4. JAVA REMOTE METHOD INVOCATION (RMI)

Java Remote Method Invocation (**Java RMI**) è una libreria di Java che permette lo sviluppo di applicazioni distribuite, fornendo la possibilità di effettuare comunicazione remota tra programmi scritti in Java. Infatti, **Java RMI** offre ad un oggetto in esecuzione su una Java Virtual Machine la possibilità di invocare metodi di un oggetto in esecuzione in una altra JVM, anche se essa si trova su una macchina differente.
Il ruolo che viene ricoperto da Java RMI all'interno della Java Platform e quello di *integration library* (libreria per l'integrazione).
Le applicazioni RMI seguono un'architettura client-server dove il server crea un certo numero di *oggetti server* accessibili da remoto e attende che gli *oggetti client* ne utilizzino i servizi, compiendo invocazioni remote sui metodi che espongono.
Gli obiettivi principali che si poneva la realizzazione di Java RMI sono:

1. Invocazione trasparente di metodi remoti:
Java RMI offre al programmatore un meccanismo semplice per l'invocazione di metodi che sono offerti da un oggetto remoto, e deve avvenire fornendo l'"illusione" che essa avvenga su un oggetto che risiede all'interno dello stesso spazio di indirizzamento utilizzato dall'oggetto che compie l'invocazione.

2. Integrazione in Java:
Il modello distribuito si integra all'interno del linguaggio Java standard, che permette di offrire un ambiente familiare allo sviluppatore e può usare gli stessi strumenti, modelli e astrazioni che vengono utilizzati per oggetti locali. Java RMI fornisce un ***garbage-collector*** distribuito in modo da preservare la modalità di gestione della memoria di Java che solleva il programmatore dal doversi occupare esplicitamente della memoria.

3. Non-trasparenza della natura locale/remota di un oggetto:
Nonostante l'obiettivo di assicurare la semplicità di uso per il programmatore, esistono diverse caratteristiche del linguaggio che non devono essere nascoste al programmatore. Quindi, il fatto che un oggetto sia remoto oppure locale deve essere chiaro ed evidente.

4. Rendere minima la complessità di client e server:
Si deve assicurare la minima complessità all'applicazione distribuita basata su Java RMI. In particolare, il livello di complicazione che viene introdotto da un oggetto distribuito per l'oggetto clienti (cioè l'oggetto che compie l'invocazione remota) e per l'oggetto server (cioè l'oggetto che riceve ed esegue l'invocazione) deve essere limitato.

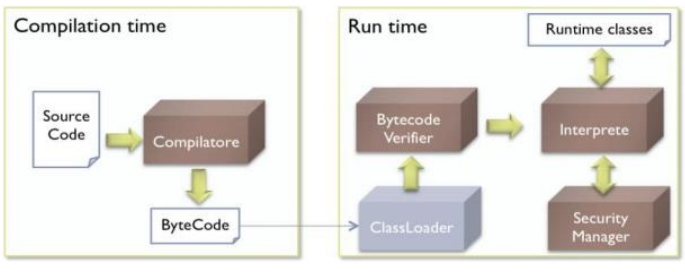
5. Preservare la sicurezza fornita da Java:
Il modello a oggetti distribuito fornito da Java RMI non deve alterare il livello di sicurezza che viene offerto dalla piattaforma Java. Infatti, sin dalla presentazione del linguaggio, la "sicurezza" e la "robustezza" del linguaggio sono state al centro delle attenzioni dei progettisti, principalmente a causa della natura distribuita del linguaggio, che prevede la esecuzione in locale di programmi che vengono scaricati dalla rete.

6. Modalità di invocazione:
Java RMI deve prevedere la possibilità che esistano diversi tipi di invocazione, fornendo quella di tipo ***unicast*** da un client verso un server ma permettendo (in futuro) anche l'estensione verso invocazioni di tipo ***multicast*** vale a dire verso diversi server replicati. Inoltre, deve essere possibile che l'oggetto server sia attivato solo al momento dell'invocazione e che i riferimenti ad oggetti persistenti siano persistenti.

7. Livelli di trasporto multipli:
Infine, Java RMI deve essere aperto verso future espansioni che prevedano che il protocollo di trasporto (basato su ***socket***) possa essere modificato.

GARBAGE COLLECTION:
Essendo la memoria una risorsa finita, bisogna gestirla al meglio. Esistono, in generale, due filosofie di gestione della memoria, la prima è gestita completamente dal programmatore mediante funzioni, come free() e malloc() in C, e quindi è più facile commettere errori, però il programmatore ha il completo controllo, mentre la seconda è fornire un servizio per la allocazione/deallocazione assolutamente trasparente al programmatore, la cosiddetta ***garbage collection***, dove tutti gli oggetti vengono automaticamente allocati/deallocati quando il sistema lo ritiene necessario, in quanto esso mantiene traccia dei riferimenti attivi ad ogni oggetto e quindi si ha maggiore produttività in quanto la progettazione e l'implementazione possono ignorare la gestione della memoria.
La **Garbage Collection** in locale funziona mantenendo e calcolando il numero di riferimenti (tramite tecnica **reference counting**) che fanno riferimento ad un oggetto, se un oggetto non è più riferito allora è candidato ad essere eliminato dall'heap alla prossima esecuzione del **Garbage Collector**.
La proposta di **Java Remote Method Invocation** fornisce un sistema di **Garbage Collection** per gli oggetti remoti. Questo meccanismo si basa su una estensione della ***garbage collection*** locale, la JVM tiene traccia di tutti i riferimenti all'oggetto remoto che risultano essere attivi (*live*). Alla prima invocazione di un oggetto, la JVM ritiene quell'oggetto referenziato e quindi da non eliminare. Al termine delle invocazioni, il client fa in modo di inviare un messaggio di dereferenziazione dell'oggetto e, la JVM server quindi considera quel riferimento debole (*weak*) e quindi passibile di eliminazione alla prossima invocazione dei ***garbage collector***.
Questa tecnica però ha alcuni problemi, ad esempio, il client potrebbe chiudersi per qualche malfunzionamento, oppure potrebbe perdere la connessione verso il server, ed il server si troverebbe con un oggetto remoto che risulta essere referenziato (e quindi da non passare al garbage collector) ma in effetti non sarà mai utilizzato dal client, e quindi rappresenta un potenziale problema di **memory leak**. A questo scopo si introduce il meccanismo di **lease**, ovvero ogni riferimento che viene assegnato al client ha un tempo di vita specificato. Al termine del periodo, se non vengono effettuate altre invocazioni, il server considera quel riferimento non più valido e quindi l'oggetto diventa *weak* e collezionabile dal garbage collector. Questo significa che, in generale, il programmatore che scrive l'oggetto client deve prevedere che il **lease** possa scadere e, per evitare che l'oggetto server sia eliminato, fornire dei metodi che (a intervalli di tempo prefissati) provveda a "rinnovare" il lease, effettuando delle chiamate fittizie a metodi che non hanno nessun effetto.

SICUREZZA IN JAVA:
La sandbox fornita dal linguaggio permette di eseguire applicazioni (e applet) in maniera tale che le operazioni che esse compiono siano controllate e ristretta così da prevenire danni dovuti ad errori di programmazione oppure da intenzionali tentativi di operazioni illegali.



Java fornisce la **sandbox** basandosi su **4 livelli di sicurezza**:

1. **Sicurezza del linguaggio**, Java è fortemente tipizzato quindi tutte le variabili hanno un tipo definito a tempo di compilazione e solamente (poche) implicite conversioni (casting) vengono effettuate dal compilatore e dalla macchina virtuale a run-time, tutte le altre operazioni di casting devono essere esplicitate dal programmatore. Poi, Java offre la gestione automatica della memoria attraverso un meccanismo di ***garbage collection***, che impedisce di esaurire lo spazio di indirizzamento del processo. L'assenza di puntatori e l'impossibilità di poter fare aritmetica o assegnamenti con i riferimenti rende impossibile effettuare accessi illegali in memoria. Inoltre, l'accesso alla memoria reale non viene determinato a tempo di compilazione ma a tempo di esecuzione, non si può conoscere in anticipo in che zona di memoria verranno memorizzati gli oggetti e non si può scrivere codice per alterarli. Infine, a tempo di esecuzione vengono controllati i limiti di array per prevenire accessi a elementi non esistenti.
2. **Classloader**, si occupa di caricare la classe a tempo di esecuzione, anche da locazioni remote. Il suo compito principale è quello di caricare la classe in un **namespace** separato rispetto a quello delle classi locali, in modo che classi del linguaggio built-in, locali, non possono essere rimpiazzare da altre. Infatti, quando si fa riferimento ad una classe, viene prima cercata tra le classi del sistema locale (built-in) e, solo successivamente, nel **namespace** della classe dalla quale viene riferita.
3. **Bytecode Verifier**, dopo che il **Classloader** ha caricato una classe per l'esecuzione, il **Bytecode verifier** controlla che essa non sia "volontariamente" ostile, che non ci siano stack underflow/overflow e violazioni alla regole specificate dai modificatori di accesso.
4. **Security Manager**, si occupa di definire i confini della sandbox. Il **Security Manager** viene interpellato dalla macchina virtuale per ciascuna operazione potenzialmente pericolosa, e fornisce le autorizzazioni sulla base della policy che ha stabilito l'utente lanciando la macchina virtuale.

4.1 MODELLO AD OGGETTI DISTRIBUITI DI JAVA RMI

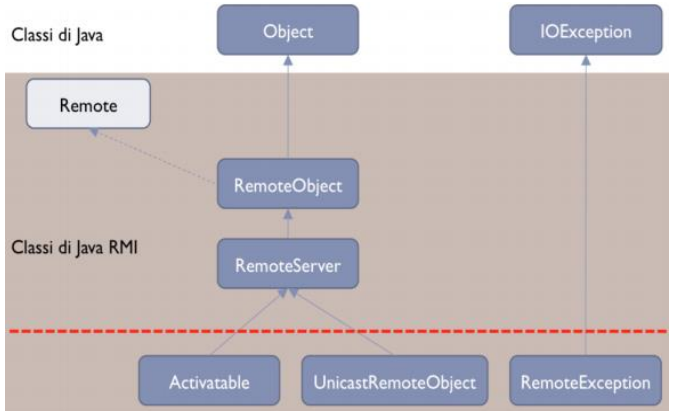
Un **oggetto remoto** è un oggetto i cui metodi possono essere acceduti da un altro spazio di indirizzamento, e potenzialmente da un'altra macchina. La descrizione dei servizi offerti da remoto da un oggetto remoto è contenuta all'interno di una **interfaccia remota** che è una interfaccia Java che dichiara i metodi remoti. Una **invocazione di metodi remoti** (**Remote Method Invocation**) rappresenta l'invocazione di un metodo su un oggetto remoto (specificato nell'interfaccia remota) e ha la stessa sintassi di un'invocazione di un metodo locale.

L'oggetto client di *oggetti remoti server* utilizza esclusivamente l'interfaccia remota dell'oggetto, non la sua implementazione, questo garantisce che le funzionalità remote risultino astratte verso il client e disaccoppia le due implementazioni, permettendo, ad esempio, evoluzioni dell'oggetto server (cioè della sua implementazione) senza che il client debba essere modificato.

4.1.1 STRUTTURA DELLE CLASSI JAVA RMI

Java RMI è contenuto in 5 package:

- **java.rmi** e **java.rmi.server** che contengono il meccanismo basilare di funzionamento delle invocazioni remote;
- **java.rmi.activation** per gli oggetti attivabili;
- **java.rmi.dgc** per la Distributed Garbage Collection;
- **java.rmi.registry** per il servizio di localizzazione.



INTERFACCE ED ECCEZIONI REMOTE:

Prima di definire un oggetto remoto, si deve definire un'interfaccia remota per l'oggetto, in modo che vengano esposti i servizi che l'oggetto remoto intende mettere a disposizione per un utilizzo da parte dei client.

Un'**interfaccia remota per Java RMI** deve estendere (implementare) l'interfaccia **java.rmi.Remote** che è un'interfaccia cosiddetta ***marker***, cioè un'interfaccia vuota che serve solamente per poter segnalare che essa definisce dei metodi accessibili da remoto.

Ogni metodo descritto in un'interfaccia remota deve essere un ***metodo remoto***, cioè deve soddisfare entrambe le seguenti condizioni:

1. Un metodo remoto deve dichiarare esplicitamente l'**eccezione java.rmi.RemoteException**, poiché la semantica dei malfunzionamenti di un oggetto remoto è diverso da quella dei malfunzionamenti di un oggetto locale. In questa maniera si forza lo sviluppo successivo, che comporterà l'invocazione di questi metodi, a gestire l'eccezione remota in maniera esplicita, in quanto il compilatore controlla che l'eccezione sia gestita.
2. I **parametri remoti** di un metodo remoto devono essere dichiarati tramite la propria interfaccia remota, non utilizzando la classe dell'implementazione remota. Questo permetterà di poter passare riferimenti remoti sia come parametri che come valori restituiti.

Il meccanismo dell'interfaccia remota aggiunge un livello ulteriore di accessibilità ai *modificatori di accesso dei metodi* (*public*, *protected*, etc.).

I metodi remoti, dichiarati in un'interfaccia remota sono più accessibili dei metodi public, che risultano accessibili ma solamente da invocazioni all'interno della stessa macchina virtuale. Infine, come nelle interfacce (anche remote) sia possibile definire delle costanti.

IMPLEMENTAZIONI REMOTE:

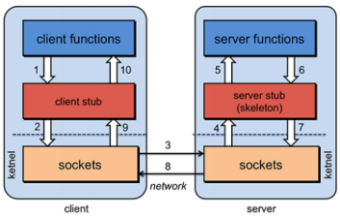
Per realizzare l'implementazione remota che deriva (*implements*) da un'interfaccia remota per offrire verso l'esterno i metodi remoti in essa definiti, si può procedere in due modi:

1. **Riuso dell'implementazione remota**, prevede che la classe che contiene l'implementazione dell'oggetto derivi esplicitamente da **java.rmi.server.UnicastRemoteObject** ereditando di conseguenza il comportamento definito dalle classi **java.rmi.server.RemoteObject** e **java.rmi.server.RemoteServer**;
2. **Classe di implementazione locale**, permette che la classe derivi il comportamento da qualche altra classe (non remota) e che si debba quindi occupare esplicitamente di esportare l'oggetto (tramite il metodo statico ***exportObject()*** di **java.rmi.server.UnicastRemoteObject**) e di implementare la semantica di alcune operazioni di ***Object*** per oggetti remoti che sono ridefiniti in **java.rmi.server.RemoteObject** e **java.rmi.server.RemoteServer**.

Va sottolineato come l'implementazione remota può implementare anche altri metodi, oltre a quelli remoti definiti nell'interfaccia remota ma che questi saranno accessibili esclusivamente in locale, secondo le direttive di accesso fornite dai modificatori di accesso.

RIFERIMENTI REMOTI:

Nel modello distribuito, gli oggetti client interagiscono con un oggetto **stub** che espone localmente le stesse interfacce remote definite dall'oggetto remoto.
In questa maniera, lo **stub** rappresenta l'interfaccia remota dell'oggetto remoto in locale, sulla macchina dove l'oggetto client è in esecuzione.
Dal punto di vista della JVM del client, il tipo dello **stub** è lo stesso di quello dell'oggetto server, quindi, il client può accedere tradizionalmente al tipo di un oggetto remoto, controllando quale interfaccia remota implementa, attraverso ***istanceof***.



CARICAMENTO DINAMICO DELLE CLASSI:

Java Remote Method Invocation permette il passaggio di oggetti come parametri, valori restituiti o eccezioni, attraverso la **serializzazione**. Questo però crea problemi, in quanto, il caricamento delle classi a tempo di esecuzione risulta essere più complesso nel momento in cui stiamo passando ad un metodo offerto da un server remoto (ad esempio) un parametro che è una istanza di una sottoclasse della classe dichiarata nella firma dei metodo. In questo caso, l'oggetto remoto può trovarsi nella situazione in cui non conosce esattamente come è strutturata la classe di cui l'oggetto passato e istanza. Questo viene risolto da Java RMI attraverso il caricamento dinamico delle classi. Quando si fa il marshalling degli oggetti per la trasmissione (ad esempio, come parametri nella invocazione da client a server), essi vengono anche annotati con il **codebase**, cioè con la Uniform Resource Locator (URL) di un server WWW da dove è possibile trovare la definizione della classe (cioè il file .class). Quando viene effettuato l'unmarshialling dell'oggetto, il **Classloader** cerca di risolvere il nome della classe nel suo contesto, poi, in caso non sia possibile, viene acceduta la definizione della classe per poter ricreare l'oggetto all'altro capo della comunicazione (nel nostro esempio, sul server).

LOCALIZZAZIONE E INVOCAZIONE DI OGGETTI REMOTI:

Per poter invocare il metodo remoto di un oggetto remoto, l'oggetto client deve avere a disposizione il riferimento remoto, può essere:

1. **Ottenuto come risultato di altre invocazioni (locali o remote) di metodi**, questa è standard per quanto riguarda le invocazioni locali;
2. **Attraverso un servizio di directory**, Java RMI fornisce un semplice meccanismo di *name server* nella classe **java.rmi.Naming**, che permette di gestire riferimenti a oggetti remoti accessibili specificando un ID (stringa). Tale classe fornisce metodi per ricercare (***lookup ()***), registrare (***bind(), unbind() e rebind()***) ed elencare (***list()***) gli identificativi registrati, accedendo al servizio attraverso una specifica che segue lo standard Uniform Resource Locator (URL).

L'invocazione di un metodo remoto ha la stessa sintassi di un'invocazione locale. Poiché i metodi remoti devono includere ***RemoteException*** nella propria firma, il codice dell'oggetto client viene forzato dal compilatore a gestire questo possibile malfunzionamento della chiamata remota, in aggiunta ad altre eccezioni che dipendono dalla semantica dell'applicazione.

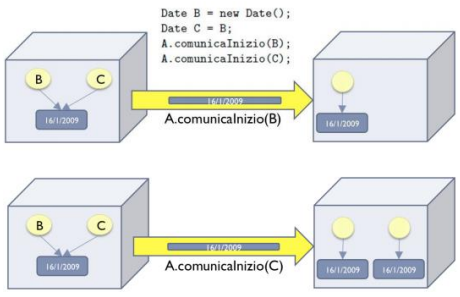
PASSAGGIO DI PARAMETRI:

Un metodo remoto può dichiarare solo parametri o valori restituiti che siano serializzabili, vale a dire che implementino l'interfaccia ***Serializable***. Le classi dei parametri o dei valori restituiti che non siano disponibili localmente, vengono scaricate dinamicamente.
Un oggetto locale passato come parametro o restituito come valore da un'invocazione remota viene **passato per copia**, vale a dire che il contenuto dell'oggetto viene copiato prima di essere serializzato dal meccanismo di serializzazione di Java. Quindi la semantica per il passaggio di parametri locali è diversa da quella di Java che, ricordiamo, passa il riferimento di un oggetto. Quindi, quando vengono passati come parametri più riferimenti allo stesso oggetto ad un'altra macchina virtuale utilizzando invocazioni diverse allora i riferimenti sulla macchina server saranno ad oggetti distinti.

Ad esempio:
Se A è un riferimento ad un oggetto remoto che offre il metodo remoto *comunicaInizio(Date x)*, e se B e C sono riferimenti a oggetti locali di tipo *Date*, dopo avere eseguito il codice:

```
Date B = new Date();
Date C = B;
A.comunicaInizio(B);
A.comunicaInizio(C);
```

Avremo che sulla macchina remota, ci saranno due oggetti diversi di tipo *Date()* a cui punteranno le due variabili, mentre sulla macchina locale, dalla quale provenivano, essi facevano riferimento allo stesso oggetto.

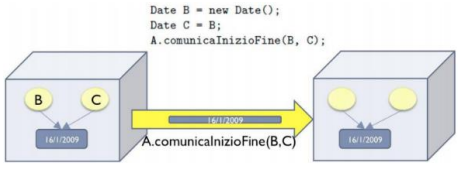


Il meccanismo implementato da Java RMI assicura la cosiddetta **integrità referenziale**, ovvero quando vengono passati più riferimenti allo stesso oggetto nella stessa invocazione, allora viene garantito che anche sulla macchina remota alla quale sono stati passati i riferimenti punteranno allo stesso oggetto.

Continuando l'esempio precedente, se l'oggetto A fornisse anche il metodo remoto *comunicaInizioFine(Date x, Date y)*, allora dopo avere eseguito il codice:

```
Date B = new Date();
Date C = B;
A.comunicaInizioFine(B, C);
```

Avremo che sulla macchina remota ci sarà un solo oggetto *Date* a cui punteranno i due riferimenti passati come parametri.



Quindi quando si passa un oggetto remoto come parametro o si ottiene come valore restituito, viene passato il suo stub, e non l'implementazione.
Questo funzionamento è realizzato col **meccanismo di Marshalling**.

MECCANISMO DI MARSHALLING:

Fare il marshalling di un oggetto in Java significa effettuare una serializzazione modificando la semantica dei riferimenti remoti (invece di un riferimento remoto viene inserito lo stub dell'oggetto remoto) e aggiungendo informazioni all'oggetto (il **codebase** della classe dell'oggetto).
Il marshalling di Java RMI si basa sulla specializzazione del meccanismo tradizionale di serializzazione effettuata da ***ObjectOutputStream***. Infatti, questa classe offre la possibilità di poter modificare il comportamento tramite il quale gli oggetti vengono scritti come stream di byte. Più precisamente, la specializzazione del meccanismo di serializzazione per il marshalling avviene modificando tre metodi della classe ***ObjectOutputStream***:

- *replaceObject()*, può definire un metodo alternativo per serializzare un oggetto sullo stream, deve essere richiamato ogni volta invocato *writeObject()*;
- *enableReplaceObject()*,restituisce un booleano e stabilisce se l'istanza deve oppure no specializzate il meccanismo di serializzazione, usando il metodo *replaceObject()*;
- *annotateClass()*, che permette di inserire informazioni addizionali sulla classe, viene usato per specificare il codebase e permettere quindi il caricamento dinamico.

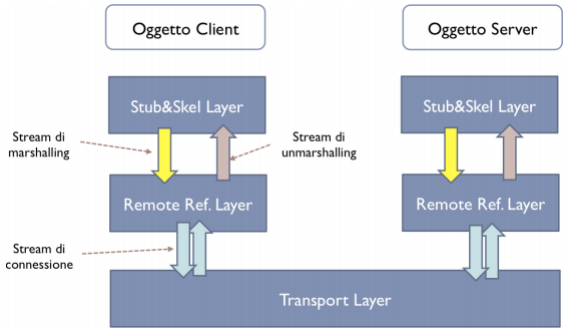
4.2 ARCHITETTURA DI JAVA RMI

Il sistema di Java RMI e strutturato su tre livelli (layer):

- **Stub/Skeleton Layer** che comprende gli stub lato client e gli skeleton lato server;
- **Remote Reference Layer** che specifica il comportamento dell'invocazione e la semantica del riferimento (unicast, multicast, etc.);
- **Transport Layer** che si occupa della connessione e della sua gestione.

L'applicazione dell'utente si trova in cima a questi livelli e interagisce (in maniera moderata) con il livello di stub/skeleton.

Un **client** che invoca un metodo su un oggetto server remoto, fa uso di uno **stub** per portare a termine la sua richiesta in quanto il riferimento remoto che è in possesso del **client** è solamente un riferimento al suo **stub**, presente in locale.
Lo **stub** implementa l'interfaccia remota dell'oggetto remoto (verso il **client**) e inoltra le richieste all'oggetto server attraverso il **remote reference layer** del **client**.
Il **remote reference layer** del **client** si occupa di gestire la semantica delle invocazione remota lato client (se, ad esempio, si tratta di una invocazione unicast o multicast).
Il **livello di trasporto** si occupa di stabilire la connessione con la macchina remota e della successiva gestione della connessione, curando il ***dispatching*** delle invocazioni verso gli oggetti remoti. A questo scopo, inoltra la richiesta al livello di reference del server.
Il livello di reference del server si occupa di inoltrare la richiesta allo **skeleton** e di curare la semantica dell'invocazione lato server (gestendo, ad esempio, i riferimenti ad oggetti persistente per curarne la loro attivazione).



STUB/SKELETON LAYER:

Essendo il livello più alto di Java RMI, esso si occupa di essere l'interfaccia tra l'applicazione (cioè le classi Java scritte dal programmatore) ed il resto dei sistema. L'interfaccia verso il basso, in direzione del **remote reference layer**, consiste nel fornire uno stream di Marshal di oggetti Java che vengono passati (per copia) al **remote reference layer**.

Lo **stub** è incaricato di:

- Iniziare la connessione con la macchina virtuale remota, chiamando il **remote reference layer**;
- Effettuare il **marshalling** verso uno stream di marshal, fornito dal **remote reference layer**;
- Attendere il risultato della invocazione;
- Effettuare l'**unmarshalling** dei valori restituiti (o delle eccezioni verificatasi);
- Restituire il valore verso l'oggetto client che ha richiesto l'invocazione.

Lo **skeleton** è incaricato di effettuare il dispatching verso l'oggetto remoto, vale a dire, curare che l'invocazione sia effettuata sull'oggetto remoto in attesa di invocazioni. Quando uno **skeleton** riceve un'invocazione in entrata, si occupa di:

- Effettuare l'**unmarshalling** dei **remote reference layer** (lato server) dei parametri per l'invocazione;
- Invocare il metodo sull'implementazione che si trova nella sua JVM;
- Effettuare il **marshalling** del valore restituito (compreso di eventuali eccezioni) verso chi ha invocato il metodo.

REMOTE REFERENCE LAYER:

Questo layer si occupa di interfacciare il livello di trasporto con quello di **stub/skeleton** fornendo e supportando la semantica dell'operazione di invocazione di un metodo. In un senso, si occupa della parte di protocollo indipendente dall'invocazioni remote specifiche, che vengono gestite da stub e da skeleton. Infatti, ogni implementazione di oggetto remoto può scegliere un protocollo generale che determina le modalità di invocazione, fissato per la durata di vita dell'oggetto. Ad esempio, diverse sarebbero le modalità permessa alle invocazioni:

- Invocazioni **unicast**, vale a dire da un singolo client verso un singolo server;
- Invocazioni **multicast**, vale a dire un singolo client fa una invocazione ad una "batteria" di server replicati, in maniera da poter garantire la ridondanza: se uno di questi è in esecuzione allora risponderà all'invocazione;
- Invocazioni di **oggetti attivabili**: le invocazioni potrebbero essere effettuate ad un oggetto remoto che è persistente, vale a dire viene attivato se arrivano delle invocazioni;
- Invocazioni con **riconnessione**: le invocazioni potrebbero tentare connessioni alternative se l'oggetto remoto originariamente contattato non risponde all'invocazione.

Il protocollo di invocazione utilizza le due componenti client e server del **remote reference layer**. Il lato client ha informazione circa il server della invocazione (se **unicast**) e comunica attraverso il livello di trasporto verso il lato server dello stesso layer.
Durante l'invocazione, da **stub** a **skeleton** e ritorno, il **remote reference layer** può intervenire per forzare uno specifico protocollo di invocazione, ad esempio, nel caso di un'invocazione multicast la parte client del livello può inoltrare la richiesta ad un insieme di server e selezionare la prima risposta che arriva, scartando le altre. Il lato server, invece viene utilizzato per l'attivazione di oggetti persistenti in caso d'invocazione remota.
Questo layer fornisce verso l'alto (lo stub/skeleton layer) un riferimento ad un oggetto che implementa l'interfaccia **java.rmi.server.RemoteServer**, che espone un metodo *invoke()* per effettuare l'inoltro dell'invocazione, che viene chiamato dallo stub.
Il **remote reference layer** interagisce verso il basso col **livello di trasporto**, verso il quale utilizza l'astrazione di una connessione orientata ai flussi (stream di connessione). Questa astrazione viene fornita dal livello di trasporto che non è obbligato a realizzare una connessione e porrebbe utilizzare protocolli connectionless senza alterare la modalità con cui il remote reference layer comunica i dati.

TRANSPORT LAYER:

Il livello di trasporto ha il compito di:

- Stabilire la connessione verso macchine con indirizzi IP remoti;
- Gestire le connessioni e monitorare il loro stato;
- Rimanere in ascolto per connessioni in arrivo;
- Gestire una tabella degli oggetti remoti che risiedono nello spazio di indirizzamento locale;
- Stabilire una connessione per le chiamate in entrata;
- Identificare l'oggetto dispatcher a cui inoltrare la connessione.

4.3 PROCESSO DI CREAZIONE DI UN PROGRAMMA JAVA RMI

Il processo per la creazione di un programma Java RMI può essere riassunto dalla Figura e si suddivide, dopo alcuni passi preliminari, in due sotto-processi:

- uno che procede verso lo sviluppo ed esecuzione del server;
- l'altro che va in direzione delle sviluppo e della esecuzione dei client.

Alcuni passi sono dipendenti dai precedenti e altri, invece, possono proseguire indipendentemente.
In particolar modo, lo sviluppo del client risulta essere solo minimamente influenzato da quello del server.
Il programmatore lato client ha necessita esclusivamente della interfaccia remota (ed eventualmente dello stub) per potere proseguire nello sviluppo del programma client.

DEFINIZIONE INTERFACCIA REMOTA:

Il primo passo consiste nella definizione di quali sono i servizi che vengono offerti dal nostro server, specificati all'interno di una interfaccia. Questa interfaccia rappresenta il "contratto" che vincola il server ad offrire determinati servizi se il client utilizza l'interfaccia che il server espone. L'implementazione, ovviamente, sarà a carico dei server e totalmente nascosta al client.

La interfaccia remota per RMI deve avere le seguenti caratteristiche:

- L'interfaccia deve derivare dalla interface mark-up *Remote*;
- Tutti i metodi remoti devono lanciare l'eccezione **java.rmi.RemoteException**.

Un classico errore che viene fatto è quello di inserire nell'interfaccia remota "a tappeto" tutti i metodi del server, anche quelli che sono locali.

IMPLEMENTAZIONE DEL SERVER:

Un oggetto remoto in Java deve essere istanza di una classe che:

- Implementi una o più interfacce remote (cioè che estendano *Remote*);
- Derivi da **java.rmi.UnicastRemoteObject**.

Il fatto che il server derivi da **UnicastRemoteObject** implica che il costruttore del nostro server debba esplicitamente essere scritto (anche vuoto) in quanto è necessario che il costruttore della sottoclasse (il nostro server) lanci esplicitamente la eccezione **UnicastRemoteObject** che viene lanciata dal costruttore della superclasse.

COMPILAZIONE DEL SERVER:

In generale, una buona IDE lo fa automaticamente, ma è importante sapere do esattamente vengono messi i file *.class* che vengono creati dal compilatore *javac*. Se si accettano i valori di default allora i sorgenti vengono messi in una directory *src* mentre i file in bytecode vengono messi in una directory *bin* che non viene visualizzata. Per poterla visualizzare è necessario utilizzare una *view Navigation*.

COMPILAZIONE CON LO STUB COMPILER rmic:

Avendo specificato interfaccia remota e server (che deriva da **UnicastRemoteObject**) si possono generare automaticamente i file che sono necessari (stub e skeleton) senza che debbano essere scritti dal programmatore. A questo scopo, RMI fornisce uno *stub compiler*, chiamato *rmic* che, eseguito sul file *.class* dei nostro server, genera lo stub (e lo skeleton). Quindi, dobbiamo eseguire lo stub sul file *.class* del nostro server. A tale scopo possiamo definire una applicazione esterna da chiamare dall'interno di Eclipse in modo da poterlo eseguire selezionando (nel Navigator) il file *.class* e generare lo stub del nostro oggetto server. Una volta fatto ciò fare refresh della cartella.

SERVIZIO DI NAMING rmiregistry:

A questo punto è necessario che il nostro oggetto remoto possa essere accessibile dai client che ne vogliono invocare i servizi. A tale scopo, Java RMI propone un servizio di naming (abbastanza semplice) chiamato **rmiregistry**. Questo programma deve essere lanciato prima di eseguire l'oggetto server, in quanto il server (tipicamente) tra le prime operazioni farà in modo di registrarsi presso il servizio di naming con una etichetta. I client che vogliono accedere ai servizi di un oggetto remoto, fanno una richiesta (operazione di lookup) al registry per ottenere un riferimento remoto da usare per le invocazioni remote. Il servizio di naming deve essere lanciato dalla directory dove si trova il file *.class* dello stub, dell'oggetto remoto e dell'interfaccia remota, quindi (nei nostri esempi) dalla directory *bin* del progetto.

ESECUZIONE SERVER:

Ora si può eseguire il server eseguendo con la macchina virtuale la classe appropriata. L'unica cosa particolare da fare è scegliere una politica di sicurezza per la macchina virtuale, operazione che non compiamo per applicazioni locali, di solito. Infatti, la macchina virtuale, per poter eseguire operazioni potenzialmente pericolose (ed accedere la rete lo è) ha bisogno che venga esplicitamente permesso in una policy che viene fornita alla macchina virtuale su linea di comando. Nel nostro caso adottiamo una politica liberale nel quale tutto è permesso:

```
grant {    permission java.securityAllPermission;    }
```

Fornire su linea di comando alla macchina virtuale il file di policy da seguire:

```
java -Djava.security.policy=policyall xxx
```

Se la macchina virtuale non trova il file di policy non protesta in maniera evidente, ma adotta una policy estremamente restrittiva, impedendoci accesso (dai server) ai servizio di naming.

REGISTRAZIONE DEL SERVER SUL SERVIZIO DI NAMING:

Appena lanciato il server, esso deve (di norma) registrarsi sul servizio di naming. Vengono usati metodi dei package **java.rmi.Naming** che prevedono semplici modalità di registrazione, richiesta e deregistrazione. Per motivi di sicurezza, non è possibile che il server si registri su un servizio di naming che non sia sul suo stesso host, quindi non è possibile avere un servizio di naming "esterno", il che riduce l'efficacia di soluzioni distribuite.

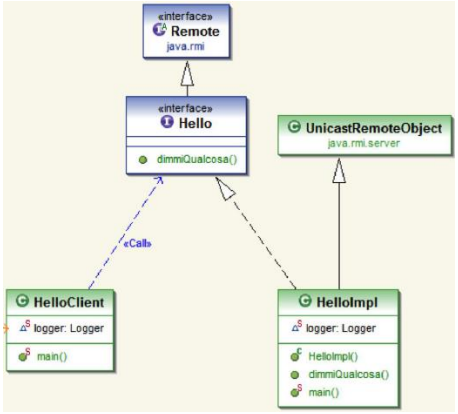
IMPLEMENTAZIONE CLIENT:

Si deve risolvere il problema della localizzazione dell'oggetto remoto, effettuata tramite i servizi di **java.rmi.Naming** che permettono di ottenere un riferimento remoto ad un oggetto server. A questo punto, l'invocazione dei metodi remoti procede in maniera tradizionale, con la sola differenza che si deve gestire l'eccezione **RemoteException** che viene lanciata da tutti i metodi remoti. L'implementazione dell'invocazione remota risulta essere a carico dello stub.

COMPILAZIONE ED ESECUZIONE DEL CLIENT:

L'unica attenzione che si deve prestare è al fatto che deve essere presente lo stub del server nella directory dove il client viene compilato. La stessa attenzione deve essere compiuta per l'esecuzione.

ESEMPIO HelloWorld in Java RMI:



La nostra applicazione consisterà di un server che offre un metodo remoto che prende come parametro il nome dell'utente che sta invocando il metodo e restituisce una stringa, che verrà stampata a video dal client.

DEFINIZIONE DELL'INTERFACCIA REMOTA:
Implementazione ***Hello.java*** →
Dobbiamo specificare una interface Java che derivi da `java.rmi.Remote` e indicare il metodo remoto con l'indicazione dei parametri e del tipo di dati restituito. In più, deve lanciale `java.rmi.RemoteException`.

```
public interface Hello extends java.rmi.Remote{
    String dimmiQualcosa(String daChi) throws java.rmi.RemoteException;
}
```

IMPLEMENTAZIONE SERVER:
Implementazione ***HelloImpl.java*** →
Ora, dobbiamo implementare il server, una convenzione utilizzata per denotare una classe che è l'implementazione di un'interfaccia, è quello di chiamare il programma come l'interfaccia, aggiungendo Impl al nome dell'interfaccia.
Notiamo la definizione di un costruttore vuoto, necessario perché (per fare il match con il costruttore della superclasse) deve lanciare la eccezione `RemoteException`.
Poi implementiamo il metodo remoto dell'interface remota col classico comportamento che ci aspettiamo
Nel main istanziamo il Security Manager di RMI che serve a poter caricare classi dinamicamente dalla rete (non serve in questo caso) se esse non sono presenti nei CLASSPHH della macchina virtuale e che implementa la politica data nel file *policyall*.
I blocco try catch serve per l'eccezioni che possono essere lanciate dalla istanziazione dell'oggetto remoto e dall'utilizzo del servizio di Naming attraverso il metodo rebind() che registra l'oggetto, con nome HelloServer sul registry attivato sullo stesso host dove eseguiremo il programma server.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Logger;
public class HelloImpl extends UnicastRemoteObject implements Hello{
    private static final long serialVersionUID = -4469091140865645865L;
    static Logger logger= Logger.getLogger("getLogger");
    //Costruttore
    public HelloImpl() throws RemoteException{  }
    //Metodo remoto dimmiQualcosa:
    public String dimmiQualcosa(String daChi) throws RemoteException {
        logger.info("Sto salutando "+daChi);
        return "Ciao!";
    }
    public static void main(String args[]) {
        System.setSecurityManager( new RMISecurityManager());
        try{
            logger.info("Creo l'oggetto remoto...");
            HelloImpl obj = new HelloImpl();
            logger.info("...ora effettuo il rebind...");
            Naming.rebind("HelloServer", obj);
            logger.info("... Pronto!");
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

COMPILAZIONE DEL SERVER: easy.

COMPILAZIONE CON STUB COMPILER rmic:

Dopo aver compilato, si ottiene la classe *HelloImpl.class*, sulla quale possiamo eseguire *rmic* configurato.

SERVIZIO DI NAMING rmiregistry:

A questo punto si può lanciare il registry di RMI.

ESECUZIONE DEL SERVER:

Possiamo lanciare il server ed ottenere dalla console le info che il server è stato creato come oggetto remoto.

REGISTRAZIONE SERVER SU SERVIZIO DI NAMING:

Con `Naming.rebind()`, porta alla registrazione dell'oggetto sul registry, indica che il server è pronto.

IMPLEMENTAZIONE CLIENT:
Tranandosi di invocazioni remote (sia l'utilizzo del servizio di naming che la vera e propria invocazione remota del metodo) questa parte del programma va raccolta in un try catch.
Viene effettuato il lookup() del server sul servizio di Naming. Questo viene effettuato passando come parametro una URL che specifica rmi: come protocollo, poi indica il server localhost ed infine indica l'id HelloServer con il quale il server si è registrato.
In questa maniera, il riferimento obj è un riferimento remoto ad un oggetto server di cui si ignora l'implementazione e si conosce esclusivamente i servizi che vengono offerti tramite la interfaccia remota Hello.
A questo punto, viene invocato il metodo remoto, passando come parametro il nome dell'utente ("Pippo") e poi si stampa quello che è stato restituito dalla invocazione remota del metodo `dimmiQualcosa()`.

```
import java.rmi.*;
import java.util.logging.Logger;
public class HelloClient {
    static Logger logger = Logger.getLogger("getLogger");
    public static void main(String args[]) {
        try {
          logger.info("Sto cercando l'oggetto");
          Hello obj=(Hello)Naming.lookup("rmi://localhost/HelloServer");
          logger.info("...Trovato! Invoco metodo...");
          String risultato = obj.dimmiQualcosa("Pippo");
          System.out.println("Ricevuto: " + risultato);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

COMPILAZIONE ED ESECUZIONE DEL CLIENT: easy.