Politecnico di Milano

Polo Territoriale di Como



Prova finale di Ingegneria del Software  
Documento di progetto – Diagrammi UML

- Gruppo 1 -

Ghielmetti Nicolò

Quaglia Ennio

Zaffaroni Leonardo

Indice

[UseCase Diagram 3](#_Toc498960422)

[Component Diagram 4](#_Toc498960423)

[Deployment Diagram 5](#_Toc498960424)

[Class Diagram 6](#_Toc498960425)

[Object Diagram 7](#_Toc498960426)

[Communication Diagram 8](#_Toc498960427)

[Activity Diagram – Cancellazione di un servizio 9](#_Toc498960428)

[Activity Diagram – Invocazione di un servizio 10](#_Toc498960429)

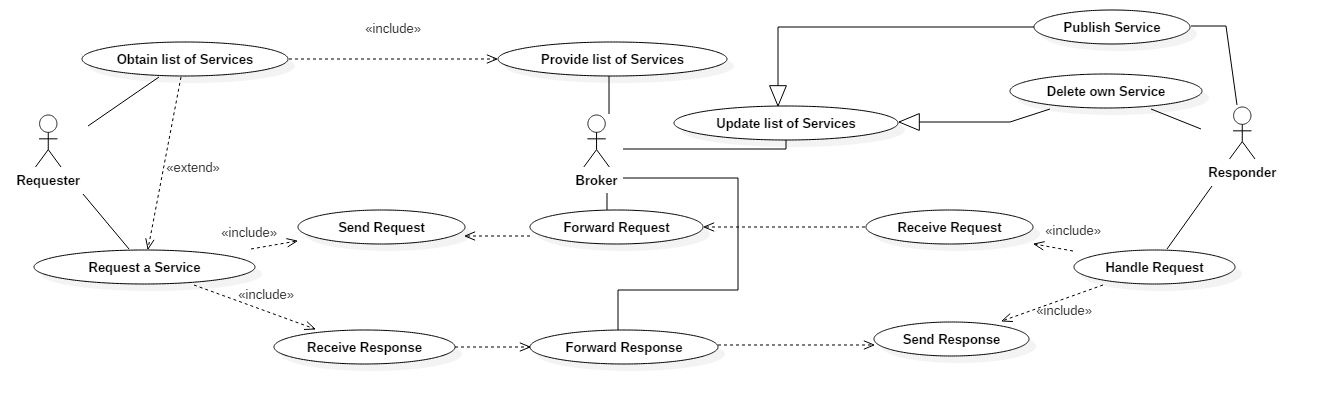
[Sequence Diagram – Richiesta della lista dei servizi 11](#_Toc498960430)

[Sequence Diagram – Invocazione di un servizio 12](#_Toc498960431)

[State Diagram – Broker 13](#_Toc498960432)

[State Diagram – Node 14](#_Toc498960433)

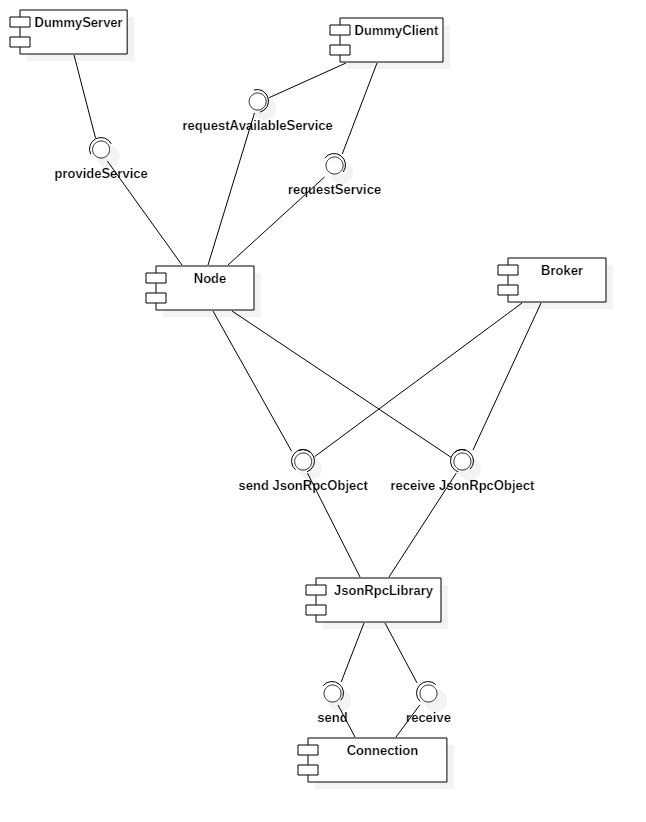
# UseCase Diagram



Il sistema si può rappresentare attraverso tre attori:

* **Requester**: è un utilizzatore di servizi. In particolare può:
  + richiedere una lista dei servizi disponibili al Broker.
  + richiedere l’invocazione di un servizio mediante la comunicazione trasparente realizzata dal Broker.
* **Responder**: è un fornitore di servizi. Ha il compito di:
  + gestire opportunamente le richieste, inviando risposte o errori adeguati.
  + gestire la sua presenza all’interno del sistema, aggiungendo e/o cancellando il proprio servizio.
* **Broker**: si occupa di gestire la comunicazione trasparente fra i vari **Requesters** e **Responders** presenti all’interno del sistema. In particolare si occupa di:
  + inoltrare richieste e risposte agli opportuni destinatari.
  + gestire la lista dei servizi disponibili mantenendola aggiornata.

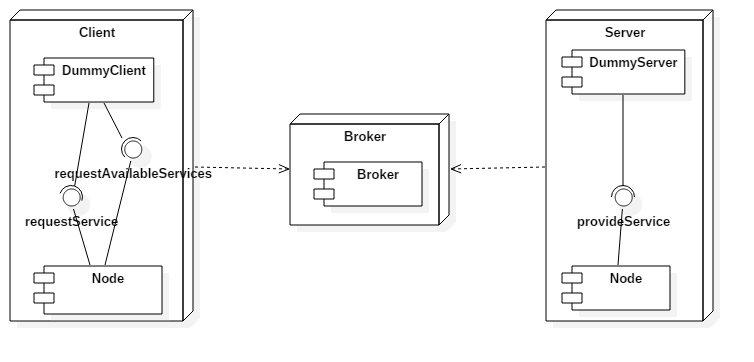
# Component Diagram



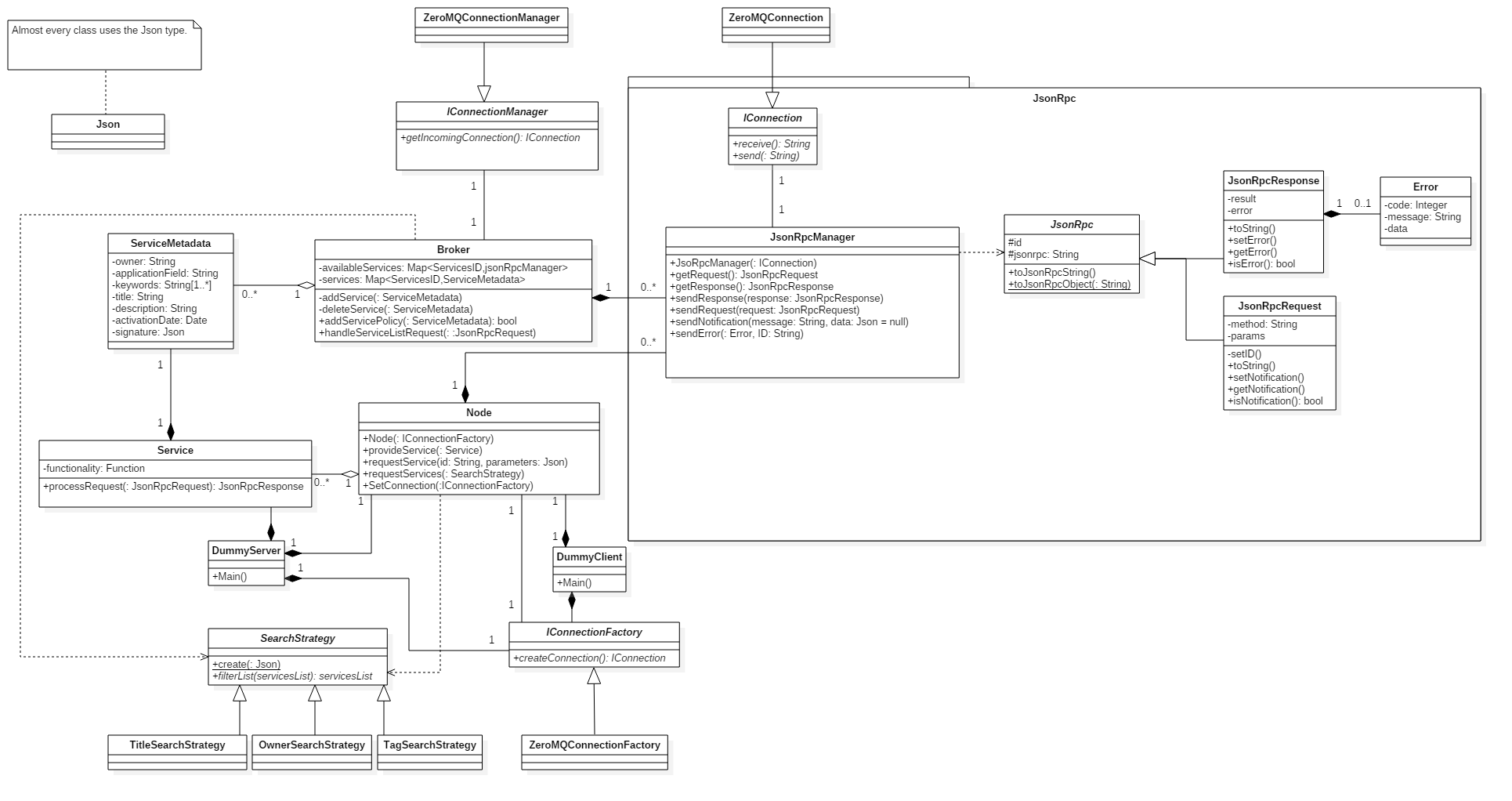
In questo diagramma vengono mostrati i moduli software in cui è diviso il sistema.

In particolare si può notare che il componente **Node** espone sia interfaccia dedicate alla funzione di Client e Server.

# Deployment Diagram



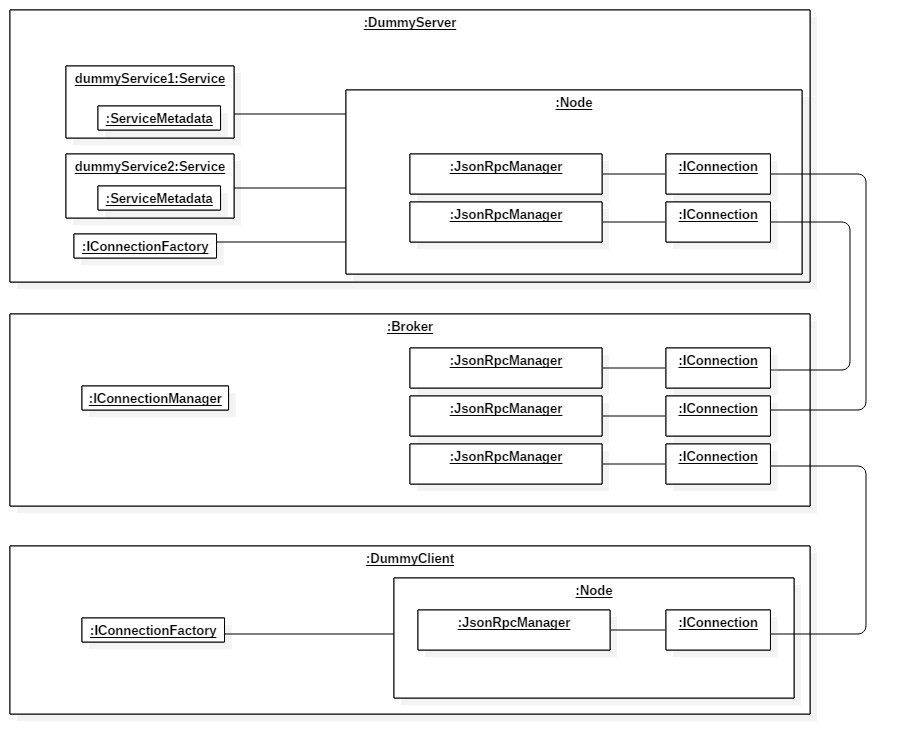
# Class Diagram



Nel class diagram sono rappresentate le classi del tema comune (rappresentata dal package JsonRpc) e del tema B. Di seguito, alcune precisazioni:

* Al fine di ottenere l’applicazione e la libreria JsonRpc agnostiche rispetto al canale trasmissivo sono state previste le seguenti interfacce:
  + **IConnection**: è un’interfaccia che si occupa di gestire una comunicazione già instaurata. La presenza di questa interfaccia è necessaria affinché i metodi di ricezione e invio di risposte e richieste siano da implementare indipendentemente dalla libreria di comunicazione.
  + **IConnectionManager**: è l’interfaccia attraverso la quale il Broker ottiene per ogni connessione in arrivo una **IConnection**.
  + **IConnctionFactory**: è l’interfaccia attraverso la quale il **Node** può instaurare una connessione con il Broker.
* Per poter consentire la ricerca da parte del client, dei servizi disponibili nel sistema, è stata prevista nel modello la classe astratta ***SearchStategy***che permette l’indicazione di criteri di ricerca*.*
* Sebbene nella specifica JSON-RPC gli oggetti JSON-RPC Notification e JSON-RPC Error siano trattati, non abbiamo ritenuto opportuno prevedere le classi JsonRpcNotification e JsonRpcError poiché sono dei casi particolari rispettivamente di **JsonRpcRequest** e **JsonRpcResponse**. Per rispettare comunque la specifica sono stati previsti dei metodi per la realizzazione dei casi particolari sopracitati.
* **JsonRpcManager** è una classe che si occupa dello scambio di messaggi, secondo lo standard JSON-RPC, per una determinata **IConnection**.
* **addServicePolicy** è un metodo astratto della classe Broker la quale concretizzazione consentirà, in funzione dei metadati passati come parametro, di stabilire se il servizio può essere aggiunto nel sistema.
* La cancellazione, la registrazione di un servizio e la richiesta della lista dei servizi disponibili nel sistema sono servizi predefiniti presenti all’interno del Broker.
* Le classi **DummyClient** e **DummyServer** sono programmi Client e Server d’esempio nel sistema.

# Object Diagram



In questo diagramma è rappresentata una possibile configurazione degli oggetti.

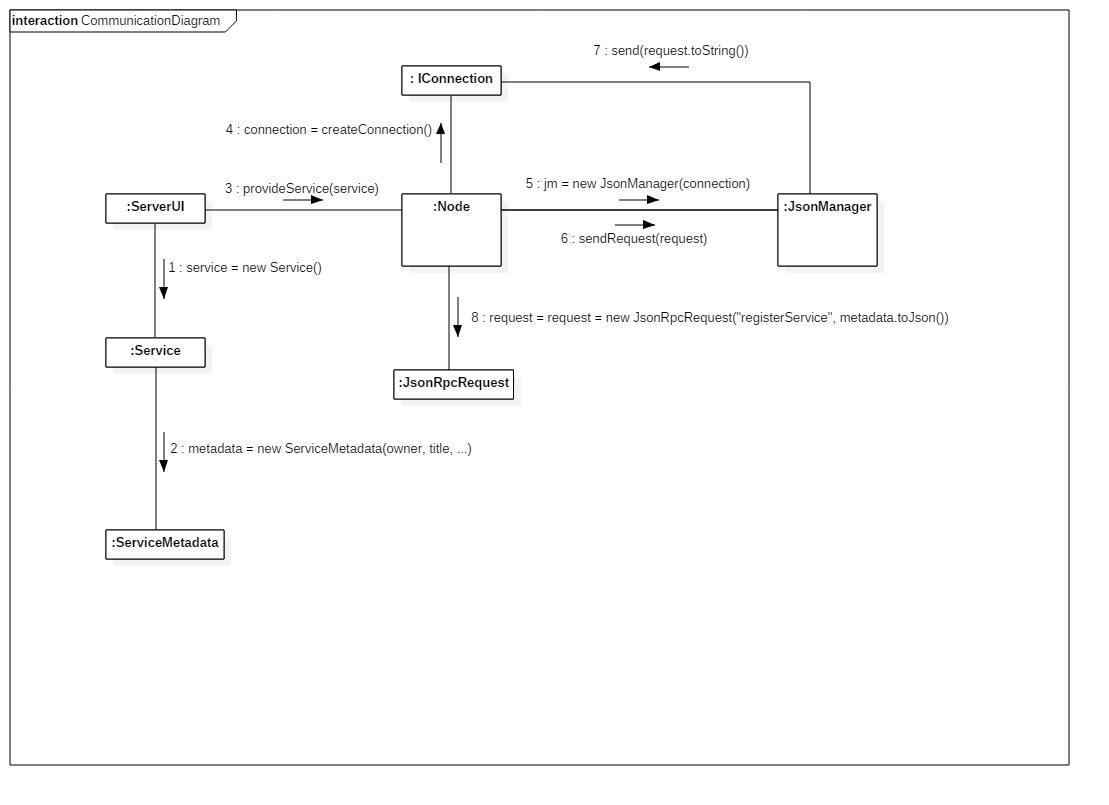
Nello specifico si possono notare tre entità principali:

* **DummyServer** è un fornitore di due servizi,
* **DummyClient** è un utilizzatore di servizi,
* il **Broker**, che fa da tramite fra i due.

La situazione corrente si è raggiunto dopo questi passaggi:

* Il **DummyServer** ha correttamente creato e aggiunto i suoi due servizi al **Node**, che a sua volta li ha registrati nel Broker.
* Il **DummyClient** ha richiesto un servizio al **Broker**. La richiesta è ancora in corso, infatti sono ancora presenti il **JsonRpcManager** con la relativa **IConnection**: nel momento in cui la richiesta è stata soddisfatta, la connessione viene chiusa e il **JsonRpcManager** distrutto.

# Communication Diagram

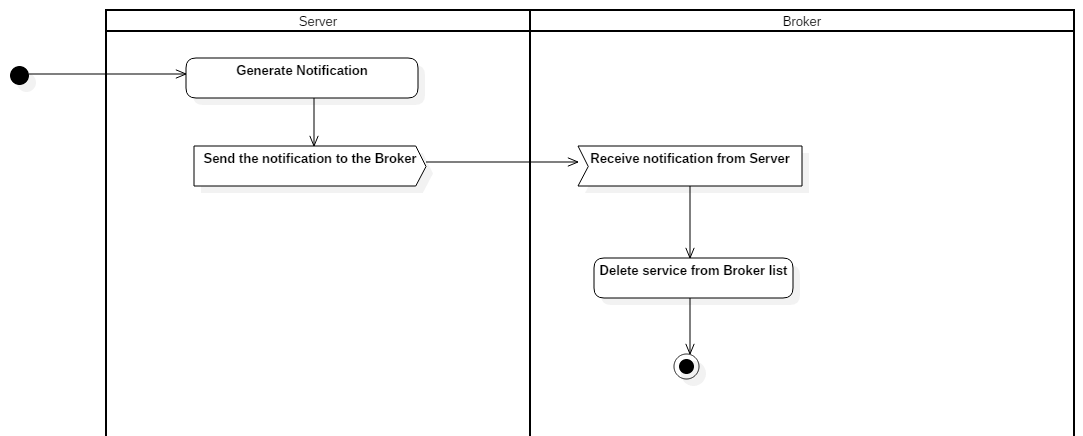
 In questo communication diagram viene descritto lo scenario in cui un nodo *Server* pubblica un nuovo servizio da lui gestito.

Viene istanziato un oggetto di tipo Service che conterrà la funzione da svolgere per erogare il servizio ed al suo interno vengono creati i metadata (singola istanza della classe *ServiceMetadata*) necessari al Broker e ai Clients per identificare tale servizio.

Attraverso l'interfaccia pubblica *provideService(…)* l'oggetto *Node* può inviare la richiesta di pubblicazione del servizio al Broker:

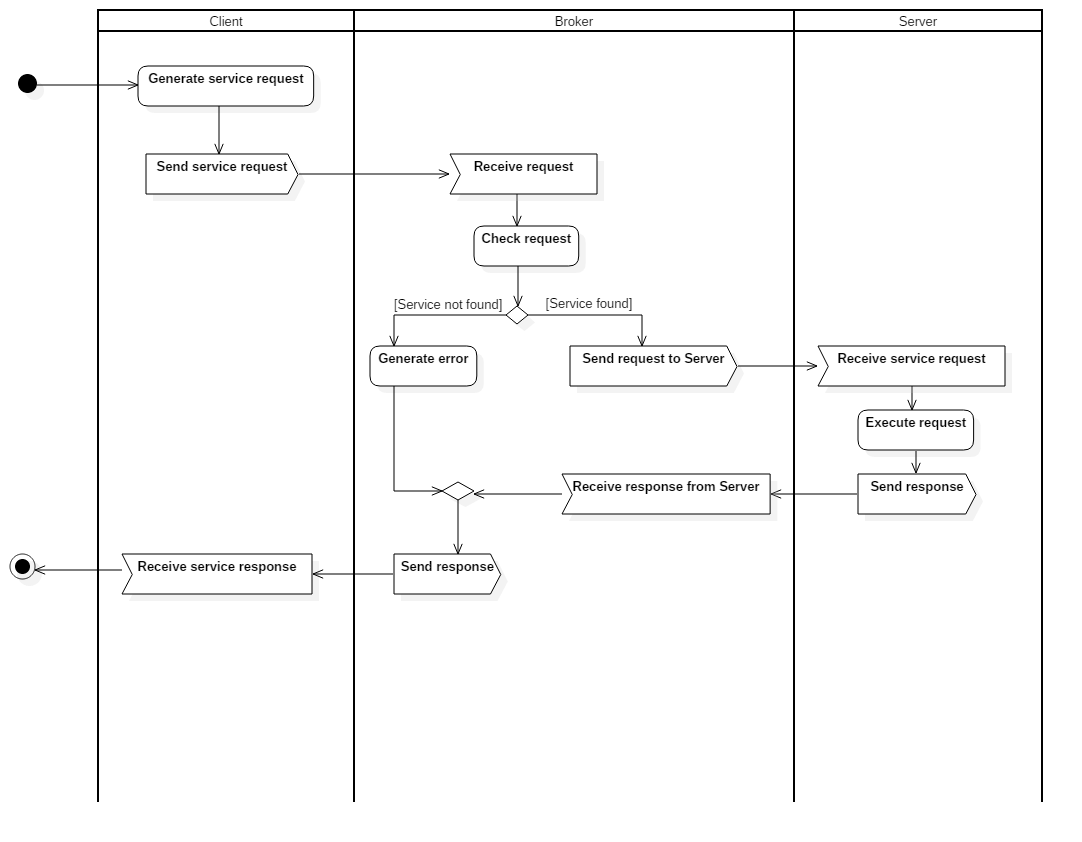
* attraverso *IConncetion* crea una connessione verso il Broker.
* converte la richiesta istanziando una *JsonRpcRequest*, specificando l’intenzione di pubblicare un servizio.
* istanzia un oggetto della classe *JsonRpcManager*, il quale, attraverso *send(..)*, l’interfaccia pubblica di *IConncetion*, invia *JsonRpcRequest* al Broker.

# Activity Diagram – Cancellazione di un servizio



Questo activity diagram descrive le operazioni che il sistema deve eseguire al fine di eliminare un servizio dalla lista dei servizi conosciuti dal *Broker*. Il *server* può eliminare un servizio quando lo richiede più opportuno, *termina il servizio e notifica il Broker dell’eliminazione*. Il *Broker* *aggiorna la lista dei servizi eliminando il servizio non più disponibile*. Non è necessario, ai fini del progetto, che il *Broker* comunichi l’avvenuta eliminazione al *Server*.

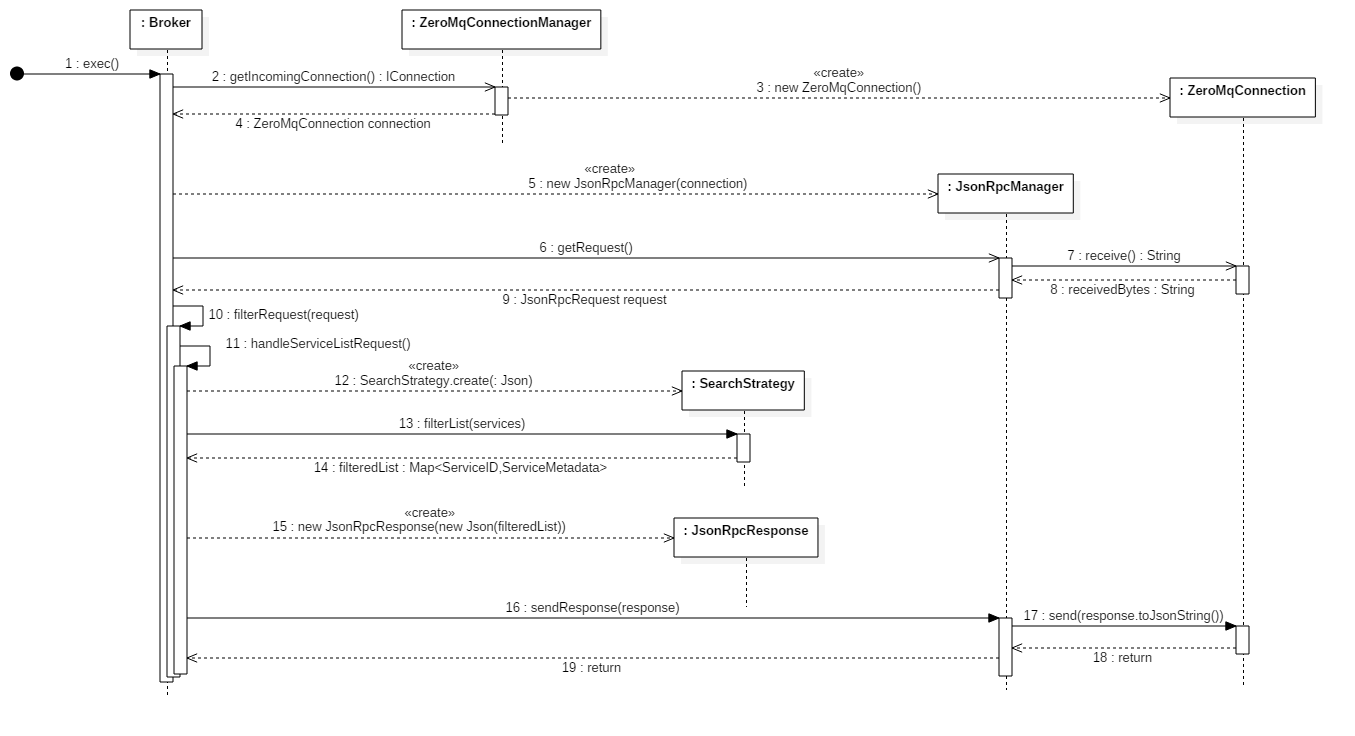
# Activity Diagram – Invocazione di un servizio

 In questo activity diagram vengono mostrate le operazioni svolte , dall’intero sistema, al fine di erogare un servizio richiesto dal *Client*. Come prima operazione il *Client genera una richiesta di servizio* e la *inoltra al Broker restando in attesa di risposta*. Il *Broker verifica la presenza di tale servizio*:

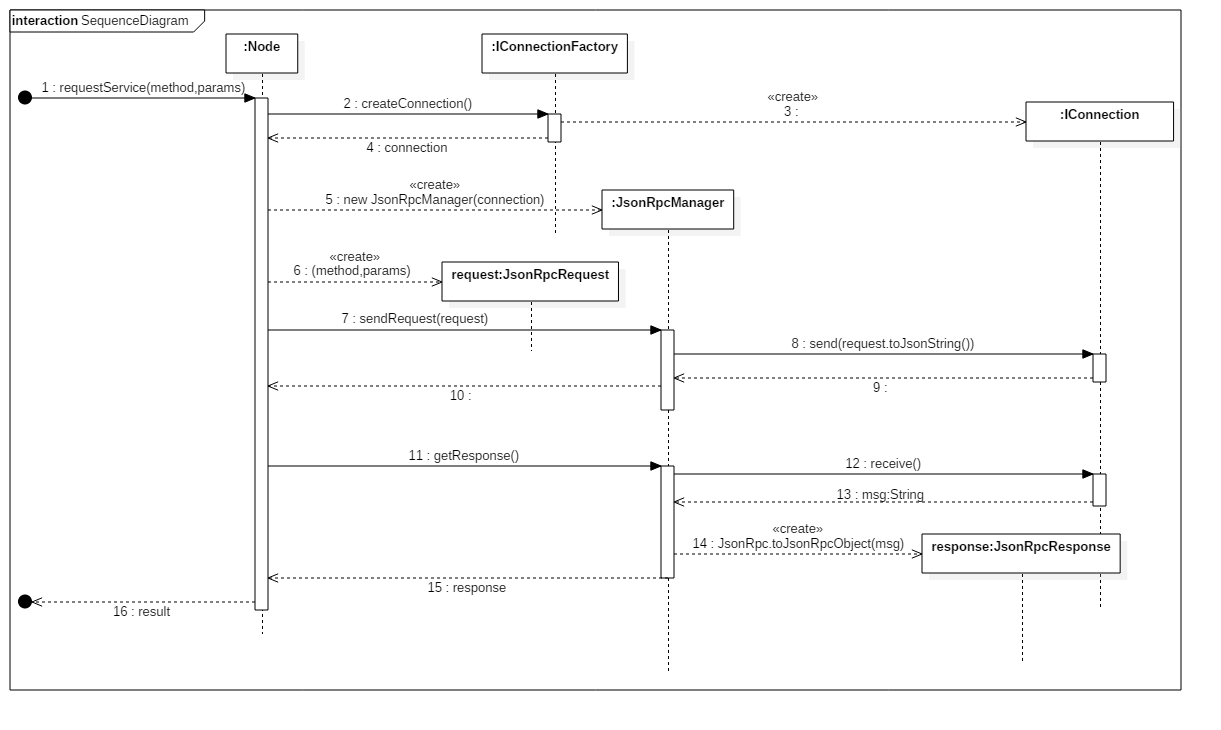
* *se presente inoltra la richiesta al Server* che lo eroga rimanendo in *attesa di una sua risposta contente il risulta del servizio richiesto.*
* altrimenti il *Broker genera una risposta di errore*.

In entrambi i casi *il Client riceve una risposta dal Broker* terminando correttamente l’operazione di richiesta di servizio.

# Sequence Diagram – Richiesta della lista dei servizi



# Sequence Diagram – Invocazione di un servizio



La possibilità di invocare un servizio viene fornita al livello superiore sotto forma di una semplice funzione, **requestService**, che accetta l’identificativo del metodo da invocare e i parametri impacchettati in un oggetto Json, e ritorna i risultato anch’esso sotto forma di Json.

I passaggi che vengono svolti per invocare il servizio sono:

1. Instaurazione di una connessione col broker, gestita dal **IConnectionFactory** (passaggi 2,3,4).
2. Costruzione del **JsonRpcManager** con la **IConnection** appena ottenuta (passaggio 5).
3. Costruzione della **JsonRpcRequest** con i parametri passati dall’utente e opportuni dati aggiuntivi(come per esempio il campo ID, generato al momento) ( passaggio 6).
4. Invio della richiesta ( passaggi 7,8,9,10).
5. Ricezione della risposta, da cui viene estratto il campo result e ritornato all’utente sotto forma di oggetto Json

( passaggi 11,12,13,14,15).

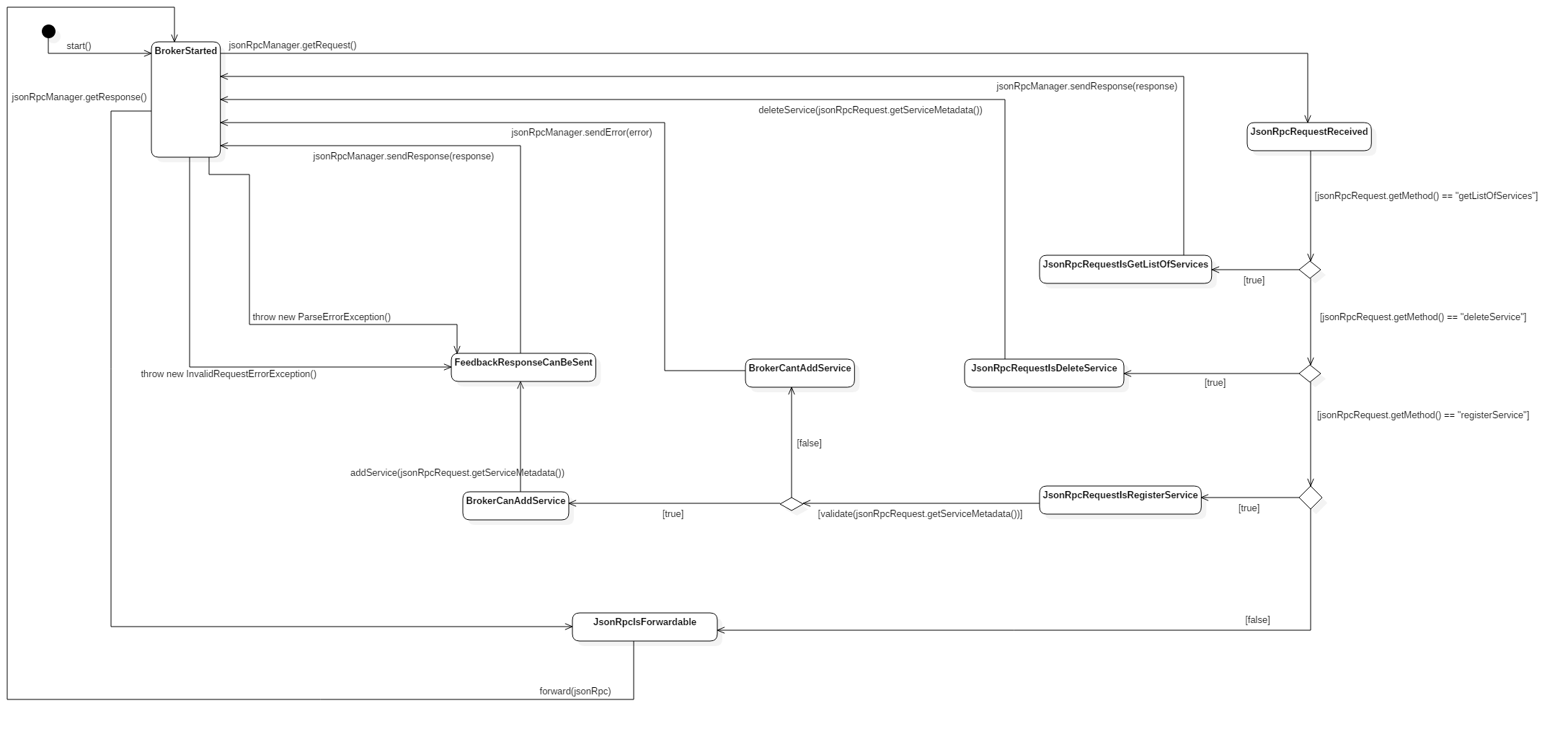
Note:

* Tutti i metodi di ricezione, quindi **getResponse()** del **JsonRpcManager**, **receive()** di **IConnection** e lo stesso **requestService(…)**, sono stati pensati come bloccanti: fintanto che non hanno un valore da ritornare bloccano l’intero flusso di istruzioni, sfruttando il fatto che le code di messaggi sono gestite già da ZeroMQ.

Per questo motivo si sfrutteranno i thread, nello specifico ogni servizio fornito, sia all’interno del **Node** che del **Broker**, sarà gestito da un thread personale, con relativo **JsonRpcManager** e **IConnection**.

* La **IConnectionFactory** potrebbe gestire le **IConnection**(s) tramite una ObjectPool, visto che le richieste hanno comunque un tempo di esecuzione limitato (nel caso una risposta non arrivi entro un certo tempo, si genererà un errore di timeout).

# State Diagram – Broker



# State Diagram – Node

