oltre all'erogazione di servizi, il server deve gestire connessione e, comunemente, anche lo stato della sessione di ciascun client



# Architetture C/S a due livelli

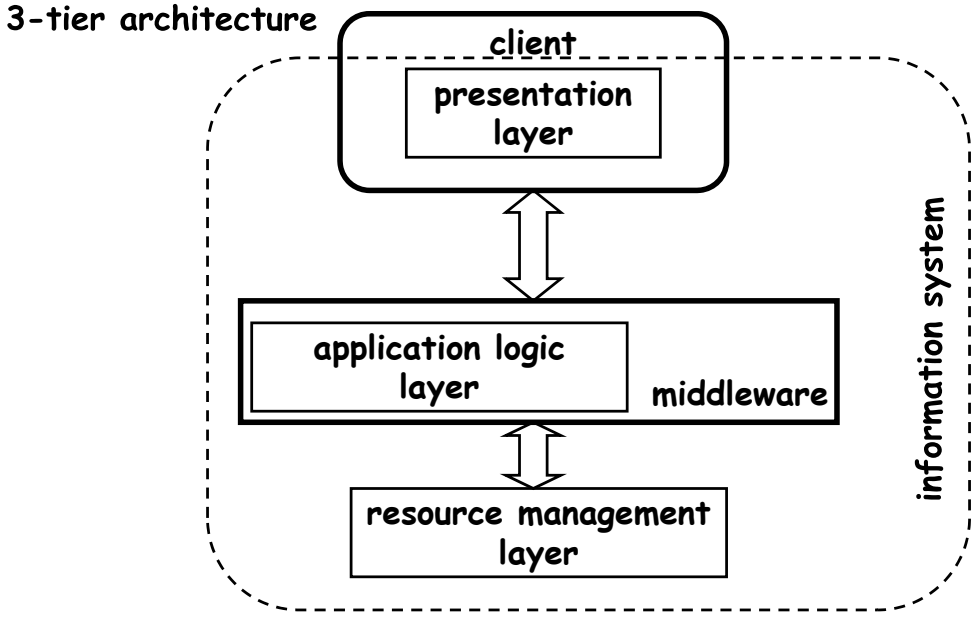
- Discussione
  - il modello client/server a due livelli è legato a sviluppi fondamentali legati al software per sistemi distribuiti
    - middleware, meccanismi di RPC, pubblicazione di interfacce, sistemi aperti
  - punto di partenza per i sistemi distribuiti moderni

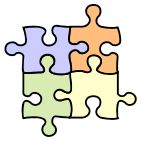




# - Architetture C/S a tre livelli

- *Architettura client-server a tre livelli* (anni '90)
  - i tre strati funzionali sono separati su tre diversi livelli di deployment

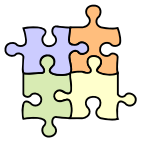




# Architetture C/S a tre livelli

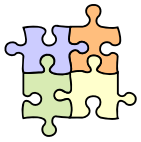
- Conseguenze
  - ☺ consente migliori prestazioni – rispetto al modello thin-client – consente una migliore distribuzione del carico di elaborazione – può compensare il maggior overhead nella comunicazione
  - ☺ architettura più scalabile – se aumenta il carico, è possibile aggiungere nuovi server – il livello intermedio può essere realizzato come un cluster di calcolatori
  - ☺ consente una maggior affidabilità
  - ☺ più semplice da gestire – rispetto al modello fat-client
  - ☹ maggior overhead nella comunicazione
  - ☹ maggior complessità



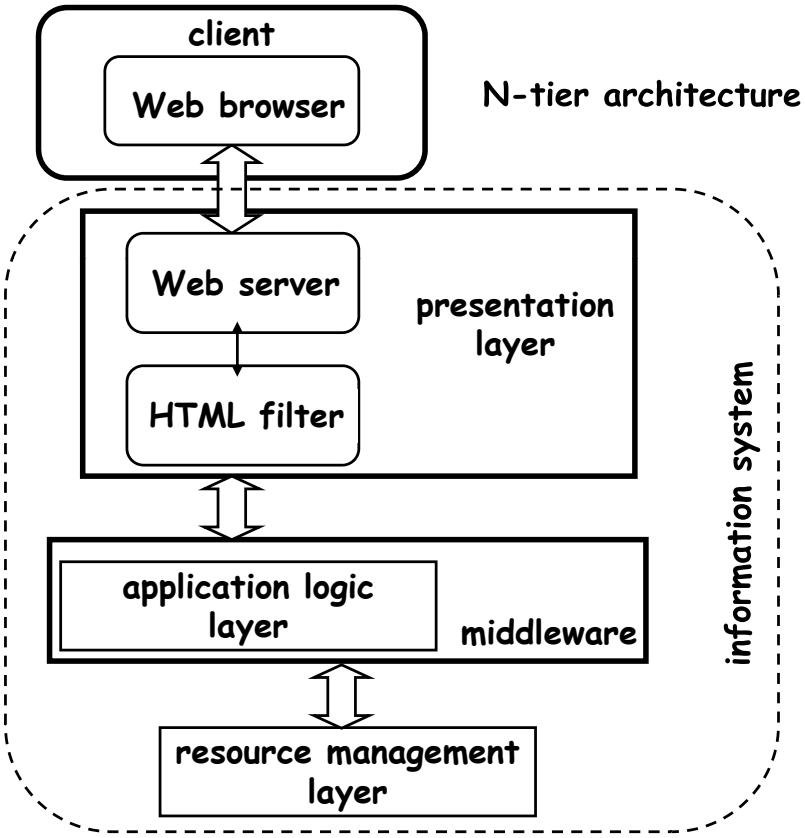


# - Architetture C/S ad N livelli

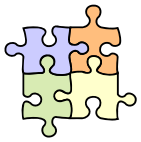
- *Architettura client-server ad N livelli*
  - generalizzazione del modello client/server a tre livelli ad un numero qualunque di livelli e server intermedi



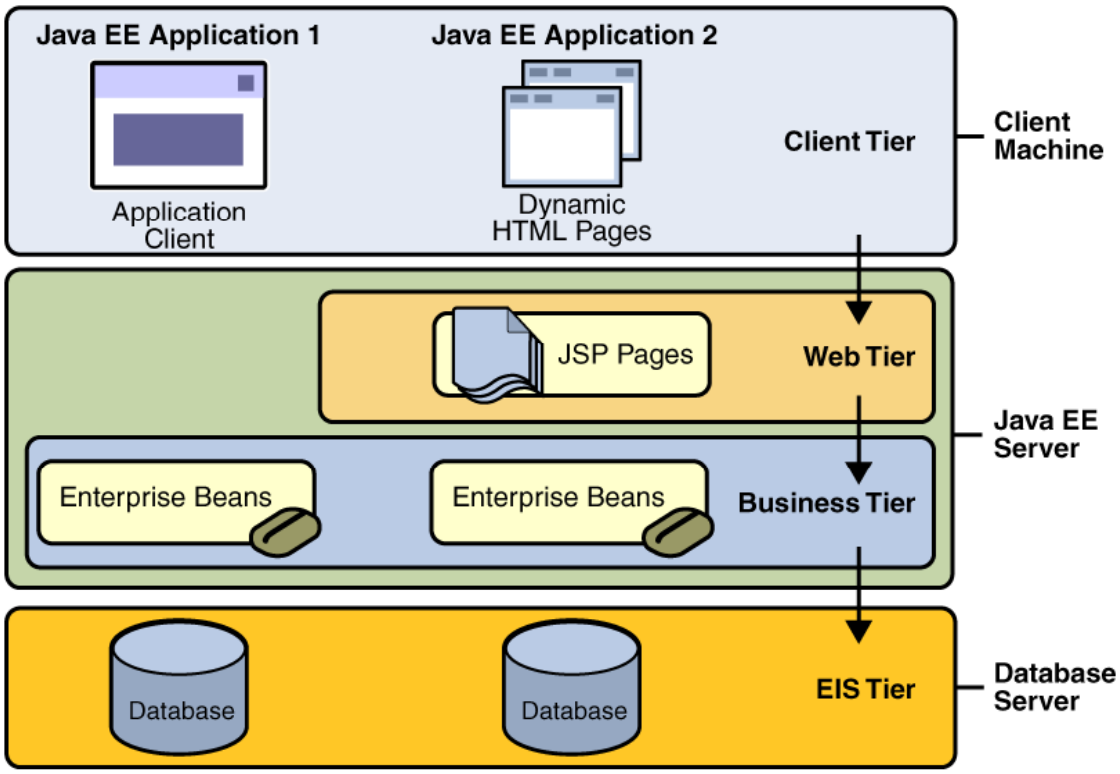
# Esempio – accesso web ad app. pre-esistente

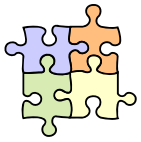




# Esempio – piattaforma Java EE

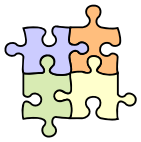




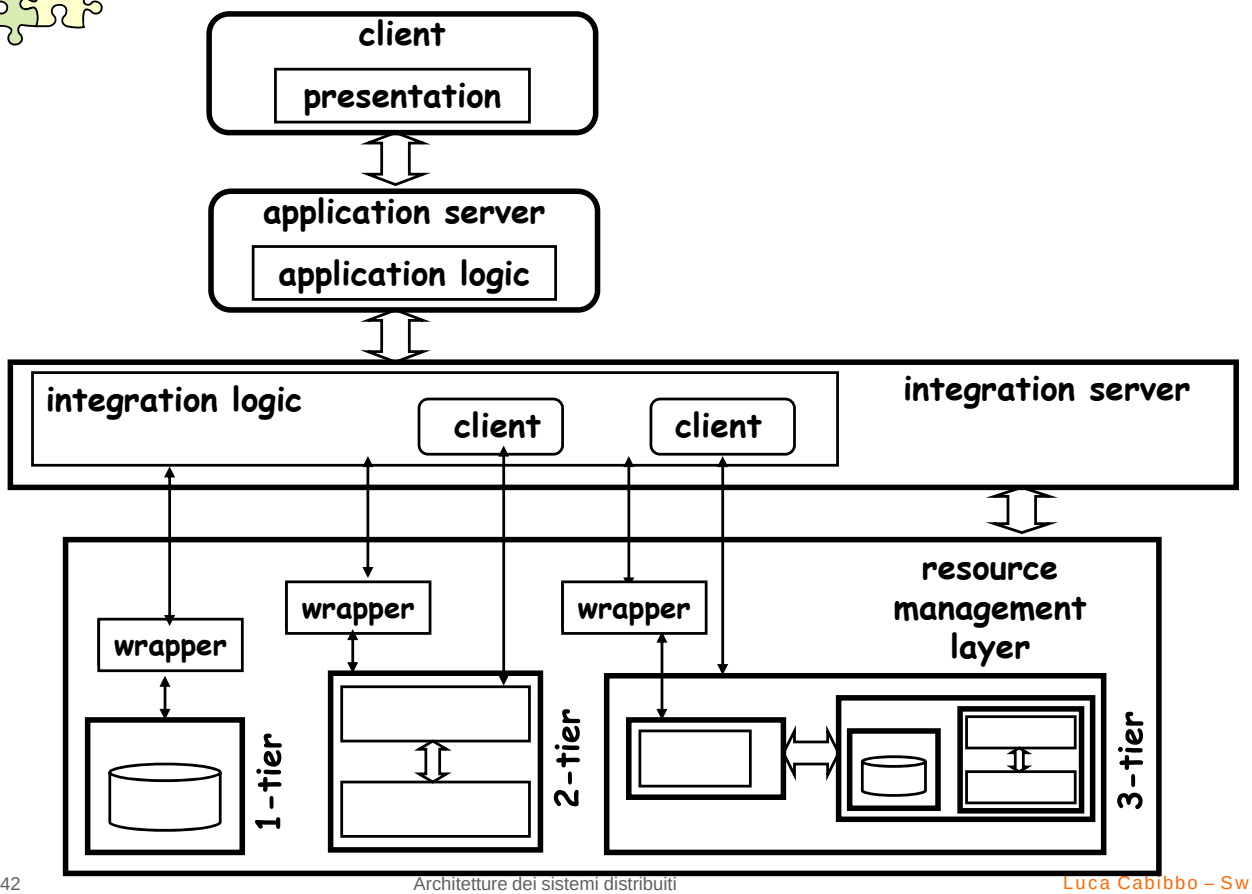
# Esempio – architettura per l'integrazione

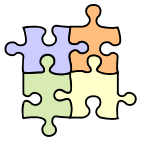
- Si consideri un'applicazione che deve accedere a dati da più basi di dati
  - un server per l'integrazione può essere collocato tra l'application server e i database server
  - il server per l'integrazione
    - accede ai dati distribuiti
    - li integra
    - li presenta all'application server come se provenissero da una singola base di dati





# Un'architettura per l'integrazione

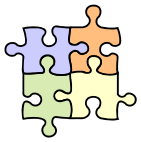




## - Uso di architetture C/S

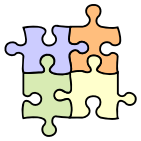
- Architetture C/S a due livelli, thin-client
  - applicazioni legacy system, in cui la separazione della logica applicativa e della gestione dei dati non è fattibile
  - applicazioni di elaborazione intensiva – senza o con poca gestione dei dati
  - applicazioni data-intensive (navigazione e interrogazione) senza o con poca logica applicativa
- Architetture C/S a due livelli, fat-client
  - applicazioni in cui l'elaborazione è fornita da software off-the-shelf (ad es., Microsoft Excel) sul client
  - applicazioni in cui è richiesta elaborazione intensiva di dati – ad es., visualizzazione di dati
  - applicazioni con funzionalità per l'utente relativamente stabili, usate in un ambiente con una buona gestione del sistema





## Uso di architetture C/S

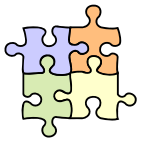
- Architetture C/S a tre o N livelli
  - applicazioni su larga scala – con centinaia o migliaia di client
  - applicazioni in cui sia i dati che i programmi sono "volatili"
  - applicazioni in cui è richiesta l'integrazione di dati/servizi da più sorgenti



# Da un livello ad N livelli

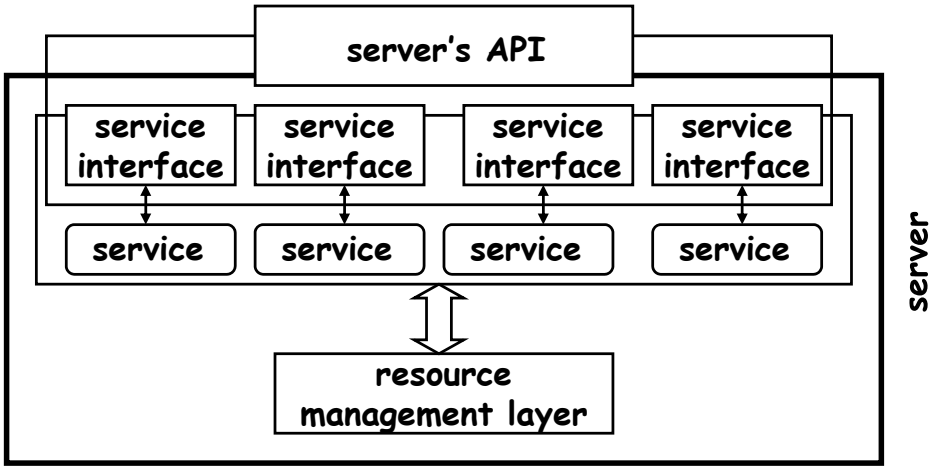
- Con l'introduzione di ciascun livello
  - ☺ l'architettura guadagna in flessibilità, funzionalità e possibilità di distribuzione
  - ☹ l'architettura potrebbe introdurre un problema di prestazioni – perché aumenta il costo della comunicazione
  - ☹ viene introdotta più complessità – in termini di gestione ed ottimizzazione



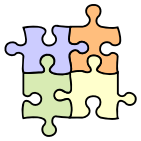


# - Interfacce e protocolli

- Nella comunicazione client/server
  - un server offre dei servizi – "pubblicati" come un'interfaccia – basata anche su un protocollo e sul formato delle richieste/risposte scambiate



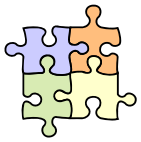

  - sempre importante chiedersi: quali protocolli? quali formati?



## - Strati e livelli

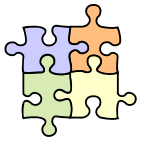
- Attenzione, le corrispondenze tra strati e livelli non sono sempre così nette
  - un client Java Swing, che parla con un server mediante RMI
    - la presentazione risiede ed è eseguita lato client
  - un client di tipo applet
    - la presentazione viene eseguita lato client – risiede nel client solo dopo che è stata scaricata completamente dal lato server
  - un'applicazione web
    - il client è un browser web – che "esegue" la presentazione (le pagine web)
    - ma le pagine web risiedono (o sono generate) lato server – dunque in un altro livello/strato





## - Servizi stateless e stateful

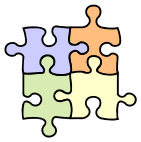
- Un servizio (ed il server che lo implementa) può essere stateless oppure stateful
  - la presenza o assenza di stato non si riferisce allo stato complessivo del server o del servizio
  - si riferisce, piuttosto, alla capacità di ricordare lo stato di una specifica conversazione (sessione) tra un client ed il server
- Si tratta di una caratteristica importante
  - ad esempio, ha impatto sulla scalabilità del livello server
  - ha anche impatto sul livello di accoppiamento tra client e server



# Servizi stateless

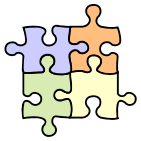
- Un servizio è *stateless* se non mantiene informazioni di stato su ciò che avviene tra richieste successive di uno stesso client
  - ad es., un servizio di previsioni del tempo
  - ogni richiesta viene gestita mediante l'esecuzione di un'operazione indipendente dalle altre richieste
  - ciascuna richiesta deve contenere tutte le informazioni necessarie a soddisfare la richiesta





# Servizi stateful

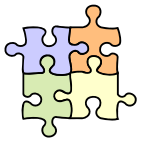
- Un servizio è *stateful* se mantiene (qualche) informazione di stato circa le diverse richieste successive da parte di uno stesso client nell'ambito di una sessione (o conversazione)
  - la gestione di una richiesta può dipendere dalla storia delle richieste precedenti
  - ciascuna richiesta contiene le informazioni necessarie a soddisfare la richiesta – nell'ambito di un protocollo in cui un servizio viene erogato sulla base di richieste multiple



# Esempio

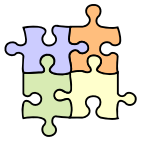
- Si consideri un servizio che consente di specificare un ordine relativo a più prodotti
  - una possibile implementazione stateless
    - l'intero ordine viene richiesto mediante una richiesta singola – magari preceduta da una serie di richieste di informazioni sui singoli prodotti
  - una possibile implementazione stateful
    - il client richiede di iniziare un ordine
    - per ciascun prodotto, il client fa una richiesta per aggiungere il prodotto all'ordine (quello definito nell'ambito della sessione)
    - infine, il client invia una richiesta in cui conferma l'ordine





# Esempio

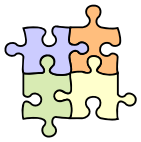
- Si consideri un servizio che consente di specificare un ordine relativo a più prodotti
  - un'altra possibile implementazione stateless
    - il client richiede di iniziare un ordine – il server applicativo restituisce il codice dell'ordine
    - per ciascun prodotto, il client fa una richiesta per aggiungere un prodotto ad un ordine – il codice dell'ordine viene passato nella richiesta
    - infine, il client invia una richiesta in cui conferma l'ordine – il codice dell'ordine viene passato nella richiesta
    - in questo caso, le informazioni sull'ordine sono (ad esempio) salvate dal server applicativo in una base di dati
      - ma il server applicativo non ha necessità di gestire informazioni circa lo stato della sessione – è stateless



# * Architetture a oggetti distribuiti

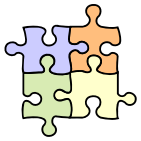
- Nelle architetture client/server, si fa distinzione tra client e server
  - i client possono richiedere servizi ai client – il viceversa non è ammesso
  - questo porta alla definizione di una struttura gerarchica di clienti-serventi
  - è possibile pensare a soluzioni meno vincolate – più flessibili?
- Le architetture a oggetti distribuiti adottano un approccio più generale – in cui
  - vengono mantenute le importanti nozioni di “servizio” e “interfaccia”
  - viene adottato un paradigma ad oggetti
  - viene rimossa la distinzione netta tra client e server





# Richiamo: Paradigma a oggetti

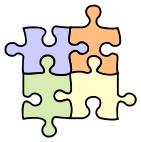
- Nel paradigma di programmazione a **oggetti**
  - ciascun *oggetto* incapsula stato e comportamento
    - il comportamento di un oggetto è descritto dalla sua *interfaccia* (definita implicitamente o esplicitamente)
      - è la specifica dei metodi che possono essere invocati pubblicamente
    - l’implementazione del comportamento è privata
    - anche lo stato di un oggetto è gestito privatamente
  - ciascun oggetto è identificato mediante un *riferimento univoco*
    - questo è necessario, in particolare, nell’invocazione di metodi
  - un programma è composto da una collezione di oggetti
    - nella programmazione ad oggetti tradizionale, tutti gli oggetti risiedono normalmente in un singolo processo



# Paradigma a oggetti distribuiti

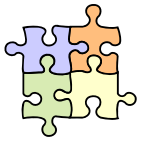
- Nel paradigma di programmazione a **oggetti distribuiti**
  - due tipi di oggetti
    - *oggetti locali* – sono visibili localmente ad un processo
    - *oggetti remoti* – possono essere distribuiti in più calcolatori/processi
  - ciascun *oggetto* incapsula stato e comportamento
  - gli oggetti remoti possono essere utilizzati mediante la loro *interfaccia remota* – deve essere definita esplicitamente
  - gli oggetti remoti sono identificati mediante un *riferimento remoto* (univoco)
    - la cui conoscenza è necessaria per invocare metodi remoti
  - un programma distribuito è composto da una collezione di oggetti, locali e remoti
    - ciascun oggetto può interagire con quelli che conosce – a lui locali o remoti (alcuni visibili globalmente)



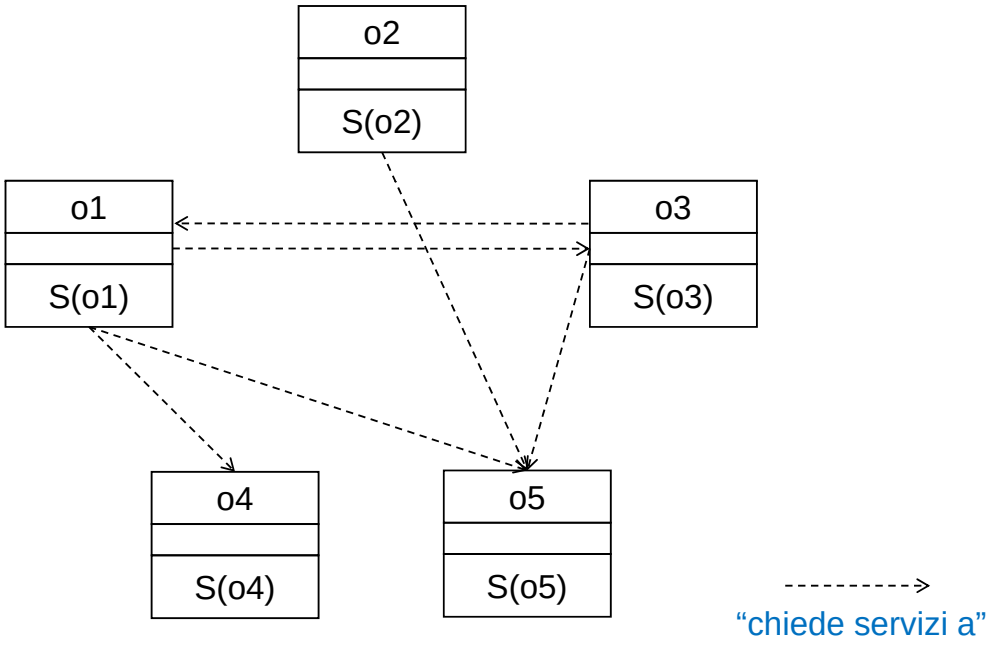


# Architetture a oggetti distribuiti

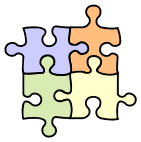
- Dunque, in un'**architettura a oggetti distribuiti** (*DOA*)
  - gli elementi del sistema sono chiamati *oggetti remoti* (o *distribuiti*)
    - in realtà, macro-oggetti
    - comunque realizzati con tecnologie ad oggetti
  - ciascun oggetto remoto fornisce dei servizi
    - descritti mediante la sua interfaccia remota
  - un oggetto può richiedere/fornire servizi ad altri oggetti
  - gli oggetti possono essere distribuiti tra diversi calcolatori



# Un'architettura a oggetti distribuiti

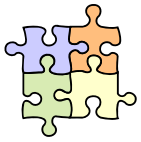




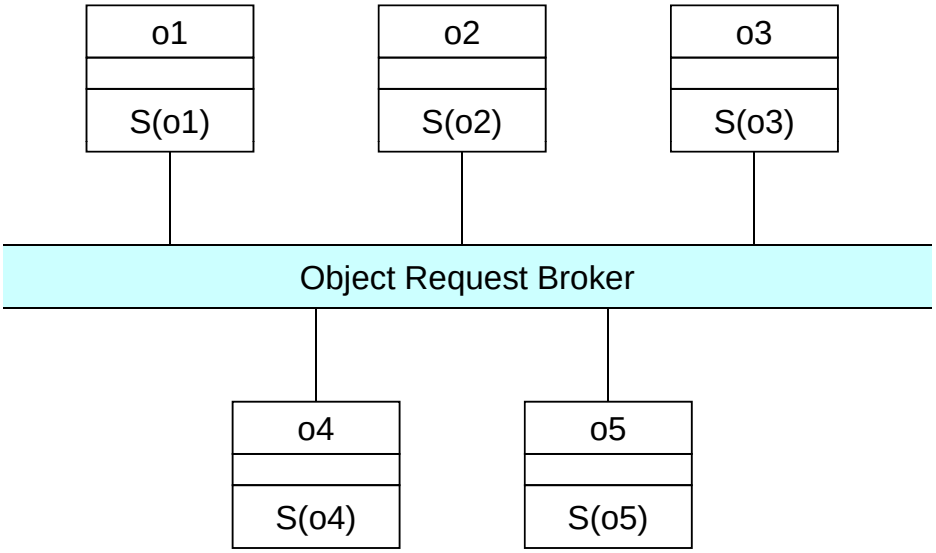



# Comunicazione tra oggetti distribuiti

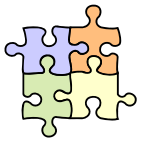
- In una DOA, la comunicazione tra oggetti distribuiti avviene mediante del middleware opportuno
  - solitamente un *broker* – nello specifico, un *object request broker* (*ORB*)
  - l'ORB agisce essenzialmente come un bus software per consentire la comunicazione tra i vari oggetti remoti (distribuiti)



# Comunicazione mediante broker

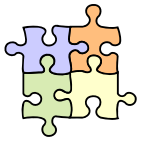




# Interazione tra oggetti distribuiti

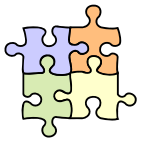
- Modalità di interazione tra oggetti distribuiti
  - gli oggetti server (oggetti che offrono servizi) devono registrare i servizi che offrono presso il broker
    - più precisamente, presso il servizio di directory offerto dal broker
  - gli oggetti client possono consultare il broker per ottenere un riferimento remoto ad un oggetto server
    - consultando il servizio di directory – ad esempio a partire da un identificatore simbolico dell'oggetto server di interesse
  - gli oggetti client possono poi fare richieste agli oggetti server
    - usando il broker come indirezione
  - "client" e "server" usati per indicare il ruolo nell'ambito di una possibile interazione
    - gli oggetti possono anche interagire come "pari"



# Caratteristiche delle DOA

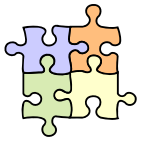
- Conseguenze
  - ☺ architettura aperta, flessibile e scalabile
  - ☺ possibile riconfigurare il sistema dinamicamente, migrando gli oggetti tra i calcolatori – consente di rimandare decisioni su dove fornire i servizi, oppure di cambiare decisioni per sostenere, ad esempio, scalabilità
  - ☺ consente l'introduzione dinamica di nuove risorse, quando richieste
  - ☺ la manutenibilità può essere favorita – con oggetti a grana piccola – coesi e poco accoppiati
  - ☺ l'affidabilità può beneficiare del fatto che lo stato degli oggetti è incapsulato





# Caratteristiche delle DOA

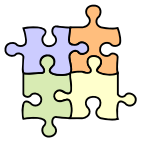
- Conseguenze
  - ☹ le prestazioni peggiorano se sono utilizzati molti oggetti a grana piccola o con servizi a grana piccola
  - 😐 le prestazioni dipendono dalla topologia e dalla grana degli oggetti e della loro interfaccia
    - bene con oggetti a grana grossa, che comunicano poco
  - 😐 la sicurezza beneficia dall'incapsulamento dei dati – ma la frammentazione dei dati influisce negativamente
  - ☹ maggior complessità rispetto ai sistemi client/server



# Usi delle DOA

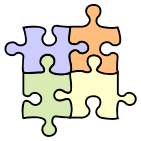
- La DOA può essere usata come un “modello logico” per strutturare ed organizzare il sistema
  - gli elementi dell’architettura sono macro-oggetti che offrono servizi e incapsulano lo stato
  - il modello a oggetti viene usato per ragionare ai vari livelli di decomposizione del sistema
- La DOA può essere usata come un approccio flessibile all’implementazione di sistemi client/server
  - l’architettura logica del sistema è client/server
  - client e server sono realizzati come oggetti distribuiti – che comunicano con una tecnologia DOA





# Architetture a oggetti distribuiti

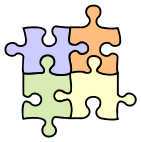
- Il funzionamento di un ORB è descritto dal pattern architetturale Broker
  - vedi dopo
- Per le tecnologie ad oggetti distribuiti, vedi anche
  - dispensa su Oggetti distribuiti e invocazione remota



# * Proxy [POSA, GoF]

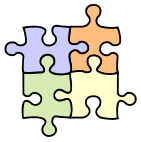
- Il design pattern **Proxy** fa comunicare i client di un componente con un rappresentante del componente – non con il componente stesso
  - l'introduzione di un tale "segnaposto" può sostenere diversi scopi – ad es., aumentare l'efficienza, semplificare l'accesso, consentire la protezione da accessi non autorizzati
- Proxy [GoF]
  - fornisce un surrogato o un segnaposto per un altro oggetto – per controllarne l'accesso



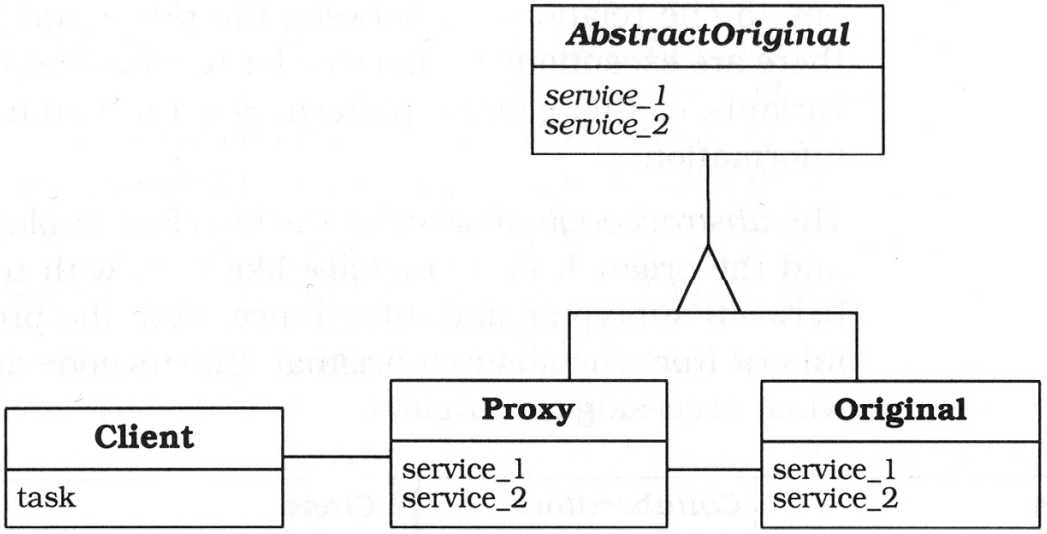


# Proxy

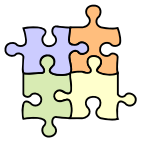
- Contesto
  - l'accesso diretto tra un client e un componente è tecnicamente possibile – ma non è l'approccio migliore
- Problema
  - l'accesso diretto ad un componente è spesso inappropriato – per motivi di accoppiamento, efficienza, sicurezza, ...
- Soluzione
  - il client viene fatto comunicare con un rappresentante del componente – il *proxy*
  - il proxy non fa solo da intermediario – ma esegue delle ulteriori pre- e post-elaborazioni
    - ad es., effettua un controllo dell'accesso



# Struttura

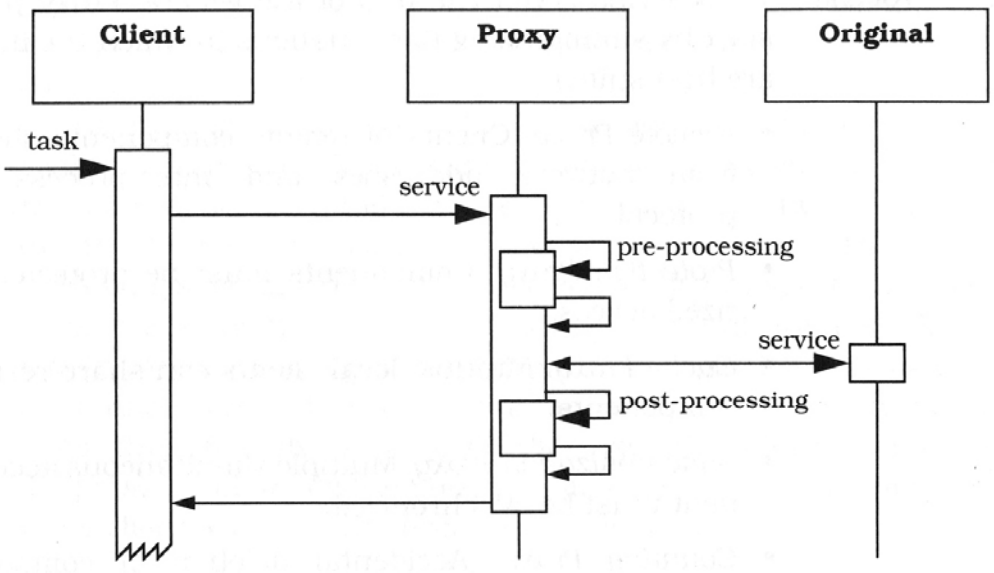


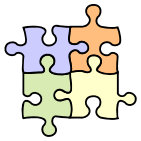


# Scenario

- Il Proxy è un intermediario tra il client e il fornitore del servizio – con la stessa interfaccia
  - ma esegue delle ulteriori pre- e post-elaborazioni

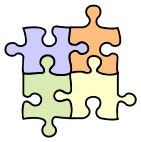




# Possibili applicazioni

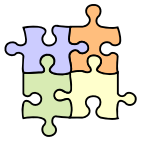
- Alcune varianti comuni di Proxy – corrispondono ad applicazioni specifiche
  - Remote Proxy – intermediario per l'accesso ad un componente remoto – usato, ad esempio, nel pattern architetturale Broker
  - Protection Proxy – gestisce il controllo degli accessi ad un componente
  - Cache Proxy – diversi client locali possono condividere i risultati da componenti remoti
  - Synchronization Proxy – sincronizza gli accessi concorrenti ad un componente
  - Virtual Proxy – sostiene l'accesso pigro ad informazioni il cui accesso è costoso
  - Firewall Proxy – protegge i client locali nell'accesso a servizi di rete
  - ...





# * Broker [POSA]

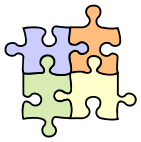
- Il pattern architetturale **Broker**
  - può essere usato per strutturare sistemi distribuiti con componenti disaccoppiati che interagiscono mediante l'invocazione di servizi remoti
  - un componente broker è responsabile del coordinamento della comunicazione
    - ad es., per inoltrare richieste e trasmettere risposte o eccezioni
- Broker è un pattern alla base di molte tecnologie ad oggetti/servizi distribuiti
  - costituisce un'infrastruttura di comunicazione che rende trasparenti all'applicazione (ed al programmatore) alcune complessità della distribuzione



# Esempio

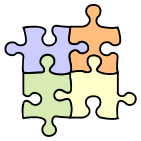
- City Information System – CIS
  - sistema di informazioni turistiche
  - portale verso altri sistemi (esterni) che effettivamente offrono servizi per turisti
    - informazioni su alberghi e ristoranti, trasporti pubblici, musei, visite guidate, ...
    - in alcuni casi con la possibilità di fare prenotazioni/acquisti
  - ci sono più sistemi esterni che possono soddisfare richieste
  - è possibile la registrazione dinamica di nuovi sistemi esterni
  - il CIS è un punto di contatto singolo per il turista verso i sistemi esterni





# Broker

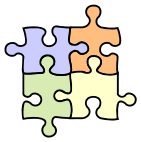
- Contesto
  - ambiente distribuito
  - più componenti erogatori di servizi – eventualmente eterogenei
- Problema
  - un sistema in grado di utilizzare dei componenti distribuiti in modo flessibile – con queste caratteristiche
    - componenti disaccoppiati ma interoperabili
    - componenti aggiunti/rimossi/sostituiti a runtime
    - trasparenza nell'accesso ai componenti – ad es., di locazione
  - in generale, le applicazioni distribuite dovrebbero gestire le problematiche connesse alla distribuzione usando un modello di programmazione che protegga le applicazioni stesse dai dettagli della rete e della posizione dei componenti in rete



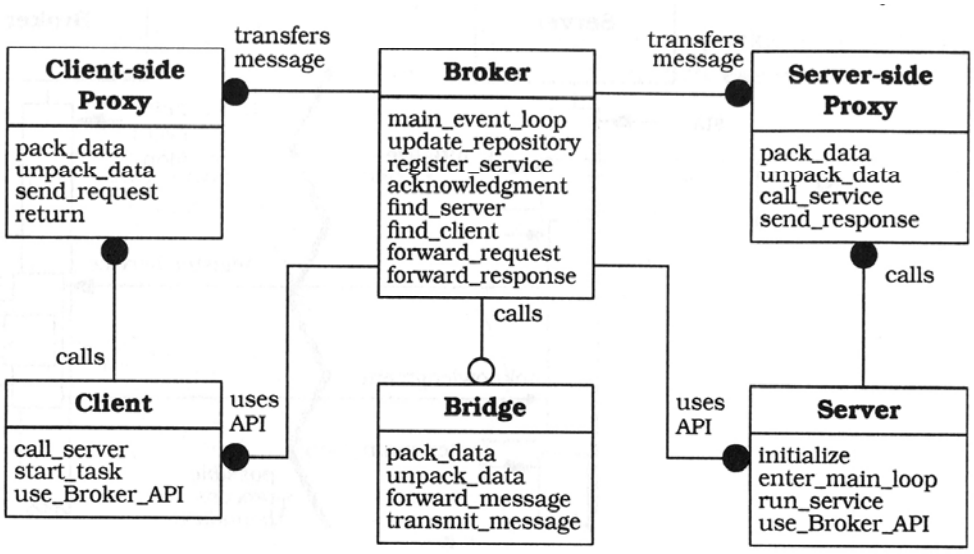
# Broker

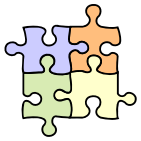
- Soluzione
  - introdurre un componente *broker* – per avere un miglior disaccoppiamento tra client e server
    - il broker incapsula i dettagli dell'infrastruttura di comunicazione in un sistema distribuito
    - definisce un modello di programmazione distribuita in cui i client possono richiedere servizi remoti come se fossero servizi locali
    - in questo modo, i dettagli della comunicazione sono separati dalle funzionalità applicative
  - dinamica dell'interazione
    - i server registrano i propri servizi presso il broker
    - i client accedono i servizi indirettamente, tramite il broker – che inoltra le loro richieste e gli trasmettono le risposte





# Struttura

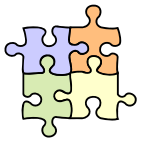




# Partecipanti (1)

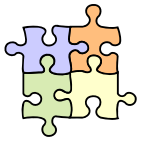
- *Server*
  - un oggetto o componente che offre servizi
  - i servizi sono esposti tramite una interfaccia – ad es., IDL
- *Client*
  - un oggetto (o componente o applicazione) che vuole fruire di servizi
  - inoltra le proprie richieste al Broker
- Client e server non nell'accezione dello stile client/server
  - ma nell'accezione DOA – in una certa interazione, un oggetto/componente client vuole fruire di un servizio fornito da un oggetto/componente server





# Partecipanti (2)

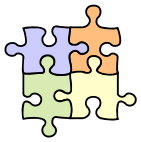
- *Broker*
  - un bus/"messaggero" responsabile della trasmissione di richieste e risposte tra client e server
  - offre a client e server (mediante delle API) funzionalità per registrare servizi e per richiedere l'esecuzione di servizi
  - può offrire altri servizi aggiuntivi – ad es., naming (directory)



# Partecipanti (3)

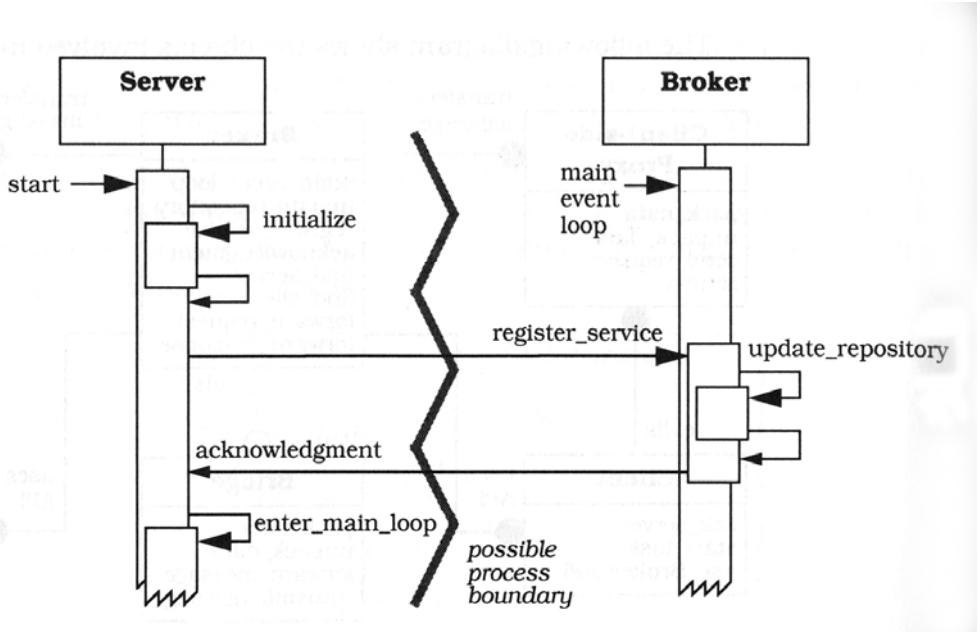
- *Proxy lato client*
  - intermediario tra client e broker
    - è il rappresentante lato client del servizio richiesto
  - fornisce trasparenza, perché un oggetto remoto appare locale al client
- *Proxy lato server*
  - intermediario tra broker e server
  - responsabile di ricevere richieste dal broker, di invocare il servizio effettivo e di trasmettere le risposte al broker
- *Bridge*
  - componente opzionale – per collegare/far interoperare più broker
  - ad es., un broker per “tecnologia”, ciascuno con il suo bridge

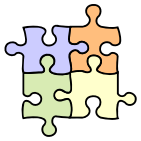




# Scenario 1

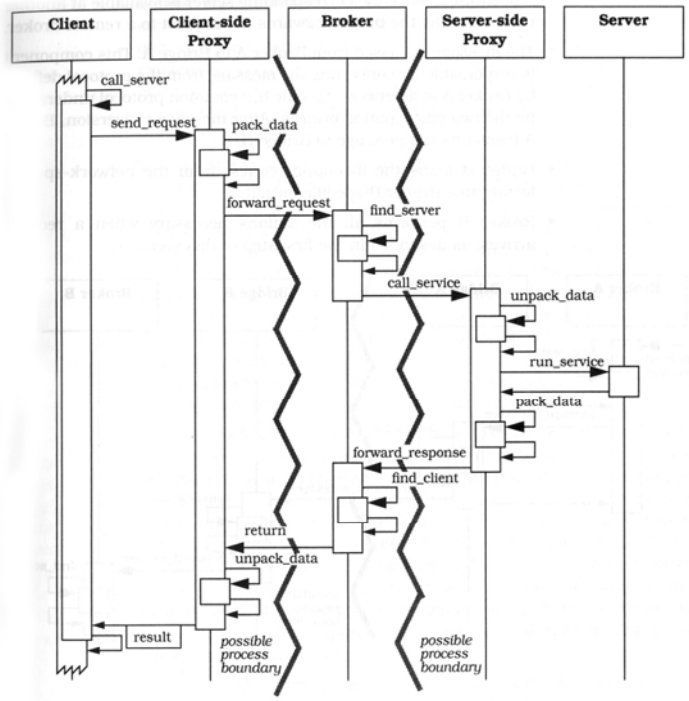
- Registrazione di un server presso un broker



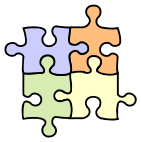


# Scenario 2

- Gestione di una richiesta da parte di un client

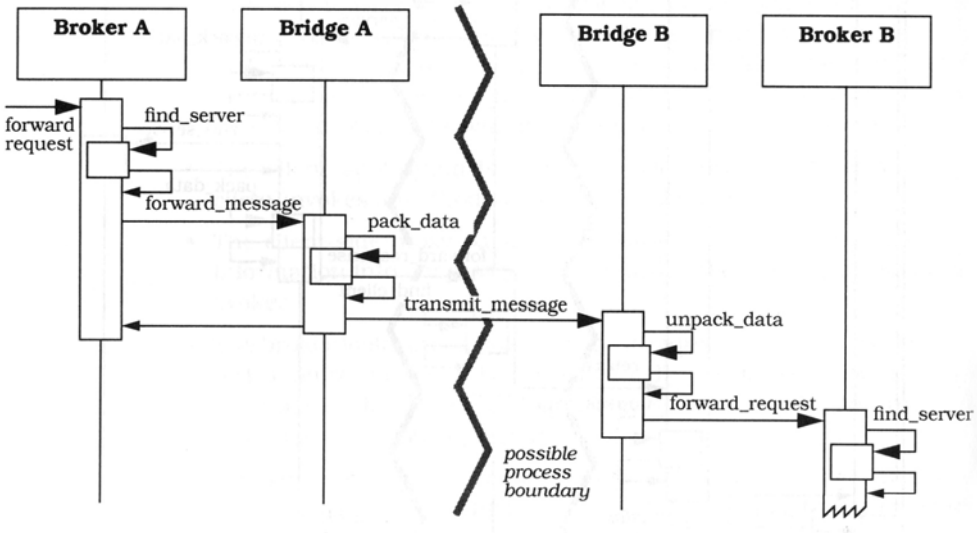


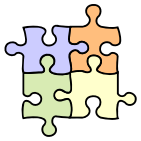


# Scenario 3

- Interazione tra broker multipli
  - ciascun broker gestisce componenti “omogenei”
  - i diversi broker comunicano mediante bridge

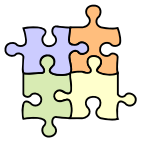




# Conseguenze

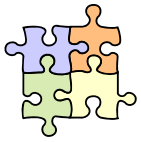
- Benefici
  - ☺ trasparenza dalla posizione – il client non ha bisogno di sapere dove si trova il server – i server possono ignorare la posizione dei loro client
  - ☺ modificabilità ed estendibilità dei componenti
  - ☺ interoperabilità tra tipi di broker diversi
  - ☺ riusabilità di servizi esistenti





# Conseguenze

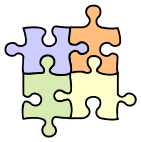
- Inconvenienti
  - ☹ riduzione delle prestazioni
  - ☹ minor tolleranza ai guasti rispetto ad una soluzione non distribuita
  - ☹ testing difficile



# Esempio: Java RMI

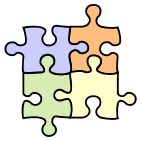
- Java RMI
  - l'architettura Java RMI è sostanzialmente basata su Broker
    - il ruolo del broker è svolto dalla JVM remota che contiene l'oggetto servente
    - il proxy lato client è lo stub generato da rmic
    - il ruolo del proxy lato server (lo skeleton) è svolto dalla JVM remota
  - la comunicazione Java RMI è normalmente basata sul protocollo JRMP
    - tuttavia, la tecnologia Java RMI-IIOP (Java RMI over Internet Inter-ORB Protocol) consente di interoperare con dei broker CORBA – si tratta di una "tecnologia bridge"





# * Architetture peer-to-peer

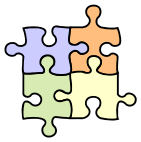
- I sistemi **peer to peer** (*p2p*)
  - sono sistemi distribuiti e decentralizzati, in cui l'elaborazione può essere svolta da ogni calcolatore della rete
  - in linea di principio, nessun distinzione tra client e server
- Il sistema complessivo è progettato per trarre vantaggio dalla potenza di calcolo e memorizzazione di un'ampia rete di calcolatori



# Architetture peer-to-peer

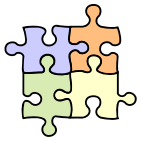
- Struttura (di base)
  - un singolo tipo di elemento – il *peer* (pari)
  - i formati e i protocolli della comunicazione tra peer sono immersi nei peer stessi
  - i peer comunicano/condividono risorse di calcolo (memorizzazione, cpu, contenuti, ...) in modo diretto
- Esempi
  - SETI@Home, Grid
  - sistemi per uso personale, ad es., per il file sharing – Napster, Gnutella, Kazaa, ...





# Caratterizzazione delle architetture p2p

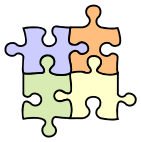
- Due aspetti caratterizzanti delle architetture peer-to-peer
  - la condivisione di risorse (memorizzazione, cpu, ...) avviene in modo diretto – e non tramite la presenza di un server centralizzato
    - ma è possibile utilizzare dei server centralizzati per compiti specifici – ad es., la localizzazione delle risorse
  - capacità di trattare l'instabilità e la connettività variabile come la norma
    - con la conseguenza di essere auto-adattabili e tolleranti ai guasti
    - questa è una delle differenziazioni tra architetture p2p e architetture Grid – in cui le risorse sono sì numerose e disperse – ma normalmente sono presenti in modo stabile ed affidabile



# Classificazione delle applicazioni p2p

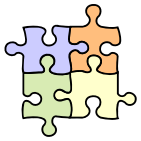
- Alcune categorie di applicazioni p2p
  - comunicazione e collaborazione
    - ad es., chat – comunicazione diretta e real-time
  - calcolo distribuito
    - ad es., seti@home
    - per condividere capacità di calcolo (processori)
    - richiede un coordinamento centrale – per suddividere l'elaborazione complessiva in elaborazioni parziali – e ricostruire i risultati complessivi da quelli parziali
  - servizi di supporto di internet
  - distribuzione di contenuti
    - la categoria più nota – quella che suscita maggior interesse
- Ci concentriamo su quest'ultima categoria





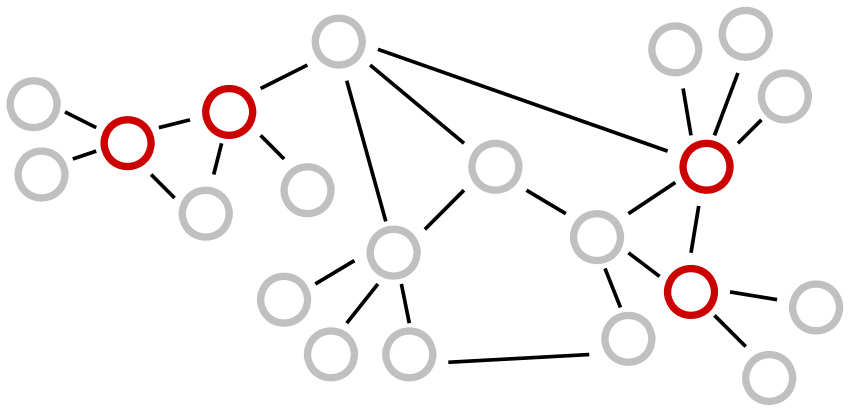
# Arch. p2p per la distribuzione di contenuti

- *Architetture decentralizzate pure*
  - i nodi (*peer*) sono tutti di un tipo – svolgono sia il ruolo di client che di server – non c'è coordinamento centrale delle loro attività
- *Architetture decentralizzate ibride*
  - c'è un server centrale che facilità l'interazione tra i peer
- *Architetture parzialmente centralizzate*
  - la maggior parte dei nodi sono *peer*
  - alcuni nodi (*super-peer*) assumono un ruolo più importante – fungono da indice centralizzato dei file condivisi dai peer
  - super-peer spesso stabiliti dinamicamente

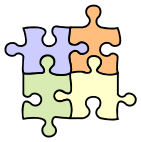


# Topologie peer-to-peer e topologie di rete

- Vengono utilizzate diverse topologie logiche per la connessione dei peer
  - si tratta di un'overlay network sulla rete fisica sottostante
- Ad esempio
  - grafo generico, albero, anello
  - con super-peer – anello o grafo centralizzato



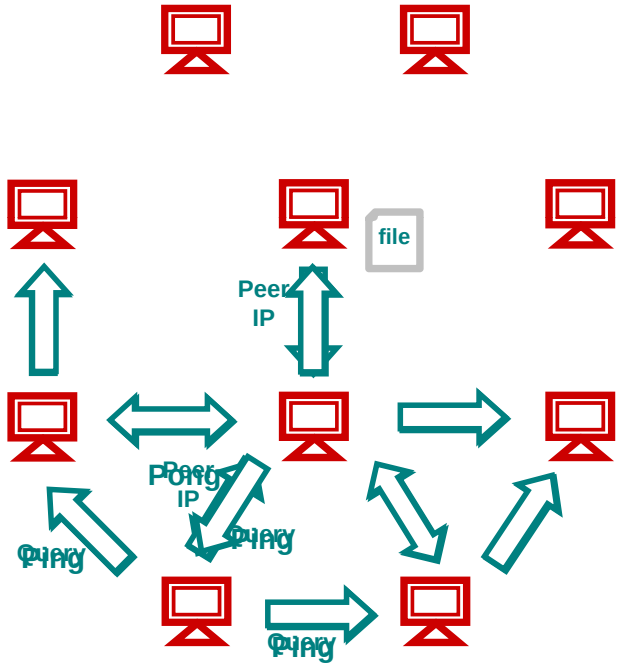




# Responsabilità dei peer

- Nelle architetture p2p (pure)
  - i nodi sono sia elementi funzionali che commutatori di comunicazione
  - possono indirizzare dati e segnali di controllo (query) da un nodo all'altro
- Un nodo
  - mantiene dati – repliche di dati
  - risponde a query
  - inoltra query a cui non sa rispondere ad altri nodi
  - restituisce dati a chi glieli richiede
- L'architettura può essere altamente ridondante

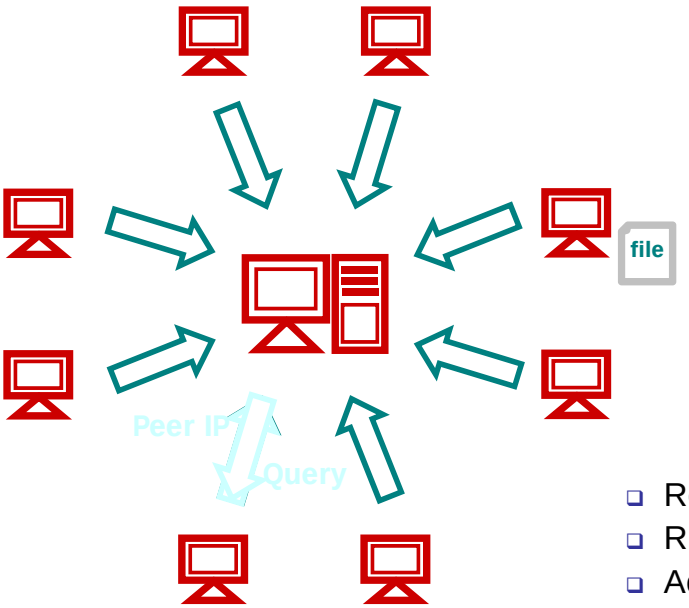
# Architettura decentralizzata pura



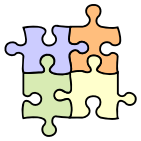

- Ping
- Pong e flooding del Ping
- Query
- Flooding Query
- Acquisizione Peer IP
- Download (diretto)



# Architettura decentralizzata ibrida



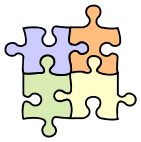

- Registrazione presso il server
- Richiesta di contenuti
- Acquisizione lista dei peer che soddisfano la richiesta
- Download (diretto) dei contenuti



# Architetture P2P

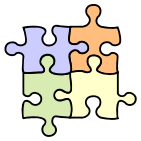
- Conseguenze
  - ☺ disponibilità ed affidabilità – sono eliminati punti di fallimento singoli
  - ☺ scalabilità
  - 😐 supporto per l'anonimato
  - ☹ possibili partizionamenti della rete
  - ☹ difficoltà a garantire una certa qualità di servizio
  - ☹ i dati condivisi devono essere statici, oppure devono essere tollerati dati non aggiornati





# * Discussione

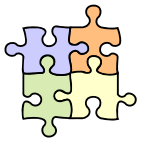
- I pattern architetturali descritti in questa dispensa sono alla base di molti sistemi distribuiti attuali
  - le tecnologie sottostanti, ed i relativi pattern di utilizzo, si sono successivamente evoluti per semplificare ulteriormente lo sviluppo dei sistemi distribuiti e favorire la loro interoperabilità



# Discussione

- Le architetture client/server e ad oggetti distribuiti
  - sono alla base di molte tecnologie per sistemi distribuiti
  - si sono nel tempo evolute per semplificare ulteriormente lo sviluppo dei sistemi distribuiti
- Le evoluzioni sono mirate a superare alcuni “limiti” di queste tecnologie di base – ad esempio
  - messaging
    - consente una modalità di interazione asincrona basata sullo scambio di messaggi
  - componenti
    - con una piattaforma che offre servizi ai suoi componenti – con la possibilità di configurarli dinamicamente
  - servizi
    - interoperabilità tra componenti in piattaforme diverse





# Discussione

- Middleware per il messaging
  - consente lo scambio asincrono di messaggi
  - sostiene accoppiamento debole, flessibilità ed affidabilità
- Middleware per componenti
  - i componenti vivono in contenitori (application server) in grado di gestire la configurazione e la distribuzione dei componenti, e fornire ad essi funzionalità di supporto
  - possibile sia la comunicazione sincrona che asincrona
- Middleware orientato ai servizi
  - enfasi sull’interoperabilità tra componenti eterogenei, sulla base di protocolli standard aperti ed universalmente accettati
  - possibile sia la comunicazione sincrona che asincrona
  - flessibilità nell’organizzazione dei suoi elementi (servizi)