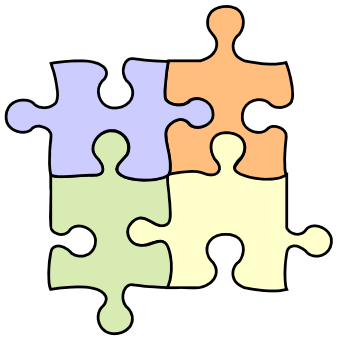
Luca Cabibbo



# Architetture Software

# Architetture dei sistemi distribuiti

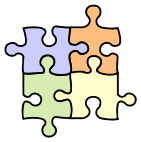
**Dispensa ASW 410**

ottobre 2014

*Un sistema distribuito è*
*un sistema in cui il fallimento di un computer*
*di cui nemmeno conosci l'esistenza*
*può rendere inutilizzabile il tuo computer.*

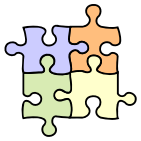
*Leslie Lamport*





## - Fonti

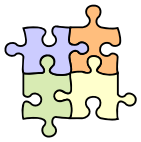
- [POSA1] Pattern-Oriented Software Architecture, 1996
- [POSA4] Pattern-Oriented Software Architecture – A Pattern Language for Distributed Computing, 2007
- [SAP] Chapter 13, Architectural Tactics and Patterns
- [Alonso et al.] Alonso, Casati, Kuno, Machirajy Web Services – Concepts, Architectures and Applications, 2004 – Chapter 1, Distributed Information Systems
- [Bernstein] Phil Bernstein, Middleware, Comm. of the ACM, 1996
- [CDK] Coulouris, Dollimore, Kindberg, Distributed Systems – Concepts and Design, 4th ed., 2005
- [Shaw] Mary Shaw, Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status, Technical report CMU/SEI-1994-TR-2, 1996



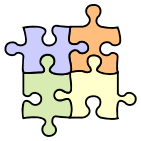
# - Obiettivi e argomenti

- Obiettivi
  - introdurre i sistemi distribuiti
  - presentare alcuni stili architetturali fondamentali per sistemi distribuiti – client/server, peer-to-peer, architettura a oggetti distribuiti
- Argomenti
  - introduzione
  - connettori
  - middleware
  - un po' di storia dei sistemi distribuiti
  - stile client/server
  - stile peer-to-peer
  - architetture a oggetti distribuiti
  - discussione





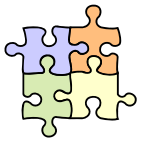
# - Wordle



# * Introduzione

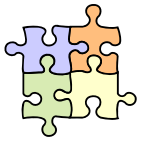
- Tutti i grandi sistemi informatici sono oggi sistemi distribuiti
- Alcune possibili definizioni – un **sistema distribuito** è
  - un sistema in cui l'elaborazione delle informazioni è distribuita su più computer – anziché centralizzata su una singola macchina
  - un sistema di elaborazione in cui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





# Benefici della distribuzione

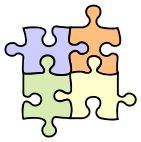
- ☺ Connettività e collaborazione
  - possibilità di condividere risorse hardware e software (compresi dati e applicazioni)
- ☺ Prestazioni e scalabilità
  - la possibilità di aggiungere risorse fornisce la capacità di migliorare le prestazioni e sostenere un carico che aumenta (scalabilità orizzontale)
- ☺ Tolleranza ai guasti
  - grazie alla possibilità di replicare risorse



# Benefici della distribuzione

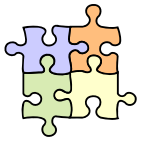
- ☺ Sistemi inerentemente distribuiti
  - alcune applicazioni sono inerentemente distribuite – non sono possibili opzioni diverse
- ☺ Apertura
  - l'uso di protocolli standard aperti favorisce l'interoperabilità di hardware e software di fornitori diversi
- ☺ Economicità
  - i sistemi distribuiti offrono spesso (*ma non sempre*) un miglior rapporto prezzo/qualità che i sistemi centralizzati basati su mainframe





# Svantaggi legati alla distribuzione

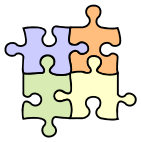
- ☹ Complessità
  - i sistemi distribuiti sono più complessi di quelli centralizzati
  - più difficile capirne e valutarne le qualità
- ☹ Sicurezza
  - l'accessibilità in rete pone problematiche di sicurezza
- ☹ Non prevedibilità
  - i tempi di risposta dipendono dal carico del sistema e dal carico della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



# Svantaggi legati alla distribuzione

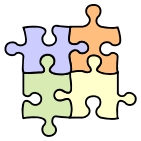
- ☹ Complessità accidentale
  - introdotta dall'uso di strumenti di sviluppo non opportuni
- ☹ Metodi e tecniche non adeguati
  - i metodi di analisi e progettazione più diffusi fanno spesso riferimento allo sviluppo di applicazioni mono-processo, mono-thread
  - i metodi di analisi e progettazione per 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

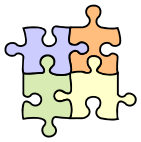
- La sfida posta dai sistemi distribuita
  - ottenere i possibili benefici
  - minimizzando gli svantaggi



# Parentesi: mainframe

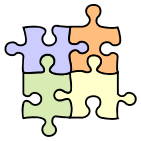
- La tecnologia dei *sistemi centrali* – più noti come *mainframe* – è ancora attuale e in continua evoluzione [Corso di Sistemi Centrali, 2007]
  - è una tecnologia "ricca di funzionalità sempre più avanzate e di innovazioni tecniche via via più sofisticate"
    - basata su tecniche/tattiche *ad hoc* per prestazioni, scalabilità, disponibilità, sicurezza, ...
  - "oggi i mainframe sono presenti e hanno un ruolo insostituibile in tutto il mondo nelle infrastrutture informatiche delle più importanti aziende industriali e finanziarie, nelle società di servizi pubbliche e private e nelle grandi istituzioni nazionali ed internazionali"
    - è una tecnologia adatta a sistemi molto grandi e complessi
  - è una tecnologia in competizione con altre tecnologie hardware moderne, come quelle dei sistemi multiprocessore e i cluster





# Parentesi: virtualizzazione

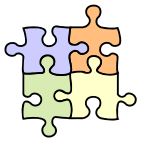
- Un'altra tecnologia importante oggi è quella della *virtualizzazione*
  - consente a un singolo server fisico di ospitare N server virtuali
    - questo "singolo server" potrebbe essere un mainframe oppure un nodo di un cluster
  - ad es., per la server consolidation – gestione più semplice – TCO inferiore – ad es., riducendo i consumi di energia e aumentando la percentuale di utilizzo
  - in ogni caso, l'organizzazione dei server virtuali può essere basata sugli stili architetturali per sistemi distribuiti



# * Connettori

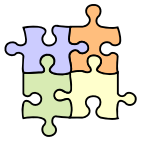
- In un'architettura software è possibile distinguere due tipi principali di elementi software
  - *componenti*
    - elementi responsabili dell'implementazione di *funzionalità* e della gestione di *dati*/*informazioni*
  - *connettori*
    - elementi responsabili delle *interazioni* tra componenti – i connettori caratterizzano assemblaggio e integrazione di componenti
- Questa distinzione riflette la sostanziale indipendenza tra gli aspetti funzionali e quelli relativi alle interazioni
  - alcune considerazioni sono state già fatte nella dispensa sull'Introduzione ai connettori – e qui vengono riassunte per comodità





# Componenti e connettori

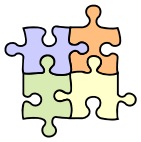
- Secondo [Shaw]
  - i *componenti* sono il luogo della computazione e dello stato
    - ogni componente ha una *specifica di interfaccia* che definisce le sue proprietà (sia funzionalità che proprietà di qualità, ad esempio circa le prestazioni)
    - ogni componente è di un qualche tipo – ad es., filtro, server, memoria, ...
    - l'interfaccia di un componente comprende la specifica dei "ruoli" (chiamati *player* ) che un componente può rivestire nell'interazione con altri componenti



# Componenti e connettori

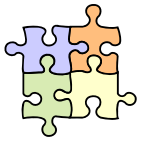
- Inoltre [Shaw]
  - i *connettori* sono il luogo delle relazioni tra componenti
    - i connettori sono mediatori di interazioni – sono “ganci” tra componenti
    - ogni connettore ha una *specifica di protocollo* che definisce le sue proprietà – queste proprietà comprendono regole sul tipo di interfacce che è in grado di mediare, nonché impegni sulle proprietà dell’interazione, come ad es. affidabilità, prestazioni e ordine in cui le cose avvengono
    - ogni connettore è di un qualche tipo – ad es., chiamata di procedura remota, pipe, evento, broadcast, ...
    - il protocollo di un connettore comprende la specifica dei *ruoli* che devono essere soddisfatti – ad es., client e server
  - la composizione dei componenti avviene mettendo in relazione player di componenti con ruoli di connettori





# Connettori

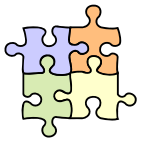
- L’esperienza ha mostrato che ci sono vari motivazioni per trattare i connettori separatamente dai componenti [Shaw]
  - la scelta e progettazione dei connettori (ovvero, delle interazioni) è importante tanto quanto quella dei componenti
    - alcune informazioni (scelte architetturali) del sistema non hanno una collocazione naturale in nessuno dei suoi componenti
  - la progettazione dei connettori può essere fatta separatamente da quella dei componenti
  - i connettori sono potenzialmente astratti – e riutilizzabili in più contesti
    - sistemi diversi riusano spesso degli stessi pattern di interazione – dunque i connettori sono spesso indipendenti dalle applicazioni
    - questo ha portato allo sviluppo degli strumenti di middleware



# * Middleware

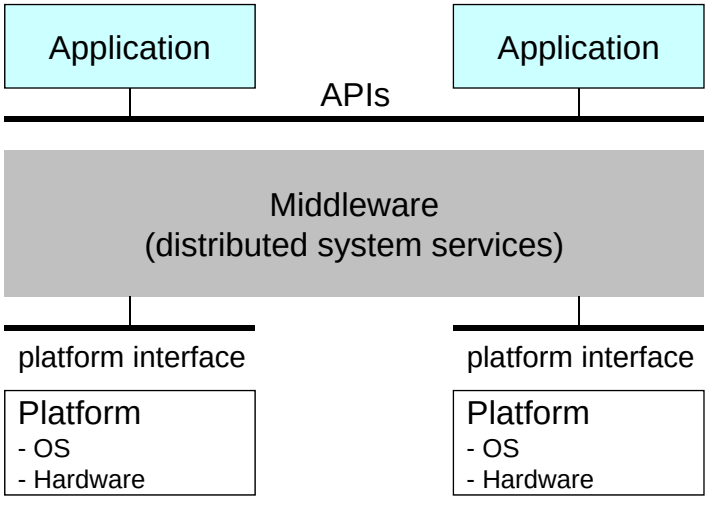
- Lo sviluppo di sistemi software distribuiti è sostenuto dagli strumenti di **middleware**
  - una classe di tecnologie software sviluppate per aiutare gli sviluppatori nella gestione della complessità e della eterogeneità presenti nei sistemi distribuiti [D.E. Bakken, Middleware, 2003]
    - uno strato software “in mezzo” – sopra al sistema operativo, ma sotto i programmi applicativi
    - ciascun middleware fornisce un’astrazione di programmazione distribuita
  - il middleware ha lo scopo di sostenere lo sviluppo dei connettori (sono anch’essi elementi software) per realizzare la comunicazione e le interazioni tra i diversi componenti software di un sistema distribuito

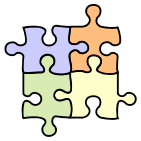




# Middleware

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

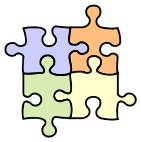




# Middleware

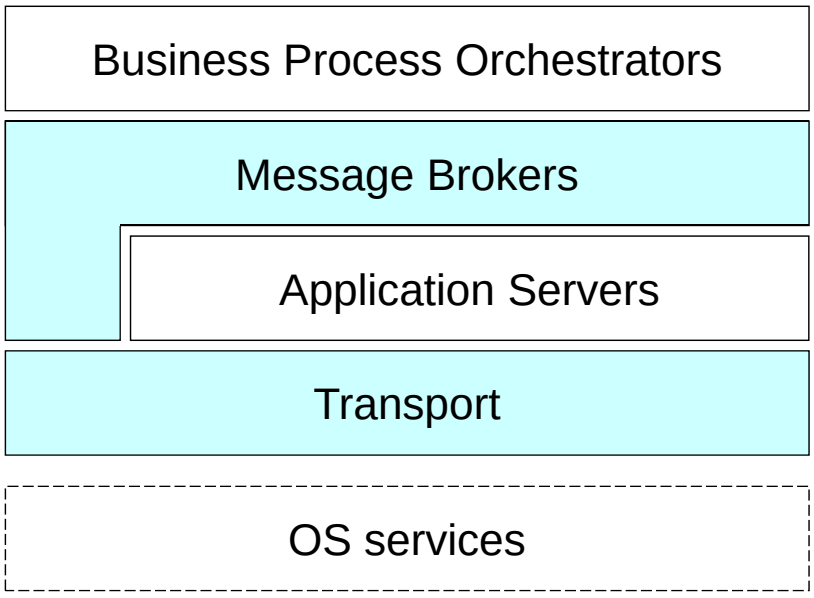
- Il middleware è il software che sostiene il collegamento (plumbing/piping/wiring) tra componenti software – e rende facilmente programmabili i sistemi distribuiti
  - ciascuno strumento di middleware offre una specifica modalità di interazione
    - ad es., RPC offre un paradigma di interazione basato sulla chiamata (sincrona) di procedure remote
    - il messaging, invece, offre un paradigma di programmazione basata sullo scambio (asincrono) di messaggi
  - sulla base di meccanismi di programmazione e API relativamente semplici

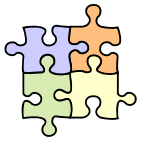




# Middleware – classificazione

- Una possibile classificazione delle tecnologie di middleware [Gorton, Essential Software Architecture, 2006]

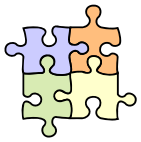




# Middleware

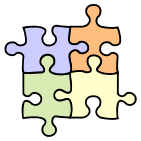
- Varie famiglie di strumenti di middleware
  - middleware per oggetti distribuiti
    - evoluzione di RPC – i componenti distribuiti sono considerati oggetti – con identità, interfaccia, incapsulamento
    - non sostiene la gestione della configurazione degli oggetti distribuiti
  - middleware message-oriented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





# Middleware

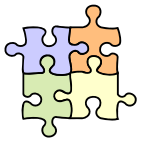
- Varie famiglie di strumenti di middleware
  - middleware per componenti
    - evoluzione del middleware per oggetti distribuiti
    - consente sia la comunicazione sincrona che asincrona
    - i componenti vivono in contenitori (application server) in grado di gestire la configurazione e la distribuzione 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 ...*



# Middleware e stili architetturali

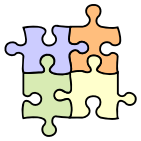
- Relazioni tra strumenti di middleware e stili architetturali
  - l'applicazione di alcuni stili architetturali è sostenuta, dal punto di vista tecnologico, da opportuni strumenti di middleware
    - ad es., lo stile C/S può essere basato su RPC, lo stile C/S a più livelli sugli application server (AS) e middleware a componenti, le SOA sui WS...
  - altri stili architetturali, invece, descrivono l'architettura (dell'infrastruttura) di alcuni strumenti di middleware
    - ad es., RMI è basato su Broker, un AS su Container
- È chiaramente utile conoscere e comprendere queste relazioni
  - per capire quale middleware utilizzare per realizzare un certo stile architetturale – e per capire come utilizzare al meglio un certo middleware
  - per comprendere il funzionamento del middleware – e per realizzare (se serve) nuovi tipi di connettori





# Middleware e trasparenza

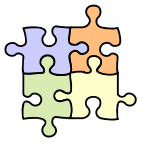
- Ciascuno strumento di middleware ha lo scopo di mascherare qualche tipo di eterogeneità comunemente presente in un sistema distribuito
  - il middleware maschera sempre l'eterogeneità delle reti e dell'hardware
  - alcuni strumenti di middleware mascherano eterogeneità nel sistema operativo e/o nei linguaggi di programmazione
  - alcuni strumenti di middleware mascherano eterogeneità nelle diverse implementazioni di uno stesso standard di middleware
    - ad es., alcune implementazione di Corba o Java EE
  - le astrazioni di programmazione offerte dal middleware possono fornire trasparenza rispetto ai seguenti aspetti
    - posizione, concorrenza, replicazione, fallimenti, mobilità
  - in alternativa, il programmatore dovrebbe farsi esplicitamente carico di queste eterogeneità e di questi aspetti



# Uso efficace del middleware

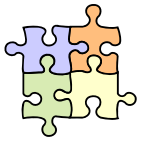
- Se utilizzato in modo opportuno, il middleware consente di affrontare e risolvere diverse problematiche significative nello sviluppo dei sistemi distribuiti
  - consente di generare automaticamente tutti (o quasi) i connettori
  - in questo modo, consente di concentrarsi sullo sviluppo della logica applicativa – e non sui dettagli della comunicazione tra componenti e della piattaforma hw/sw utilizzata
- Per aumentare effettivamente la produttività, il middleware scelto deve essere utilizzato in modo corretto
  - la decisione del middleware da utilizzare richiede delle considerazioni e delle decisioni esplicite
  - è comunque necessaria una buona comprensione del paradigma di comunicazione implementato dal middleware, nonché della sua struttura e dei suoi principi di funzionamento





# Uso efficace del middleware

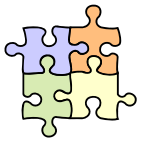
- Malgrado i molti benefici offerti dal middleware, il middleware non è una panacea per i sistemi distribuiti
  - il middleware non può risolvere “magicamente” i problemi derivanti da decisioni di progetto povere
    - che potrebbero invece avere conseguenze negative su stabilità, prestazioni e scalabilità
    - ad esempio, chi sviluppa applicazioni distribuite deve conoscere le differenza tra una chiamata di procedura remota e una chiamata locale
    - inoltre, le applicazioni distribuite devono essere preparate a gestire fallimenti della rete e guasti nei server
  - inoltre il middleware è focalizzato solo sulla comunicazione tra componenti
    - le responsabilità di natura applicativa sono completamente fuori dalla sua portata



## - Di che cosa parleremo

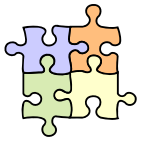
- Nel seguito di questa dispensa sono descritti alcuni stili architetturali fondamentali per sistemi distribuiti
  - stile client/server
  - stile peer-to-peer
- Altri stili architetturali per sistemi distribuiti sono descritti in ulteriori dispense
  - architetture a oggetti distribuiti, a componenti, a servizi
  - messaging per l'integrazione di applicazioni
  - broker come stile architetturale fondamentale utilizzato nella realizzazione di infrastrutture di middleware





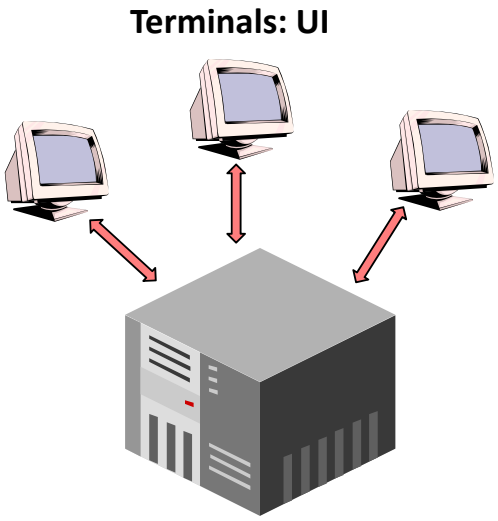
## * Un po' di storia dei sistemi distribuiti

- Un po' di storia delle architetture client/server (e della loro evoluzione)
- Faremo riferimento soprattutto al contesto molto diffuso delle *applicazioni* di tipo *enterprise* – con queste caratteristiche
  - logica applicativa complessa
  - dati complessi e persistenti, di grandi dimensioni
  - transazioni
  - molti utenti concorrenti
  - requisiti di sicurezza
  - non considereremo invece altri contesti comuni – come ad esempio quello del calcolo scientifico

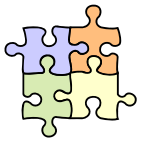


# Applicazioni basate su mainframe – anni '70

- Approccio
  - il mainframe gestisce tutta la logica applicativa e le risorse
  - interazione con gli utenti tramite terminali "stupidi"
- Vantaggi
  - unica tecnologia adeguata (?)
  - semplicità di deployment
- Problemi
  - possibilità di interazione (UI) limitata
  - ogni client richiede risorse nel mainframe – **la scalabilità è limitata**

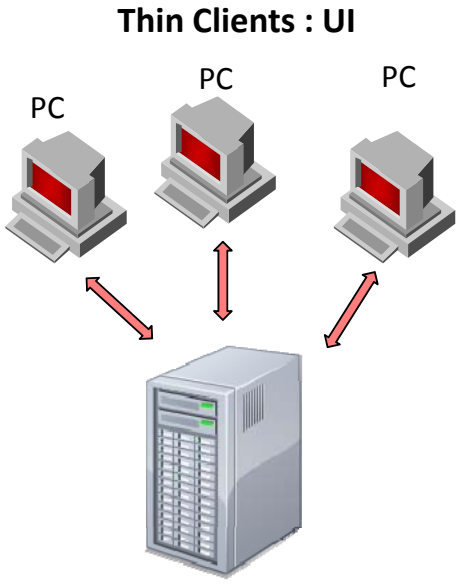


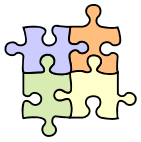


# App. C/S a 2 livelli (thin client) – in. anni '80

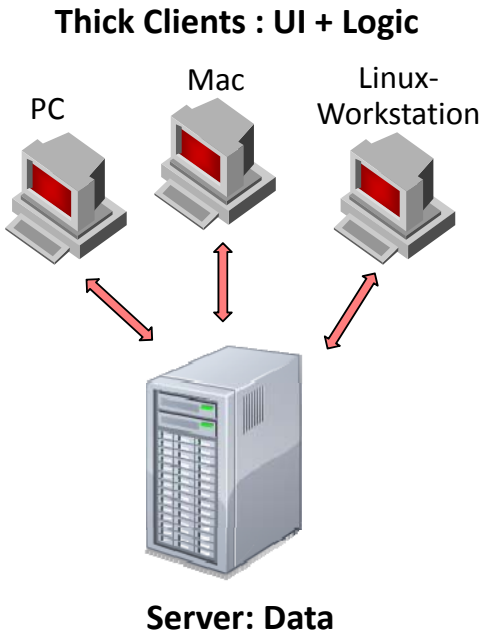
- Approccio
  - disponibilità dei primi PC
  - UI trasferita ai client
  - logica applicativa e dati gestiti dal server centrale
- Vantaggi
  - migliore esperienza di uso (GUI)
- Problemi
  - il server gestisce tutta la logica applicativa – inoltre, ogni client richiede risorse nel server (una connessione e altre risorse) – **la scalabilità è limitata**



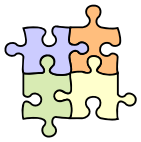


# App. C/S a 2 livelli (thick client) – f. anni '80

- Approccio
  - disponibilità di PC più potenti
  - logica trasferita (in parte) ai client
  - dati gestiti dal server centrale
- Vantaggi
  - migliore esperienza di uso (GUI)
- Problemi
  - ogni client richiede risorse nel server – una connessione e altre risorse – **la scalabilità è limitata**

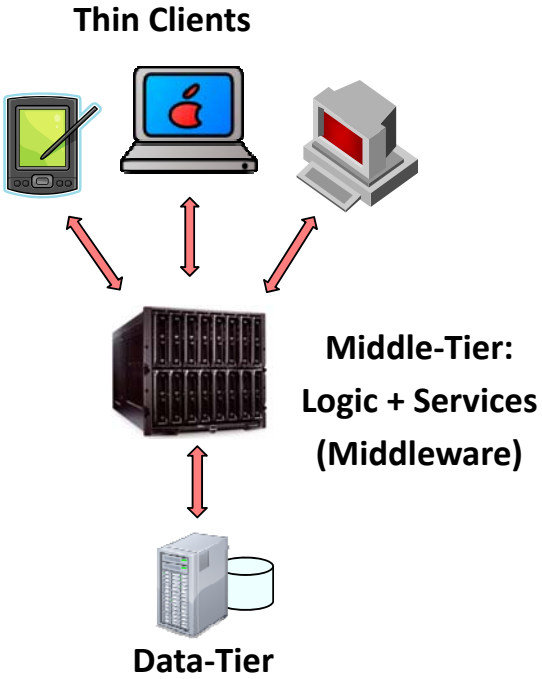


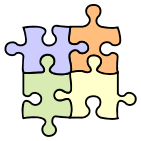


# App. C/S a 3 livelli – anni '90

- Approccio
  - logica trasferita al server intermedio
  - connessioni stateless (non permanenti) tra client e server intermedio
- Vantaggi
  - logica applicativa a fattor comune
  - middleware fornisce servizi comuni – sicurezza, pooling, ...
  - risorse condivise tra client – **la scalabilità è migliorata**
- Problemi
  - modello di programmazione più complesso

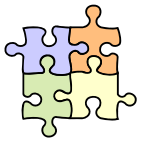




# Alcune considerazioni

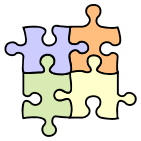
- Che cosa insegna la storia dei sistemi distribuiti (almeno fino a questo punto)?
  - cambiamenti nelle architetture hardware possono avere un impatto sulle architetture software
    - ad es., disponibilità dei PC ⇒ architetture C/S
    - ad es., grandi datacenter, tecnologie per la virtualizzazione e la gestione dei datacenter ⇒ cloud computing
  - nuovi requisiti possono influenzare le architetture hardware e software
    - ad es., applicazioni web ⇒ necessità di servire molti client ⇒ architetture C/S a più livelli ⇒ modello di programmazione con servizi stateless
  - qual è oggi il contesto tecnologico dell'hardware e di business? quali requisiti influenzano le architetture software?





# * Stile client/server

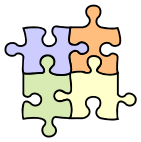
- Contesto
  - ci sono delle risorse oppure dei servizi condivisi
  - un gran numero di client distribuiti vogliono accedere a queste risorse/servizi
  - è necessario controllare l'accesso a queste risorse e servizi, e la qualità del servizio
- Problema
  - si vuole sostenere modificabilità e riuso di un insieme di risorse o servizi condivisi – questo può avvenire gestendo l'accesso a queste risorse e servizi, mettendo a fattor comune i servizi comuni – e rendendo possibile una loro modifica in una singola locazione o comunque in un numero piccolo di locazioni
  - si vogliono migliorare scalabilità e disponibilità di queste risorse e servizi – centralizzandone il controllo, anche se le risorse sono poi distribuite tra più server fisici



# Stile client/server

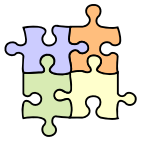
- Soluzione – organizza il sistema come
  - un insieme di *servizi* – ciascun servizio è caratterizzato da un'interfaccia, basata su un protocollo richiesta/risposta
  - un insieme di *server* – i server sono componenti (processi) che forniscono/erogano servizi
    - per un servizio, ci può essere un server centralizzato – oppure più server distribuiti
  - un insieme di *client* – i client sono componenti (processi) fruitori di servizi
    - ci possono essere una molteplicità di client
  - i clienti interagiscono richiedendo servizi ai server
    - i client sono attivi – i server reattivi
    - un server può essere acceduto in modo concorrente da molti client
  - alcuni componenti possono agire sia da client che da server





# Stile client/server

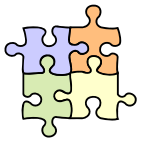
- Dunque, nello stile client-server
  - due tipi di componenti – client e server
  - il principale tipo di connettore è un connettore per l'interazione tra client e server – implementa un protocollo richiesta/risposta, usato per l'invocazione di un servizio
  - questo stile separa le applicazioni client dai servizi che i client devono usare
  - poiché i server possono essere acceduti da qualunque client – è possibile aggiungere nuovi client al sistema
  - inoltre, i server possono essere replicati per fornire scalabilità e disponibilità



# Stile client/server

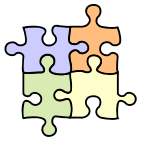
- Esempi di uso
  - molti servizi di Internet, accesso alle basi di dati, ...
- Conseguenze – in prima approssimazione
  - ☺ condivisione di risorse, centralizzazione di elaborazione complessa o sensibile, ...
  - ☹ overhead della comunicazione
  - ☹ il server può essere un collo di bottiglia per prestazioni e scalabilità – può anche essere un punto di fallimento singolo per la disponibilità





# Stile client/server

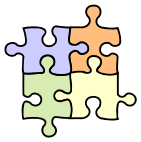
- Nello stile client/server, l'assunzione comune è che gli elementi client e server sono componenti software – ovvero, dei processi logici
  - lo stile client-server viene comunemente adottato nel contesto della *vista logica/funzionale* o della *vista della concorrenza*
- Tuttavia, è spesso utile/necessario descrivere la particolare modalità di applicazione dello stile client/server anche con riferimento alla *vista di deployment*
  - processi diversi possono essere allocati su processori/computer diversi
  - è anche possibile che processi diversi siano allocati sullo stesso processore/computer



# Livelli

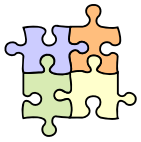
- Esistono diversi tipi di architetture client/server (C/S)
  - le architetture C/S sono normalmente organizzate a “livelli”
  - un **livello** (*tier*) corrisponde a un nodo o gruppo di nodi di calcolo su cui è distribuito il sistema
    - punto di vista del deployment
  - il sistema è organizzato come una sequenza di livelli
    - ciascun livello funge da server per i suoi client – 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 (o gruppi di nodi) fisici





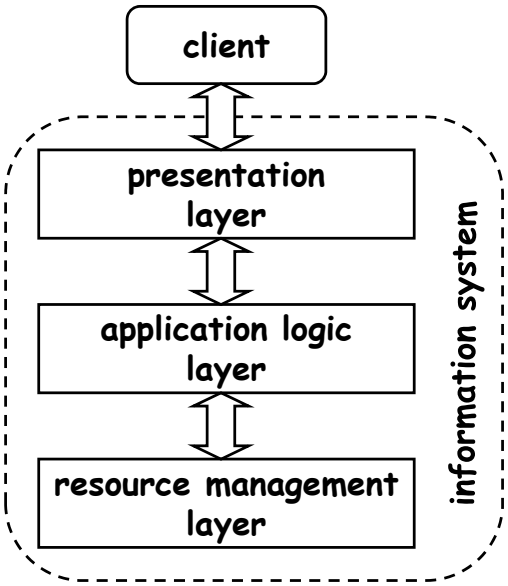
# Livelli e strati

- 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
  - inoltre, il software in ciascun livello è spesso organizzato internamente a strati
- È 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

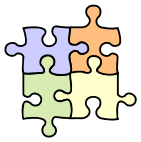


# Architettura a strati di riferimento

- In particolare, nel seguito faremo riferimento ad un sistema che, dal punto di vista funzionale, è organizzato secondo un'architettura a tre strati

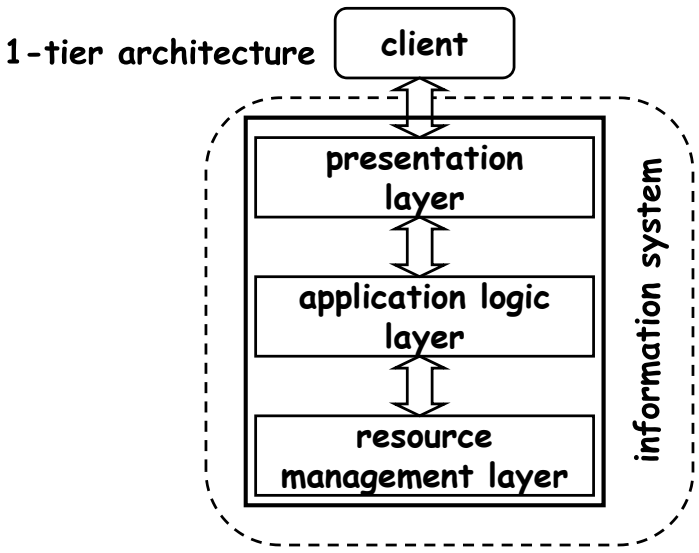


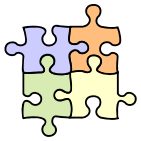


# - Arch. basata su mainframe – anni '70

- *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

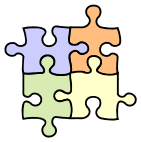




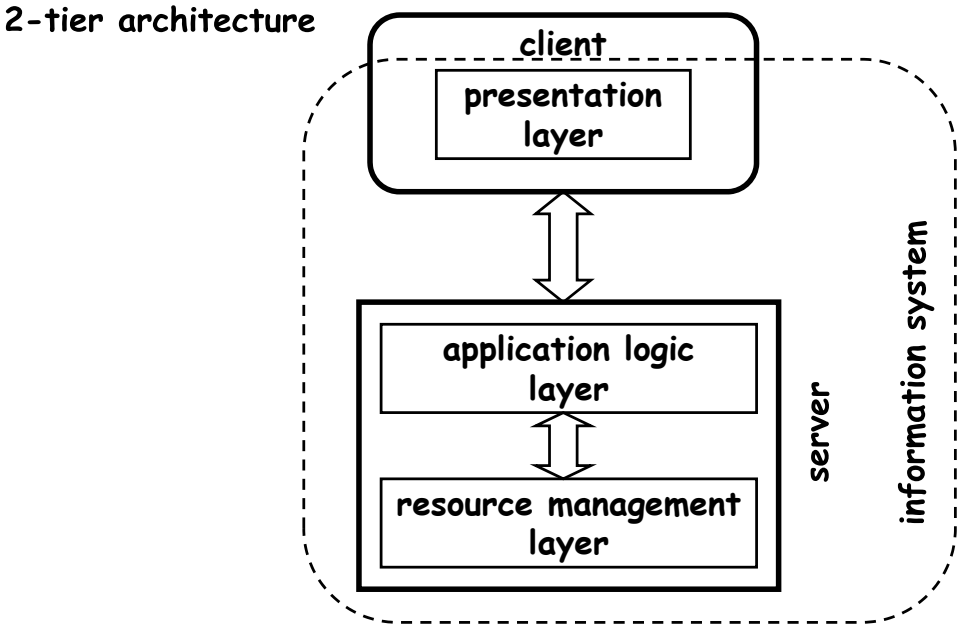
# - Architetture C/S a due livelli – anni '80

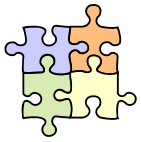
- *Architettura client-server a due livelli* – anni '80
  - le responsabilità sono distribuite su due livelli
    - un livello server
    - un livello client
- Due varianti principali per le architetture C/S a due livelli
  - *modello thin-client*
    - server – responsabile della logica applicativa e gestione dei dati
    - client – responsabile dell'esecuzione del software di presentazione
  - *modello thick-client* (o *fat-client*)
    - server – responsabile della gestione dei dati
    - client – responsabile di presentazione e logica applicativa



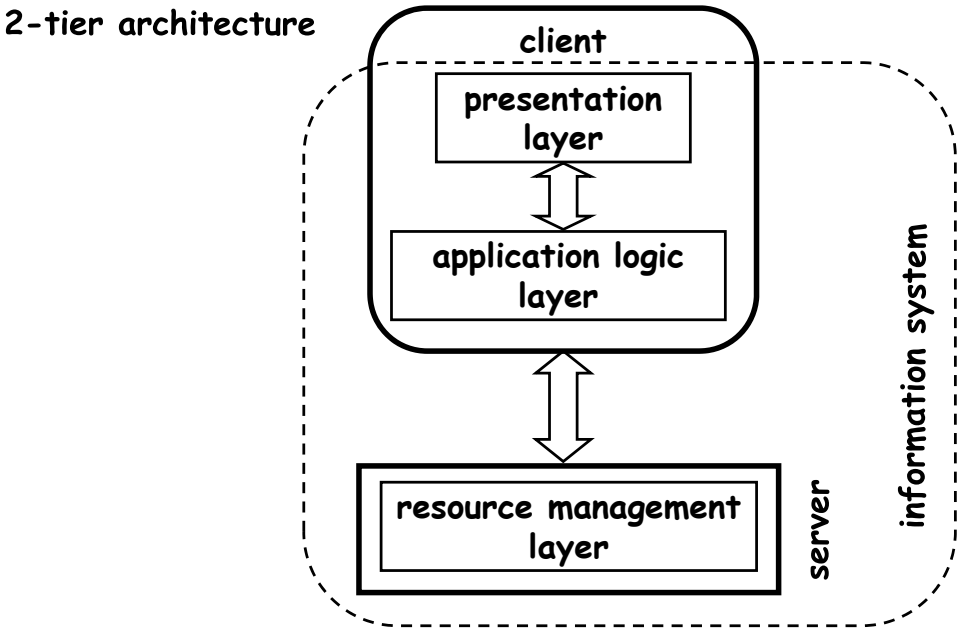


# Architettura C/S a due livelli thin-client

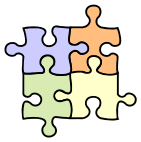




# Architettura C/S a due livelli thick-client

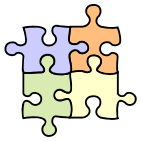




# Architetture C/S a due livelli

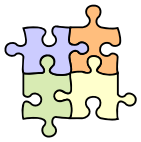
- Conseguenze
  - ☺ il modello thin-client è stata una soluzione semplice per la migrazione da legacy system ad un'architettura client/server
  - ☺ il modello thick-client ha saputo utilizzare l'aumentata potenza di calcolo dei PC degli anni '80
  - ☺ possibili più client – di tipo diverso
- Conseguenze
  - ☹ 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 allocare risorse per gestire la connessione con ciascun client e, spesso, anche lo stato della sessione di ciascun client



# Architetture C/S a due livelli

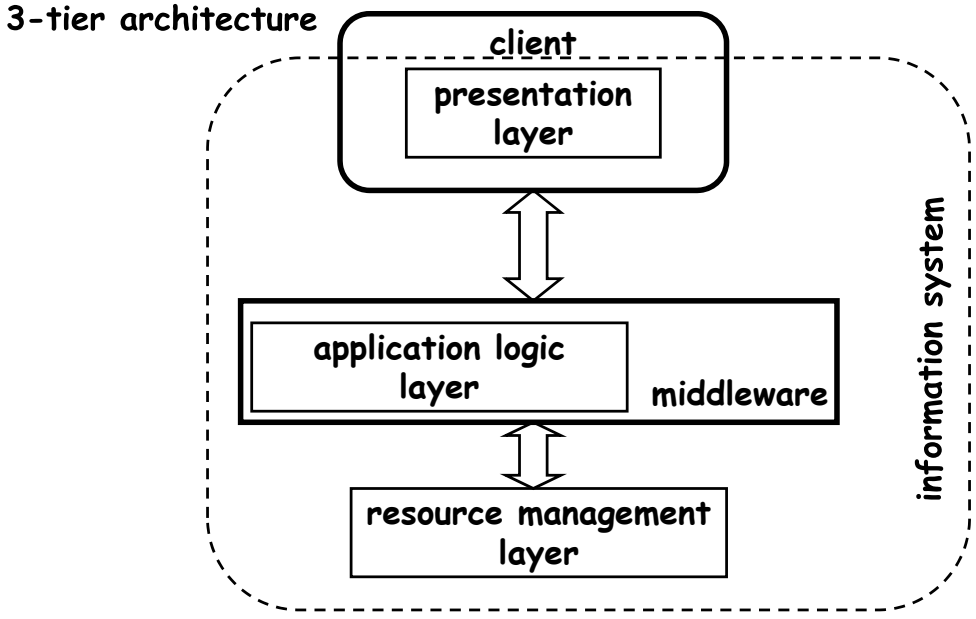
- Discussione
  - il modello client/server a due livelli è alla base di sviluppi fondamentali del software per sistemi distribuiti
    - middleware, meccanismi di RPC, pubblicazione di interfacce, sistemi aperti
  - punto di partenza per i sistemi distribuiti moderni

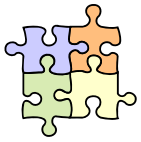




# - Architettura C/S a tre livelli – anni '90

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

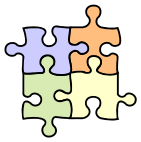




# Architetture C/S a tre livelli

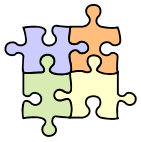
- Conseguenze
  - ☺ consente migliori prestazioni – rispetto al modello thin-client – consente una migliore distribuzione del carico di elaborazione – può compensare il maggior overhead nella comunicazione
  - ☺ supporto per scalabilità e disponibilità – il livello intermedio è costituito da un cluster di calcolatori – scalabilità "orizzontale"
  - ☺ più semplice da gestire – rispetto al modello fat-client
    - in pratica, tutti questi vantaggi sono ottenuti sulla base di nuove soluzioni tecnologiche realizzate dal middleware, che realizza l'infrastruttura per sostenere queste qualità
  - ☹ maggior complessità e maggior overhead nella comunicazione
  - ☹ può essere difficile decidere come allocare le responsabilità ai diversi livelli – inoltre è complesso e costoso cambiare questa decisione dopo che il sistema è stato costruito



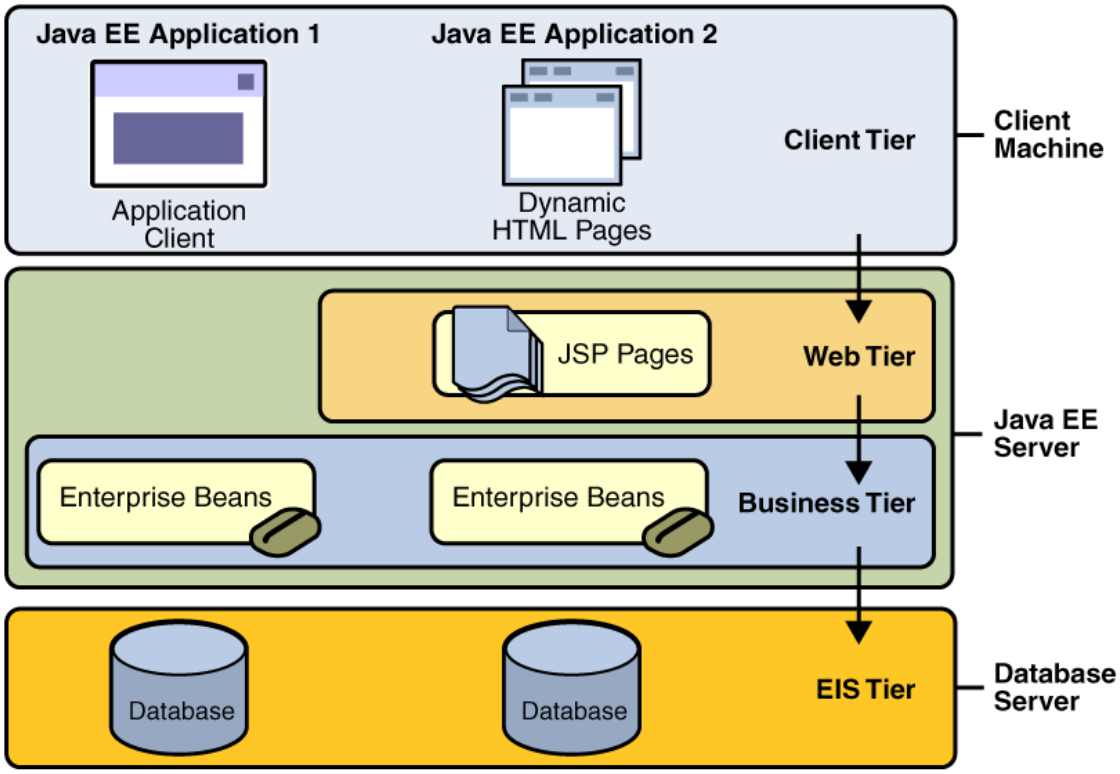


# - Architetture C/S a N livelli

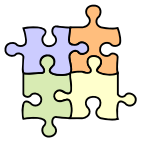
- *Architettura client-server ad N livelli*
  - generalizzazione del modello client/server a tre livelli a un numero qualunque di livelli e server intermedi
- L'architettura a più livelli – *multi-tier* – ha lo scopo di distribuire le capacità di calcolo in sotto-insiemi distinti ed indipendenti
  - di solito questo ha lo scopo di ottimizzare l'uso delle risorse oppure l'ambiente di esecuzione dei server
  - ciascun gruppo di risorse è chiamato un *livello* – *tier*
    - si noti che i livelli non sono componenti software – piuttosto, ciascun livello è un raggruppamento logico di componenti
  - per la scelta dei livelli, sono possibili diversi criteri di raggruppamento
    - ad esempio, avere delle stesse responsabilità a runtime, oppure condividere uno stesso ambiente di esecuzione



# Esempio – piattaforma Java EE

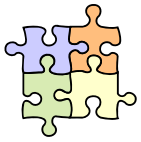




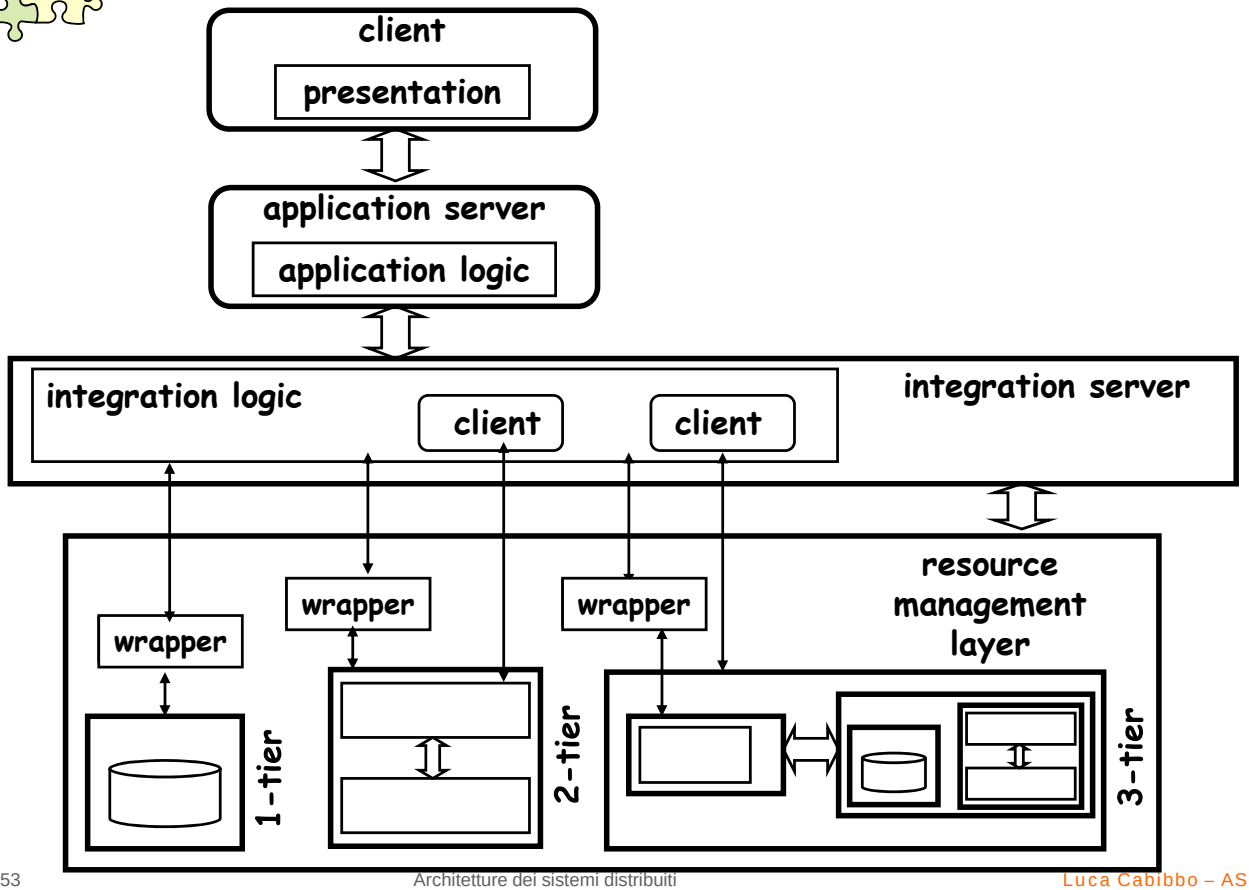



# Esempio – architettura per l'integrazione

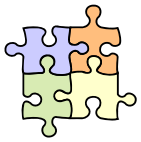
- Si consideri un'applicazione che deve accedere a dati da più basi di dati
  - un server per l'integrazione può essere collocato tra l'application server e i database server
  - il server per l'integrazione
    - accede ai dati distribuiti
    - li integra
    - li presenta all'application server come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



# Un'architettura (ad hoc) per l'integrazione

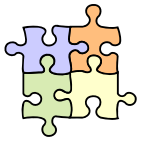




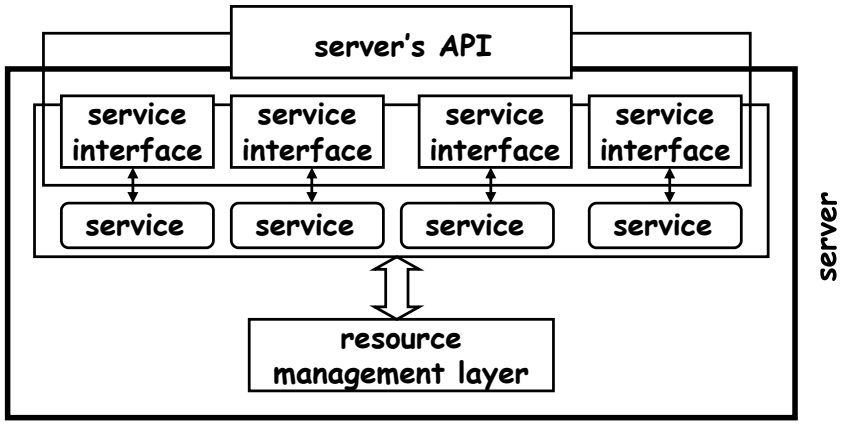
# - Ancora su strati e livelli

- 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



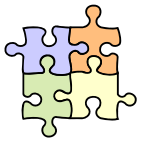
# - Interfacce e protocolli

- Nello stile client/server
  - i servizi offerti da un server vanno “pubblicati” mediante un’*interfaccia*
    - basata anche su un protocollo e sul formato delle richieste/risposte scambiate – che definiscono un connettore



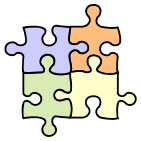

  - sempre importante chiedersi: quale la caratterizzazione del connettore? ma anche: quali le possibilità?





# - Interfacce fornite e interfacce richieste

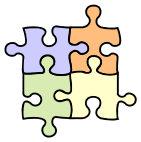
- Nello stile client/server, i client richiedono l’erogazione di servizi offerti dai server
  - i servizi offerti da un server costituiscono l’*interfaccia fornita* del server
  - i servizi richiesti da un client costituiscono l’*interfaccia richiesta* del client
- Inoltre
  - un server può erogare più servizi – ovvero, avere più interfacce fornite
  - un client può richiedere più servizi – ovvero, avere più interfacce richieste
  - in un’architettura a più livelli, alcuni componenti possono avere sia interfacce fornite che interfacce richieste



# - Servizi stateless e stateful

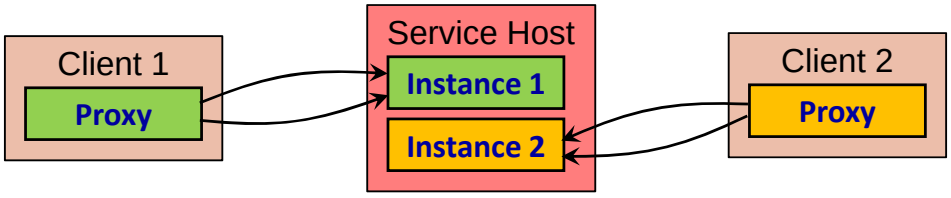
- Un servizio (e il server che lo implementa) può essere stateless oppure stateful
  - la presenza o assenza di stato non si riferisce allo stato complessivo del server o del servizio
  - si riferisce, piuttosto, alla capacità di ricordare lo stato di una specifica conversazione (sessione) tra un client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

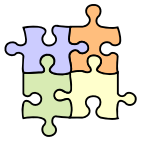




# Servizi stateful

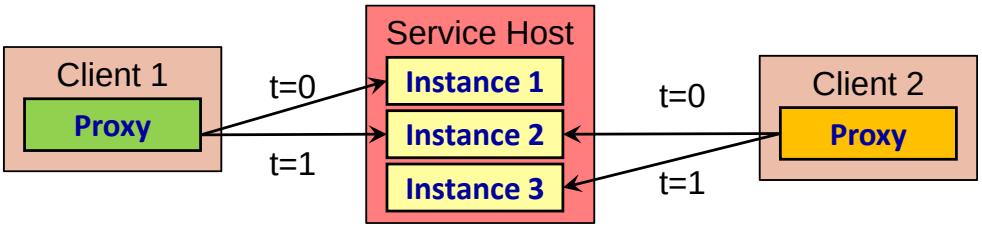
- Un servizio è *stateful* se mantiene (qualche) informazione di stato circa le diverse richieste successive da parte di uno stesso client nell'ambito di una sessione (o conversazione)
  - utile quando la gestione di una richiesta deve poter dipendere dalla storia delle richieste precedenti da parte di quel client
  - il server deve gestire un'istanza del servizio (che occupa risorse) per ciascun client, tutta la durata della sua sessione
    - dal punto di vista del programmatore, la gestione delle informazioni di sessione è più semplice
    - impatto negativo sulla scalabilità



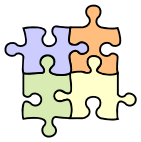


# Servizi stateless

- Un servizio è *stateless* se non mantiene informazioni di stato su ciò che avviene tra richieste successive di uno stesso client
  - adeguato quando la gestione di una richiesta è indipendente dalla storia delle richieste precedenti da parte di quel client
    - ad es., un servizio di previsioni del tempo
  - ogni richiesta può essere gestita indipendentemente dalle altre richieste
    - le risorse del server possono essere condivise tra i diversi client – il server può fare pooling di istanze di serventi
    - impatto positivo sulla scalabilità

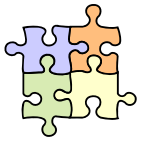




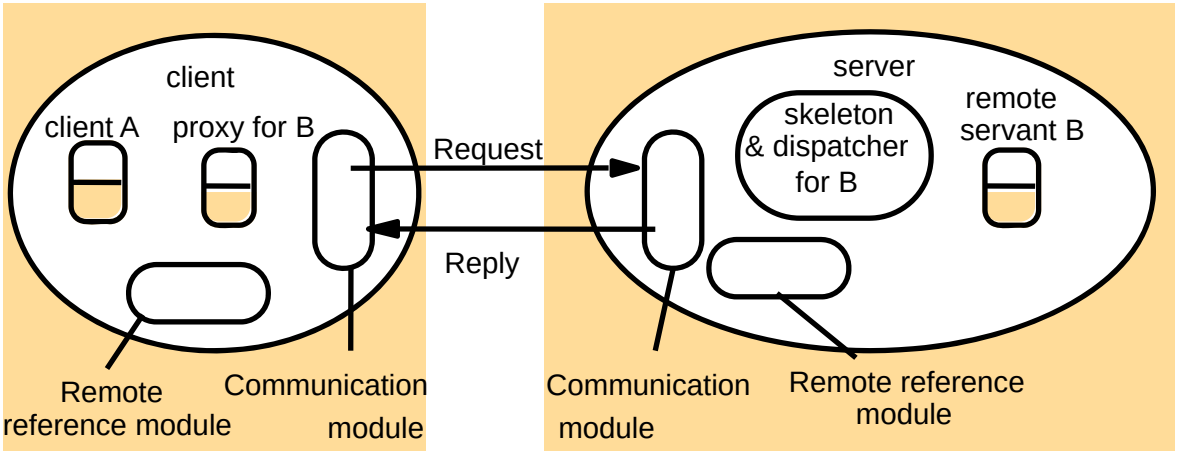
# Implementazione stateless di servizi stateful

- Un servizio che (all'apparenza) è *stateful* può anche essere implementato come un servizio stateless
  - le informazioni di sessione possono essere gestite al di fuori del servizio – ad es., in una base di dati
    - in tal caso, se il servizio non gestisce più direttamente le informazioni di sessione, allora è un servizio stateless
  - è necessario però adottare un protocollo e assegnare delle responsabilità in modo opportuno – ad esempio (ma non è l'unica possibilità)
    - il protocollo può prevedere che il client acquisisca l'id della sua sessione e poi lo ripeta a ogni richiesta
    - in ciascuna richiesta, il servente stateless recupera lo stato della sessione (da dove è gestito) e serve la richiesta
    - i serventi possono essere condivisi tra i diversi client
    - dal punto di vista del programmatore, la gestione delle informazioni di sessione è più complessa

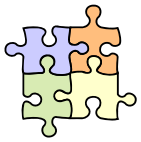


# - Implementazione di un protocollo C/S

- Una possibile implementazione di un protocollo richiesta-risposta in un'architettura client-server
  - Remote Procedure Call (RPC)

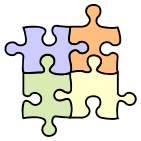




# - Ulteriori considerazioni

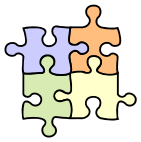
- Nello stile client-server
  - l'invocazione dei servizi è di solito sincrona
    - ovvero, il client effettua una richiesta – e si blocca fino a quando la richiesta non è stata servita
  - l'interazione è iniziata dai client – dunque, è asimmetrica
    - il client deve conoscere l'identità del server – ma il server non deve conoscere l'identità dei suoi client
- Tuttavia, sono possibili delle varianti
  - ad esempio, un browser web non si blocca in attesa dei dati che ha richiesto
  - è possibile che il server possa iniziare delle azioni nei confronti dei suoi client
    - sulla base di meccanismi di registrazione a procedure di notifica o callback



# * Stile peer-to-peer

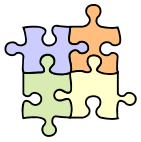
- I sistemi peer-to-peer (p2p) sono sistemi distribuiti e decentralizzati
  - sulla base di numerosi nodi della rete p2p, che condividono le proprie risorse di calcolo – come processori, capacità di memorizzazione, contenuti, …
  - capacità di trattare l'instabilità come la norma
- Esempi
  - comunicazione e collaborazione – ad es., chat – comunicazione diretta e real-time
  - condivisione di contenuti – la categoria più nota
  - calcolo distribuito – ad es., seti@home, grid, cloud – per condividere capacità di calcolo (processori)
  - sistemi per la gestione e la replicazione di dati distribuiti





# Stile peer-to-peer

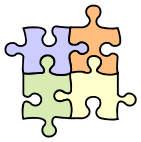
- Contesto
  - ci sono diverse entità distribuite – ciascuna entità fornisce delle proprie risorse computazionali
  - queste entità devono poter cooperare e collaborare per fornire dei servizi a una comunità distribuita di utenti
  - ciascuna di queste entità è considerata ugualmente importante nel poter avviare interazioni con le altre entità
- Problema
  - si vogliono organizzare un insieme di entità computazionali distribuite affinché possano condividere i loro servizi
  - queste entità sono tra di loro "equivalenti" o comunque "pari"
  - si vogliono connettere queste entità sulla base di un protocollo comune
  - si vogliono sostenere scalabilità e disponibilità



# Stile peer-to-peer

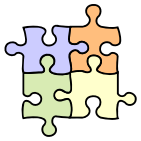
- Soluzione
  - organizza il sistema (o il servizio) sulla base di un insieme di componenti che interagiscono direttamente come “pari” (*peer*)
    - i peer sono tutti ugualmente importanti – nessun peer o gruppo di peer può essere critico per la salute del sistema/servizio
  - la comunicazione è di solito basata su delle interazioni richiesta-risposta – ma senza l’asimmetria dello stile client/server
    - ciascun peer fornisce (offre) e consuma (richiede) servizi simili – utilizzando uno stesso protocollo
    - ogni peer può agire sia come “client” che come “server” del servizio – ogni componente può interagire con ogni altro componente, chiedendogli i suoi servizi – un’interazione può essere avviata da ognuno dei partecipanti
    - talvolta l’interazione consiste solo nell’inoltro di dati – senza il bisogno di una risposta





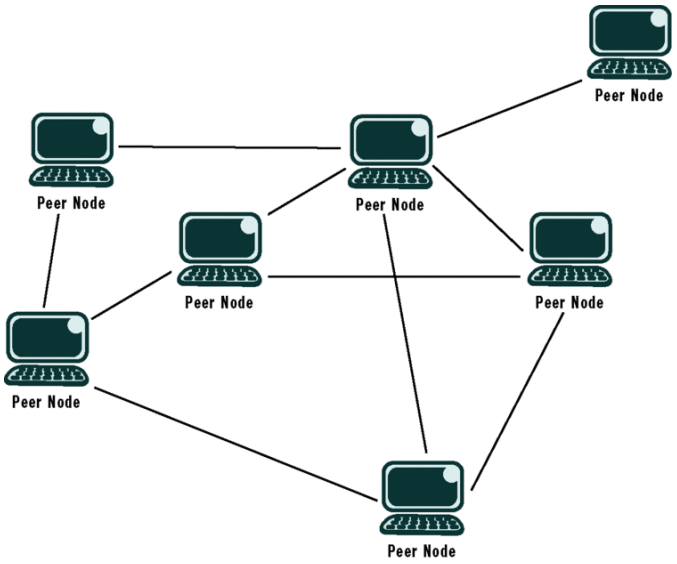
# Stile peer-to-peer

- Lo stile peer-to-peer riflette i meccanismi inerentemente bidirezionali che possono sussistere tra due o più generiche entità (ad es., persone o organizzazioni) che tra loro interagiscono come pari
  - ciascun peer fornisce e consuma servizi simili – e fornisce agli altri peer interfacce per consumare e fornire questi servizi
  - i connettori peer-to-peer implicano dunque delle interazioni bidirezionali
  - i servizi sono relativi alla gestione di risorse che si vogliono condividere
    - ad es., dati o risorse computazionali

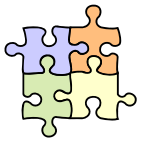


# Esempio

- Una rete peer-to-peer
  - i collegamenti mostrano una relazione di adiacenza “logica” tra nodi – che può essere diversa dall’adiacenza “fisica” – e che può variare dinamicamente

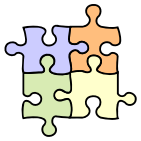




# Scenari

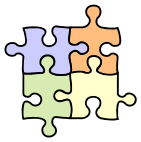
- Avvio
  - un peer si connette alla rete peer-to-peer (p2p) – per scoprire altri peer con cui poter interagire – ma anche per comunicare la propria presenza e disponibilità ad interagire
- Richiesta di servizi
  - un peer può poi avviare delle interazioni per ottenere dei servizi – chiedendo questi servizi ai peer che ha scoperto
    - ad es., la richiesta relativa alla *ricerca* di una risorsa (è un servizio comune nelle reti peer-to-peer) – questa richiesta può essere propagata ad altri peer adiacenti – di solito per un numero limitato di hop
    - la richiesta per il *consumo* di una risorsa viene invece di solito diretta ad uno o più peer specifici



# Scenari

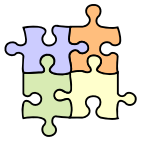
- Super-nodi
  - una rete p2p può avere dei peer specializzati (super-nodi) che forniscono servizi comuni agli altri peer – ad es., la ricerca
- Aggiunta di peer
  - i peer possono essere aggiunti dinamicamente alla rete p2p
  - questo favorisce la scalabilità (orizzontale) del servizio





# Scenari

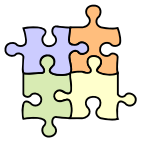
- Rimozione di peer
  - è anche possibile che i peer vengano rimossi dinamicamente dalle rete p2p – ma questo non dovrebbe compromettere la disponibilità del servizio
  - questo è possibile se i diversi peer hanno capacità sovrapponibili – ovvero, se la stessa risorsa (dati o servizio) è fornita da più peer
    - un peer può interagire con più peer per ottenere una certa risorsa – il carico viene distribuito
    - se un peer che sta fornendo la risorsa viene rimosso, il peer richiedente potrà ottenere la risorsa dai peer rimanenti
    - non è necessaria la centralizzazione di risorse



# Discussione

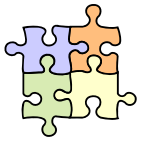
- I punti di debolezza dello stile peer-to-peer sono fortemente correlati alle sue forze
  - poiché i sistemi peer-to-peer sono fortemente decentralizzati, è più complesso (o impossibile) gestire la sicurezza, la consistenza dei dati, gestire e controllare la disponibilità dei dati e dei servizi, effettuare backup e recovery
  - in molti casi è difficile fornire garanzie di qualità – soprattutto se i peer possono venire e andare a piacere
- Tuttavia, lo stile peer-to-peer può essere usato per fornire dei buoni livelli di qualità del servizi *in ambienti opportunamente controllati*
  - ad esempio, è il caso di molte elaborazioni distribuite sui nodi di un data center in ambienti grid o cloud





# Esempio: basi di dati distribuite

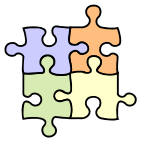
- Alcuni sistemi per la gestione di basi di dati distribuite (ad es., i database NoSQL) usano una combinazione dei seguenti meccanismi per sostenere scalabilità e disponibilità
  - replicazione dei dati su più nodi di un cluster – insieme a una politica per garantire la consistenza delle diverse copie dei dati
    - replicazione master-slave (non è p2p) – la replicazione avviene sulla base di un'organizzazione gerarchica dei nodi
    - replicazione peer-to-peer – una soluzione spesso più efficace per la propagazione degli aggiornamenti – fornisce inoltre una miglior tolleranza ai guasti, e scalabilità orizzontale (possibilità di aggiungere dinamicamente nodi al cluster)
  - distribuzione (sharding) dei dati sui nodi del cluster
    - l'uso di tecniche di hashing distribuito (DHT) consente di combinare distribuzione e replicazione dei dati



# * Architetture a oggetti distribuiti

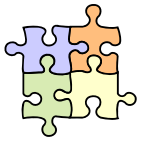
- Nelle architetture client/server, i client e i server sono organizzati secondo una struttura gerarchica
  - è possibile pensare a soluzioni più flessibili – meno vincolate – in cui i diversi componenti software possono interagire anche come “pari”
- Le architetture a oggetti distribuiti adottano un approccio più generale – in cui
  - sono mantenute le importanti nozioni di “servizio” e “interfaccia” – i servizi vengono ancora consumati sulla base di un protocollo richiesta/risposta
  - viene adottato un paradigma a oggetti
  - viene rimossa la distinzione statica tra client e server – ma, in ciascuna singola interazione, si continua a distinguere tra “client” e “server” (nell’ambito della specifica interazione)





# Richiamo: Paradigma a oggetti

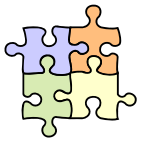
- Nel paradigma di programmazione a **oggetti** (non distribuiti!)
  - ciascun *oggetto* incapsula stato e comportamento
    - il comportamento di un oggetto è descritto dalla sua *interfaccia* (definita implicitamente o esplicitamente)
      - è la specifica dei metodi che possono essere invocati pubblicamente
    - 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

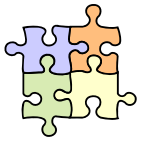
- Nel paradigma di programmazione a **oggetti distribuiti**
  - due tipi di oggetti
    - *oggetti locali* – sono visibili localmente a un processo
    - *oggetti remoti* – possono essere distribuiti in più computer/processi
  - ciascun *oggetto* incapsula stato e comportamento
  - gli oggetti remoti 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)



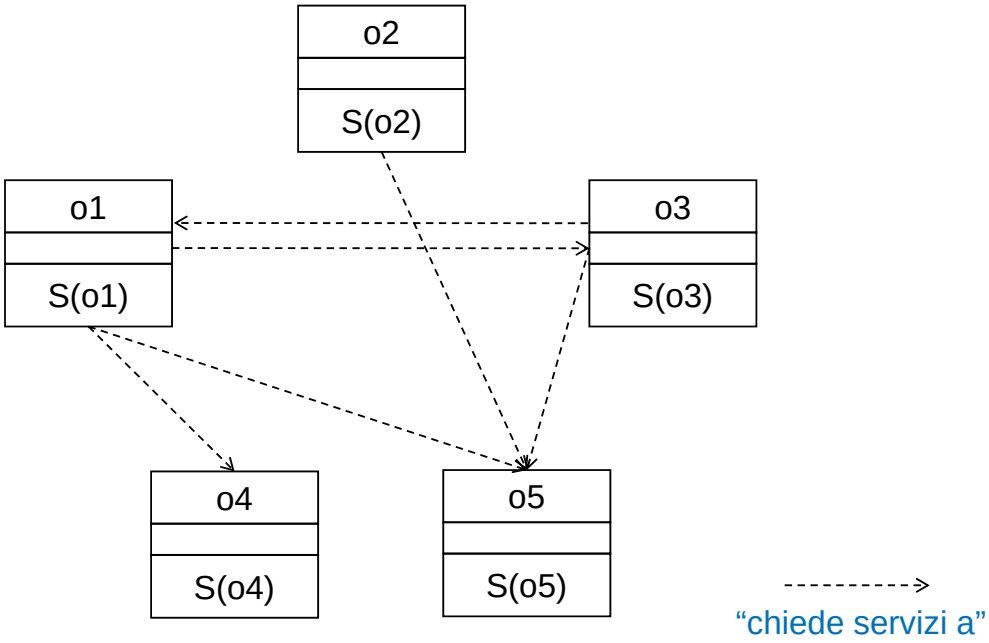


# Architetture a oggetti distribuiti

- Dunque, in un'**architettura a oggetti distribuiti** (*DOA*)
  - gli elementi del sistema sono chiamati *oggetti remoti* (o *distribuiti*)
    - attenzione, si tratta di solito di "macro-oggetti" – nel senso che, comunemente, un oggetto remoto definisce una facade verso un gruppo di oggetti "tradizionali" (che gli sono locali)
    - comunque realizzati con tecnologie a 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 computer, in modo flessibile

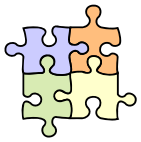


# Un'architettura a oggetti distribuiti



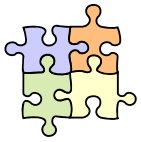

- Nota – si tratta di una vista "logica", "funzionale"



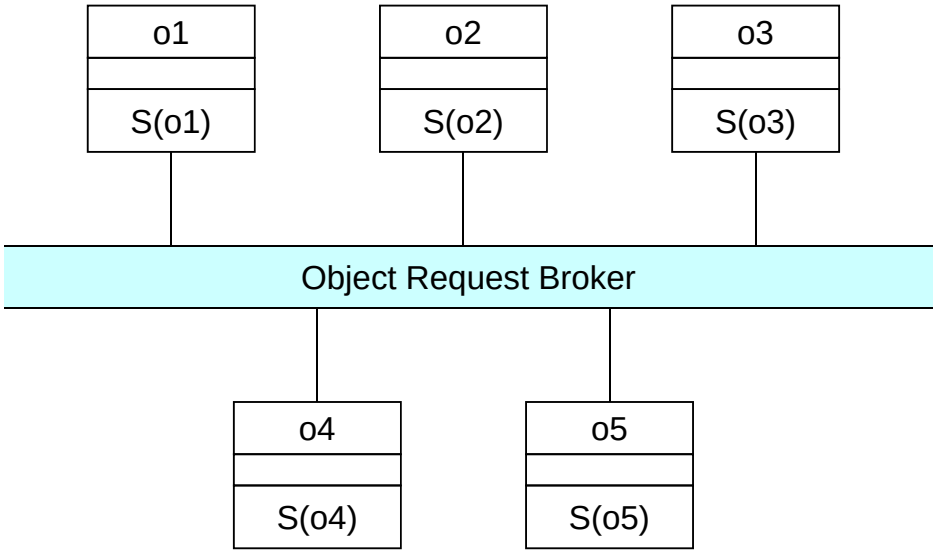


# Comunicazione tra oggetti distribuiti

- In una DOA, la comunicazione tra oggetti distribuiti – basata su un paradigma di comunicazione di tipo RMI – avviene mediante un'infrastruttura di comunicazione opportuna – ovvero, 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)

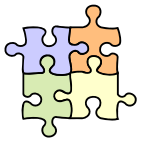


# Comunicazione mediante broker



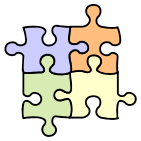

- Questa è, invece, una vista di deployment
  - mostra l'infrastruttura di comunicazione
  - potrebbe mostrare anche i nodi di calcolo





# Interazione tra oggetti distribuiti

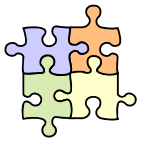
- Modalità di interazione tra oggetti distribuiti
  - gli oggetti server (ovvero, gli oggetti che offrono servizi) possono registrare i servizi che offrono presso il broker
    - più precisamente, presso il servizio di directory gestito dal broker
  - gli oggetti client possono consultare il broker per ottenere un riferimento remoto a 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

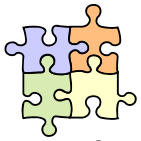
- Conseguenze
  - ☺ architettura aperta, flessibile e scalabile
  - ☺ possibile riconfigurare il sistema dinamicamente, migrando gli oggetti tra computer – questo consente di rimandare decisioni su dove fornire i servizi, oppure di cambiare decisioni per sostenere, ad esempio, scalabilità
  - ☺ consente 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

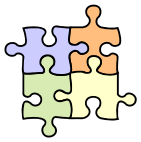
- Conseguenze
  - ☹ le prestazioni peggiorano se sono utilizzati molti oggetti a grana piccola o con servizi a grana piccola
  - 😐 le prestazioni dipendono dalla topologia e dalla grana degli oggetti e della loro interfaccia
    - bene con oggetti a grana grossa, che comunicano poco
  - 😐 la sicurezza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



# Usi delle DOA

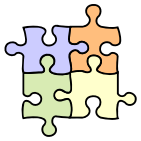
- È possibile identificare due modalità principali per l'uso delle DOA
  - la DOA può essere usata come un "modello logico" per strutturare e organizzare il sistema
    - gli elementi dell'architettura sono macro-oggetti che offrono servizi e incapsulano lo stato
    - il modello a oggetti viene usato per ragionare ai vari livelli di decomposizione del sistema
  - le tecnologie DOA possono essere usate come base (flessibile) per l'implementazione di sistemi client/server
    - ovvero, si adotta un'architettura "logica" per il sistema di tipo client/server – ma "tecnologicamente" 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
  - descritto nel seguito
- Per le tecnologie a oggetti distribuiti, vedi anche
  - dispensa su Oggetti distribuiti e invocazione remota



# * Discussione

- Gli stili architetturali descritti in questa dispensa – client/server, peer-to-peer e a oggetti distribuiti – sono alla base di molti sistemi distribuiti attuali
  - le tecnologie sottostanti, e i relativi pattern di utilizzo, si sono successivamente evoluti per semplificare ulteriormente lo sviluppo dei sistemi distribuiti e favorire la loro interoperabilità