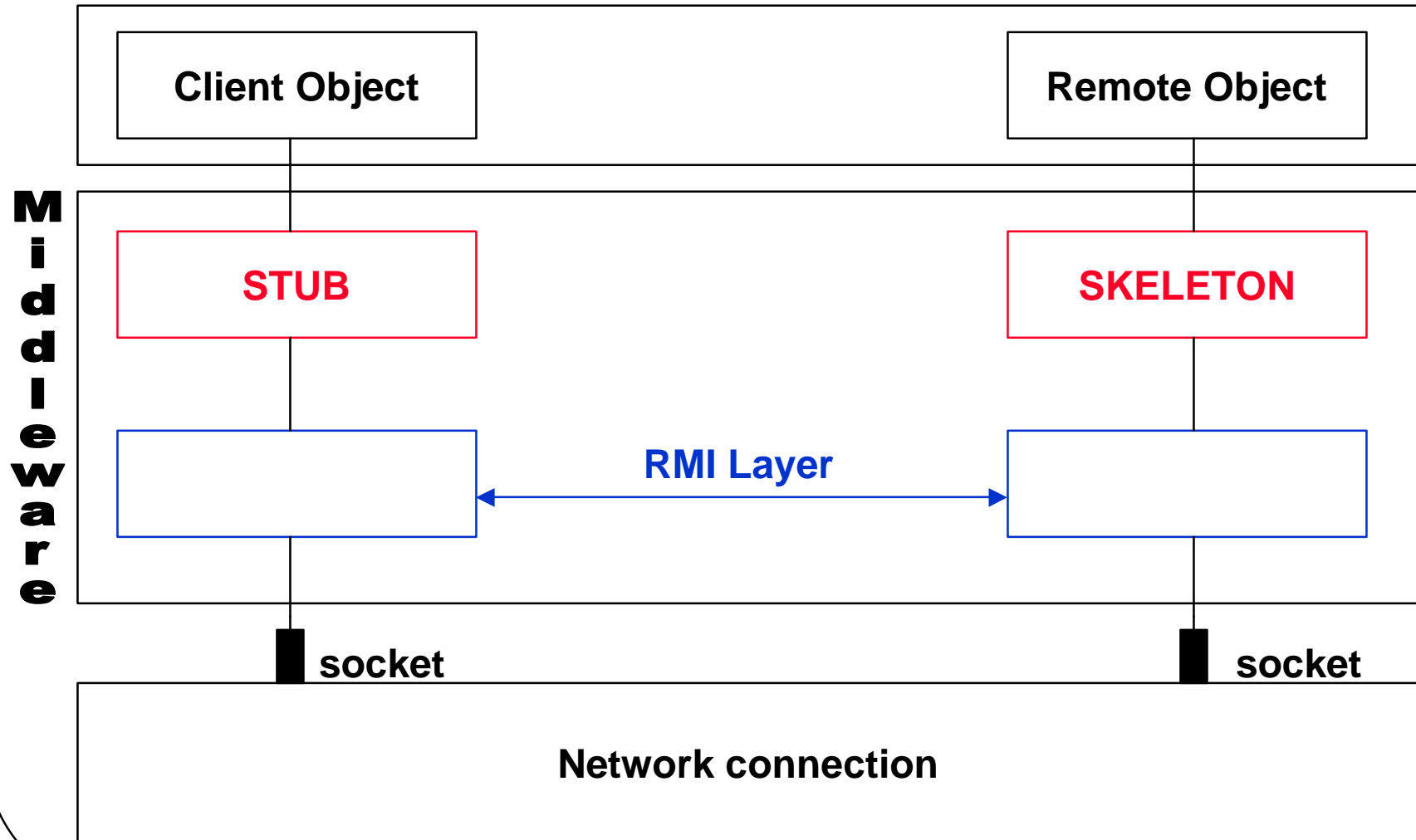


Java RMI (Remote Method Invocation)

- È un meccanismo RPC per la programmazione in Java (invocazione di metodi su oggetti remoti)
- *trasporto*: rete TCP/IP (al momento, **solo TCP**)
- *rappresentazione dei dati*: **Serializzazione di Java**
- *generazione automatica degli stub* (**rmic**)
- *Garbage collection distribuito automatico*
- Documentazione:
 - RMI Specification (inclusa nella documentazione Java SDK)

Il modello RMI



Elementi del Modello

- **Stub:** rappresentazione locale dell'oggetto remoto (surrogato) generata automaticamente
- **Skeleton:** codice per il dispatching delle invocazioni di metodo all'oggetto remoto, generato automaticamente (non più necessario in ambiente Java2)
- **RMI Layer:** strato di middleware RMI comune (indipendente dal tipo di oggetto). Si occupa di:
 - instaurare un collegamento logico fra gli oggetti
 - codificare/decodificare richieste e risposte

Passaggio di Parametri e Valore di ritorno

- Il passaggio di oggetti remoti (quelli esposti tramite RMI) e non remoti avviene diversamente:
- Oggetti non remoti:
 - passati per valore (trasferiti usando la Serializzazione)
 - devono essere oggetti serializzabili
- Oggetti remoti:
 - passati per riferimento (viene trasferito uno stub che permette l'accesso all'oggetto).
- In ogni caso, viene garantita la “referential integrity”

Localizzazione di Oggetti Remoti

- Localizzare un oggetto remoto significa ottenere un riferimento ad esso.
- Questo può avvenire in due modi diversi:
 - interrogando un RMI registry presente sull'host remoto (funziona in modo simile al port mapper, ma fornisce riferimenti ad oggetti remoti ed è basato su nomi)
 - tramite il valore di ritorno di una chiamata ad un altro oggetto remoto di cui si ha già il riferimento
- E' bene limitare l'uso del RMI registry alla registrazione di un singolo oggetto, dal quale si possano ottenere i riferimenti agli altri.

Accesso al Registry

- Il registry è un server RMI che deve essere lanciato:
> `rmiregistry &`
- L'accesso al Registry dai programmi avviene attraverso l'interfaccia `java.rmi.registry.Registry`
- La classe Naming implementa l'accesso al registry per le operazioni di registrazione e lookup.

Registrazione e Lookup

- Il Registry associa ad ogni oggetto registrato un nome locale, che viene interpretato come nome di una risorsa in un url:

rmi://nomehost/nomeoggetto

- Per la registrazione si può usare

`Naming.bind("nomeoggetto")`

- Per il lookup:

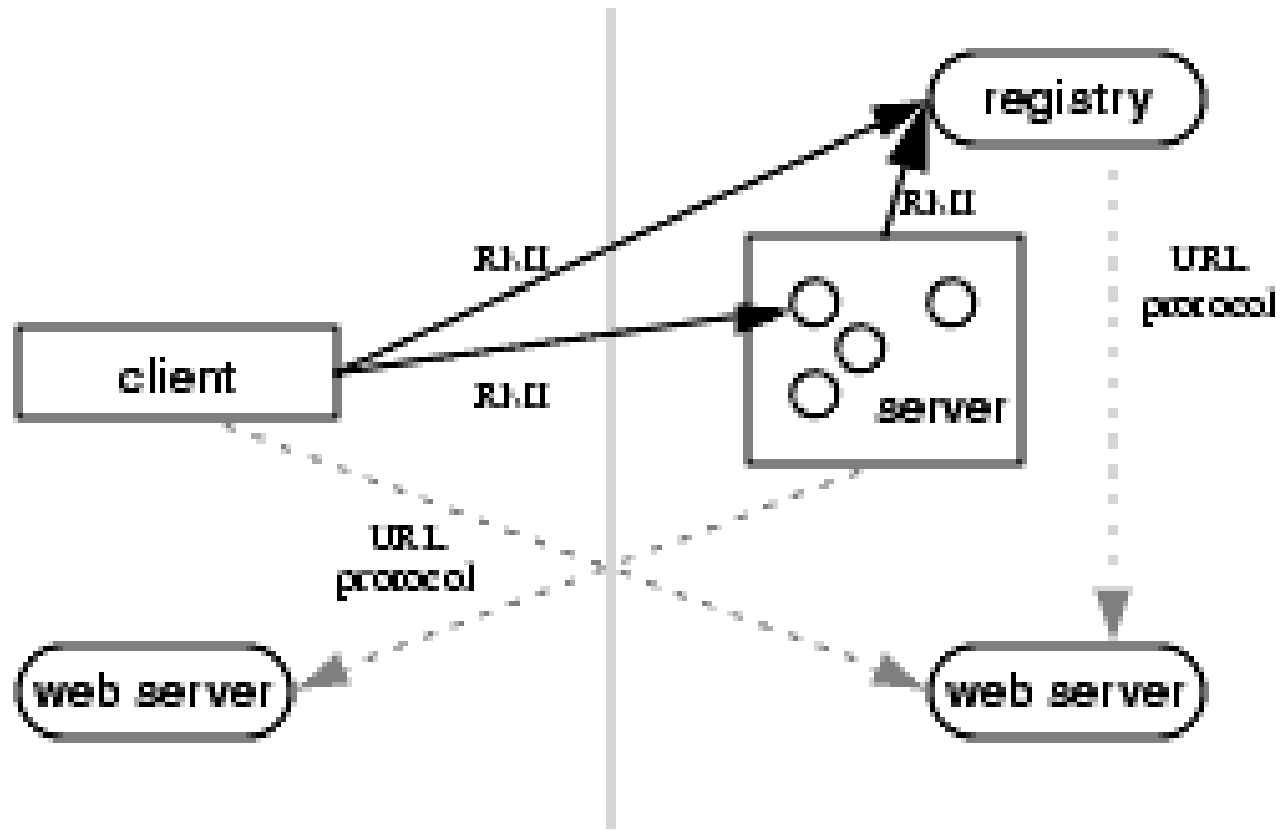
`Naming.lookup("rmi://nomehost/nomeoggetto")`

Download Automatico delle Classi

- Chi riceve oggetti serializzati deve avere accesso alle classi (bytecode) di tutti gli oggetti trasferiti
- Per evitare la complicazione di installare i file .class ad ogni modifica
 - è possibile fare in modo che le classi necessarie vengano scaricate automaticamente da un server http.
 - Trattandosi di codice scaricato via rete, è necessario installare un security manager che consenta questa operazione in modo controllato.
 - La classe di libreria RMI SecurityManager fornisce un esempio di security manager per RMI.

Download Automatico delle Classi

- La serializzazione usata in RMI è stata modificata in modo da includere anche informazioni su come scaricare i file .class



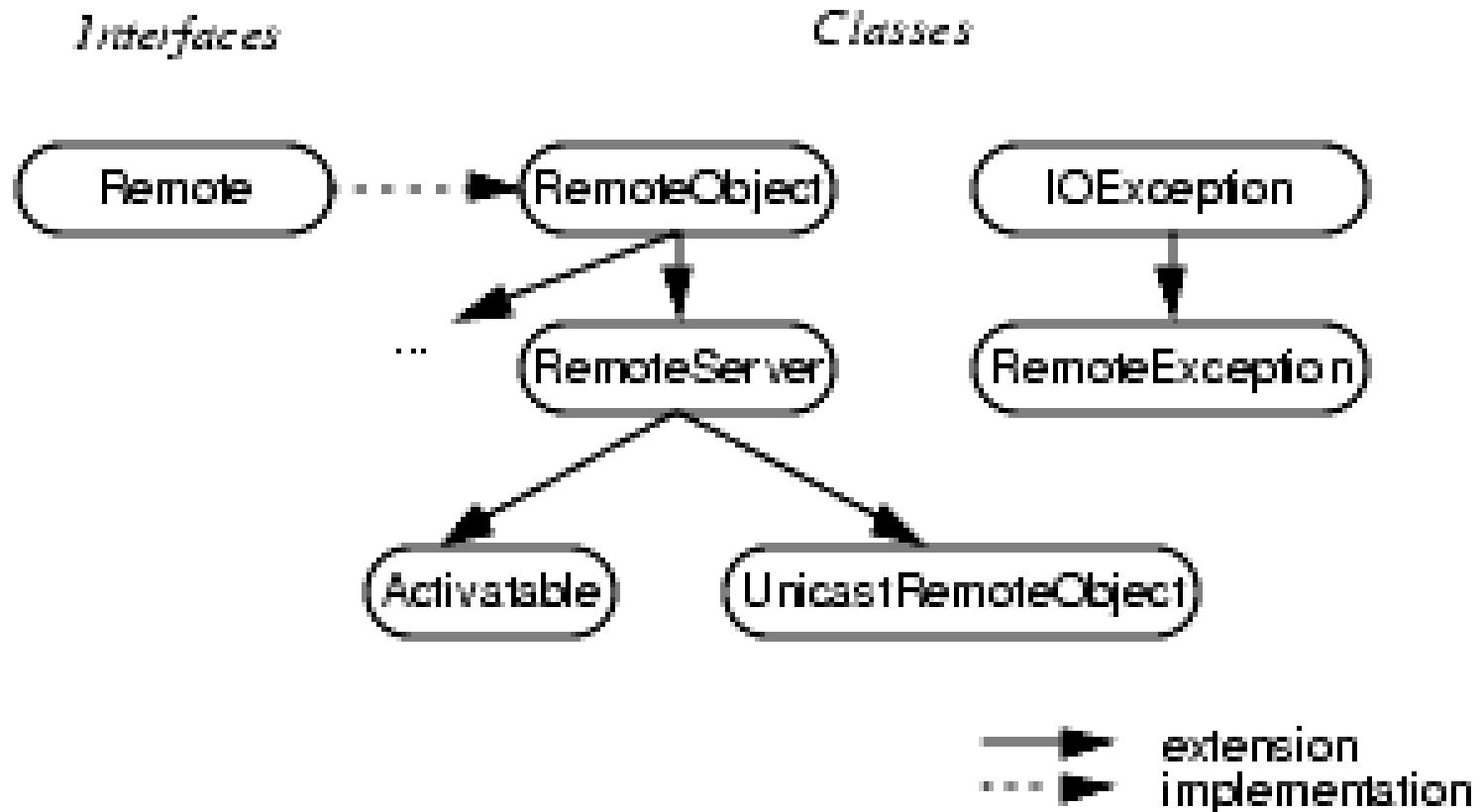
Concorrenza e Sincronizzazione

- La specifica RMI prevede che il server *possa* eseguire le invocazioni di metodi in modalità multi-threaded:
 - i metodi esposti a chiamate remote devono essere thread-safe
 - garantire questo è a carico del programmatore

Generazione di Stub e Skeleton

- Stub e Skeleton vengono generati automaticamente da `rmic`
- A differenza di `rpcgen`, `rmic` non richiede un file di specifica:
 - la generazione viene fatta a partire dal file `.class` della classe che realizza l'oggetto remoto.
 - E' necessario separare interfaccia e implementazione della classe che realizza l'oggetto remoto
 - Il client viene scritto facendo riferimento solo all'interfaccia

Classi e Interfacce della libreria java.rmi



Esempio: Data.java

```
import java.rmi.*;

public interface Data extends Remote {
    public String getData() throws RemoteException;
    public long getTime() throws RemoteException;
}
```

Tutti i metodi devono lanciare
la RemoteException



Esempio: DataImpl.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.text.*;

public class DataImpl extends UnicastRemoteObject implements Data {
    public DataImpl() throws RemoteException {
        super();
    }

    public String getData() throws RemoteException {
        return DateFormat.getDateTimeInstance().format(new Date());
    }

    public long getTime() throws RemoteException {
        return System.currentTimeMillis();
    }
}
```

Esempio: DataServer.java

```
import java.rmi.*;
import java.rmi.server.*;
public class DataServer {
    public static void main(String[] args) {
        try {
            System.out.println("Constructing server impl...");
            DataImpl d = new DataImpl();

            System.out.println("Binding server impl to registry...");
            Naming.rebind("Data",d);

            System.out.println("Waiting for clients...");

        } catch (Exception e) {
            System.out.println("DataServer Error: " + e);
        }
    }
}
```

Esempio: DataClient.java

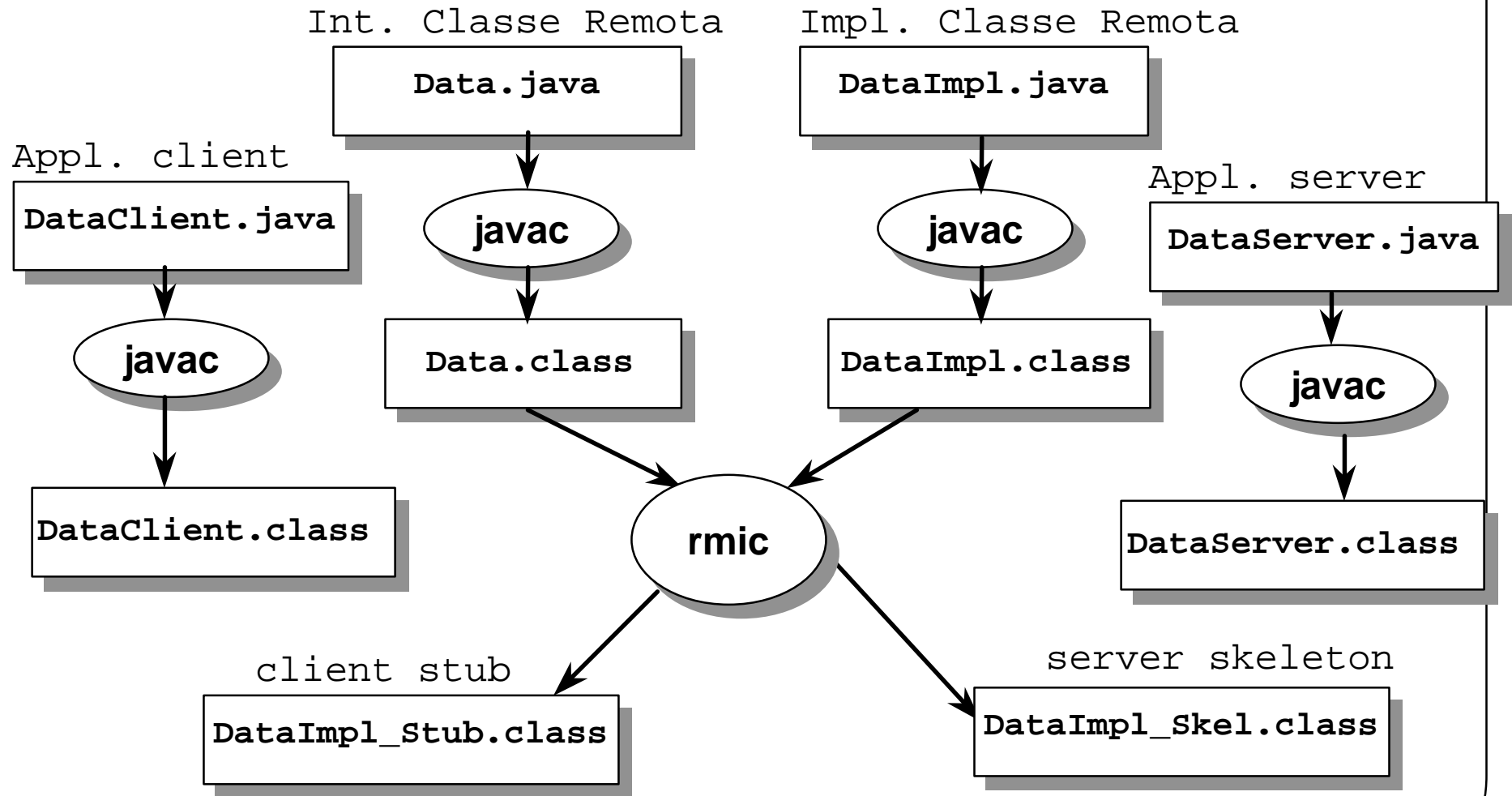
```
import java.rmi.*;
import java.rmi.server.*;
public class DataClient {

    public static void main(String[] args) {
        System.setSecurityManager(new RMISecurityManager());
        String url = "rmi://localhost/Data";
        try {
            Data d = (Data)Naming.lookup(url);
            System.out.println(d.getData());
            System.out.println(d.getTime());
        } catch (Exception e) {
            System.out.println("DataClient Error: " + e);
        }
    }
}
```

Esempio: client.policy

```
grant {  
    permission java.net.SocketPermission  
        "*:1024-65535", "connect";  
};
```

Compilazione



Deployment

- Client
 - Data.class
 - DataClient.class
- Server
 - Data.class
 - DataImpl.class
 - DataServer.class
 - DataImpl_Stub.class
- Web server
 - Data.class
 - DataImpl_Stub.class

Autenticazione

- Non sono previsti meccanismi di autenticazione specifici per RMI
- Tuttavia è possibile attivare RMI su socket SSL
 - I socket SSL sono estensioni dei normali socket
 - Il metodo `exportObject` della classe `UnicastRemoteObject` permette di specificare una `RMIClientSocketFactory` e una `RMISServerSocketFactory` (oggetti che creano i socket da usare)

Estensioni dei socket

- Per usare meccanismi di autenticazione/encryption o compressione su connessioni tramite socket è possibile estendere le classi Socket e ServerSocket
- La libreria javax.net.ssl offre l'implementazione di socket SSL, usabile tramite le classi
 - SSLSocketFactory
 - SSLServerSocketFactory

Esempio: Creazione di un socket (client)

```
try {  
    SSLSocketFactory factory =  
        (SSLSocketFactory)SSLSocketFactory.getDefault();  
    SSLSocket socket =  
        (SSLSocket)factory.createSocket("www.verisign.com", 443);  
    socket.startHandshake();  
} catch (IOException e) {  
    ...  
}
```

Il pattern RMI Factory

- È una tecnica che si può usare per creare oggetti remoti
 - si richiede la creazione dell'oggetto remoto ad un oggetto "fabbrica", anch'esso remoto
 - la fabbrica crea l'oggetto e ne restituisce un reference
 - Poiché il reference risulta di tipo remoto, il chiamante ottiene un reference ad uno stub collegato al nuovo oggetto
- In alternativa si può usare il concetto di oggetto attivabile

Uso di RMI negli Applet