



Java Messaging Service e Enterprise Service Bus

Alma Mater Studiorum - Università di Bologna
CdS Laurea Magistrale in Ingegneria Informatica
I Ciclo - A.A. 2017/2018

Corso di Sistemi Distribuiti M

07 - Java Messaging Service (JMS) e Cenni su Enterprise Service Bus (ESB)

Docente: Paolo Bellavista
paolo.bellavista@unibo.it

<http://lia.disi.unibo.it/Courses/sd1718-info/>
<http://lia.disi.unibo.it/Staff/PaoloBellavista/>



Perché Utilizzare Servizi di Messaging?

- ❑ Comunicazione **disaccoppiata** (o loosely coupled)
- ❑ Comunicazione **asincrona**
- ❑ Messaggi come strumento principale di comunicazione fra applicazioni (**modello a scambio di messaggi**)
- ❑ È il software di supporto allo scambio di messaggi a fornire le funzionalità di base necessarie
 - Message Oriented Middleware (MOM), Messaging system, Messaging server, Messaging provider, JMS provider



Perché usare sistemi di messaging?

- ❑ Indipendenza dalla **piattaforma**
- ❑ Indipendenza dalla **locazione di rete**
- ❑ Appropriato per lavorare in **ambienti eterogenei**
- ❑ Anonimità
 - **Who?**
 - **Where?**
 - **When?**

} **non importano**
disaccoppiamento nello spazio e nel tempo
- ❑ Fortemente diverso rispetto a sistemi basati su RPC
 - CORBA
 - RMI



Perché sistemi di messaging?

❑ **Scalabilità**

- Capacità di gestire un **numero elevato di clienti**
 - Senza cambiamenti nella logica applicativa
 - Senza cambiamenti nell'architettura
 - Senza (grosso) degrado nello throughput di sistema

Si tendono a incrementare le capacità hardware del sistema di messaging se si desidera una maggiore scalabilità complessiva

❑ **Robustezza**

- consumatori possono avere un fault
- produttori possono avere un fault
- rete può avere un fault

Senza problemi (almeno per un po' di tempo...) per il sistema di messaging



Esempi di Applicazioni di Messaging

- ❑ ***Transazioni commerciali*** che usano carte di credito
- ❑ Report con ***previsioni del tempo***
- ❑ ***Workflow***
- ❑ ***Gestione di dispositivi di rete***
- ❑ Gestione di supply chain
- ❑ Customer care
- ❑ ...



Caratteristiche Usuali per i Servizi di Messaging

- ❑ Supporto a **due modelli di messaging**
 - **Point-to-point**
 - **Publish/Subscribe**
- ❑ Affidabilità (in che senso?)
- ❑ Operazioni con logica **transazionale**
- ❑ Messaging distribuito, ovviamente (che cosa significa?)
- ❑ Sicurezza
- ❑ Alcuni sistemi di messaging supportano:
 - **Consegna con qualità** (garanzie/indicazioni di real-time)
 - Transazioni sicure
 - **Auditing, metering, load balancing, ...**



❑ **Point-to-Point**

- Un messaggio è **consumato da un singolo ricevente**
- Utilizzato quando ogni messaggio prodotto deve essere processato con successo da un singolo consumatore

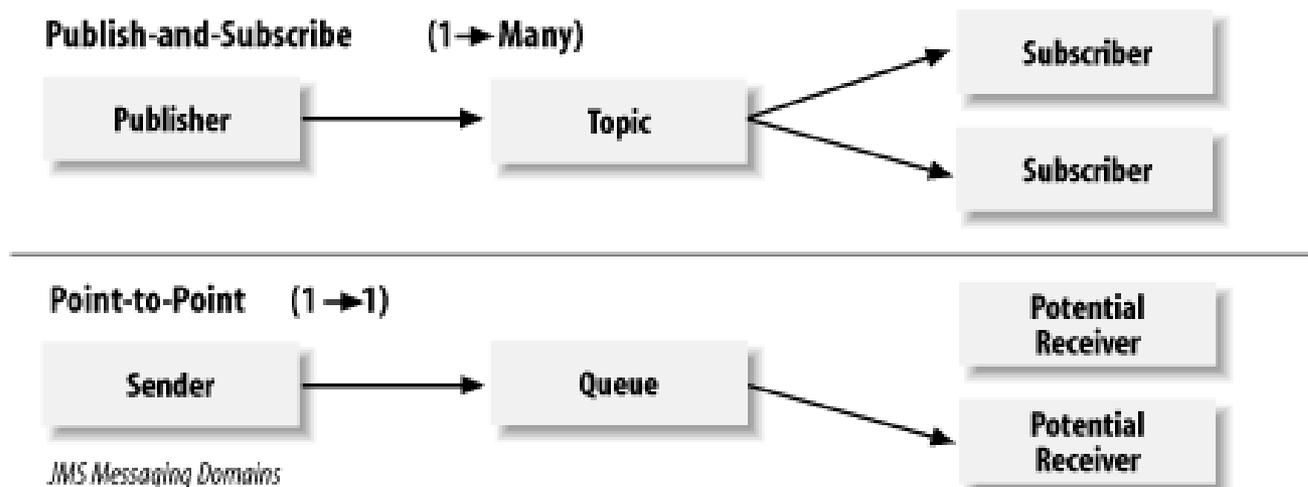
❑ **Publish/Subscribe**

- Un messaggio **consumato da riceventi multipli**
- Ad esempio, una applicazione di bacheca per richieste di lavoro. Creazione di un nuovo topic (argomento) di nome “new hire”; diverse applicazioni possono sottoscrivere (subscribe/abbonarsi) il proprio interesse al topic “new hire”



Point-to-Point

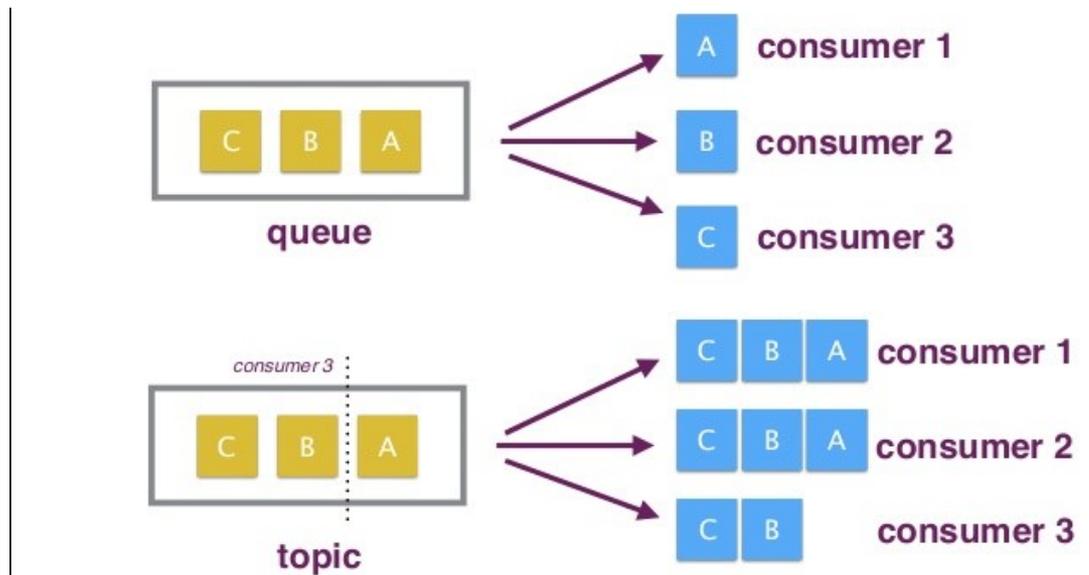
- ❑ Un messaggio è consumato da un ***singolo ricevente***
- ❑ Ci possono essere produttori multipli, ovviamente
- ❑ La “destinazione” di un messaggio è una ***coda con nome (named queue)***
- ❑ FIFO (per lo stesso livello di priorità)
- ❑ Produttori inviano messaggi a named queue specificando un livello di priorità desiderato





Publish/Subscribe (Pub/Sub)

- ❑ Un messaggio consumato da **riceventi multipli**
- ❑ La “destinazione” di un messaggio è **un argomento con nome (named topic)**
- ❑ Produttori pubblicano su un topic
- ❑ Consumatori si “abbonano” a un topic





Affidabilità dello Scambio di Messaggi

- ❑ **Alcune garanzie** nella consegna dei messaggi
 - Gradi differenti di affidabilità (reliability) sono possibili
 - **Produttore può specificare diversi livelli di reliability**
 - **Affidabilità più elevata** va tipicamente di pari passo con **minore throughput**
- ❑ Di solito i supporti ai sistemi di messaging utilizzano **storage persistente per preservare i messaggi**



Operazioni sui Messaggi con Proprietà Transazionali

❑ **Produzione transazionale**

- Produttore può raggruppare una **serie di messaggi in un'unica transazione**
- O tutti i messaggi sono **accodati** con successo o nessuno

❑ **Consumo transazionale**

- Consumatore riceve un **gruppo di messaggi** come **serie** di oggetti con proprietà transazionale
- Fino a che **tutti i messaggi non sono stati consegnati e ricevuti con successo**, i messaggi rimangono **mantenuti permanentemente nella loro queue o topic**

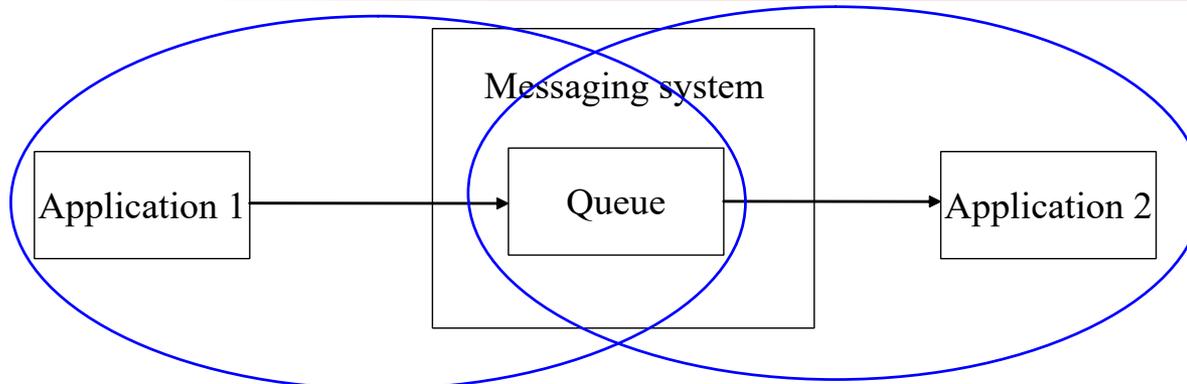


Scope della Transazionalità

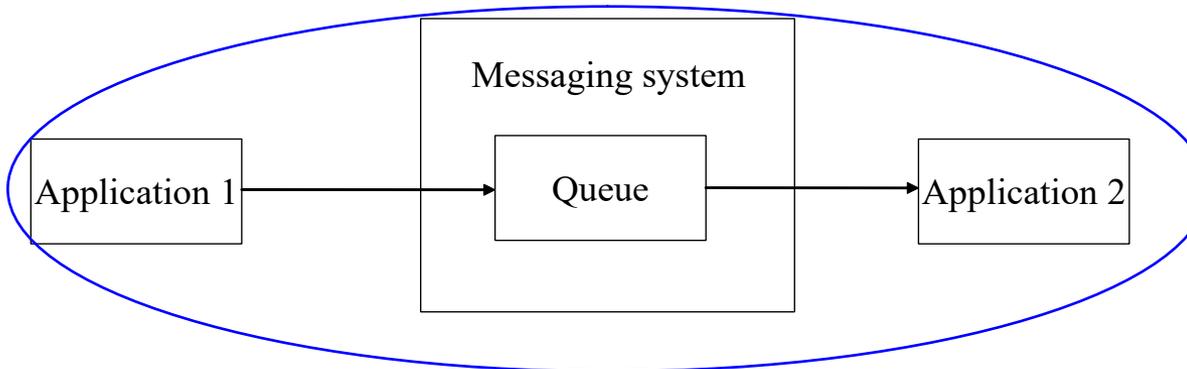
- ❑ **Scope client-to-messaging system**
 - Le proprietà di transazionalità riguardano ***l'interazione fra ogni cliente e il sistema di messaging***
 - JMS supporta questo tipo di scope
- ❑ **Scope client-to-client**
 - Le proprietà di transazionalità riguardano ***l'insieme delle applicazioni produttore-consumatore per quel gruppo di messaggi***
 - Non supportato da JMS



Scope della Transazionalità



client-to-messaging system



client-to-client

Ovviamente ***sistema di messaging può essere distribuito*** a sua volta
Sistemi di enterprise messaging possono realizzare una ***infrastruttura in cui i messaggi sono scambiati fra server nel distribuito***



Supporto alla Sicurezza

❑ **Autenticazione**

- I sistemi di messaging richiedono usualmente ai clienti di presentare **certificati digitali** con signature

❑ **Confidenzialità** dei messaggi

- Usualmente **encryption** dei messaggi

❑ **Integrità** dei messaggi

- Usualmente integrità dei dati tramite **digest** dei messaggi

❑ Sicurezza è **gestita in modo dipendente dal vendor** del sistema di messaging (ad es. non specificata affatto in JMS)



- ❑ JMS come ***insieme di interfacce Java*** (e associata definizione di semantica) che specificano come un cliente JMS possa accedere alle ***funzionalità di un sistema di messaging generico***
- ❑ Supporto alla ***produzione, distribuzione e consegna di messaggi***
- ❑ Supporto a ***diverse semantiche*** per message delivery
 - ***Sincrona/asincrona (bloccante/non-bloccante), con proprietà transazionali, garantita, durevole***
- ❑ Supporto sia a modello ***Point-to-Point (reliable queue)*** che ***Publish/Subscribe***
 - Selettori di messaggio lato ricevente
 - 5 tipologie di messaggi possibili

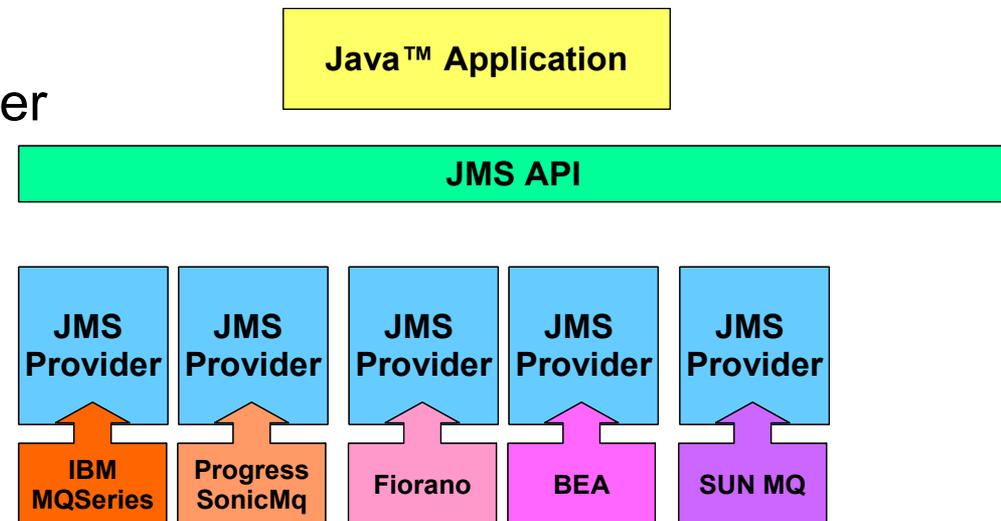


Obiettivi di Design di JMS

JMS è parte della piattaforma J2EE; MA NON
necessita di EJB container per essere usato (è “solo”
fortemente integrato)

Obiettivi:

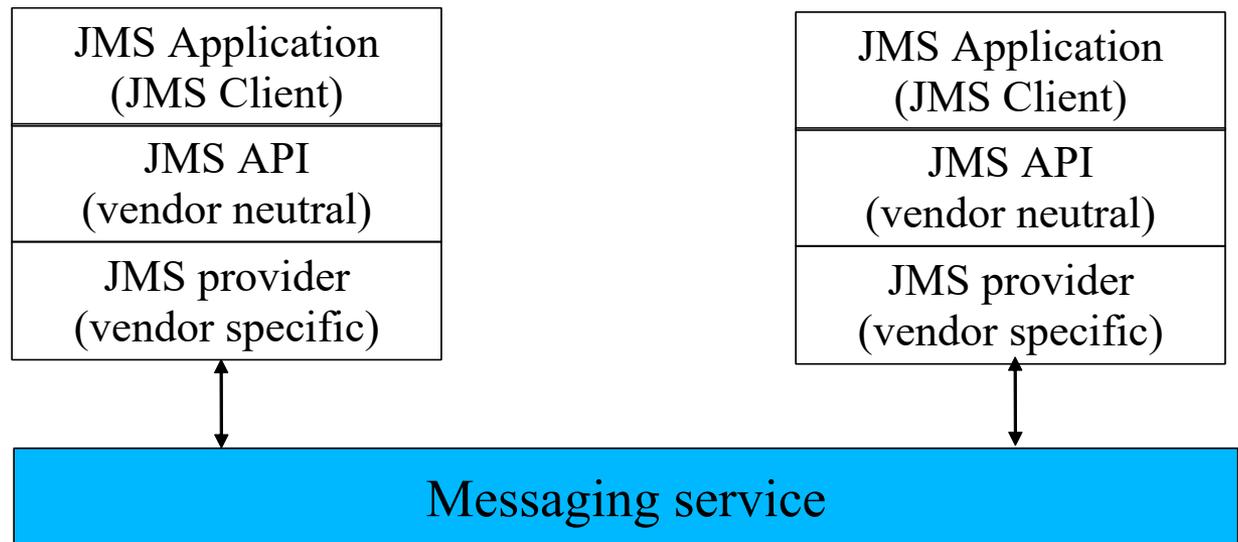
- ❑ **Consistenza** con le API dei **sistemi di messaging esistenti**
- ❑ **Indipendenza** dal vendor del sistema di messaging
- ❑ Copertura della maggior parte delle funzionalità comuni nei sistemi di messaging
- ❑ Promuovere tecnologia Java per sistemi messaging





Componenti Architeturali di JMS

- ❑ Clienti JMS e non-JMS
- ❑ Messaggi
- ❑ Provider JMS (sistema di messaging)
- ❑ Oggetti amministrati tramite **JNDI**
 - *Destination*
 - *ConnectionFactory*





Domini JMS (Stili di Messaging)

❑ **Point-to-Point**

- i messaggi in una queue possono essere ***persistenti o non persistenti***

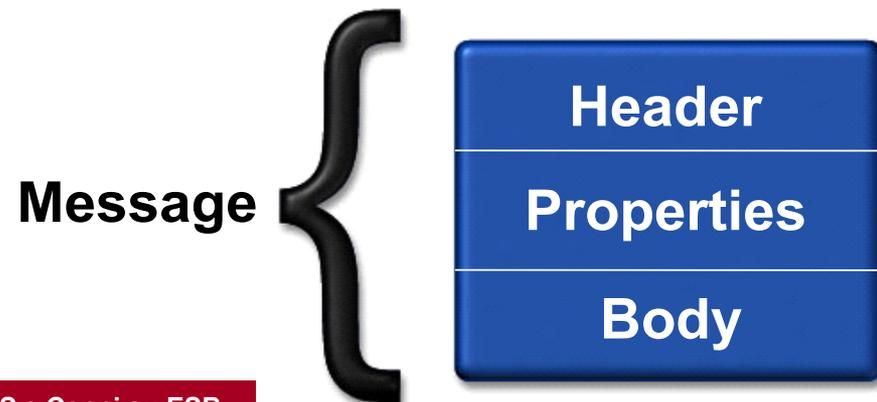
❑ **Pub/Sub**

- ***Non durevoli*** (i messaggi sono disponibili solo durante l'intervallo di tempo in cui il ricevente è attivo; se il ricevente non è connesso, la semantica è di ***poter perdere ogni messaggio prodotto in sua assenza***)
- ***Durevole*** (i messaggi sono ***mantenuti dal sistema***, che fa le veci dei riceventi non connessi al tempo della produzione dei messaggi; il ricevente non perde mai messaggi quando disconnesso)



Messaggi JMS

- ❑ Messaggi come modalità di comunicazione disaccoppiata fra le applicazioni
- ❑ I veri **formati** che attualmente sono utilizzati per l'encoding dei messaggi sono fortemente dipendenti dal provider del sistema di messaging
 - ***Un sistema di messaging può interoperare completamente solo al suo interno***
- ❑ JMS fornisce quindi **solo un modello astratto e unificato** per la rappresentazione interoperabile dei messaggi attraverso le sue interfacce





Message Header + Proprietà

- ❑ Utilizzato per *l'identificazione del messaggio e il suo routing*
- ❑ Include la **destination** e
 - modalità di consegna (persistente, non persistente), timestamp, priorità, campo ReplyTo

Elenco delle proprietà: JMSDestination, JMSDeliveryMode (persistente o no), JMSMessageID, JMSTimeStamp, JMSRedelivered, JMSExpiration, JMSPriority, JMSCorrelationID, JMSReplyTo (destinazione fornita dal produttore, dove inviare la risposta), JMSType (tipo del corpo del messaggio)

Proprietà dei messaggi (coppie nome/valore) possono essere: campi application-specific, campi dipendenti da e specifici di un particolare sistema di messaging, campi opzionali



Corpo del Messaggio (Message Body)

- ❑ Mantiene, ovviamente, il **contenuto del messaggio**
- ❑ Supporto per **diversi tipi di contenuto**, ogni tipo definito da una interfaccia:
 - StreamMessage, MapMessage, TextMessage, ObjectMessage, BytesMessage

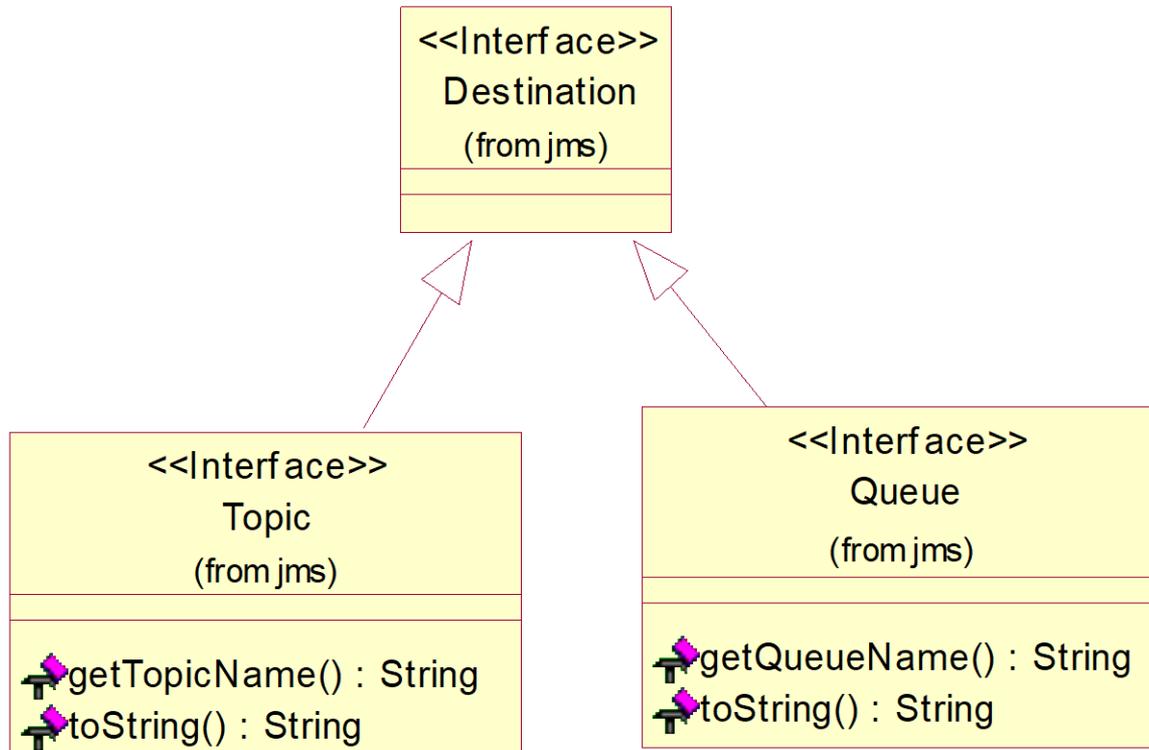
Ad esempio:

- **StreamMessage** contiene valori primitivi e supporta lettura sequenziale
- **MapMessage** contiene coppie nome/valore e supporta lettura sequenziale o by name
- **BytesMessage** contiene byte “non interpretati” e viene utilizzato di solito per fare match con formati preesistenti



Interfaccia Destination

- ❑ Rappresenta ***l'astrazione di un topic o di una queue*** (non di un ricevitore di messaggi!)
- ❑ Interfacce figlie per ***Queue*** e ***Topic***



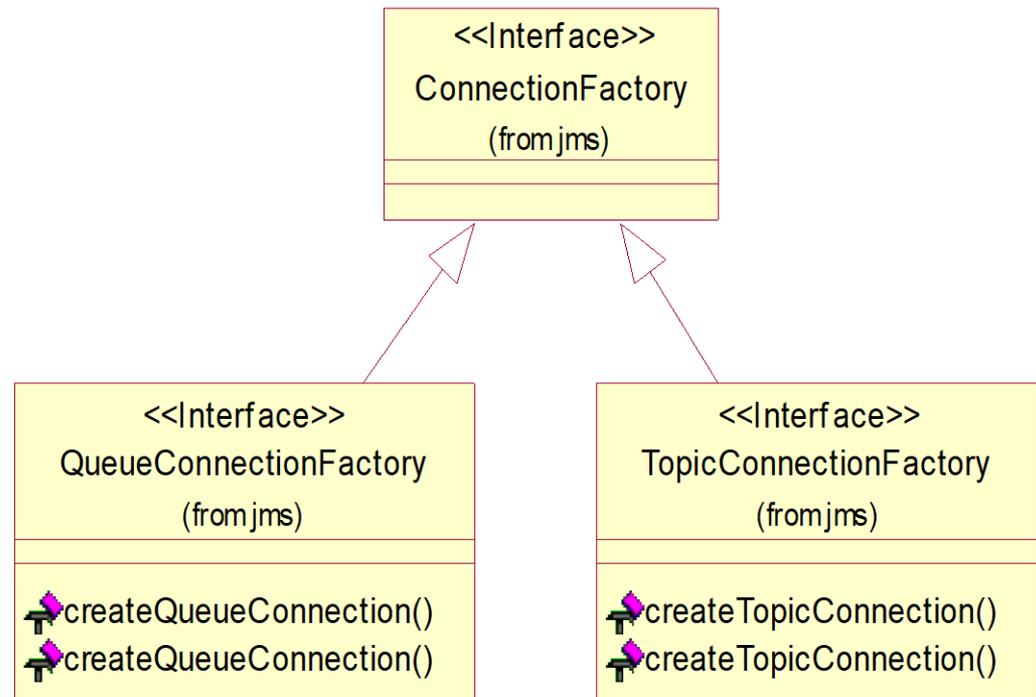


Interfaccia ConnectionFactory

- ❑ **Classe factory** per creare una **connessione provider-specific verso il server JMS**

Simile al gestore di driver (java.sql.DriverManager) in JDBC, per chi se lo ricorda...

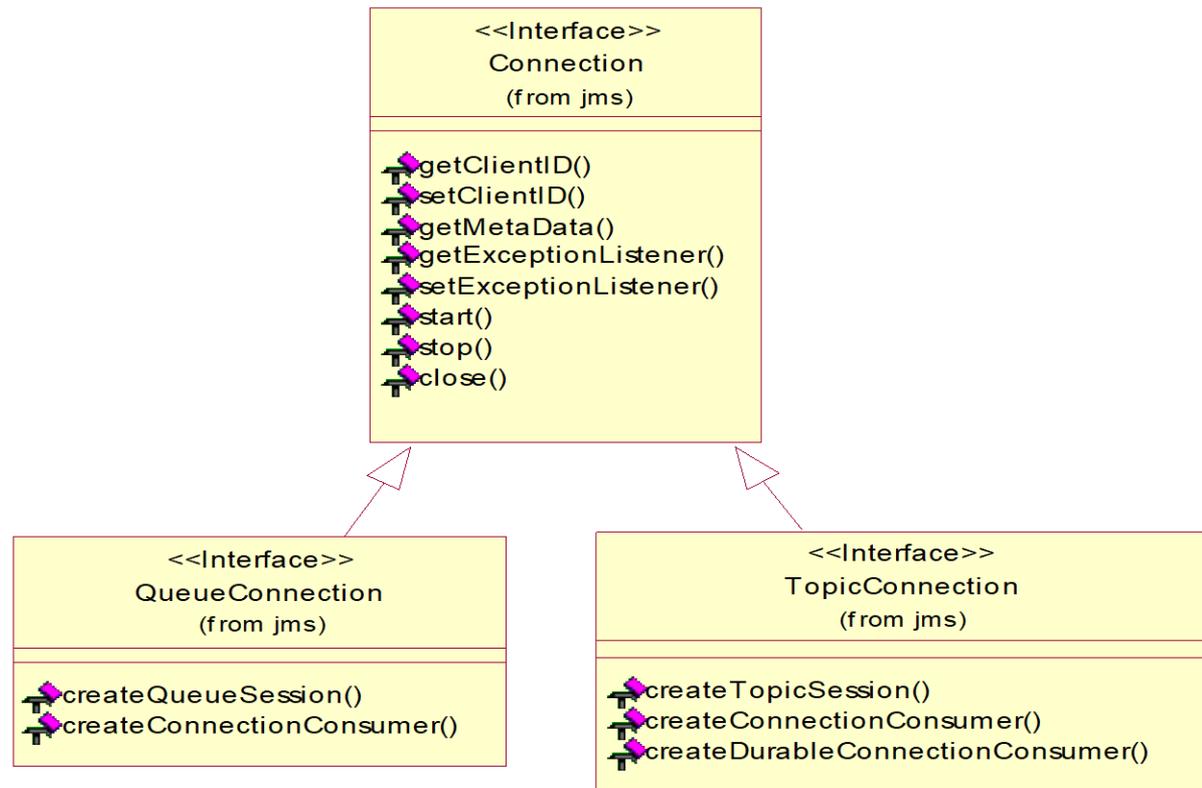
- ❑ **Interfacce figlie** per **QueueConnectionFactory** e **TopicConnectionFactory**





Interfaccia Connection

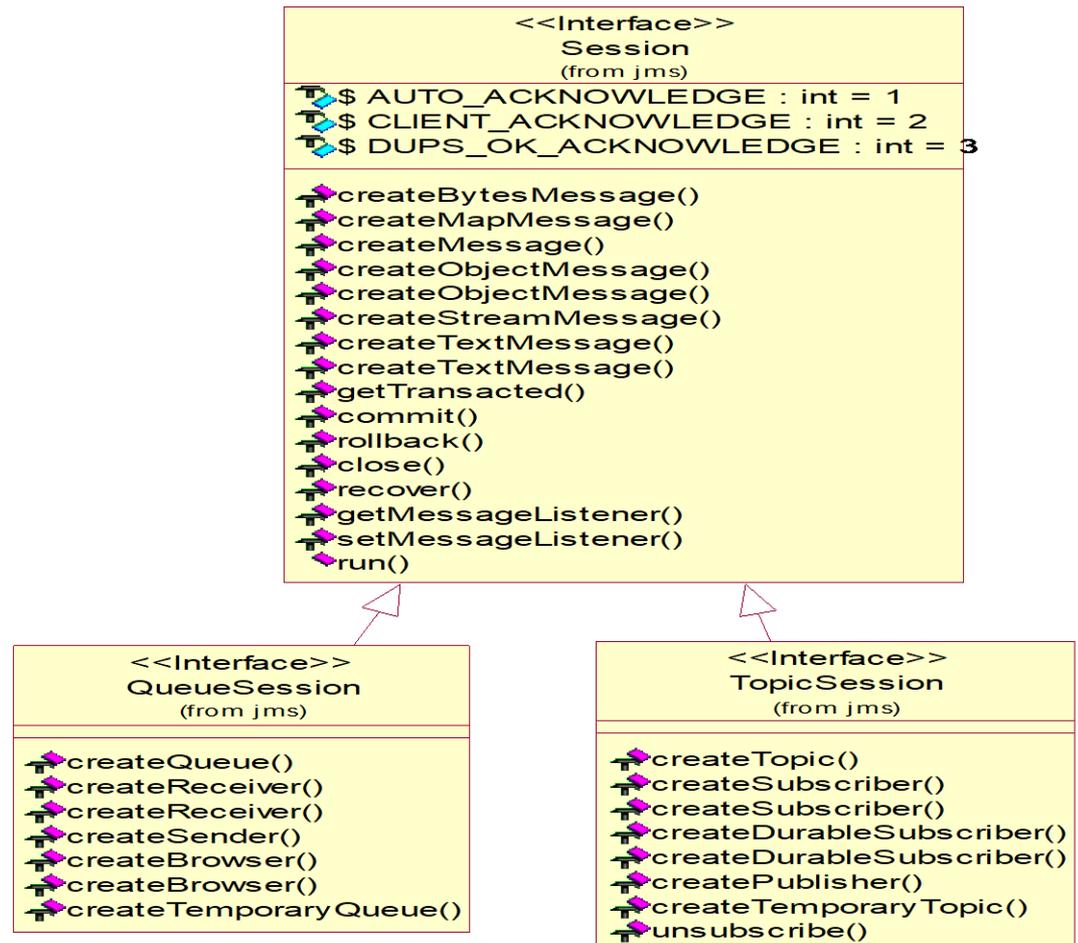
- ❑ **Astrazione** che rappresenta un **singolo canale di comunicazione verso il provider JMS**
- ❑ Connessione viene creata da un oggetto ConnectionFactory
- ❑ Connessione **dovrebbe essere chiusa** quando si è **terminato di utilizzare** la r





Interfaccia Session

- ❑ **Creata da un oggetto Connection**
- ❑ Una volta connessi al JMS provider attraverso una Connection, **tutte le operazioni si svolgono nel contesto una Session attiva**
- ❑ Una **sessione è single-threaded**, ovvero ogni operazione di invio e ricezione messaggio avviene **in modo serializzato**
- ❑ Le sessioni realizzano un **contesto "limitato" con "proprietà transazionali"**



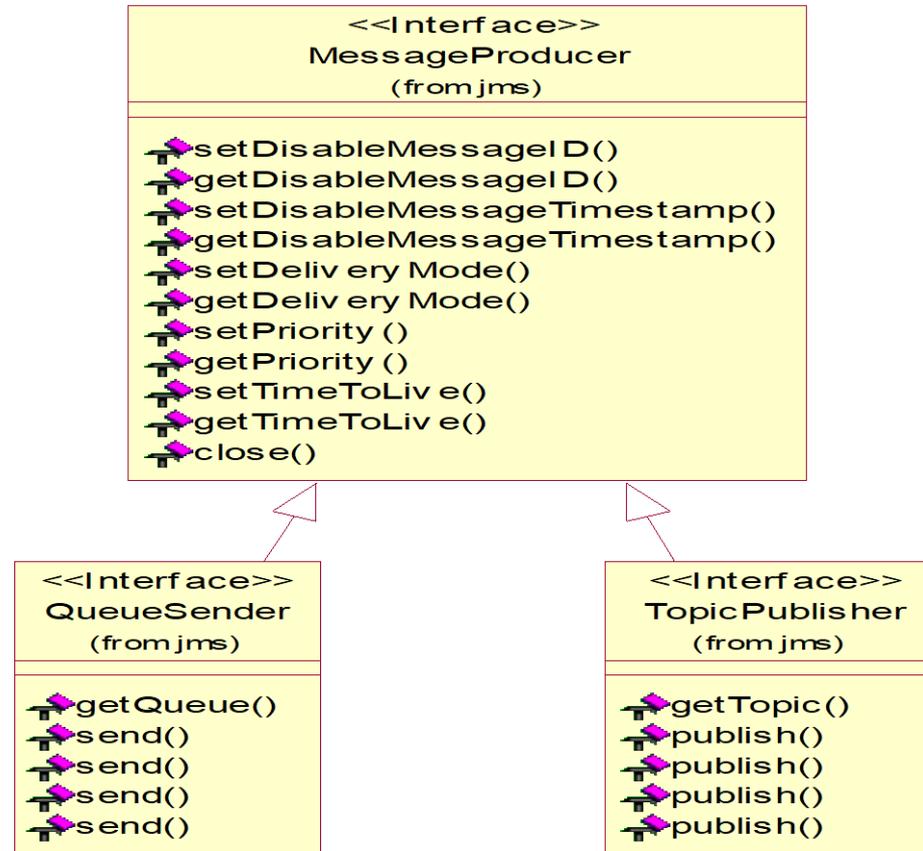


Interfacce MessageProducer e MessageConsumer

Per inviare un messaggio verso una Destination, il cliente deve **richiedere esplicitamente all'oggetto Session di creare un oggetto MessageProducer**

Analogamente interfaccia MessageConsumer

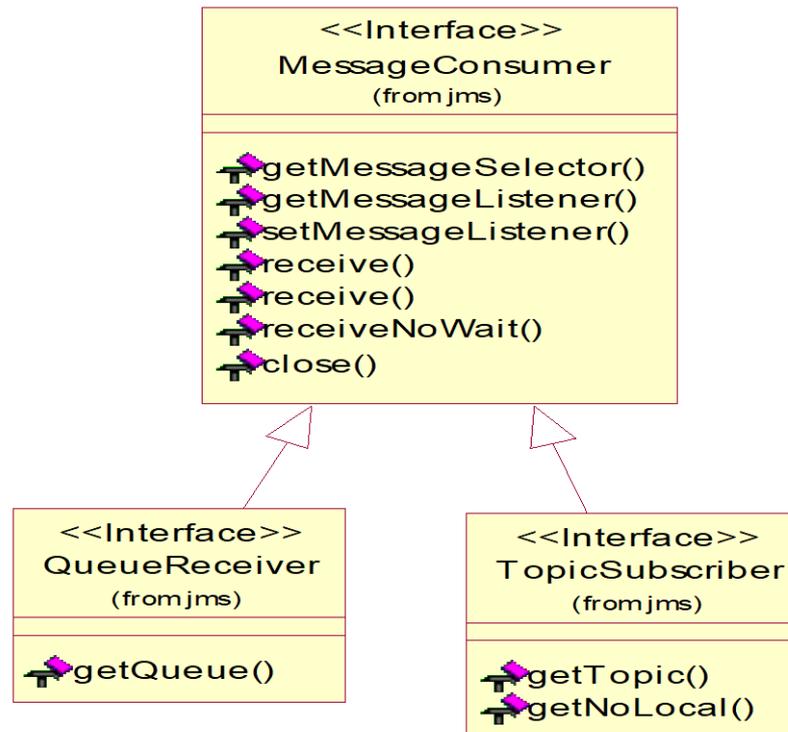
- ❑ Clienti che vogliono ricevere messaggi creano un oggetto MessageConsumer (collegato ad un oggetto Destination) attraverso Session
- ❑ **Due modalità di ricezione dei messaggi: blocking, non-blocking**





Modalità di Ricezione Blocking e Non-blocking

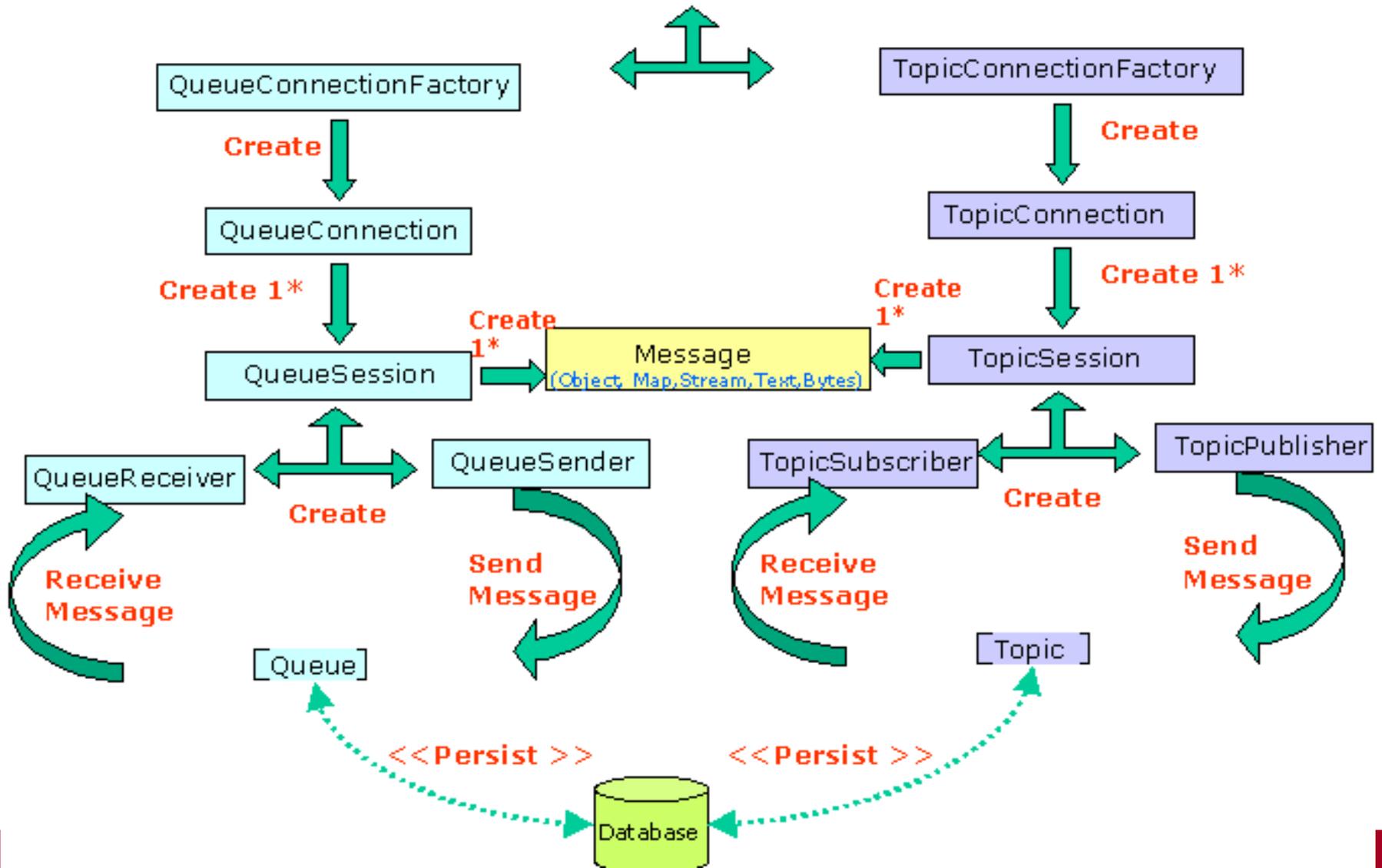
- ❑ **Modalità *blocking***: solito metodo [receive\(\)](#) bloccante
- ❑ **Modalità *non blocking***:
 - Cliente registra un oggetto [MessageListener](#)
 - Quando un messaggio è disponibile, il provider JMS richiama il metodo [onMessage\(\)](#) di [MessageListener](#) (**callback**)

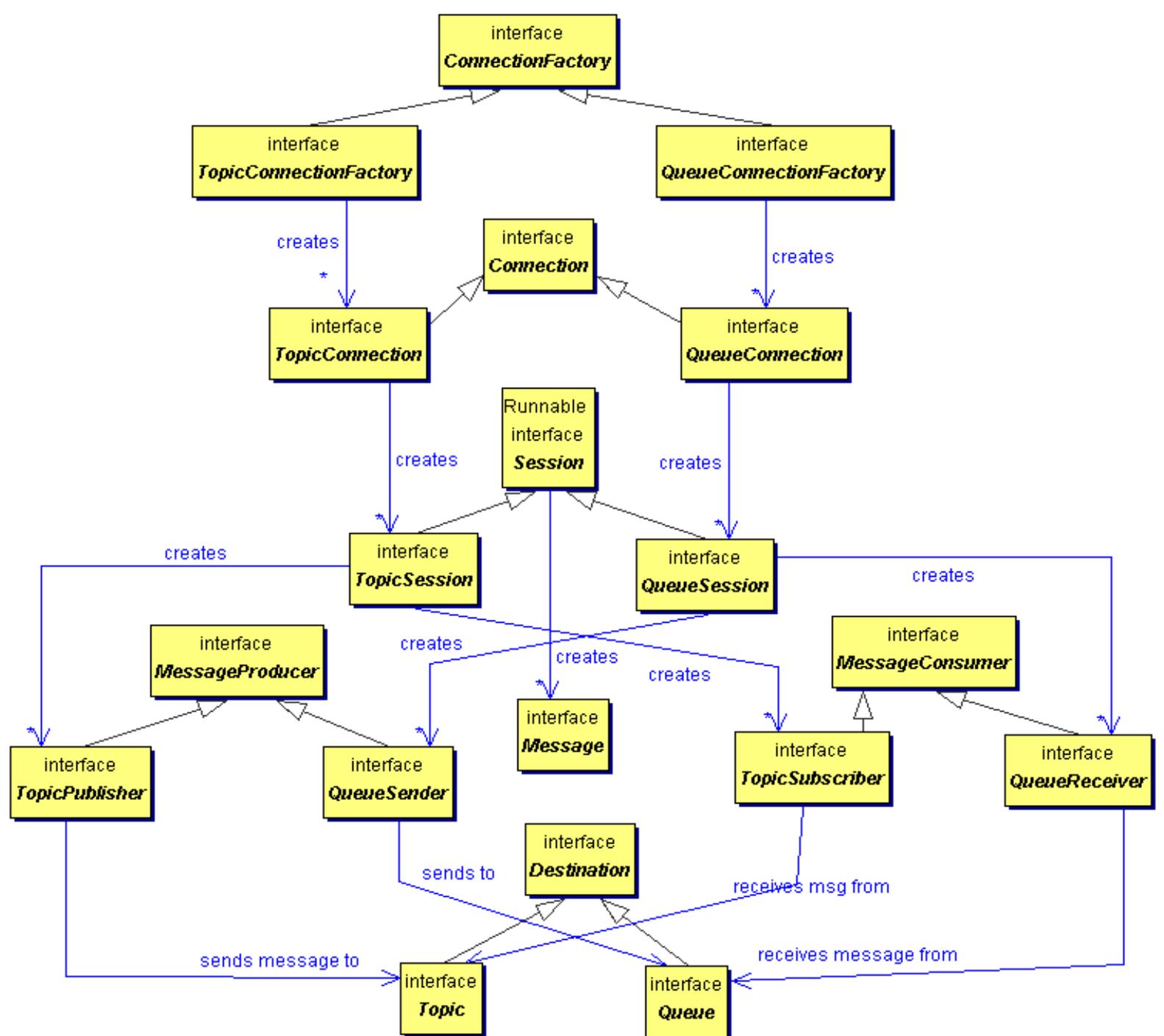




Riassunto “grafico” delle API JMS

<<Lookup from JNDI context>>







Esempi Pratici: Passi per Costruire una Applicazione JMS Sender

1. Ottenere un oggetto `ConnectionFactory` e un oggetto `Destination` (Topic o Queue) attraverso JNDI
2. Creare una `Connection`
3. Creare una `Session` per inviare/ricevere messaggi
4. Creare un oggetto `MessageProducer` (`TopicPublisher` o `QueueSender`)
5. Avviare la `Connection`
6. Inviare o pubblicare messaggi
7. Chiudere `Session` e `Connection`



Oggetti ConnectionFactory e Destination via JNDI

```
// Ottiene oggetto InitialContext
Context jndiContext = new InitialContext();

// Trova l'oggetto ConnectionFactory via JNDI
TopicConnectionFactory factory =
    (TopicConnectionFactory) jndiContext.lookup(
        "MyTopicConnectionFactory");

// Trova l'oggetto Destination via JNDI
// (Topic o Queue)
Topic weatherTopic =
    (Topic) jndiContext.lookup("WeatherData");
```



Oggetti Connection e Session

```
// Richiede la creazione di un oggetto Connection  
// all'oggetto ConnectionFactory
```

```
TopicConnection topicConnection =  
    factory.createTopicConnection();
```

```
// Crea un oggetto Session da Connection:  
// primo parametro controlla transazionalità  
// secondo specifica il tipo di ack
```

```
TopicSession session =  
    topicConnection.createTopicSession (false,  
        session.CLIENT_ACKNOWLEDGE);
```



MessageProducer e Avvio della Connection

```
// Richiede la creazione di un oggetto MessageProducer
// all'oggetto Session
// TopicPublisher per Pub/Sub
// QueueSender per Point-to-Point
TopicPublisher publisher =
    session.createPublisher(weatherTopic);

// Avvia la Connection
// Fino a che la connessione non è avviata, il
// flusso dei messaggi non comincia: di solito
// Connection viene avviata prima dell'invocazione
// dei metodi per la trasmissione messaggi
topicConnection.start();
// Creazione del messaggio
TextMessage message = session.createMessage();
message.setText("text:35 degrees");
// Invio del messaggio
publisher.publish(message);
```



Esempi Pratici: Passi per Ricevente JMS (non-blocking)

1. Ottenere oggetti `ConnectionFactory` e `Destination` (Topic o Queue) tramite JNDI
2. Creare un oggetto `Connection`
3. Creare un oggetto `Session` per inviare/ricevere messaggi
4. ***Creare un oggetto `MessageConsumer` (`TopicSubscriber` o `QueueReceiver`)***
5. ***Registrare `MessageListener` per modalità non-blocking***
6. Avviare la `Connection`
7. Chiudere `Session` e `Connection`



Oggetti TopicSubscriber e MessageListener

```
// Crea oggetto Subscriber da Session
```

```
TopicSubscriber subscriber =  
    session.createSubscriber(weatherTopic);
```

```
// Crea oggetto MessageListener
```

```
WeatherListener myListener  
    = new WeatherListener();
```

```
// Registra MessageListener per l'oggetto
```

```
// TopicSubscriber desiderato
```

```
subscriber.setMessageListener(myListener);
```



Affidabilità dei Messaggi

- ❑ Ad esempio, modalità “molto” affidabile per ***l’invio di un messaggio***: messaggio ***PERSISTENT*** all’interno di una ***transazione***
- ❑ Ad esempio, modalità “molto” affidabile per ***consumo di un messaggio***: ricezione all’interno di una transazione, sia nel caso di ricezione da queue che tramite “abbonamento” durevole a un topic

Basic Reliability

- Controllo ***ack*** dei messaggi
- Utilizzo di ***messaggi persistenti***
- Configurazione dei ***livelli di priorità***
- Consentire ***expiration*** di messaggi

Advanced Reliability

- “Abbonamenti” durevoli (durable subscription)
- Utilizzo di ***transazioni “locali”***



Ricezione e ACK

Alla ricezione, il cliente riceve e processa il messaggio

❑ ***Dopo, se necessario, ACK del messaggio***

- Acknowledgment è cominciato dal provider JMS o dal cliente, in dipendenza dalla **modalità di ACK scelta per la sessione**

❑ In sessioni ***con transazionalità (transacted)***

- **ACK automatico al commitment** della transazione
- In caso di roll-back della transazione, in seguito tutti i messaggi già consumati prima del roll-back sono **ri-inviati**

❑ In sessioni ***non-transacted***

- **ACK (quando e come) dipende** dal valore specificato come **secondo parametro** del metodo `createSession()`



Tipi di ACK (in ricezione)

- ❑ **Auto acknowledgment** (AUTO_ACKNOWLEDGE)
 - ACK automatico dopo il ritorno con successo dei metodi `MessageConsumer.receive()` o `MessageListener.onMessage()`. Possono esserci messaggi duplicati?

- ❑ **Client acknowledgment** (CLIENT_ACKNOWLEDGE)
 - Il cliente deve esplicitamente invocare il metodo `acknowledge()` dell'oggetto `Message`. ACK di un messaggio è cumulativo sui precedenti non-ack'ed. Possono esserci messaggi duplicati?

- ❑ **Lazy acknowledgment** (DUPS_OK_ACKNOWLEDGE)
 - Overhead minimo per provider JMS; invio “saltuario” di ack anche cumulativi da parte di JMS; **possibilità di messaggi duplicati** (*duplicate-tolerant application, idempotenza*)



ACK in produzione

Anche ***ack***, con modalità decisamente più semplici e gradi di libertà decisamente più limitati, ***in produzione***

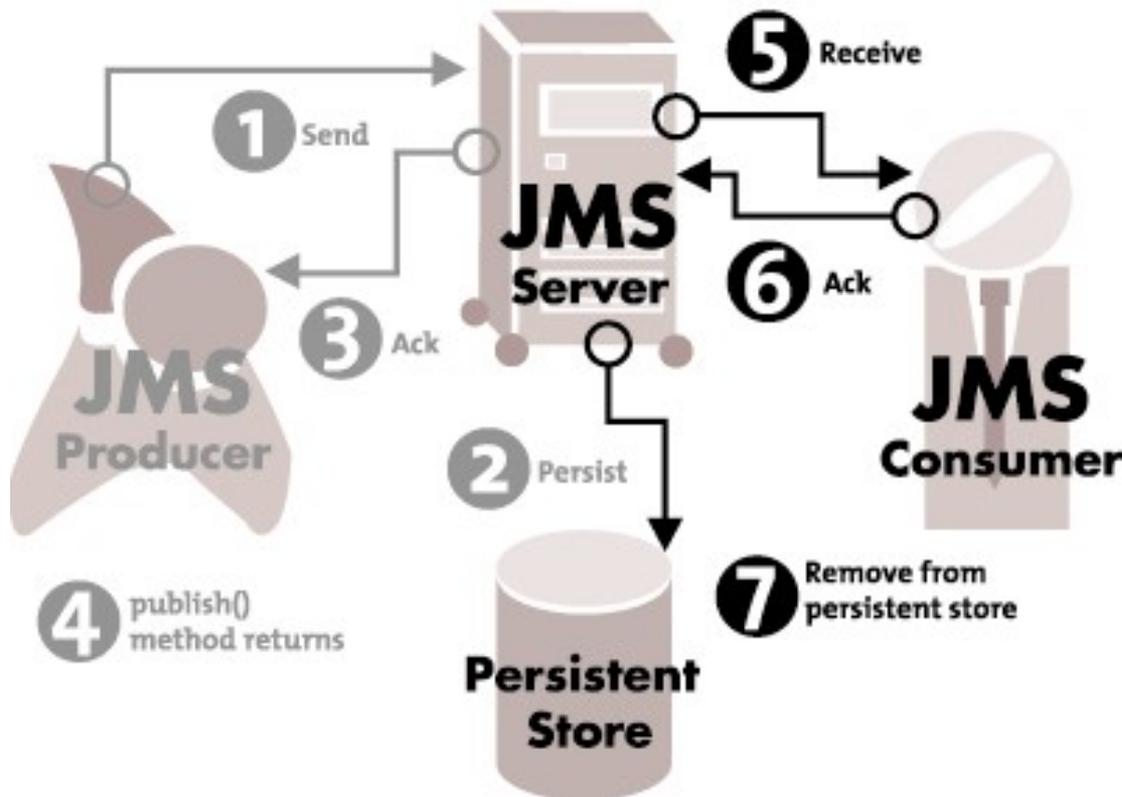
- ***Ack bloccante per send(), localmente*** al produttore
- Ritrasmissioni messaggi anche lato produzione

Differenze di performance fra

- Ack in produzione in sessione transacted
- Ack «normale»
- Dipendentemente da persistent o meno?



Ad esempio, per AUTO_ACK



- ❑ Prospettiva lato ***produttore e consumatore***
- ❑ Differenze fra caso ***persistent e non-persistent***
- ❑ Quando ci possono essere ***messaggi duplicati?***
- ❑ Quando ci può essere ***perdita di messaggi?***
- ❑ Inoltre, ***tre casi di ack differenziati***



Persistenza: 2 Modalità di Consegna

❑ **PERSISTENT**

- Default
- Specifica al **provider JMS di garantire** che il messaggio **non sia perso quando in transito**, ad esempio a causa di un guasto del provider JMS

❑ **NON_PERSISTENT**

- **NON richiede la memorizzazione dei messaggi** lato JMS provider
- Migliori risultati di performance

Metodo [SetDeliveryMode\(\)](#) nell'interfaccia [MessageProducer](#)

- `producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);`
- **forma estesa:** `producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);`



Priorità e Expiration nella Consegna dei Messaggi

- ❑ 10 livelli di priorità
 - da 0 (più basso) a 9 (più alto)
 - default = 4
- ❑ Uso del metodo `setPriority()` dell'interfaccia `MessageProducer`, ad esempio `producer.setPriority(7)` ;
o la forma estesa `producer.send(message, DeliveryMode.NON_PERSISTENT, 7, 10000)` ;
- ❑ Expiration: possibilità di **configurare TTL** tramite `setTimeToLive()` dell'interfaccia `MessageProducer`
 - `producer.setTimeToLive(60000)` ;
 - o forma estesa, `producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 60000)` ;



Configurazione Livelli di Affidabilità

Spesso scelte di default o prese alla creazione di Destination

Basic Reliability

- ❑ Persistenza
 - A livello di singolo messaggio, ad es. interfaccia MessageProducer
- ❑ Controllo degli ACK
 - A livello di sessione, interfaccia Session
- ❑ Livelli di priorità
 - A livello di singolo messaggio, ad es. interfaccia MessageProducer
- ❑ Expiration time
 - A livello di singolo messaggio, ad es. interfaccia MessageProducer

Advanced Reliability

- ❑ Sottoscrizione durevole
 - A livello di sessione, interfaccia Session
- ❑ Transazionalità
 - A livello di sessione, interfaccia Session



Come Funzionano le Durable Subscription?

- ❑ Un durable subscriber si registra specificando una ***identità univoca***
- ❑ In seguito, oggetti subscriber che hanno la medesima identità “recuperano l’abbonamento” (*subscription resume*) ***esattamente nello stato in cui è stato lasciato*** dal subscriber precedente
- ❑ Se un durable subscription ***non ha clienti attivi***, il provider JMS ***mantiene i messaggi*** fino a che questi non vengono ***effettivamente consegnati*** oppure alla loro ***expiration***
- ❑ All’interno di una singola applicazione Java, ***una sola session*** può avere *durable subscription* a un ***determinato named topic*** ☹️ in un determinato istante



Transazioni JMS

- ❑ **Lo scope delle transazioni in JMS è SOLO fra clienti e sistema di messaging**, non fra produttori e consumatori
 - Un gruppo di messaggi all'interno di una singola transazione è consegnato come una unica unità (*lato produttore*)
 - Un gruppo di messaggi in una transazione è ricevuto come una unica unità (*lato consumatore*)
- ❑ Transazioni "locali"
 - **Controllate dall'oggetto Session**
 - Transazione **comincia implicitamente quando l'oggetto di sessione è creato**
 - Transazione termina all'invocazione di `Session.commit()` o `Session.abort()`
 - La sessione è transazionale se si specifica il **flag appropriato** all'atto della creazione. Ad esempio:
`QueueConnection.createQueueSession(true, ..)`



Transazioni Distribuite in JMS

□ Transazioni “distribuite”

- Devono essere **coordinate da un transactional manager esterno** come Java Transactions API (JTA)
- Applicazioni possono controllare la transazione attraverso metodi JTA
 - Utilizzo di `Session.commit()` e `Session.rollback()` è **non consentito**
- In questo modo, **operazioni di messaging possono essere combinate con transazioni DB** in una singola transazione complessiva



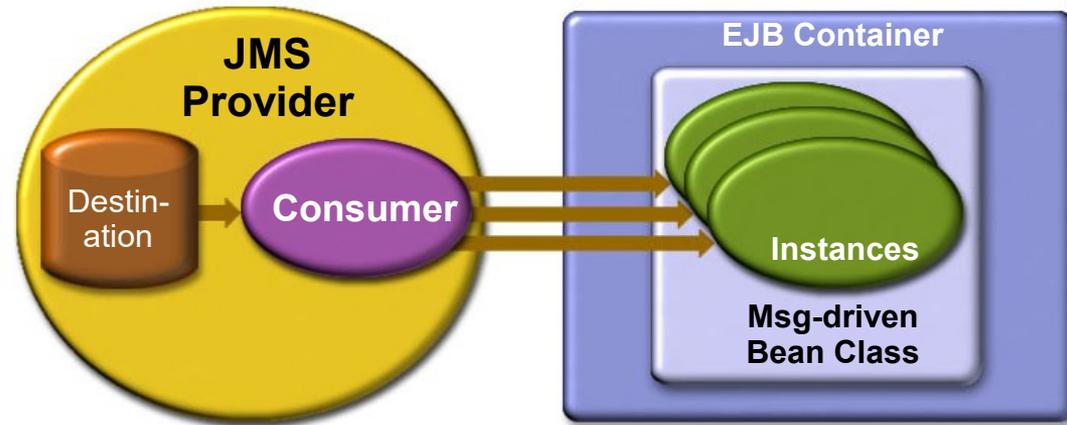
Selettori di Messaggi JMS

- ❑ Lato receiver, le applicazioni JMS possono utilizzare ***selettori per scegliere i soli messaggi che sono potenzialmente di loro interesse***
- ❑ Selettori come stringhe SQL92 che specificano ***regole di filtering***
- ❑ Selettori non possono riferire il contenuto dei messaggi, ma solo proprietà e header

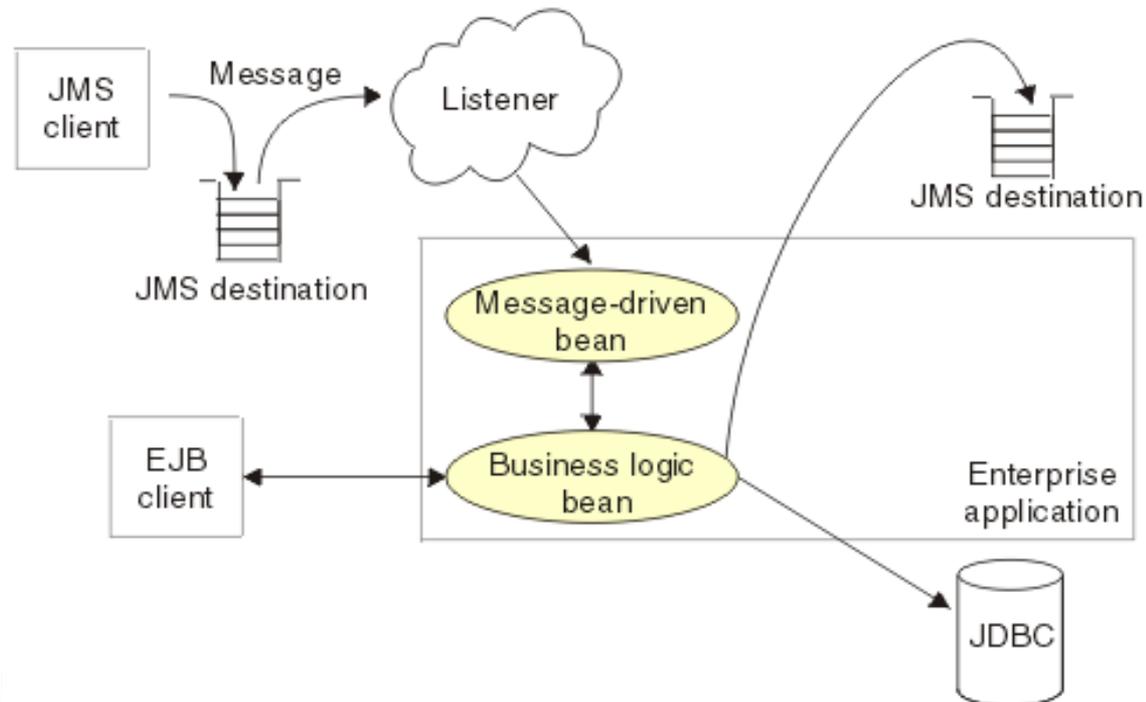


JMS e Message Driven Bean

MDB *istanziato in modo asincrono (o prelevato dal pool di istanze)*
quando un messaggio è ricevuto



Logica di business nel metodo di ricezione del messaggio, ad esempio per modificare entity bean o produrre una email di conferma





Generalizzando “in modo ardito”: Integrazione tramite ESB

- ❑ Problemi nel campo dell’integrazione
 - diversi ambienti di esecuzione, di management, ...
 - sistemi e servizi proprietari differenti, anche legacy
 - confini fisici

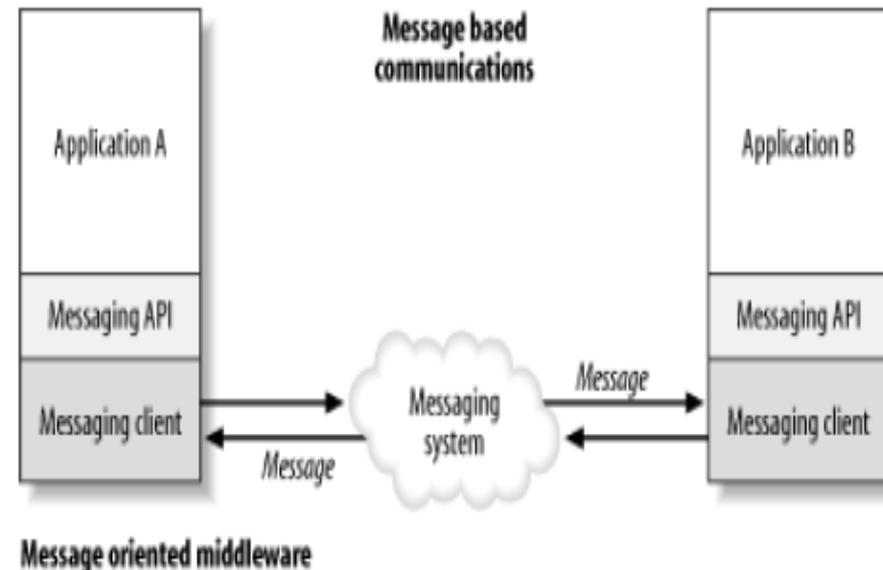
- ❑ *“A new form of **enterprise service bus (ESB) infrastructure – combining message-oriented middleware, Web services, transformation and routing intelligence** - will be running in the majority of enterprises by 2005.”* [Roy Schulte, Vice President of Gartner Inc., 2002]

- ❑ *ESB come infrastruttura software per l’integrazione, basata su “standard”, che combina messaging, Web services, data transformation e routing intelligence per connettere tra loro in modo **debolmente accoppiato** e affidabile un numero significativo di applicazioni eterogenee, **mappate come servizi***
[David Chappell, ESB, O’Reilly, 2004]



Message Oriented Middleware ed ESB

- ❑ Infrastruttura per la comunicazione tra applicazioni basata sullo **scambio di messaggi**
 - Modello sincrono vs. **modello asincrono**
 - Modello p2p vs. **pub-sub**
- ❑ Caratteristiche generali
 - **Disaccoppiamento**
 - Gestione dei “topic”
 - Controllo degli accessi
 - Struttura messaggi
 - **QoS configurabile**





Service Oriented Architecture (SOA)

Paradigma basato su:

- ❑ **Servizi autonomi**
- ❑ **Interfacce** che definiscono contratti tra Consumer e Provider
- ❑ **Messaggi** che compongono le operazioni invocabili sui servizi
- ❑ **Registri** dei servizi
- ❑ Possibilità di **comporre i servizi in processi di business**

Obiettivo è ottenere:

- ❑ **Accoppiamento debole**, e quindi...
- ❑ **Flessibilità** di business
- ❑ **Interoperabilità** tra le applicazioni
- ❑ **Indipendenza** rispetto alle tecnologie di implementazione



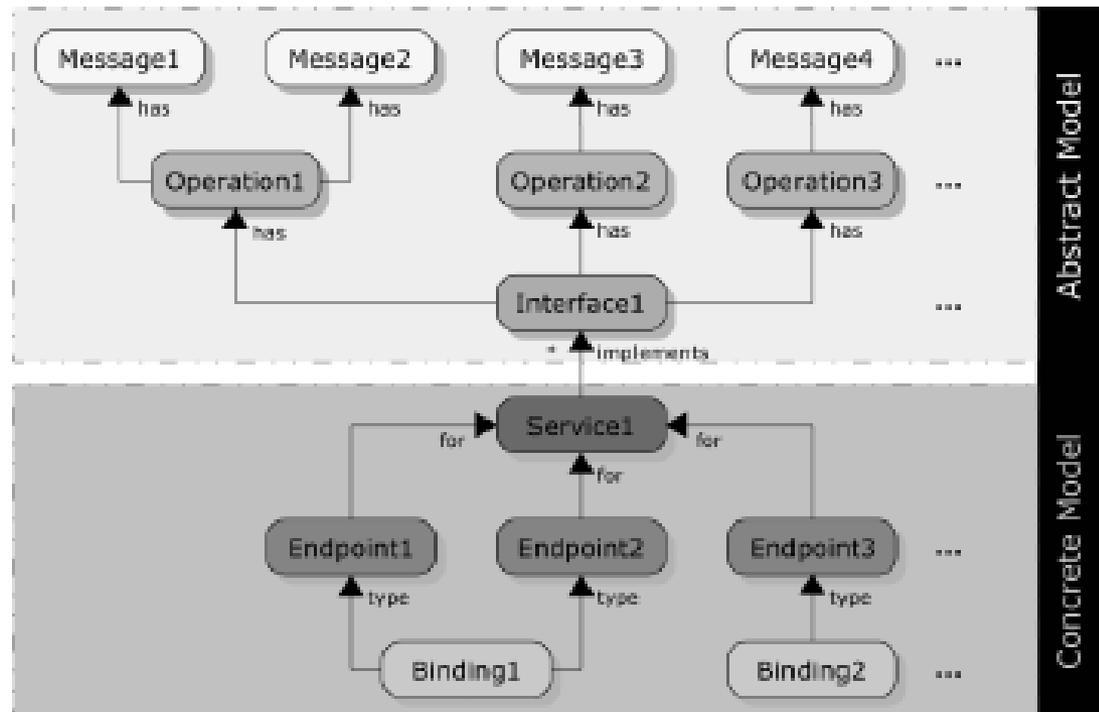
Web Service

- ❑ **Infrastruttura** per l'interazione tra applicazioni basata sul concetto di “**servizio**”
- ❑ Sfrutta **essenzialmente tre tecnologie**:
 - **SOAP** → descrizione messaggi scambiati e binding protocollo di trasporto utilizzato (usualmente HTTP)
 - **WSDL** → descrizione servizio svolto dal provider
 - **UDDI** → discovery di servizi → directory service (pattern “find-bind-invoke”)
- ❑ SOAP e WSDL si basano su XML



Web Service Description Language (WSDL)

WSDL e chiara separazione fra **livello astratto** (definizione operazioni di servizio e struttura messaggi) e **livello concreto** (binding – per ogni interfaccia, uno o più endpoint con indirizzo di rete e protocollo), tipico di tutte le soluzioni SOA





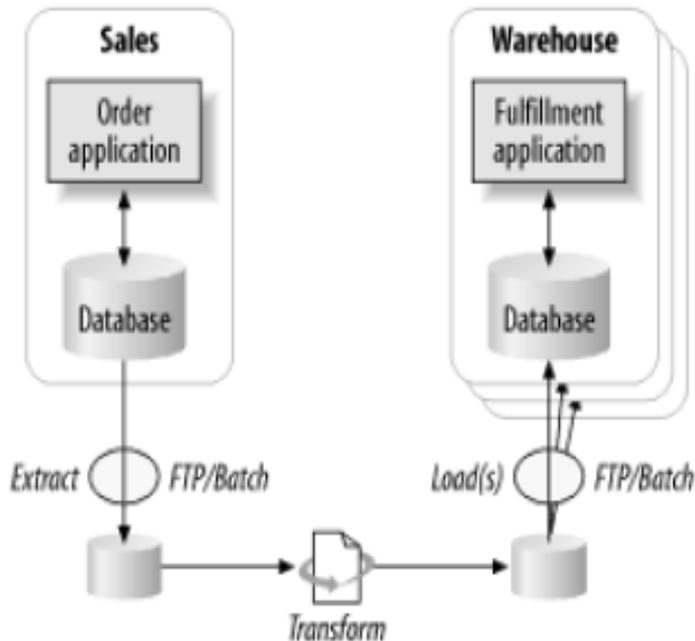
Approccio Convenzionale all'Integrazione

- ❑ Solo **10% delle applicazioni è integrato** (dati Gartner Inc.) e solo 15% di queste sfruttano middleware ad hoc...
- ❑ Com'è collegato il restante 85%? Perché le tecnologie passate si sono rivelate inadeguate?

Architettura “casuale”

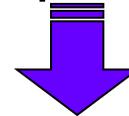
- ❑ È il risultato della composizione di diverse soluzioni adottate per i diversi sistemi nel corso degli anni
- ❑ Col tempo presenta:
 - alti costi di mantenimento
 - **rigidità** (applicazioni tightly-coupled)
 - prestazioni insoddisfacenti (**scarsa scalabilità**)

Enterprise Application Integration (EAI)



Approccio *Extract, Transform, and Load*

- ❑ Download e upload continui

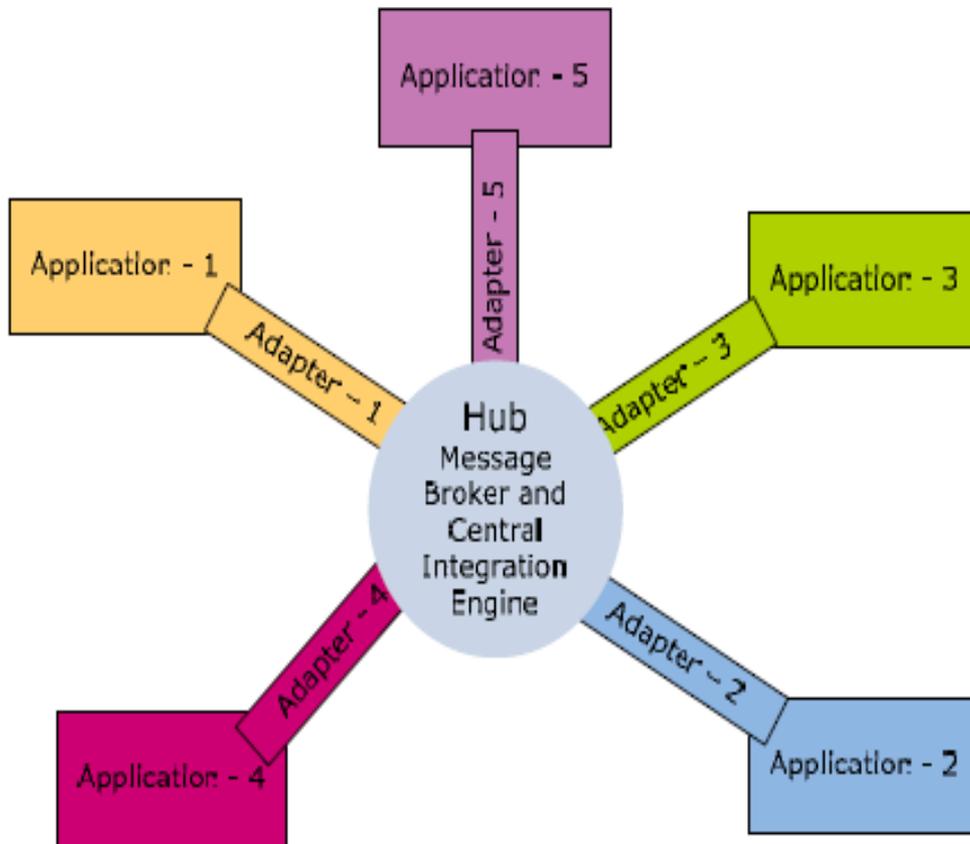


rischio di introdurre incoerenze

- ❑ Alta latenza del processo

- ❑ Applicazione *principi architetturali* allo scopo di *integrare efficacemente* applicazioni di un'organizzazione
- ❑ **Broker + orchestration engine**
- ❑ Due topologie principali: **hub-and-spoke o bus**
- ❑ Implementazioni generalmente proprietarie e alto costo

EAI: Hub-and-Spoke

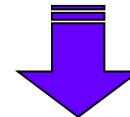


□ PRO:

- facilità di gestione (centralizzata)

□ CONTRO:

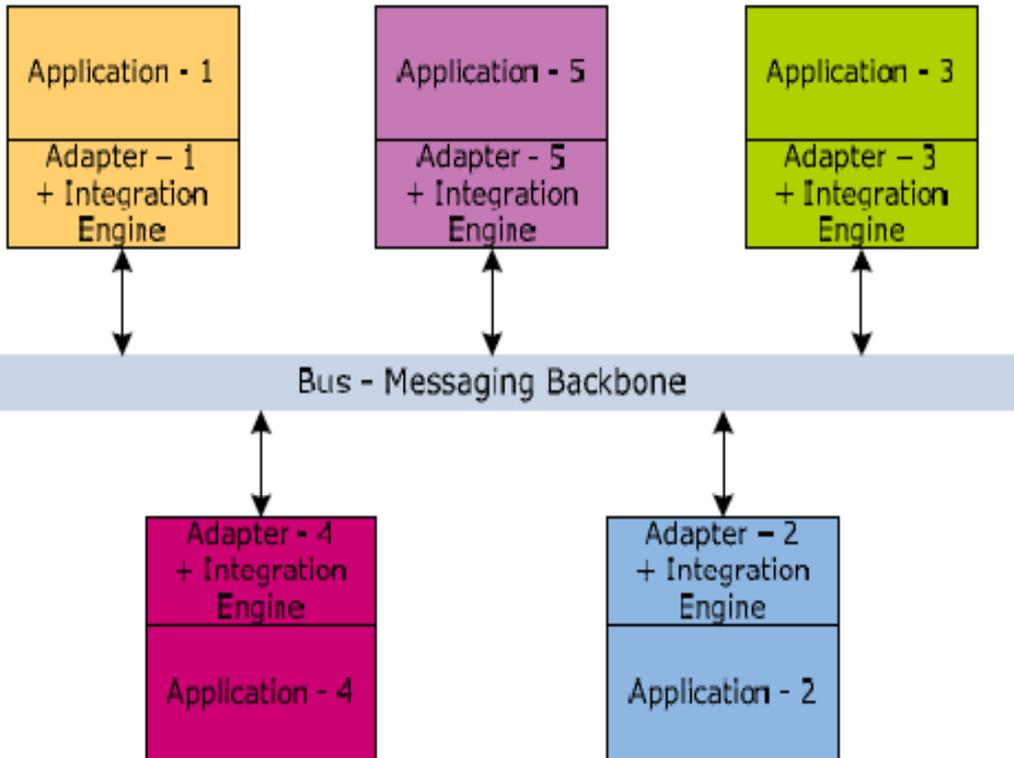
- hub punto critico di centralizzazione
- ridotta scalabilità



architettura federata



EAI: Bus di Interconnessione



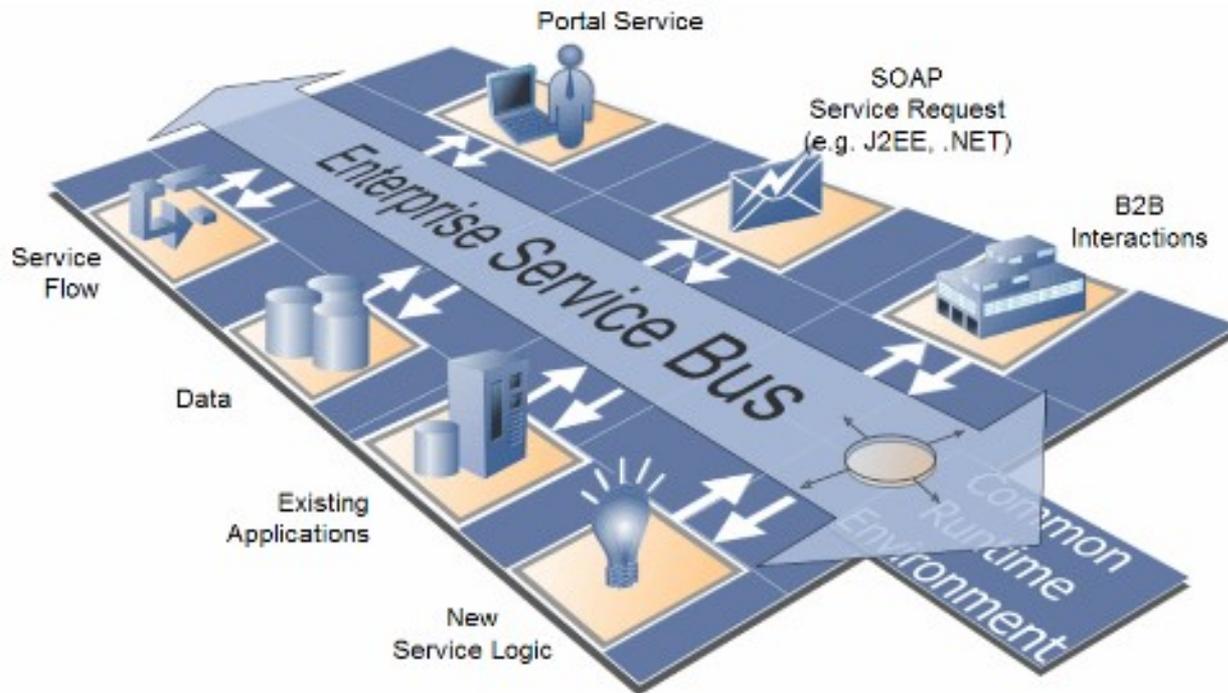
□ PRO:

- maggiore scalabilità (architettura meno centralizzata)

□ CONTRO:

- a costo di maggiori difficoltà di gestione

Enterprise Service Bus



Middleware per l'integrazione di servizi basato sul paradigma SOA

Caratteristiche:

- ❑ ***Uniformità*** nell'accesso ai servizi
- ❑ Capacità di ***orchestrarne l'integrazione*** mediandone le incompatibilità
- ❑ Funge da ***registro dei servizi***
- ❑ Agisce come ***punto centralizzato di gestione***

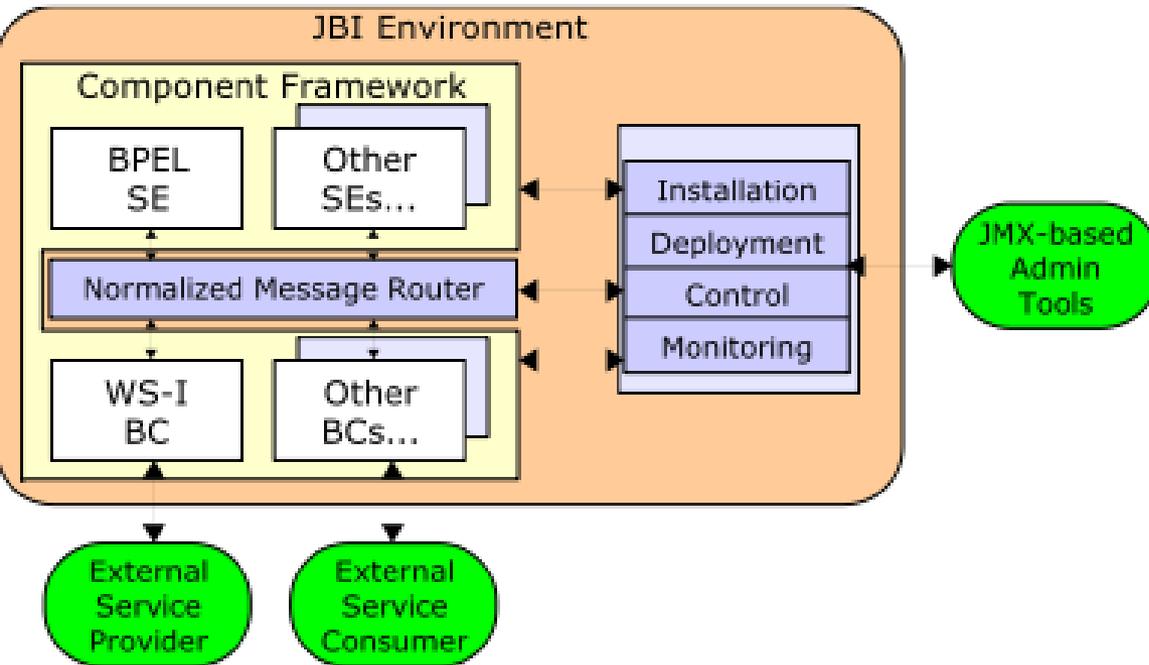


ESB: Concetti Chiave

- ❑ Architettura altamente distribuita e integrazione basata su standard
- ❑ **Servizi di orchestration**
- ❑ **Autonomia** delle singole applicazioni
- ❑ Real-time throughput; servizi di auditing e logging
- ❑ Consente adozione incrementale

Invocazione dei servizi:

- ❑ Servizi completamente disaccoppiati
- ❑ Pattern “find-bind-invoke” è gestito automaticamente dall’infrastruttura
- ❑ Progettista deve solo definire **itinerario logico che i messaggi devono seguire**; servizi si “limitano” a inviare e ricevere messaggi...



- ❑ **JSR 208**, Java Business Integration (JBI), 2005 (specifica Java di ESB Standard)
- ❑ **Servizi offerti e consumati da componenti**
- ❑ Interazione tra componenti mediata da **Normalized Message Router (NMR)**
- ❑ **Gestione** attraverso strumenti **JMX-compliant**

Tipologie di componenti:

- ❑ **Service Engine** – responsabili della logica di business, offrono servizi implementati in Java; forniscono **logica di integrazione e di trasformazione verso altri componenti**; a loro volta possono utilizzare i servizi degli altri SE
- ❑ **Binding Component** – **consentono fruizione di servizi esterni** all'environment da parte di servizi interni e viceversa. Fungono da **adattatori di protocollo**



Normalized Message Router (NMR)

Comunicazione tra componenti all'interno del bus **NON** è **diretta**. NMR che agisce da mediatore fra i vari componenti

❑ **Compito dell'NMR**

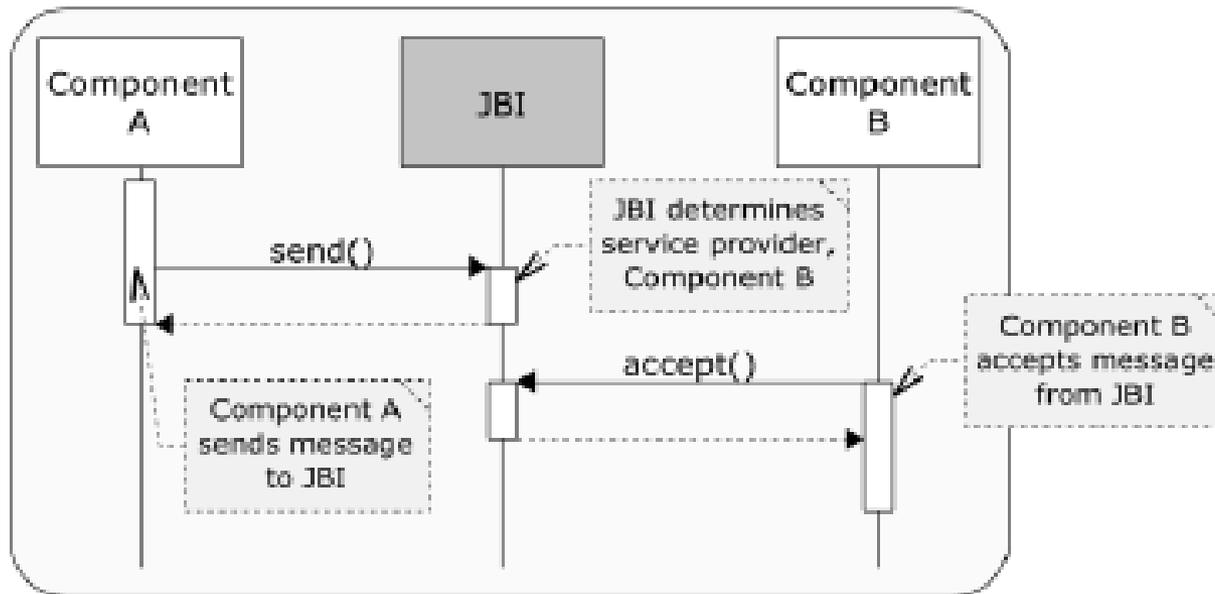
- Routing dei messaggi tra 2 o più componenti
- Disaccoppiare Service Consumer da Service Provider garantendo un basso accoppiamento tra i componenti JBI

Messaggi in formato XML

- ❑ **Comunicazione "technology-neutral" tra endpoint.** Normalized message scambiati sono definiti in formato indipendente e neutrale da qualsiasi specifica applicazione, tecnologia o protocollo di comunicazione
- ❑ Trasformazioni di formato → **trasformazioni XSLT**



Interposizione di JBI nello Scambio di Messaggi



Componenti SOA e modello a scambio di messaggi basato su interposizione:

- Elevato grado di disaccoppiamento tra componenti
- Possibilità di operare su messaggi (trasformazioni) in modo trasparente



JBI Message Exchange Pattern

JBI supporta almeno **4 pattern di scambio messaggi**:

- ❑ **In-Only** per interazione one-way
- ❑ **Robust In-Only** per possibilità di segnalare fault a livello applicativo
- ❑ **In-Out** per interazione request-response con possibilità fault lato provider
- ❑ **In Optional-Out** per provider con risposta opzionale e possibilità di segnalare fault da provider/consumer

