Sommario

Obiettivo. Da inserire.
Conoscenze pregresse. Da inserire *background*.
Metodo. Da inserire.
Risultati. Da inserire e limiti.
Conclusioni e sviluppo futuri. Da inserire.
Applicazioni. Attuate/possibili da inserire.
Destinatari. Studenti di Ingegneria Informatica Magistrale, primo anno, 2° semestre, Università degli studi di Tor Vergata.

Introduzione

Enterprise Service Bus

Un *Enterprise Service Bus*, ESB, è una parte centrale delle architetture SOA; esso si pone al centro di tale architettura fornendo un *middleware* di comunicazione intelligente tra applicazioni anche eterogenee, evitando la connessione di tipo *point-to-point* che tanti problemi crea nel momento della manutenzione e della modiche di parti delle applicazioni. In effetti, se non si utilizzasse alcun supporto all'integrazione, l'unica soluzione per collegare un insieme di funzionalità eterogenee sarebbe, appunto, l'utilizzo dell'architettura point to point; in tal caso la complessità e il costo di mantenimento aumenterebbero ad ogni aggiunta o eliminazione di applicazioni. Viceversa, anche a parità di architettura, l'impiego di un ESB ne semplificherebbe la gestione, incrementandone la flessibilità, e ne faciliterebbe aggiornamenti e modiche, così anche diminuendone il tempo di entrata sul mercato dei sistemi.
I vantaggi connessi all'impiego di ESB risaltano ancora di più in ambienti eterogenei, quando sono utilizzati, magari in gran numero, protocolli diversi; in tal caso, l'ESB maschera le eterogeneità attraverso l'impiego della opportuna serie di "adapter".

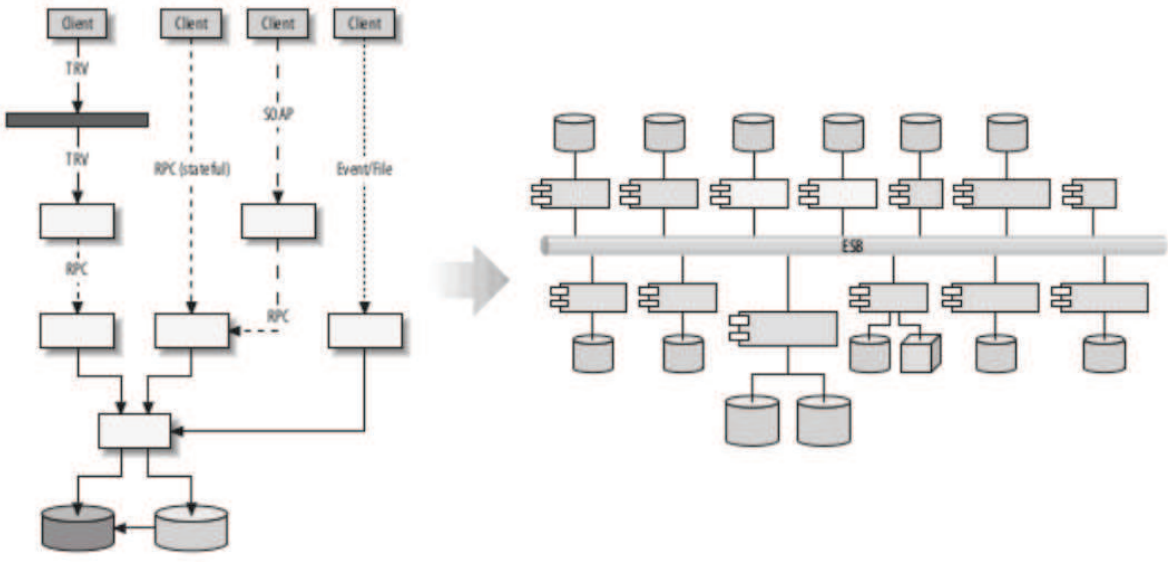

Figura 1: ……

Si possono dunque riassumere i vantaggi dell'utilizzo di un ESB nei seguenti punti:

- maggiore flessibilità per integrare applicazioni eterogenee (cambiamento di requisiti);
- maggiore portabilità;

- implementazione più distribuita, anche geograficamente, rispetto a quella prevista per un classico *broker*;
- *patching* incrementale con assenza di *downtime*.

Il costo di tutto ciò è quello che si ha ogni qual volta si utilizza l'indirezione: *overhead* e conseguente decremento della velocità di comunicazione.

Le funzionalità di un ESB possono essere riassunte nel seguente elenco:

- *location trasparency*: l'ESB disaccoppia il cliente del servizio dal fornitore ottenendo di conseguenza la trasparenza alla locazione;
- *transport protocol conversion*: capacità di accettare un tipo di protocollo nei confronti del client (SOAP, HTTP ...) e comunicare al service provider attraverso un protocollo diverso;
- *message trasformation*: capacità di convertire la struttura e il formato del *payload* dalla richiesta effettuata dal *client* nel formato effettivamente gestibile dal *service provider*;
- *message routing*: capacità di smistare una richiesta verso un particolare service provider utilizzando criteri deterministici o probabilistici (*content based*, *policy based* ...);
- *message enhancement*: l'ESB può dover incrementare, modificare o cancellare informazioni contenute in un messaggio prima che questo sia compatibile con il destinatario. Da notare che, in questo caso, si modica il payload, spesso andando a prendere informazioni da database; nel *message trasformation*, invece, ciò che viene cambiato è l'*header*;
- *security*: capacità di proteggere i servizi dei service provider da accessi non autorizzati;
- *monitoring and management*: caratteristiche necessarie per impostare l'ESB perché garantisca alte prestazioni e per monitorare il tutto;
- *message processing*: capacità di gestire delle code che possano essere interposte tra client e applicazione che il client sta usando (si assicura il *delivery* del messaggio di risposta al client);
- *service orchestration*: capacità di gestire processi di business complessi che richiedono la coordinazione di diversi servizi di business per portare a termine una singola richiesta (qui il modello dei servizi è inteso coordinato centralmente; se invece i servizi si coordinassero direttamente fra loro si preferirebbe parlare di *service coreography*);
- *transaction management*: capacità di trattare una richiesta ad un servizio di business come se fosse una singola unità di lavoro, un'entità atomica; sono possibili meccanismi di compensazione.

Ovviamente non esiste un prodotto monolitico capace di offrire tutte le funzionalità descritte: un ESB è principalmente composto da 4 componenti:

- *orchestrator*: si occupa di tutta la logica necessaria a realizzare il processo di business; si occupa dell'orchestrazione dei servizi;

- *rules engine*: gestisce tutte le regole, applicate nelle funzionalità di routing, nelle funzioni di trasformazione dei messaggi e nella funzionalità di arricchimento del messaggio;
- *service registry*: svolge la funzione fondamentale di eseguire il *mapping* dei servizi, si occupa della trasparenza alla locazione;
- *mediator*: si occupa delle altre funzionalità: *routing*, trasformazione dei messaggi, arricchimento, trasformazione del protocollo, sicurezza e gestione delle transazioni.

Spring Integration

Dopo un'analisi approfondita sugli ESB *open-source* attualmente presenti sul mercato e dopo aver individuato i punti di forza e i problemi di ciascuno, con riferimento a un impiego in corsi universitari di Master, la scelta fatta è stata quella di utilizzare Spring integration, la soluzione ESB del noto *Framework* Spring. Tale scelta è stata motivata sia per il supporto completo che viene dato a tutti gli altri moduli di Spring, sia per la semplicità e leggerezza del framework stesso. Spring integration permette di sviluppare la propria soluzione direttamente sull'applicazione senza la necessità di altri contenitori. Spring integration si basa sui contenuti del libro *Enterprise Integration Patterns* [24]; ciò ha permesso di utilizzare *pattern* noti che hanno ottimizzato l'intera struttura.

Spring integration mette a disposizioni moltissimi tipi di *end-point*; quelli utilizzati in ERMES-QIP sono:

Service activator

Invoca un metodo tramite un *bean* ogni qualvolta un messaggio arriva sul canale di input. Se il metodo ha un valore di ritorno, allora tale valore sarà inviato al canale di output; esempio di configurazione:

```
<int:service-activator
input-channel="positions-channel"
ref="newTradeActivator"
method="processNewPosition">
</int:service-activator>
<bean id="newTradeActivator
"class="com.endpoints.common.NewTradeActivator" />
```

Message enricher

Si arricchisce il messaggio in arrivo con informazioni addizionali, si invia poi l'oggetto aggiornato ai *downstream consumer*. Esistono due tipologie di *enricher*, l'*header enricher* e il *payload enricher*; la configurazione del primo è riportata di seguito.

```
<int:header-enricher input-channel="in" output-channel="out">
    <int:header name="foo" value="123"/>
    <int:header name="bar" ref="someBean"/>
</int:header-enricher>
```

In questo caso si procede alla modica di parti dell'header del messaggio; tali modiche vengono indicate nella configurazione indicando la parte dell'header da cambiare e poi come va cambiata; ovviamente sono presenti: un input channel che indica l'entrata all'enricher per i messaggi da modificare, e un output channel verso cui vanno i messaggi appena modificati.

```
<int:enricher id="findUserEnricher"
              input-channel="findUserEnricherChannel"
              request-channel="findUserServiceChannel">
    <int:property name="email"    expression="payload.email"/>
    <int:property name="password" expression="payload.password"/>
</int:enricher>
```

Nella configurazione riportata qui sopra, invece, si utilizza un payload enricher; tale componente si occupa di modificare il contenuto del messaggio, indicando la *property* da cambiare e il nuovo valore che deve essere inserito.

Gateway

Un tale tipo di end-point risulta fondamentale per una comunicazione (tra un client e un server, per esempio) in cui non sia necessario conoscere il sistema di *messaging*. Quando si usa un gateway non devono essere usate le componenti di messaggistica, verrà utilizzata semplicemente una interfaccia che esporrà le funzionalità.
Esistono due differenti tipi di gateway, quelli sincroni e quelli asincroni; nel primo caso la chiamata del messaggio sarà bloccata fino a quando il processo non è completato.

```
<int:gateway id="tradeGateway"
default-request-channel="trades-in-channel"
default-reply-channel="trades-out-channel"
service-interface="com.endpoints.gateway.ITradeGateway" />
```

L'interfaccia in questo caso è:

```
public interface ITradeGateway {
public Trade processTrade(Trade t);
}
```

Nell'esempio appena visto il client rimarrà bloccato fino a quando non riceverà una risposta. Se si vuole al contrario che il client non abbia questo comportamento si deve optare per un gateway asincrono. Per far ciò basta eseguire un cambiamento per ciò che riguarda il tipo dell'oggetto di ritorno:

```
import java.util.concurrent.Future;
public interface ITradeGatewayAsync {
public Future<Trade> processTrade(Trade t);
}
```

Per la maggior parte dei gateway offerti da Spring Integration (supporto ad HTTP, JMS,
JDBC ecc.) esiste un'ulteriore differenziazione: gateway *inbound* e *outbound*: se una richiesta in ingresso deve essere servita in *multi-threading* e si desidera che l'invocante rimanga all'oscuro del sistema di messaggistica, un inbound-gateway fornisce la soluzione. Un outbound-gateway invece fa in modo che un messaggio in arrivo possa essere utilizzato in una chiamata sincrona e che il risultato venga inviato sul canale di risposta.
Di seguito un esempio di Outbound Gateway:

```
<int-jms:outbound-gateway request-channel="toJMS"
reply-channel="jmsReplies"
request-destination-name="examples.gateway.queue"/>
```

Qualsiasi messaggio venga inviato al *request channel* sarà convertito in un messaggio del tipo indicato nel gateway (nel caso appena visto in un messaggio

Java Message Service, JMS) ed inviato alla *request-destination* del gateway. Il *reply channel* è il canale in cui tutti i messaggi di risposta vengono inviati dopo essere stati convertiti.
Si passa ora ad analizzare la variante inbound:

```
1 <int-jms:inbound-gateway id="exampleGateway"
  request-destination-name="someQueue"
3 request-channel="requestChannel"/>
```

Tale gateway rimane in ascolto di messaggi (nell'esempio JMS), mappa ogni messaggio ricevuto in un messaggio di Spring Integration, e invia quest'ultimo messaggio verso un *message channel*. Tutto ciò è quasi identico al lavoro svolto da un altro end-point di Spring integration: lo i*nbound-adapter*; la differenza è che, in questo caso, il gateway attende che dal downstream torni una risposta.
Quando un messaggio di risposta viene restituito all'inbound gateway, questo viene immagazzinato in un messaggio (nell'esempio di tipo JMS). Il gateway invia quindi il messaggio alla *reply destination*. In questo progetto sono stati utilizzati due differenti tipologie di Gateway:

- Http-gateway: gateway creati per supportare richieste HTTP; la variante inbound gestisce le richieste HTTP e permette di rispondere a queste ultime con un reply message.
- l'outbound-HTTP-gateway invece permette l'esecuzione di un richiesta HTTP; JDBC-gateway: fornisce supporto per l'invio e la ricezione di messaggi in formato di *query* nei confronti di un database.

Transformer
Non tutte le applicazioni sono in grado di intendere i dati che stanno ricevendo; a volte, per attendere alle richieste di business, i messaggi devono essere trasformati prima di essere consumati. Per esempio, un *producer* usa come payload del messaggio che invia un oggetto Java e il *consumer* è interessato a un payload in JSON. Il *transformer* si occupa esattamente di tale operazione di cambiamento.

```
1 <transformer id="testTransformer" ref="testTransformerBean"
  input-channel="inChannel"
              method="transform" output-channel="outChannel"/>
3 <beans:bean id="testTransformerBean" class="org.foo.TestTransformer" />
```

Nella precedente definizione di un transformer, in maniera molto semplice sono indicati i *channel* di ingresso e di uscita, il metodo in riferimento al quale viene eseguita la trasformazione e la classe associata a quel determinato transformer.

Filter

I consumer hanno diversi requisiti, come per esempio esigere determinati tipi di messaggi. Spring Integration Framework usa i *filter* per decidere se le applicazioni dovrebbero ricevere o meno determinati messaggi in arrivo.

```
1 <filter input-channel="input" ref="selector" output-channel="output"
  method="accept"/>

3 <bean id="selector" class="example.MessageSelectorImpl"/>
```

In XML si specificano i channel di input e output, la classe di riferimento e, di quest'ultima, il metodo boolean che permetterà di eseguire o meno la scelta. La classe riferita deve implementare l'interfaccia seguente:

```
1 public interface MessageSelector {

3     boolean accept(Message<?> message);

5 }
```

Router

Uno dei requisiti per un flusso è inviare messaggi a uno o più channel basandosi su criteri prestabiliti. Un router può essere usato per distribuire messaggi a destinazioni multiple. C'è una differenza sostanziale tra *router* e filter: mentre un filter decide solo se eseguire o meno il *forward* di un messaggio verso una destinazione in base ad un semplice test booleano, il router esegue il forward del messaggio ad uno o più canali basandosi sul contenuto.

```
1 <router method="someMethod" input-channel="input3"
  default-output-channel="defaultOutput3">
    <beans:bean class="org.foo.MyCustomRouter"/>
3 </router>
```

Nel riportato XML si notano, oltre al canale di input, un canale di default di uscita e il riferimento ad un bean. Nella classe riferita al bean sarà presente la logica tramite cui il router eseguirà le scelte.

Nel seguente esempio di tale classe si può notare che, nel metodo *select*, viene inserita una determinata logica in base alla quale si restituisce una stringa che identica il nome del corretto channel di uscita.

```
1 public class ERMESRouter {
    final static Logger logger = LoggerFactory
3       .getLogger(ERMESRouter.class);
    public String select(Message<Request> message) {
5     Request t = message.getPayload();
      if (t.getOriginAdress().equals(""))
7       return "updateChannel";
...
```

Sono molti i pattern interessanti che vengono utilizzati per la creazione di ERMES-QIP. Primo tra tutti il pattern che permette quanto segue:

- interrogare il service registry di ERMES-QIP per trovare una risposta alla domanda contenuta nel messaggio in arrivo;
- inserire la risposta nel messaggio stesso.

Di seguito un esempio di tale pattern preso dal livello 3 di ERMES-QIP:

```
<int:channel id="replyEnricherChannel" />

<int:chain input-channel="jDoraChannel" output-channel="outputLevel3Channel">
  <int:enricher request-channel="replyEnricherChannel">

    <int:property name="resolvedAdress" expression="payload.resolvedAdress" />
    <int:property name="tag" expression="payload.tag" />
  </int:enricher>

  <int:service-activator ref="jDoraService"
    method="receive" />

</int:chain>

<bean id="jDoraService" class="it.service.JDoraService" />

<int-jdbc:outbound-gateway data-source="dataSource"
  request-channel="replyEnricherChannel"
  query="select tag,resolvedAdress from t where content = :payload.content"
   reply-channel="afterDBChannel" max-rows-per-poll="1"
   row-mapper="requestMapper">
 </int-jdbc:outbound-gateway>

 <int:service-activator input-channel="afterDBChannel"
   ref="enricherService" method="receive" />

 <bean id="enricherService" class="it.enricher.EnricherService" />
```

Innanzitutto si crea una *chain*, cioè una struttura all'interno della quale tutte le componenti vengono automaticamente e ordinatamente collegate con dei channel. Una chain richiede solo di indicare il channel di input e quello di output, tale componente permette di limitare la creazione di channel quando parti di configurazione sono sequenziali. Nella chain è presente un *Enricher* e un service *Activator*: il messaggio arriva nello Enricher, viene modificato passando dal channel r*eplyEnricherChannel*, entra nel service activator e attraverso questo giunge al channel di output. La parte di arricchimento avviene tramite la componente presente all'altro estremo del replyEnricherChannel: un jdbc outbound-gateway che esegue una query al service registry di ERMES-QIP chiedendo di accedere alla riga avente per *content* la voce content del payload del messaggio.

La componente chiede *tag* e indirizzo risolto di quella riga e ne inserisce i valori nel messaggio, in particolare negli attributi "tag" e "*resolved address*", rispettivamente . tutte le dinamiche appena descritte:

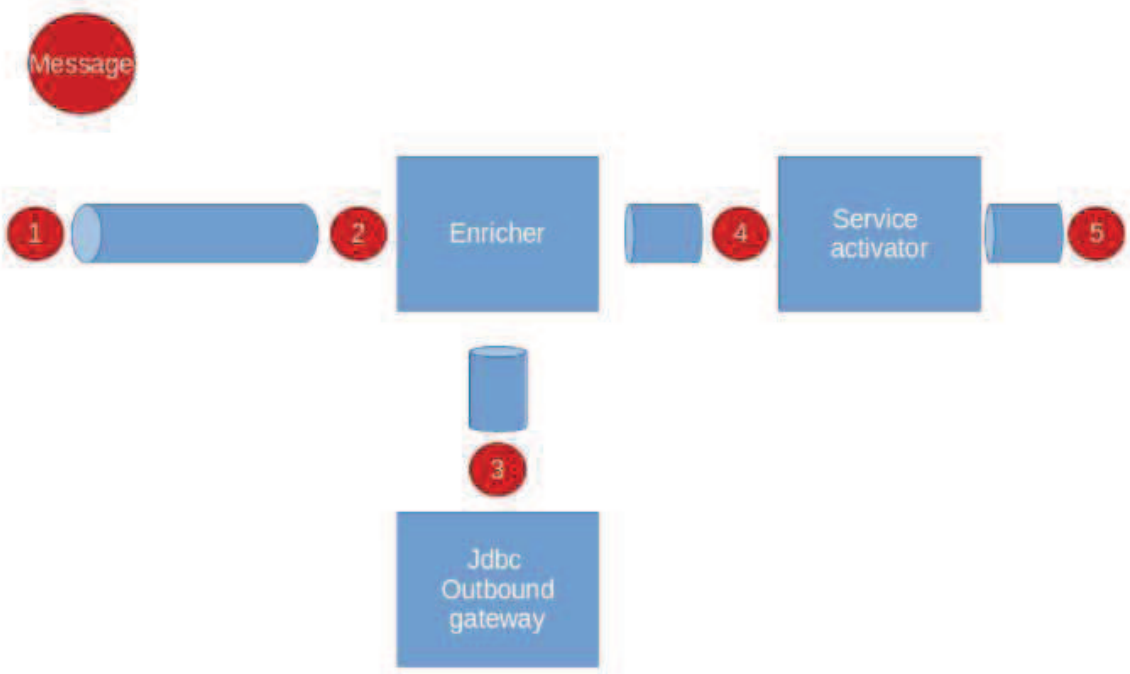

Figura 2. ….

Architettura generale del sistema

L'architettura pensata per il sistema comprende un insieme di funzionalità capaci di dare un supporto completo a sistemi di *decision-making* (GQM+ inizialmente). Nell'esempio di applicazione con GQM+, esiste un servizio web che gestisce ciascuna fase di tale approccio.
Ci sono poi ulteriori servizi web con ruoli specifici, quali creazione di grafi, elaborazione dei dati, immagazzinamento di grandi quantità di questi ultimi, ecc. Tali servizi saranno denominati di " secondo livello" per differenziarli da quelli creati per il supporto diretto degli approcci di decision making.
Tale struttura modulare e dinamica raccoglie tutti i vantaggi di un approccio SOA e vede in ESB ERMES-QIP l'unità centrale che si occupa del coordinamento e della gestione del tutto.

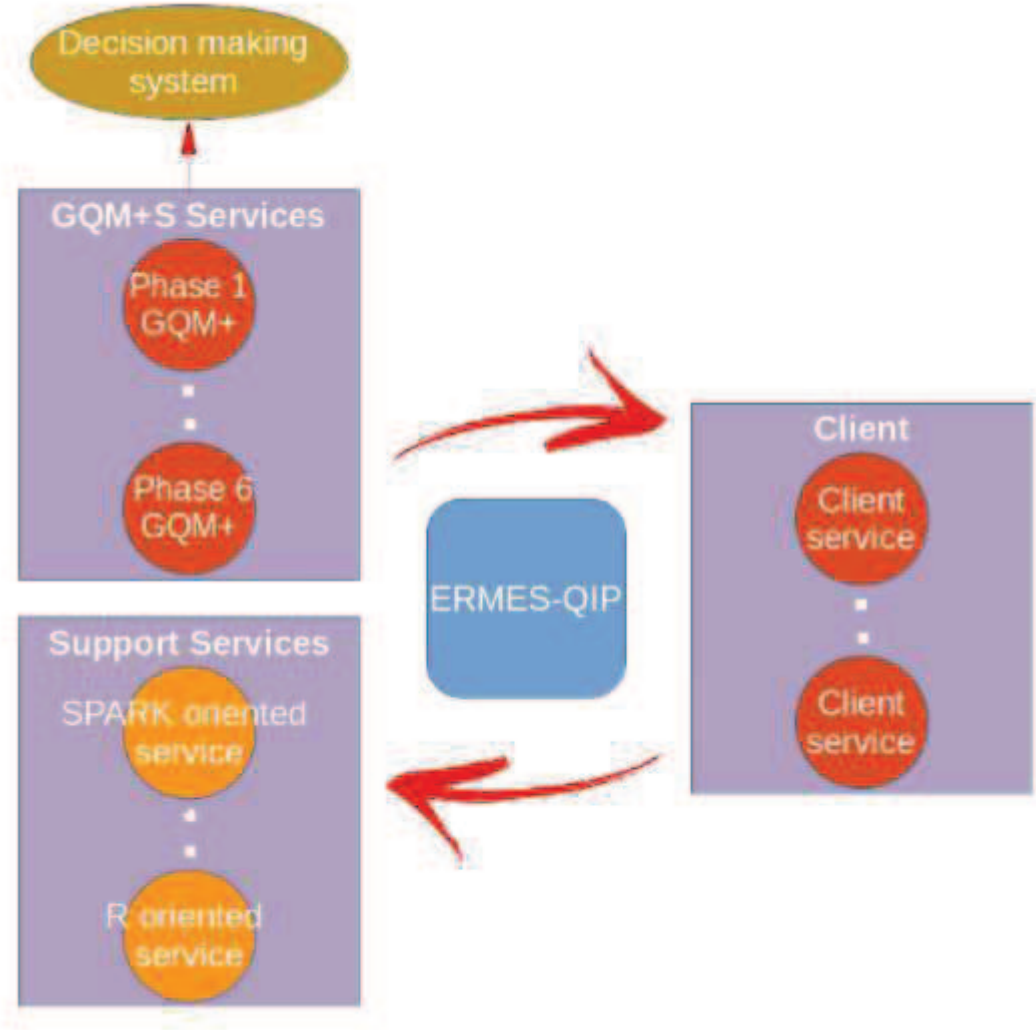

Figura 3. ...

Struttura di ERMES-QIP

ERMES-QIP si articola in tre diversi livelli:

- Livello 1: ha il maggior grado di astrazione, il ruolo di mittente è assegnato al client e quello di destinatario a un servizio; il client, senza dover minimamente conoscere la posizione e tanto meno la sintassi di comunicazione di un determinato servizio, utilizza quest'ultimo servendosi del middleware ERMES-QIP;
- livello 2 : in questo caso, mittente e destinatario sono servizi web: un servizio potrebbe dover utilizzarne un altro; si ipotizzi, per esempio, che un servizio abbia la necessità di eseguire dei calcoli matematici o voglia ottenere un grafico. Anche in questo caso ERMES-QIP si pone come middleware tra i due

servizi consentendo di nuovo trasparenza alla locazione e permettendo ai due servizi di utilizzare sintassi anche differenti per la comunicazione;

- livello 3-JDora : la tipologia del dominio (*Measurement-based decision-making system*) richiede che alcuni servizi utilizzino gli stessi oggetti, al fine di leggerli, modificarli o crearli. Una componente che si inserisce in ciascun servizio (JDora) permette ai servizi medesimi di condividere uno spazio delle classi e delle loro istanze. Ciò si può ottenere facendo in modo che ciascun servizio possa conoscere l'*owner* di ciascuna delle classi che esso utilizza; perché queste informazioni siano conosciute da tutti i servizi, siano aggiornate e mantenute in maniera corretta, si utilizza ERMES-QIP. Questo si occupa di mantenere tutte le corrispondenze class - class owner, permettendo il loro aggiornamento e la loro divulgazione ai servizi interessati. Questa soluzione di fatto trasforma l'architettura in un ibrido in cui è presente anche un file system condiviso;
- livello 3-Direct : soluzione alternativa alla precedente, prevede che gli oggetti condivisi siano consegnati al servizio che ne ha bisogno direttamente da ERMES-QIP; l'ESB infatti accoglie la richiesta di un determinato oggetto da parte di un servizio e ha la capacità di andare ad ottenerlo dal servizio che lo possiede. Tale soluzione supporta anche un sistema di *versioning* in modo tale da abilitare modiche parallele anche dello stesso oggetto.

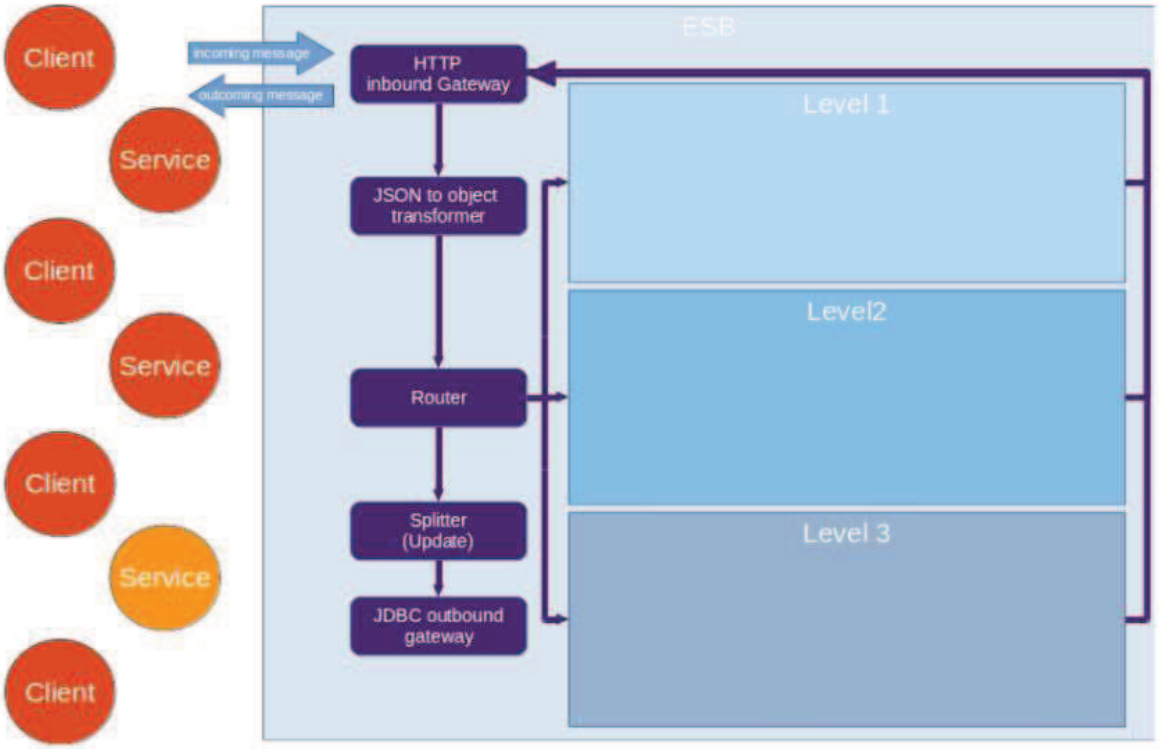

Figura 4. ...

Una parte dell'ESB (come in figura) è comune ai tre livelli: il messaggio HTTP arriva ad un http-inbound-gateway, questo lo riceve e re-inoltra verso il JSON-to-Object-Transformer; questo end-point prende il contenuto JSON del messaggio e lo trasforma in un POJO, in particolare in un'istanza della classe Request. Tale oggetto, molto semplice e basilare, è utilizzato da tutti i fruitori dell'ESB per eseguire richieste.

```
2  public class Request {

4    private String tag;

6    private ArrayList<String> content;

8    private String resolvedAdress;

10   private String originAdress;

12   private UUID id;

14

     public Request(String tag,ArrayList<String> content, String originAdress,
     String resolvedAdress, UUID id) {
16     super();
       this.tag = tag;
18     this.content = content;
       this.originAdress = originAdress;
20     this.resolvedAdress = resolvedAdress;
       this.id= id;
22   }
...
```

Esiste un attributo "tag" in Request che serve proprio a differenziare il tipo di richiesta rispetto ai tre livelli. L'attributo "content" serve per ospitare il contenuto della richiesta, il quale può essere il nome della classe da risolvere nel livello 3, identificatori dei dati e dell'azione da eseguire su di essi a livello 2 o, piuttosto, la richiesta da parte di un client nel livello 1. L'attributo "*originAdress*" serve per ospitare l'indirizzo del mittente della richiesta; l'attributo "*resolvedAdress*" può ospitare l'indirizzo di destinazione del messaggio o l'indirizzo di risposta ad una specifica richiesta fatta all'ESB. Infine, è presente un attributo "ID" per etichettare