



Luca Cabibbo  
Architettura  
dei Sistemi  
Software

# Introduzione ai connettori

**dispensa asw810**  
ottobre 2023

*Relationships among the elements  
are what give systems their added value.*  
*Eberhardt Rechtin*



## - Riferimenti

- Luca Cabibbo. **Architettura del Software: Strutture e Qualità**. Edizioni Efestò, 2021.
  - Capitolo 21, **Introduzione ai sistemi distribuiti**
- Shaw, M. **Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status**. Technical report CMU/SEI-1994-TR-2, 1996.
- Bernstein, P. **Middleware**. Communications of the ACM, 1996.



## - Obiettivi e argomenti

### □ Obiettivi

- introdurre i connettori
- presentare alcuni semplici esempi di connettori
- introdurre brevemente il middleware
- motivare le successive dispense di questa parte del corso

### □ Argomenti

- introduzione
- introduzione ai connettori
- componenti e connettori: esempi
- introduzione al middleware
- discussione



## \* Introduzione

- Questa parte del corso (dispense 8xx) è formata da un insieme di “esercitazioni” che esemplificano e discutono alcuni aspetti “tecnologici” dell’architettura del software
  - connettori e middleware
  - il framework Spring, Spring Boot e Spring Cloud
  - architettura esagonale
  - invocazione remota (gRPC e REST)
  - comunicazione asincrona (Kafka)
  - componenti (Spring)
  - servizi REST
  - container (Docker)
  - orchestrazione di container (Kubernetes)
- in questa edizione del corso, le esercitazioni enfatizzano i microservizi con Spring Boot, Docker e Kubernetes



## \* Introduzione ai connettori

- Nell'architettura di un sistema software è possibile distinguere due tipi principali di elementi software
  - *componenti*
    - elementi responsabili dell'implementazione di *funzionalità* e della gestione di *dati* e *informazioni*
  - *connettori*
    - elementi responsabili delle *interazioni* tra componenti – i connettori caratterizzano assemblaggio e integrazione di componenti
- Sia i componenti che i connettori sono “elementi software”
  - dunque, entrambi questi tipi di elementi sono basati su del “codice”

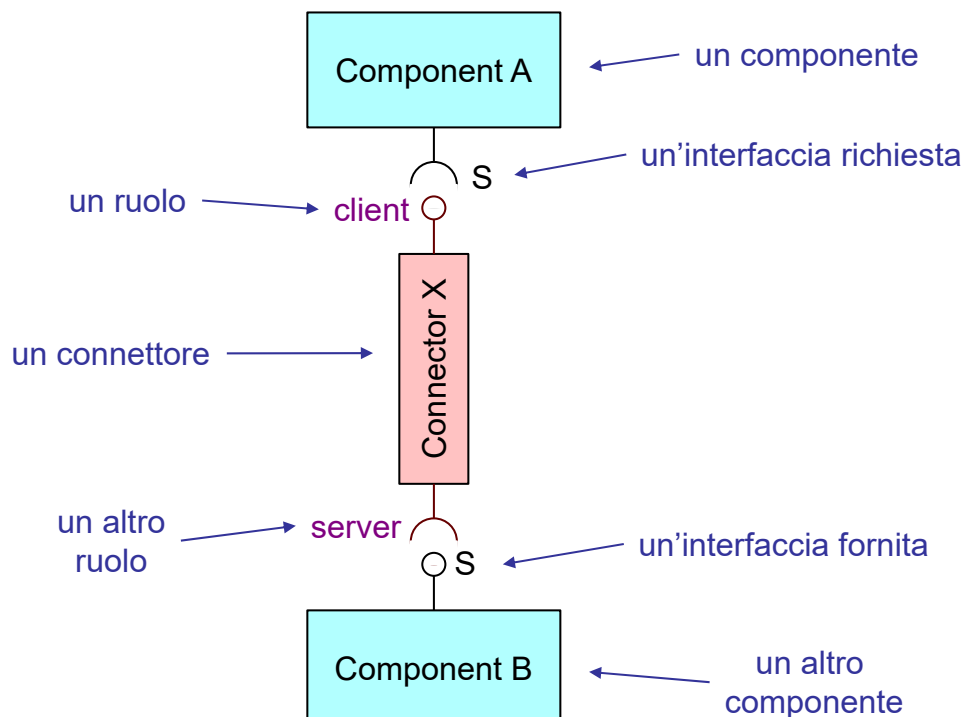
5

Introduzione ai connettori

Luca Cabibbo ASW



## Componenti e connettori



6

Introduzione ai connettori

Luca Cabibbo ASW



# Perché i connettori?

- Ecco alcune motivazioni per la distinzione tra componenti e connettori – dimostrate dall’esperienza pratica
  - i componenti si occupano di aspetti funzionali – i connettori delle interazioni
  - la scelta e progettazione dei connettori (ovvero, delle interazioni) è importante tanto quanto quella dei componenti
    - i connettori possono essere responsabili di qualità importanti di un sistema
  - la progettazione dei connettori può effettivamente essere fatta separatamente da quella dei componenti
  - i connettori sono tipicamente indipendenti dalle applicazioni
    - per questo, la generalizzazione di connettori di uso comune ha portato allo sviluppo dei servizi di middleware – tecnologie software per l’implementazione di connettori, utili soprattutto nello sviluppo di sistemi distribuiti



## \* Componenti e connettori: esempi

- In pratica, un sistema software comprende diversi componenti “funzionali” e diversi connettori
  - ogni componente può avere delle dipendenze (ovvero, dipendere) da altri componenti
    - intuitivamente, ogni componente dipende (mediante delle interfacce) dai componenti a cui vuole richiedere dei servizi
  - queste dipendenze vanno soddisfatte
    - ogni componente deve poter effettivamente richiedere servizi ai componenti da cui dipende, interagendo con essi
  - i connettori hanno a che fare con la gestione di queste interazioni tra componenti – e spesso anche con la gestione delle dipendenze tra componenti
  - questo verrà mostrato nei seguenti esempi



## - Studio di caso di riferimento

- Si supponga di aver identificato un'interfaccia e due componenti
  - un'interfaccia che definisce un servizio **Service**
  - un componente **ServiceImpl** (o **Servant** o **Server**) in grado di fornire il servizio **Service**
  - un componente **Client**, che richiede l'erogazione del servizio **Service** – a **ServiceImpl** o a chiunque sappia erogarlo



- si dice che l'interfaccia **Service** è
  - un'interfaccia fornita da **ServiceImpl**
  - un'interfaccia richiesta da **Client**
- A che cosa corrisponde/può corrispondere ciò nel codice?

9

Introduzione ai connettori

Luca Cabibbo ASW



## - Servizio e implementazione del servizio (tutte le versioni)

- Supponiamo, per semplicità, di avere la seguente definizione di **Service**
  - questa interfaccia è limitata ai soli aspetti “funzionali” del servizio

```
package asw.intro.service;  
  
/* Interfaccia del servizio Service. */  
public interface Service {  
    /* Fa qualcosa con arg. */  
    public String alpha(String arg);  
}
```

in **violetto**  
indichiamo il  
codice relativo ad  
aspetti funzionali

10

Introduzione ai connettori

Luca Cabibbo ASW



## Servizio e implementazione del servizio (tutte le versioni)

- Inoltre, supponiamo di avere una qualche implementazione **ServiceImpl** di **Service**

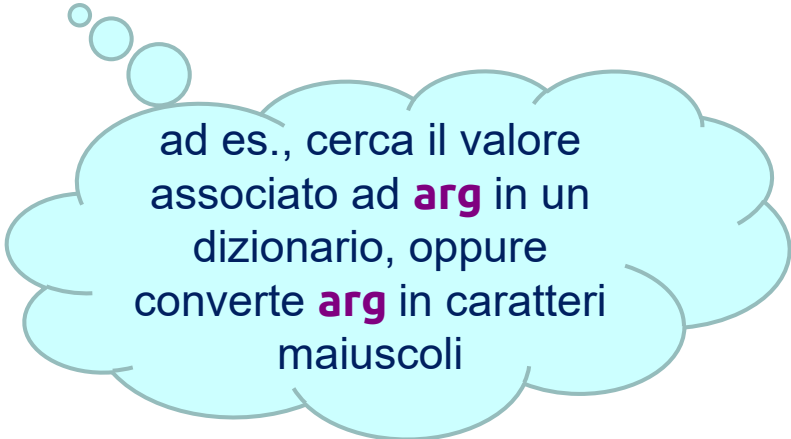
```
package asw.intro.server;

import asw.intro.service.Service;

/* Implementazione del servizio Service. */
public class ServiceImpl implements Service {

    public String alpha(String arg) {
        ... fa qualcosa con arg ...
    }

}
```



ad es., cerca il valore associato ad **arg** in un dizionario, oppure converte **arg** in caratteri maiuscoli



## - Una prima versione (versione a)

- Inoltre, un'applicazione **Main** che crea e avvia un **Client**

```
package asw.intro.main;

import asw.intro.client.Client;

/* Applicazione che crea e avvia il client. */
public class Main {

    /* Crea e avvia un oggetto Client. */
    public static void main(String[] args) {
        Client client = new Client();
        client.run();
    }

}
```



## Una prima versione

- Infine, un **Client** – in cui viene utilizzato il **Service**
  - ad es., come segue

```
package asw.intro.client;

import asw.intro.service.Service;

/* Client del servizio Service. */
public class Client {

    private Service service;

    public Client() {

    }

    public void run(...) {
        ... service.alpha(...) ...
    }

}
```

in **ROSSO**  
indichiamo  
l'invocazione del  
servizio – è  
ancora un aspetto  
funzionale



## Una prima versione

- Infine, un **Client** – in cui viene utilizzato il **Service**
  - ad es., come segue

```
package asw.intro.client;

import asw.intro.service.Service;

/* Client del servizio Service. */
public class Client {

    private Service service;

    public Client() {

    }

    public void run(...) {
        ... service.alpha(...) ...
    }

}
```

questa variabile  
d'istanza  
rappresenta una  
**dipendenza**, che  
deve essere  
soddisfatta prima  
di poter eseguire  
**run()**



## Una prima versione

- Il connettore più semplice è la *chiamata di procedura* – o *invocazione di metodo* nei linguaggi OO

```
package asw.intro.client;

import asw.intro.service.Service;
import asw.intro.server.ServiceImpl;

/* Client del servizio Service. */
public class Client {

    private Service service;

    public Client() {
        this.service = new ServiceImpl();
    }

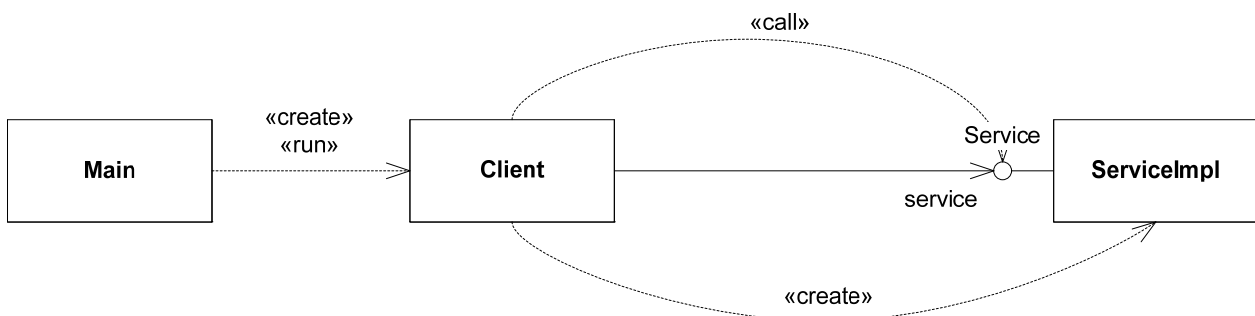
    public void run(...) {
        ... service.alpha(...) ...
    }

}
```

in **nero**  
indichiamo il  
codice relativo  
al connettore –  
ovvero,  
l'interazione tra  
componenti



## Una prima versione







## Una prima versione

- Caratteristiche di questa prima soluzione
  - una *chiamata di procedura locale*
    - **Client** e **ServiceImpl** vivono nello stesso processo – tuttavia, le scelte funzionali dovrebbero essere indipendenti da scelte relative, ad es., a concorrenza e deployment
  - una chiamata sincrona
    - durante l'esecuzione del **Service**, il **Client** rimane in attesa di **ServiceImpl** – non c'è nessuna concorrenza
  - il **Client** è accoppiato alla particolare implementazione del **Service** offerta da **ServiceImpl**
    - non c'è indipendenza dall'implementazione del servizio
  - il **Client** è responsabile di soddisfare le proprie dipendenze
  - **Client** e **ServiceImpl** devono essere scritti nello stesso linguaggio di programmazione
    - anche questo potrebbe essere un vincolo indesiderato



## - Uso di una factory (versione b)

- Concentriamoci, per ora, sull'eliminazione della dipendenza tra il **Client** e la particolare implementazione **ServiceImpl** del **Service**
  - si può rompere questa dipendenza utilizzando degli oggetti di supporto – in particolare, applicando degli opportuni design pattern
    - ad es., usando una factory – ovvero, un singleton che si occupa della creazione di un'istanza del **Service**



## Uso di una factory

- La **ServiceFactory** incapsula la creazione di un **Service**

```
package asw.intro.connector;

import asw.intro.service.Service;
import asw.intro.server.ServiceImpl;

/* Factory per il servizio Service. */
public class ServiceFactory {

    private static ServiceFactory instance = null;    // la factory
    private Service service = null;    // il servizio

    private ServiceFactory() { }    // costruttore privato

    public static synchronized ServiceFactory getInstance() {
        if (instance==null) { instance = new ServiceFactory(); }
        return instance;
    }

    /* Factory method per il servizio Service. */
    public synchronized Service getService() {
        if (service==null) { service = new ServiceImpl(); }
        return service;
    }
}
}
```

19

Introduzione ai connettori

Luca Cabibbo ASW



## Uso di una factory

- Ecco la nuova versione per il client – il connettore sarà ancora una chiamata di procedura

```
package asw.intro.client;

import asw.intro.service.Service;
import asw.intro.connector.*;

/* Client del servizio Service. */
public class Client {

    private Service service;

    public Client() {
        this.service = ServiceFactory.getInstance().getService();
    }

    public void run(...) {
        ... service.alpha(...) ...
    }

}
}
```

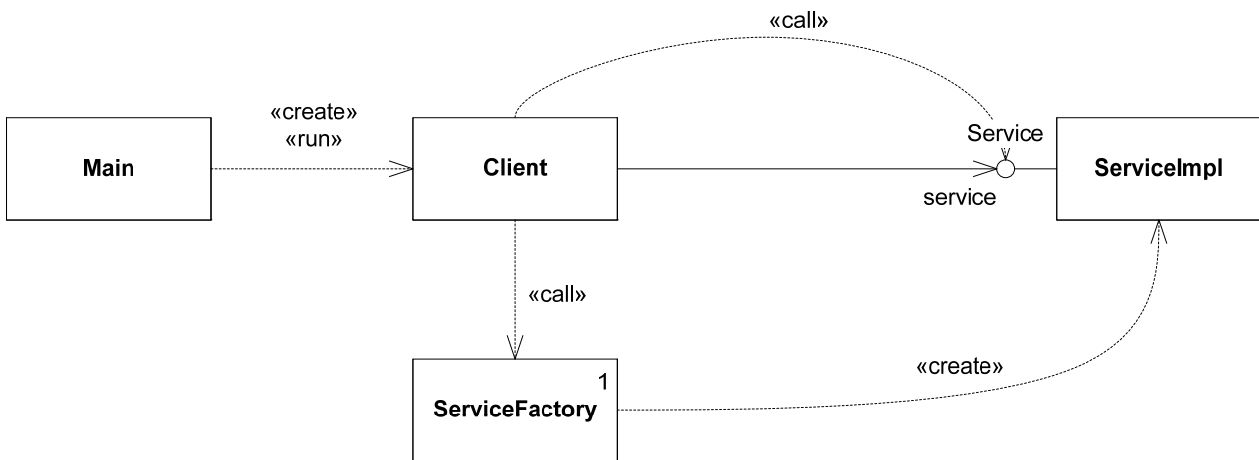
20

Introduzione ai connettori

Luca Cabibbo ASW



## Uso di una factory



21

Introduzione ai connettori

Luca Cabibbo ASW



## Uso di una factory

- Alcune caratteristiche della nuova soluzione
  - in pratica, si tratta ancora di una chiamata di procedura locale e di una chiamata sincrona
  - tuttavia, non c'è più l'accoppiamento diretto tra il **Client** e la particolare implementazione **ServiceImpl** del **Service**
    - l'accoppiamento è ora localizzato nella factory – che possiamo considerare parte del connettore
    - l'accoppiamento può essere ulteriormente ridotto usando per la factory un progetto data-driven
      - ad es., usando un file di configurazione con il nome della classe che implementa il servizio che si vuole utilizzare
  - il **Client** è ancora responsabile di soddisfare le proprie dipendenze – e nel farlo dipende dalla factory
  - c'è ancora il vincolo che **Client** e **ServiceImpl** devono essere scritti nello stesso linguaggio di programmazione

22

Introduzione ai connettori

Luca Cabibbo ASW



## - Iniezione delle dipendenze (versione c)

- Ci sono anche altri modi per eliminare la dipendenza tra il **Client** e la particolare implementazione **ServiceImpl** del **Service**
  - in particolare, l'*iniezione delle dipendenze* è un design pattern che prevede quanto segue
    - ciascun oggetto definisce le sue dipendenze (altri oggetti da cui dipende)
    - le dipendenze di un oggetto non vengono risolte dall'oggetto stesso, ma piuttosto vengono risolte da altri oggetti di supporto, mediante un'iniezione delle dipendenze (di solito al momento della creazione dell'oggetto)
    - l'iniezione delle dipendenze può essere scritta nel codice, oppure basata su file di configurazione o su annotazioni
  - chiamato anche *inversione del controllo* – poiché un oggetto non deve creare o cercare da solo gli oggetti da cui dipende
  - vediamo una possibilità

23

Introduzione ai connettori

Luca Cabibbo ASW



## Iniezione delle dipendenze

- Ecco la nuova versione per il client – che ora dipende solo dal servizio, e non più dal connettore

```
package asw.intro.client;

import asw.intro.service.Service;

/* Client del servizio Service. */
public class Client {

    private Service service;

    public Client() {}

    public void setService(Service service) {
        this.service = service;
    }

    public void run(...) {
        ... service.alpha(...) ...
    }

}
```

l'iniezione della dipendenza avverrà tramite questo metodo setter

24

Introduzione ai connettori

Luca Cabibbo ASW



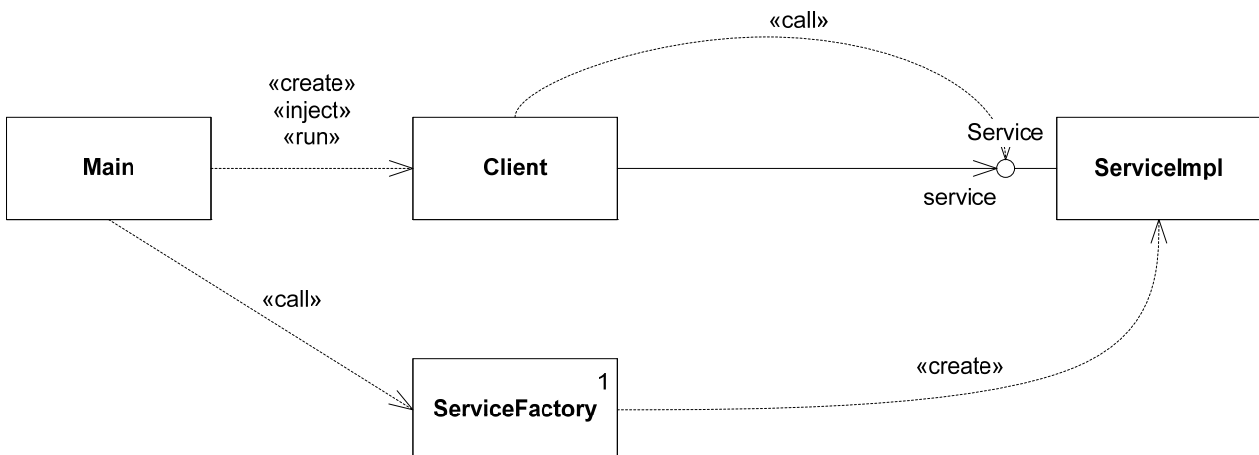
# Iniezione delle dipendenze

- L'iniezione della dipendenza viene effettuata, in questo caso, dall'applicazione **Main**

```
package asw.intro.main;  
  
import asw.intro.client.Client;  
import asw.intro.service.Service;  
import asw.intro.connector.ServiceFactory;  
  
/* Applicazione che crea e avvia il client. */  
public class Main {  
  
    /* Crea e avvia un oggetto Client. */  
    public static void main(String[] args) {  
        Service service = ServiceFactory.getInstance().getService();  
        Client client = new Client();  
        /* iniezione della dipendenza service */  
        client.setService(service);  
        client.run();  
    }  
}
```



# Iniezione delle dipendenze





# Iniezione delle dipendenze

- Alcune caratteristiche della nuova soluzione
  - rispetto a quanto fatto in precedenza, ora il **Client** non dipende più dalla factory
    - il **Client** non è più responsabile di soddisfare le proprie dipendenze
    - in pratica, nel codice del “componente” **Client** non c'è più nessuna traccia di codice “connettore”
  - ciò che è “connettore” è ora completamente separato da ciò che è “componente”!
- Varianti
  - nell'esempio mostrato, la dipendenza **service** viene iniettata tramite il metodo **setService()**
    - è anche possibile effettuare l'iniezione delle dipendenze mediante il costruttore – in questo caso, mediante un costruttore **Client(Service service)**

27

Introduzione ai connettori

Luca Cabibbo ASW



## - Uso di un application context (versione d)

- Può essere utile avere un oggetto che si occupa sia della creazione di tutti gli oggetti – nel nostro esempio, non solo del servizio, ma anche del client – che delle iniezioni di tutte le dipendenze necessarie
  - un tale oggetto è di solito chiamato un *application context*
  - un application context ha le seguenti responsabilità
    - fornire l'accesso a un certo insieme di oggetti
    - creare questi oggetti – se e quando è necessario
    - risolvere le dipendenze tra di essi – mediante iniezione delle dipendenze
    - gli oggetti vengono forniti con le loro dipendenze soddisfatte

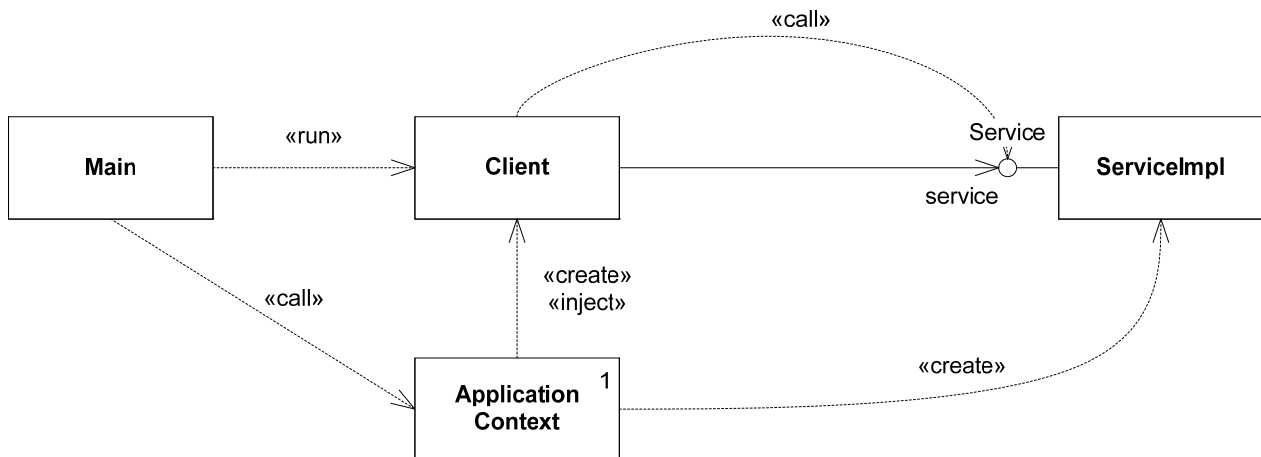
28

Introduzione ai connettori

Luca Cabibbo ASW



## Uso di un application context



29

Introduzione ai connettori

Luca Cabibbo ASW



## Uso di un application context

- L'application context è simile alla service factory
  - ma ora ha anche un metodo per creare il client

```
package asw.intro.context;
import asw.intro. ...
/* Application context. */
public class ApplicationContext {
    ... variabile, costruttore e metodo per singleton, come prima ...
    ... variabile per service e metodo getService, come prima ...
    /* Factory method per il client Client.
    * Ogni volta viene restituito un nuovo client. */
    public Client getClient() {
        Client client = new Client();
        client.setService( this.getService() );
        return client;
    }
}
```

30

Introduzione ai connettori

Luca Cabibbo ASW



## Uso di un application context

- La classe **Main** è semplificata

```
package asw.intro.main;

import asw.client.Client;
import asw.intro.context.ApplicationContext;

/* Applicazione che ottiene e avvia il client. */
public class Main {

    /* Crea e avvia un oggetto Client. */
    public static void main(String[] args) {
        Client client = ApplicationContext.getInstance().getClient();
        client.run();
    }
}
```



## Uso di un application context

- La creazione di oggetti e l'iniezione delle dipendenze sono compiti comuni nelle applicazioni complesse
  - questi compiti possono essere svolti da un opportuno framework – anziché dover scrivere ogni volta del codice apposito
  - ad es., è possibile usare **Spring Framework** (<https://spring.io/>)
  - in pratica, questo framework consente di
    - utilizzare un “application context” predefinito
    - specificare la configurazione dei componenti (“bean”) di interesse, insieme alle loro dipendenze (ad es., mediante un file di configurazione)
    - creare un bean (con le dipendenze già soddisfatte) a partire dal suo nome
  - il framework Spring verrà introdotto in una successiva dispensa





## - Uso di proxy (versione e)

- Discutiamo ora di un altro aspetto importante
  - nelle soluzioni precedenti, la factory (o l'application context) crea un'istanza del servizio e ne restituisce un riferimento che viene memorizzato direttamente dal client
  - è invece spesso utile (e pertanto comune) utilizzare anche un ulteriore intermediario, un *proxy* – un rappresentante del servizio presso il client
    - il proxy, nella versione che viene ora presentata, per semplicità
      - memorizza un riferimento al vero servizio
      - gira tutte le chiamate che riceve al vero servizio
    - la factory (o l'application context) restituisce (o inietta) non il vero servizio, ma piuttosto un suo proxy
    - il client memorizza un riferimento al proxy – pensando che sia un riferimento al servizio



## Uso di proxy

- Il **ServiceProxy** è un'indirezione verso il vero servizio, che implementa la stessa interfaccia del servizio

```
package asw.intro.connector;
import asw.intro.service.Service;
public class ServiceProxy implements Service {
    private Service service;    // il vero servizio
    public ServiceProxy(Service service) {
        this.service = service;
    }
    /* questo è proprio il metodo alpha che verrà invocato
    * dal client (anche se il client penserà di parlare
    * direttamente con il servizio) */
    public String alpha(String arg) {
        /* chiama il vero servizio */
        return service.alpha(arg);
    }
}
```



## Uso di proxy

- L'application context (o la service factory) crea la **ServiceImpl** (se necessario) – ma restituisce un **ServiceProxy**

```
package asw.intro.context;
import asw.intro. ...
/* Application context. */
public class ApplicationContext {
    ... variabile, costruttore e metodo per singleton, come prima ...
    ... metodo getClient, come prima ...
    private Service service = null;
    /* Factory method per il Servizio.
     * Restituisce un proxy al servizio. */
    public synchronized Service getService() {
        if (service==null) { service = new ServiceImpl(); }
        return new ServiceProxy( service );
    }
}
```



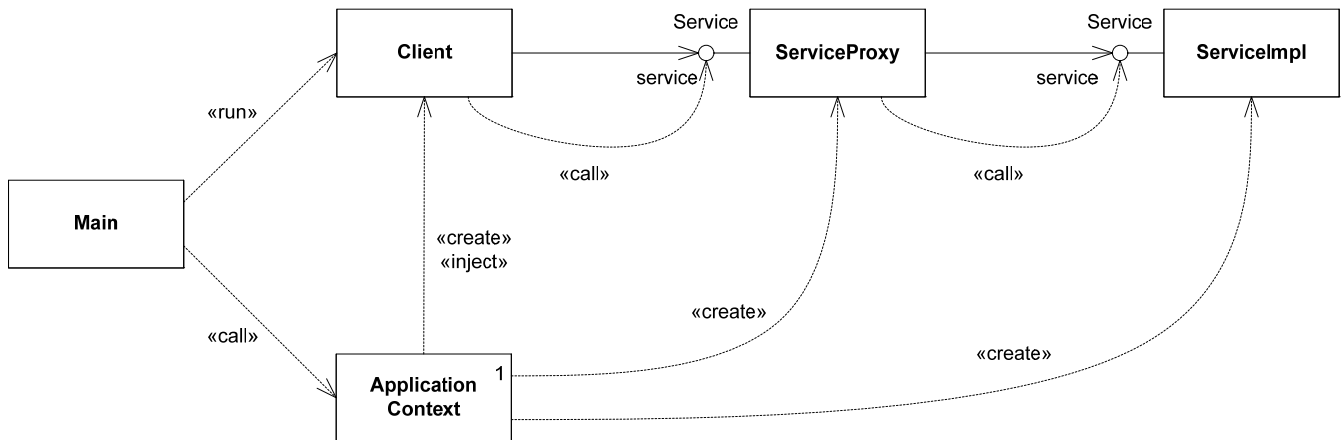
## Uso di proxy

- Non cambia né il **Client** né l'applicazione **Main**
  - il connettore è ancora, a tutti gli effetti, una chiamata di procedura locale

```
package asw.intro.client;
import asw.intro.service.Service;
/* Client del servizio Service. */
public class Client {
    private Service service;
    public Client() { }
    public void setService(Service service) {
        this.service = service;
    }
    public void run(...) {
        ... service.alpha(...) ...
    }
}
```



## Uso di proxy



37

Introduzione ai connettori

Luca Cabibbo ASW



## Uso di proxy

- ❑ Che cosa è cambiato?
  - in pratica, sembra tutto come prima – ovvero, sembra che abbiamo “cambiato tutto, per non cambiare nulla”
- ❑ Abbiamo però introdotto un elemento (il proxy) a cui possiamo dare delle responsabilità accessorie “non funzionali” (“da connettore”)
  - ad es., se si vuole effettuare il logging degli accessi al servizio, le istruzioni per la gestione del logging possono essere localizzate proprio nel proxy
    - senza cambiare né client né servizio – ovvero, i componenti che si occupano degli aspetti funzionali
  - oppure, se la computazione eseguita dal servizio è onerosa, possiamo fare caching delle richieste e delle risposte
    - aggiungendo codice nel proxy (nell’es., nel metodo **alpha**)
    - senza cambiare né client né servizio

38

Introduzione ai connettori

Luca Cabibbo ASW



## - Una versione distribuita (versione f)

- Per esemplificare questa importante osservazione – “al proxy possiamo dare delle responsabilità accessorie non funzionali, da connettore” – consideriamo ora l’eliminazione della co-localizzazione tra **Client** e **ServiceImpl**
  - è possibile gestire una comunicazione remota tra client e servizio usando un meccanismo di comunicazione interprocesso
    - ad es., i socket – un meccanismo di IPC, fornito dal sistema operativo, basato sullo scambio di messaggi (datagrammi, con UDP) oppure su un canale di comunicazione bidirezionale (con TCP)
  - in questa dispensa siamo interessati a comprendere **chi** può gestire questa comunicazione remota
    - l’aspetto di **come** gestire la comunicazione remota è invece discusso in una successiva dispensa

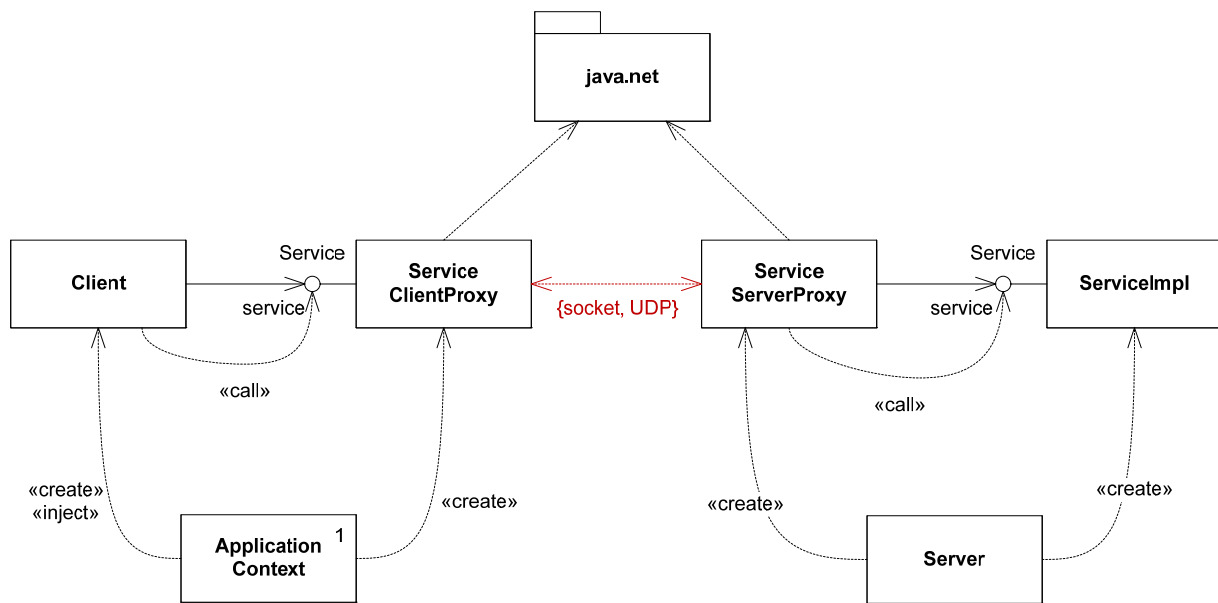


## Una versione distribuita

- La comunicazione remota tra **Client** e **ServiceImpl** può essere separata dagli aspetti funzionali tramite una coppia di *remote proxy*
  - un *remote proxy lato client*
    - un intermediario che si occupa dell’interazione remota tra **Client** e **ServiceImpl** – che vive nello stesso processo del **Client**
  - un *remote proxy lato server*
    - un altro intermediario che si occupa dell’interazione remota – che vive nello stesso processo di **ServiceImpl**



## Una versione distribuita



41

Introduzione ai connettori

Luca Cabibbo ASW



## Una versione distribuita

- Utilizzando i socket UDP, client e server possono comunicare scambiandosi messaggi – richiesta e risposta – come segue
  - il componente client vuole invocare un'operazione del componente server – ma invoca un'operazione del suo proxy
  - il proxy lato client forma un messaggio di richiesta che codifica l'invocazione – con il nome dell'operazione e l'elenco dei parametri – e invia questo messaggio al proxy lato server
  - il proxy lato server riceve il messaggio di richiesta e lo decodifica – identifica l'operazione e l'elenco dei parametri e invoca l'operazione del componente server
  - il server esegue l'operazione invocata, e restituisce un risultato
  - il proxy lato server forma un messaggio di risposta che codifica il risultato – e lo invia al proxy lato client
  - il proxy lato client riceve il messaggio di risposta e lo decodifica – estrae il risultato e lo restituisce al componente client
  - il client ha così ricevuto il risultato della sua invocazione

42

Introduzione ai connettori

Luca Cabibbo ASW



## Una versione distribuita

- Il **ServiceClientProxy** è un “remote proxy” lato client, con la seguente struttura

```
package asw.intro.client.connector;

import asw.intro.service.Service;
import java.net.*;    // per le socket

/* remote proxy lato client per il servizio */
public class ServiceClientProxy implements Service {

    private InetAddress address;    // indirizzo del server
    private int port;              // porta per il servizio

    public ServiceClientProxy(InetAddress address, int port) {
        this.address = address;    this.port = port;
    }

    public String alpha(String arg) {
        ... segue ...
    }

}
```



## Una versione distribuita

- Il metodo **alpha** del “remote proxy” lato client

```
/* questo è proprio il metodo alpha invocato
 * dal client (anche se il client pensa di parlare
 * direttamente con l'implementazione del service) */
public String alpha(String arg) {

    ... crea un datagramma di richiesta che codifica
        l'invocazione del servizio, con i relativi parametri ...
    ... invia il datagramma di richiesta ...
    ... riceve il datagramma di risposta ...
    ... estrae il risultato dal datagramma di risposta ...
    return reply;

}
```



## Una versione distribuita

- Lato client, l'application context (o la service factory) può essere usato per incapsulare la creazione del proxy lato client
  - si può usare un approccio data-driven per quanto riguarda, ad es., l'indirizzo di rete e la porta del server
    - qui ipotizziamo che il server sia in esecuzione su **10.11.1.121** e ascolti sulla porta **6789**
    - l'approccio data-driven può essere usato anche per specificare il tipo del proxy da creare – se sono disponibili più proxy relativi a modalità di connessione diverse
  - tuttavia, l'application context (o la factory) lato client non si può più occupare della creazione del vero **ServiceImpl** – poiché questi vive nel processo del server
- Il **Client** e il **ServiceImpl** (i “componenti”) possono rimanere ancora immutati rispetto alla versione precedente



## Una versione distribuita

```
package asw.intro.client.context;
import asw.intro. ...
/* Application context. */
public class ApplicationContext {
    ... variabile, costruttore e metodo per singleton, come prima ...
    ... metodo getClient, come prima ...
    private Service service = null;
    /* Factory method per il Servizio.
     * Restituisce un remote proxy al servizio. */
    public Service getService() {
        InetAddress address = InetAddress.getByName("10.11.1.121");
        int port = 6789;
        Service proxy = new ServiceClientProxy(address, port);
        return proxy;
    }
}
```



## Una versione distribuita

- Il “remote proxy” lato client non comunica direttamente con il **ServiceImpl**
  - piuttosto, è necessario un ulteriore intermediario – un “remote proxy” lato server
  - il “remote proxy” lato server
    - riceve richieste – tramite socket – dal proxy lato client, e le decodifica ed effettua le corrispondenti invocazioni a **ServiceImpl**
    - riceve risultati da **ServiceImpl** – e li invia al proxy lato client sotto forma di risposte



## Una versione distribuita

- Struttura del “remote proxy” lato server

```
package asw.intro.server.connector;

import asw.intro.service.Service;
import asw.intro.server.ServiceImpl;
import java.net.*;

/* remote proxy lato server per il servizio */
public class ServiceServerProxy {

    private Service service;    // il vero servizio
    private int port;          // porta per il servizio

    public ServiceServerProxy(Service service, int port) {
        this.service = service;    this.port = port;
    }

    public void run() {
        ... segue ...
    }

}
```





## Una versione distribuita

- Il metodo **run** del “remote proxy” lato server
  - per semplicità, un server “sequenziale” – anche in questo caso si potrebbe fare di meglio

```
public void run() {  
    ... crea la socket su cui ricevere le richieste ...  
    while (true) {  
        ... aspetta un datagramma di richiesta ...  
        ... estrae l'invocazione dal datagramma di richiesta ...  
        ... effettua l'invocazione del servizio  
          e ottiene il risultato ...  
        ... crea il datagramma di risposta ...  
        ... invia il datagramma di risposta ...  
    }  
}
```



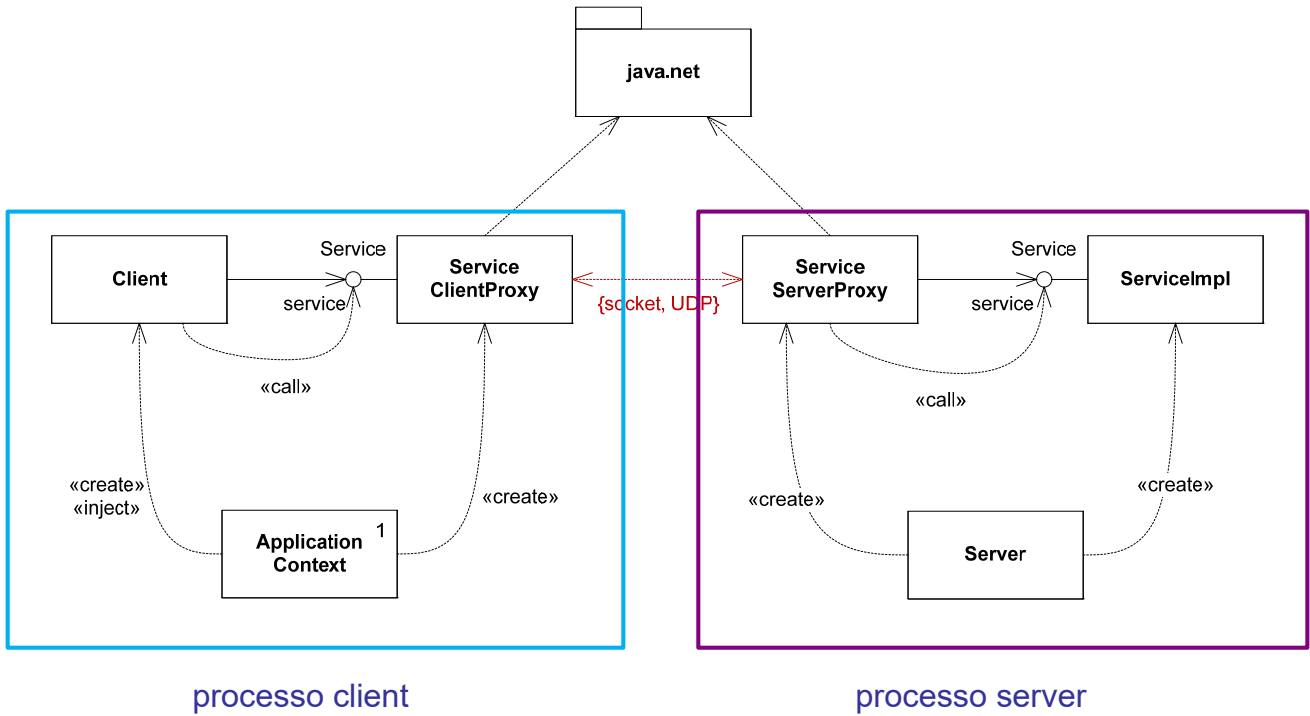
## Una versione distribuita

- Lato server, serve un oggetto **Server** – supponiamo che il server venga eseguito su **10.11.1.121** e ascolti sulla porta **6789**
  - il **Server** è responsabile della creazione di **ServiceImpl**
  - è responsabile della creazione e dell'avvio del proxy lato server

```
package asw.intro.server.connector;  
  
import asw.intro.service.Service;  
import asw.intro.server.ServiceImpl;  
  
/* server per il servizio */  
public class Server {  
  
    public static void main(String[] args) {  
        Service service = new ServiceImpl();  
        int port = 6789;  
        ServiceServerProxy server =  
            new ServiceServerProxy(service, port);  
        server.run();  
    }  
}
```

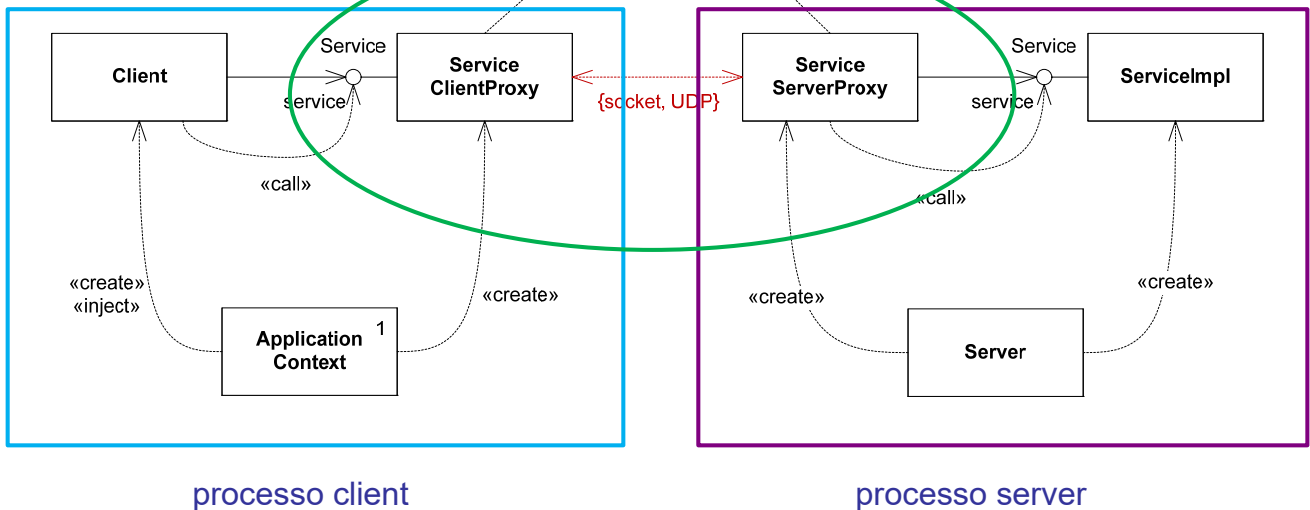


# Una versione distribuita



# Una versione distribuita

Anche se sono due classi distinte, sono parte della definizione di un unico connettore!





## Una versione distribuita

- Alcune caratteristiche di questa soluzione
  - si tratta ancora di una chiamata di procedura
    - è però una *chiamata di procedura remota*
  - si tratta ancora di una chiamata sincrona
  - l'accoppiamento tra **Client** e **ServiceImpl** è localizzato nel connettore – che comprende diverse classi
    - attenzione, a livello di codice, non c'è più nessun accoppiamento diretto, se non tramite l'interfaccia **Service**
    - l'accoppiamento – localizzato nel connettore – è finito nel protocollo di comunicazione adottato per la comunicazione remota
  - in linea di principio, **Client** e **ServiceImpl** potrebbero essere scritti con linguaggi di programmazione diversi, in esecuzione in ambienti hardware/software differenti – grazie al supporto “universale” dei socket



## - Discussione

- L'esempio (con le sue varianti) ha mostrato la comunicazione – locale e remota – tra due componenti
  - scopo dell'esempio era soprattutto mostrare che gli aspetti funzionali (codice in “violetto” e “rosso”) possono – e in genere devono – essere considerati separatamente da quelli relativi all'interazione/comunicazione (codice in “nero”)
  - nello specifico, l'esempio ha mostrato (anche se in modo parziale) l'uso dei socket come API per realizzare connettori nel contesto dei sistemi distribuiti
    - ma, come vedremo, spesso si utilizzano soluzioni basate su API e librerie più ricche per l'implementazione dei connettori – il cosiddetto middleware
  - inoltre, l'esempio ha mostrato l'uso di factory, application context, proxy e dell'iniezione delle dipendenze – sono dei pattern che è comune incontrare nell'uso degli strumenti di middleware



## Discussione

- Alcune domande relative all'esempio visto
  - come realizzare il logging degli accessi al servizio?
    - questi aspetti possono essere gestiti dal connettore
  - come rendere confidenziale lo scambio di messaggi in rete – in particolare, sulla base di meccanismi di cifratura?
    - questi aspetti possono essere gestiti dal connettore
  - come realizzare un meccanismo di autenticazione e autorizzazioni basato su id e password?
    - questi aspetti possono essere gestiti dal connettore
  - che cosa fare, ad es., se si verifica un guasto nella rete o si perde un datagramma UDP? usare socket TCP è una soluzione o no?
    - questi aspetti possono essere gestiti dal connettore



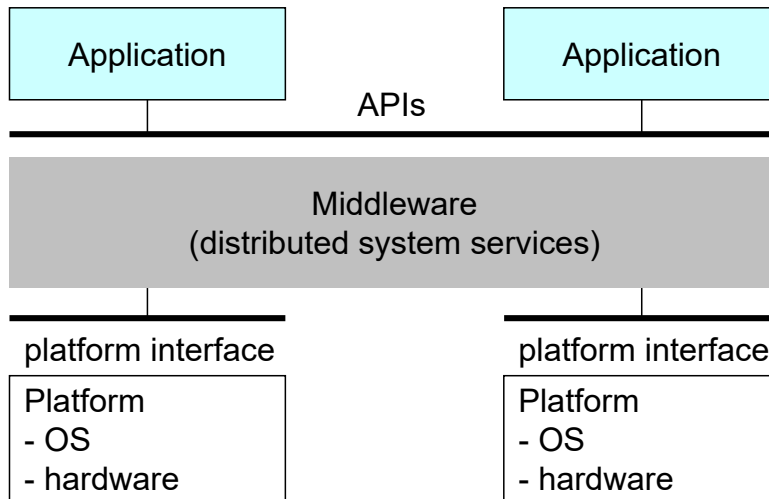
## \* Introduzione al middleware

- Meccanismi di base come i socket possono essere considerati l'“assembler” per lo sviluppo dei connettori
  - usando questi meccanismi e altre librerie di base è possibile realizzare connettori molto complessi – che gestiscono qualità complesse del software – separati dai componenti, che si possono così occupare dei soli aspetti funzionali
- In pratica, diversi connettori di uso comune sono stati generalizzati – definendo la classe degli strumenti di middleware
  - il middleware è un insieme di tecnologie che offre una molteplicità di paradigmi di interazione tra componenti – sostenendo e semplificando la realizzazione dei connettori
  - il middleware è uno strato software “in mezzo”
    - tra componenti distribuiti
    - ma anche sotto ai componenti applicativi e sopra al sistema operativo e la piattaforma



# Middleware

- Un **servizio di middleware** è [Bernstein]
  - un servizio distribuito, general-purpose e multi-piattaforma
  - che si colloca tra piattaforme e applicazioni
  - fornisce un'astrazione di programmazione distribuita – di solito implementa un insieme di protocolli e API standard
  - per aiutare a risolvere problemi di eterogeneità e distribuzione



57

Introduzione ai connettori

Luca Cabibbo ASW



## Middleware – esempi

- Alcuni esempi di (famiglie di) prodotti di middleware
  - invocazione remota (RPC e RMI)
  - comunicazione asincrona (MOM)
  - SQL Stored Procedures
  - ORB
  - component middleware (EJB)
  - Web Services
  - REST
  - ...

58

Introduzione ai connettori

Luca Cabibbo ASW



## \* Discussione

- Che cosa vedremo in questa parte del corso (dispense 8xx)
  - alcune tecnologie di middleware
    - di ciascun servizio di middleware, vedremo solo alcuni semplici esempi di uso
    - lo scopo è introdurre ed esemplificare alcune tecnologie rappresentative, le loro principali finalità, alcuni dei problemi affrontati e risolti, alcuni dei problemi non risolti o che devono essere presi in considerazione dal programmatore
    - talvolta cercheremo anche di comprendere la modalità di funzionamento e/o la struttura interna di questi strumenti
  - inoltre, vedremo alcune tecnologie e strumenti per gli ambienti di esecuzione
    - macchine virtuali, container e orchestrazione di container



## Discussione

- Tecnologie di middleware che vedremo in questa parte del corso
  - socket
    - meccanismo di base della IPC
    - non è uno strumento di middleware – ma consente di discutere alcune problematiche affrontate dal middleware
  - chiamata di procedura remota (RPC) e invocazione remota (RMI)
  - comunicazione asincrona
    - basata sullo scambio di messaggi o sulla notifica di eventi
  - componenti
  - servizi REST
  - soprattutto nel contesto del framework Spring, di Spring Boot e di Spring Cloud