Registri RMI

- Per poter interagire con un server remoto, un generico client deve essere in possesso di una sua referenza remota (ovvero un oggetto stub).
- Un *servizio di Naming* è una risorsa centralizzata che può essere utilizzata da un client come meccanismo di bootstrap per ricavare una referenza remota ad un server.
- Un servizio di Naming associa nomi logici (stringhe) di server remoti alle loro referenze remote. Dal punto di vista dei clients un servizio di Naming agisce come una sorta di “guida telefonica”: dato il nome di un server se ne può ricavare la sua referenza remota.
- Un servizio di Naming deve essere facilmente localizzabile (attraverso indirizzo IP e porta TCP/IP),
- Un *registro RMI* è un particolare servizio di naming che associa nomi di server (essenzialmente stringhe case-sensitive) alle loro refenze remote, cioè ai loro stub.

- In alternativa al registro RMI, esistono altri servizi di naming COSnaming, che vedremo in seguito.

- L'utilizzo di un registro RMI avviene in tre passi:

  - L'host in cui gira il server remoto si occupa di lanciare il registro RMI (che è esso stesso un server remoto), o da linea di comando attraverso il comando `rmiregistry`, oppure direttamente da programma;
  - Il medesimo host si occupa di registrare i server remoti (associando nomi a referenze remote) nel server RMI;
  - Il client accede il registro per ricavare le referenze remote ai server con cui intende interagire.

- Abbiamo detto che il registro RMI è esso stesso un server remoto. Ma allora siamo al punto di partenza? Il client deve comunque ricavare una referenza remota per accedere al registro RMI?

- In realtà i registri RMI sono server remoti particolari a cui si può accedere conoscendo solo l'IP dell'host su cui sono in esecuzione ed il numero di porta su cui sono in ascolto.

La classe java.rmi.Naming

RMI fornisce la classe `java.rmi.Naming` per interagire con un registro RMI preventivamente lanciato da linea di comando col comando `rmiregistry`. La classe `Naming` contiene i seguenti 5 metodi:

```
public static void bind(String nome, Remote obj) throws
       RemoteException, AlreadyBoundException,
       MalformedURLException;

public static void rebind(String nome, Remote obj) throws
       RemoteException, MalformedURLException;

public static void unbind(String nome) throws
```

```
RemoteException, NotBoundException;
MalformedURLException;
```

```
public static String[] list(String nome) throws
        RemoteException, MalformedURLException;

public static Remote lookup(String nome) throws
        RemoteException, NotBoundException,
        MalformedURLException;
```

- Si noti che poichè i metodi sono statici, ovvero di pertinenza della classe, non è quindi necessario creare un'istanza di `java.rmi.Naming` per invocarli.

- Questi metodi si dividono in due categorie: quelli invocati dal codice di installazione presso il server, e quelli invocati dal client.

- I metodi `bind()`, `rebind()`, `unbind()` vengono invocati dal codice di set-up del server per inserire o rimuovere (referenze a) servers nel registro RMI. Il parametro `nome` è una combinazione della locazione del registro RMI e del nome logico del server. Più precisamenete è un URL con la seguente sintassi:

`//registryHost:port/logical_name`

dove:

- `registryHost` è l'host su cui è in esecuzione il registro; tale informazione è opzionale e può essere omessa[2]
- `port` è la porta dell'host dove è in esecuzione il registro; anche tale informazione è opzionale e può essere omessa, in tal caso si assume che il registro stia alla porta di default 1099.
- `logical_name` è il nome del server remoto che si vuole registrare, ad esempio `HelloServer`, tale nome ovviamente deve essere univoco.

Dopo aver fatto il parsing del parametro `nome` tali metodi instaurano una connessione via socket con il registro RMI indicato. Il registro RMI è anch'esso un server remoto in ascolto sulla porta indicata. Nei metodi `bind()` e `rebind()` il `nome` del server ed il

[2] In realtà, come vedremo, tale informazione è superflua.

suo stub `obj` vengono passati al registro RMI previa serializzazione dello stub. Il registro RMI deserializza lo stub e memorizza la coppia (`logical_name`, stub) al suo interno.

- Tutti questi metodi possono lanciare `MalformedURLException` nel caso in cui il parametro `nome` non rispetti la sintassi.
- Possono lanciare `RemoteException` per una serie di ragioni, come ad esempio se non viene trovata nel codebase la classe dello stub deserializzato (`StubNotFoundException`).
- Se si utilizza il metodo `bind()`, ed esiste già nel registro un oggetto col quel `nome`, allora il metodo ritorna un'eccezione `AlreadyBoundException`. In tal caso va utilizzata la `rebind()`.
- La `unbind()` semplicemente rimuove dal registro il server passato per argomento ed emette un'eccezione `NotBoundException` in caso il server non sia presente nel registro

• I metodi `list()` e `lookup()` sono utilizzati dal client per interrogare il registro RMI. Entrambi i metodi, dopo aver fatto un parsing del parametro, aprono una connessione socket verso il registro RMI.

- Più precisamente il metodo `list()` prende come argomento un URL contenenente solo l'indirizzo dell'host ed il numero di porta dove è in esecuzione il registro RMI e fa un'invocazione remota al registro RMI senza passare alcun parametro. Il metodo `list()` ritorna un array di stringhe dei nomi dei server remoti presenti nel registro RMI.

- Il metodo `lookup()`, invece, passa al registro RMI un nome di server remoto e riceve una copia serializzata dello stub associato a quel server remoto nel registro RMI. Tale metodo può tornare un'eccezione `NotBoundException` se il server non è presente nel registro, Nota che tale metodo può tornare anche un'eccezione `StubNotFoundException` se in fase di deserializzazione, durante la `lookup()`, non è possibile caricare dinamicamente, attraverso il *codebase*, la classe dell'oggetto stub ricevuto.

I registri RMI sono server remoti

- Ogni registro RMI non è altro che un server remoto.
- Più precisamente un registro RMI è un server remoto della classe `sun.rmi.registry.RegistryImpl` che implementa l'interfaccia remota `java.rmi.registry.Registry`.
- L'interfaccia `Registry` definisce 5 metodi (non statici!!!) simili a quelli visti per la classe `java.rmi.Naming`:

```
public interface Registry extends Remote {
  public static final int REGISTRY_PORT = 1099;
  public Remote lookup(String nome) throws RemoteException,
         NotBoundException;
  public void bind(String nome, Remote obj) throws
         RemoteException, AlreadyBoundException;
  public void unbind(String nome) throws RemoteException,
         NotBoundException;
  public void rebind(String nome, Remote obj) throws
```

```
        RemoteException;
  public String[] list() throws RemoteException;
}
```

- I metodi sono simili a quelli visti per la classe `Naming`, a meno delle seguenti differenze:
  - Il parametro `nome` in cui viene passato il server non ha una struttura gerarchica ma è una semplice stringa che identifica il nome del server. Ad esempio se scriviamo "//:2333/HelloServer" si intende un server che si chiama in tal modo e non un server che si chiama HelloServer alla porta 2333.
  - La ragione per cui non è necessario indicare la porta, è perchè questi metodi vengono invocati su registri ben precisi, in esecuzione su porte ben precise, e non sulla classe Naming.
  - Per la stessa ragione, al contrario di quanto avviene per la classe Naming, il metodo list non ha parametri.
- **Domanda:** Ma visto che i registri RMI sono server remoti che im-

plementano un'interfaccia remota, per quale ragione è stata definita la classe `Naming` con i suoi metodi statici?

- **Risposta:** Per poter interagire con i registri RMI pur non avendo una loro referenza remota. Infatti, la classe `Naming` funziona da "public mirror" per un qualsiasi registro RMI lanciato da linea di comando attraverso il comando `rmiregistry`. Infatti, poichè i metodi sono statici, si possono invocare direttamente a partire dalla classe senza far uso di una referenza remota al registro RMI.

- I registri RMI si possono comunque creare e gestire da programma.

- La classe `LocateRegistry` serve a gestire registri RMI, ovvero oggetti di tipo `Registry`.

- Ecco alcuni dei metodi statici implementati da `LocateRegistry`:

```
public static Registry createRegistry(int port)
public static Registry createRegistry(int port,
   RMIClientSocketFactory csf, RMIServerSocketFactory ssf)
```

```
public static Registry getRegistry(String host, int port)
public static Registry getRegistry(String host, int port,
      RMIClientSocketFactory csf)
```

- I metodi `create` servono appunto per lanciare (creare) un registro RMI da programma. Tali metodi ritornano un'istanza della classe `RegistryImpl`, ovvero un registro RMI, in esecuzione sulla porta indicata, eventualmente utilizzando i sockets passati tra i parametri.

- I metodi `getRegistry` creano e ritornano una referenza (stub) ad un registro remoto su un certo host ad una certa porta. Il metodo ritornerà uno stub anche nel caso in cui non vi sia alcun registro in esecuzione.

- Una volta ottenuta uno stub ad un dato registro RMI, possiamo invocare i metodi dell'interfaccia `Registry`. Solo allora verrà creata una connessione col registro.

- Se però a fronte di uno stub ritornato da un `getRegistry()` non esiste il registro RMI corrispondente presso il server allora verrà ritornata un'eccezione remota.

- **IMPORTANTE LIMITAZIONE**: attraverso i metodi `LocateRegistry.createRegistry` è possibile creare un solo registro RMI per JVM. Se in una medesima JVM vogliamo utilizzare più registri RMI allora li dovremo lanciare da linea di comando.

Le ragioni per voler creare un'istanza del registro RMI, da programma, nella stessa JVM del server, utilizzando un metodo `createRegistry`, possono essere molteplici:

- Per consentire al registro di condividere l'environment java del server remoto (ad esempio il codebase).

- Se si vuole che il registro termini insieme all'applicazione del server. In tal caso gli stub diventeranno referenze pendenti e potranno essere recuperate dal garbage collector.

- Se si vuole utilizzare nel registro dei socket non standard.

Tipici errori nel lancio del registro RMI

- Il registro RMI deve poter accedere l'interfaccia remota per poter inserire correttamente lo stub al suo interno. Ciò vuol dire che quando viene lanciato deve poter accedere al bytecode dell'interfaccia remota, o direttamente oppure settando appropriatamente il codebase, e mettendo l'interfaccia remota nell'aria condivisa indicata nel codebase.

- In caso di caricamento dinamico degli stub da parte dei clients (come avviene nella maggior parte dei casi) il registro RMI non deve poter accedere alle classi stub generate con rmic, altrimenti il meccanismo di caricamento dinamico degli stub da parte del client non funziona correttamente. Di conseguenza un modo per far funzionare correttamente il vostro server è quello di lanciare il registro in una directory che veda l'interfaccia remota ma non gli stub e digitare:

  - `unnset CLASSPATH` (oppure unsetenv CLASSPATH - per resettare la variabile CLASSPATH).

– `rmiregistry 2378 &` (per lanciare il registro RMI alla porta 2378).

Sicurezza e registri RMI

Si pensi alla situazione in cui un hacker scriva un programma che perlustri Internet in cerca di registri RMI. Ovvero, un programma che tenti di connettersi su ogni porta per ogni host su sui si imbatte. Quando trova un registro RMI il programma invoca il metodo `list()` per ricavare i nomi dei server in esecuzione su quell'host. Dopodichè invoca la rebind per cambiare le referenze remote associate ad ogni server sul registro RMI!!

Questo esempio mostra che se non si effettua una restrizione sull'accesso ad un servizio di naming allora l'host che espleta tale servizio diventa estremamente vulnerabile.

La soluzione a tale problema è stata quella di imporre che le operazioni di `bind()`, `unbind()` e `rebind()` possano essere effettuate solo da processi che girano sulla stessa macchina in cui gira il registro RMI. Indipendentemente dal fatto che si utilizzi `Naming` o `Registry`.