Firewall

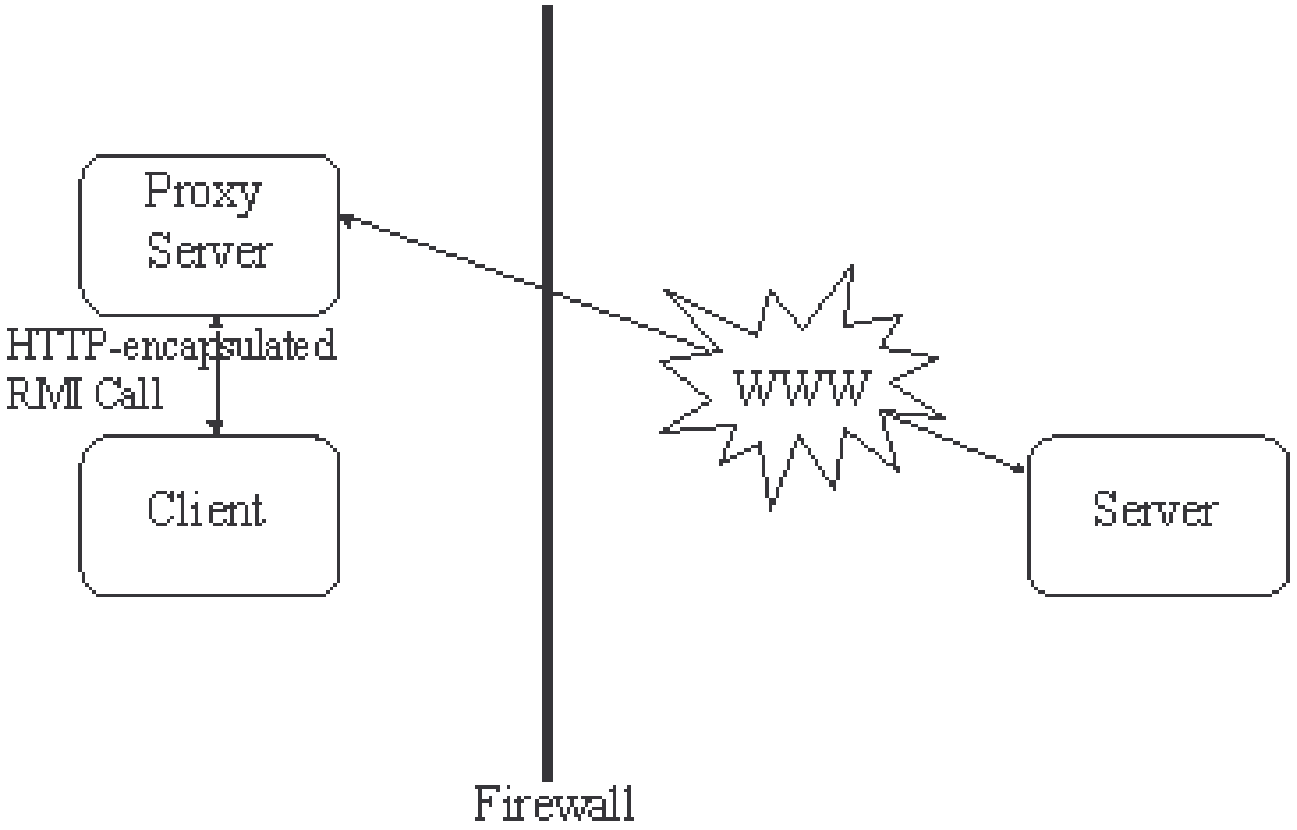
- Ogni applicazione di rete, che opera fuori da quelli che sono i confini di una rete locale, incontra inevitabilmente i cosiddetti *firewall*
- Tipicamente i firewall bloccano tutto il traffico di rete, eccetto quello destinato ad alcune porte generalmente aperte (Es. la porta 80 utilizzata dal protocollo HTTP)
- Siccome RMI transport layer apre connessioni socket dinamiche tra client e server, il traffico è tipicamente bloccato dai firewall .

Nota: Il Transport Layer esegue la connessione vera e propria tra le macchine utilizzando le specifiche standard di networking di Java, e quindi i *socket* con il protocollo *TCP/IP*

RMI e Firewall

- Una soluzione è offerta dal transport layer di RMI stesso
- Per oltrepassare i firewall, RMI si avvale del *tunneling* HTTP, incapsulando la chiamata RMI all' interno del *body* di una richiesta HTTP POST
- Un HTTP Proxy Server deve essere presente nell'infrastruttura lato client
- I possibili scenari:
  - Il client RMI, il server, o entrambi si trovano ad operare dietro un firewall

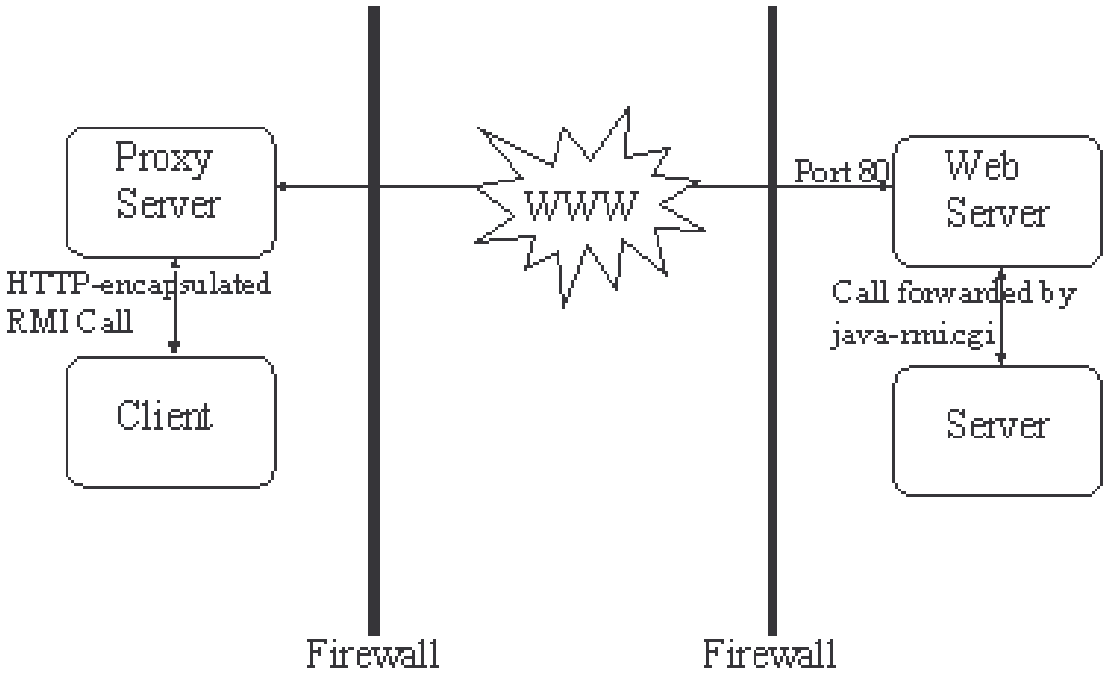
Firewall: il client opera dietro un firewall



Firewall: il client opera dietro un firewall

- Quando il transport layer tenta di stabilire una connessione con il server, viene bloccato dal firewall. Quando accade questo, RMI trasport layer automaticamente riprova, incapsulando i dati della chiamata all'interno del *body* di una richiesta HTTP POST
- La porta usata può essere quella dove RMI server è in ascolto
- L' RMI trasport layer è in ascolto con un server socket che è capace di decodificare e capire la chiamata RMI che si trova all'interno della richiesta HTTP POST
- La risposta è quindi inviata indietro al client sempre tramite HTTP

Firewall: firewall sia lato client che server



- La figura mostra lo scenario di un client e un server RMI che operano entrambi dietro i firewall, o quando il proxy server (lato client) può inoltrare dati al server solo attraverso la porta 80 HTTP

Firewall: firewall sia lato client che server

- Il client, in questo caso, *non può* più inviare le chiamate incapsulate in HTTP a una porta qualsiasi, perchè il server è anche dietro un firewall
- Quindi, il trasport layer RMI incapsula la chiamata nel body di una richiesta HTTP POST e spedisce la richiesta (attraverso il server proxy) alla porta 80 del server *HTTP*
- Questo causa l'esecuzione di uno script CGI, java-rmi.cgi, che a sua volta invoca una JVM localmente al server, la quale prende il contenuto del pacchetto HTTP e inoltra la chiamata al processo RMI server sulla porta designata.

  http://hostname:80/cgi-bin/java-rmi?forward=<port>

Firewall

- **Le risposte del server sono inviate indietro come pacchetti HTTP REPLY verso la porta client originaria, dove vengono estratte le informazioni e inviate verso lo Stub RMI appropriato**

- **Note:**
  - **Piuttosto che fare uso di script CGI per l' inoltro delle chiamate verso il server RMI, sarebbe più efficiente usare le Servlet**
  - **Usando il tunneling si ha un significativo degrado delle prestazioni**
  - **Non si può usare il meccanismo delle callback in presenza di firewall in quanto siamo costretti ad usare il tunneling. In pratica non si può effettuare una chiamate di un metodo remoto verso un client che si trova ad operare dietro un firewall.**

Distributed Garbage Collection

- Uno degli aspetti più comodi della piattaforma Java è il non preoccuparsi dell'allocazione di memoria
- La JVM ha un garbage collector (spazzino) che si preoccupa di liberare memoria attraverso l'eliminazione degli oggetti che non sono più usati da un programma in esecuzione
- Progettare un garbage collector efficiente su una singola macchina è difficile; progettare un garbage collector distribuito è molto difficile
- RMI fornisce un garbage collector distribuito basato sulla tecnica del *reference counting*

Distributed Garbage Collection

- Questo sistema funziona facendo sì che il server tenga traccia dei client che hanno un riferimento attivo all' oggetto remoto in esecuzione sul server
- Quando un oggetto remoto viene referenziato, il server marca l'oggetto come "dirty", mentre quando un client rilascia il riferimento, l'oggetto è marcato come "clean".
- I meccanismi per il DGC (distributed garbage collector) sono nascosti all'interno dello stub (skeleton) layer
- Comunque, un oggetto remoto può implementare l'interfaccia java.rmi.server.Unreferenced (metodo unreferenced), tramite la quale può ricevere notifiche se non c'è più alcun client che abbia un riferimento attivo

Distributed Garbage Collection

- In aggiunta al meccanismo “reference counting ”, un riferimento client attivo ha uno specifico *lease time*
- Se un client non aggiorna la connessione all’oggetto remoto prima che scada il lease time, il riferimento è considerarto non attivo (morto) e l’oggetto remoto può essere distrutto
- Il lease time è controllato attraverso la proprietà di sistema java.rmi.dgc.leaseValue
- Questo valore è in millisecondi e di default è impostato a 10 minuti
- A causa di questi meccanismi un client deve essere preparato a trattare con oggetti remoti che non esistono

Esempio DGC

- Ci sono due oggetti remoti, Hello e MessageObject
- Le implementazioni di questi oggetti sono destinate a stampare informazioni quando questi oggetti vengono creati, distrutti, non referenziati e finalized
- MessageObjectImpl e HelloImpl implementano il metodo *finalize*. Questo metodo viene chiamato un istante prima che il garbage collector locale distrugga l'oggetto per liberare il suo spazio di memoria. Con questo metodo rilascia qualsiasi risorsa (non solo memoria) in possesso dell' oggetto (es. uno stream)

DGC Example

- L'esempio può essere eseguito con il settaggio esplicito della dimensione dell' heap java e il valore del "leaseValue" per il DGC.

```
java –Xmx512m –Djava.rmi.dgc.leaseValue=10000 RMIServer
```

dove l' unità di tempo per il *leaseValue* è in millisecondi

DGC Example: interface Hello

```
import java.rmi.*;
public interface Hello extends java.rmi.Remote
{
   String sayHello() throws RemoteException;
   MessageObject getMessageObject() throws RemoteException;
}
```

DGC Example: HelloImpl

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloImpl extends UnicastRemoteObject
   implements Hello, Unreferenced
{
   public HelloImpl() throws RemoteException
    {   super();
    }
    public String sayHello() throws RemoteException
    {   return "Hello!";
    }
    public MessageObject getMessageObject()
        throws RemoteException
    {   return new MessageObjectImpl();
    }
   [to be continued…]
```

DGC Example: HelloImpl

```
public void unreferenced()
 {     System.out.println( "HelloImpl: Unreferenced" );
 }
public void finalize() throws Throwable
 {     super.finalize();
       System.out.println( "HelloImpl: Finalize called" );
 }
} // class HelloImpl
```

DGC Example: interface MessageObject

```
import java.io.*;
import java.rmi.server.*;
public interface MessageObject extends java.rmi.Remote
{
   int getNumberFromObject() throws
                              java.rmi.RemoteException;
   int getNumberFromClass() throws
                              java.rmi.RemoteException;
}
```

DGC Example: MessageObjectImpl

```
public class MessageObjectImpl extends UnicastRemoteObject
   implements MessageObject, Unreferenced
{
   static  int number = 0;
   static int totalNumber = 0;
   private int objNumber;
   public MessageObjectImpl() throws RemoteException
   {
      objNumber = ++number;
      totalNumber++;
      System.out.println( "MessageObject:
           Class Number is #" + totalNumber +
                    " Object Number is #" +  objNumber );
    }
   public int getNumberFromObject()
   {
      return objNumber;
    }
```

[to be continued…]

DGC Example: MessageObjectImpl

```
public int getNumberFromClass()
{ return totalNumber;
}

public void finalize() throws Throwable
{ super.finalize();
  totalNumber--;
  System.out.println( "MessageObject: Finalize for
                                    object #: " + objNumber );
}
public void unreferenced()
{
  System.out.println( "MessageObject: Unreferenced for
                                    object #: " + objNumber );
}
} // class MessageObjectImpl
```

DGC Example: RMIServer

```
import java.net.*;
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.LocateRegistry;
public class RMIServer
{  private static final int PORT = 10007;
   private static final String HOST_NAME = "…";
   private static RMIServer  rmi;

   public static void main ( String[] args )
   {
     try
     {
       rmi = new RMIServer();
     }
         [to be continued…]
```

DGC Example: RMIServer

```
catch ( java.rmi.UnknownHostException uhe )
        { System.out.println( "The host computer name you
              have specified, " + HOST_NAME + " does not
                  match your real computer name." );
        }
        catch ( RemoteException re )
        { System.out.println( "Error starting service" );
          System.out.println( "" + re );
        }
        catch ( MalformedURLException mURLe )
        { System.out.println( "Internal error" + mURLe );
        }
        catch ( NotBoundException nbe )
        { System.out.println( "Not Bound" );
          System.out.println( "" + nbe );
        }
}// main
        [to be continued…]
```

DGC Example: RMIServer

```
public RMIServer() throws RemoteException,
                  MalformedURLException, NotBoundException
   {
     LocateRegistry.createRegistry( PORT );
     System.out.println( "Registry created on host computer" +
           HOST_NAME + " on port " + Integer.toString(PORT) );
     Hello hello = new HelloImpl();
     System.out.println( "Remote Hello service
                        implementation object created" );
     String urlString = "//" + HOST_NAME + ":" +
           PORT + "/" + "Hello";
     Naming.rebind( urlString, hello );
     System.out.println( "Bindings Finished, waiting for
           client requests." );
     }
}  // class RMIServer
```

DGC Example: RMIClient

```
import java.util.Date;
import java.net.MalformedURLException;
import java.rmi.*;
public class RMIClient
{  private static final int PORT = 10007;
   private static final String HOST_NAME = "name";
   private static RMIClient  rmi;
   public static void main ( String[] args )
   {  rmi = new RMIClient();
   }
   public RMIClient()
   {  try

      {
         Hello hello = (Hello)Naming.lookup( "//" +
         HOST_NAME + ":" + PORT + "/" + "Hello" );

               [to be continued…]
```

DGC Example: RMIClient

```
   System.out.println( "HelloService lookup successful" );
   System.out.println( "Message from Server: " +
                       hello.sayHello() );
   MessageObject mo;
      for ( int i = 0; i< 1000; i++ )
      {
         mo = hello.getMessageObject();
         System.out.println( "MessageObject: Class
               Number is #" + mo.getNumberFromClass() +
                     " Object Number is #" +
                           mo.getNumberFromObject() );
         mo = null;
         Thread.sleep(500);
      }
    }
    catch ( Exception e){System.out.println(e);}
 }
}  // class RMIClient
```

SocketFactory

- La comunicazione tra oggetti distribuiti potrebbe avvenire attraverso Socket differenti da quelli di default, ad esempio per criptare o comprimere i dati inviati.
- E' possibile associare i propri Socket di comunicazione ad un oggetto remoto quando viene esportato

- Interfaccia RMIClientSocketFactory

Socket	createSocket (String host, int port)

Crea un client socket per la connessione all' host e alla porta specificati .

- Interfaccia RMIServerSocketFactory

ServerSocket	createServerSocket (int port)

Crea un server socket sulla porta specificata (port=0 porta anonima)

RMIClientSocketFactory interface

- Un' istanza di RMIClientSocketFactory è usata da RMI in modo da ottenere un client socket
- Un oggetto remoto può essere associato con un' istanza di questa classe quando viene creato o esportato attraverso il costruttore o attraverso il metodo exportObject di java.rmi.server.UnicastRemoteObject (anche attraverso java.rmi.activation.Activatable)
- Un' istanza di RMIClientSocketFactory verrà usata per creare connessioni ad oggetti remoti sui quali effettuare chiamate a metodi remoti

RMIClientSocketFactory interface

- Un' implementazione di questa interfaccia dovrebbe essere serializzabile e dovrebbe implementare Object.equals(java.lang.Object) per ritornare vero quando viene passata un' istanza che rappresenta lo stesso (funzionalità equivalenti) client socket factory, e falso altrimenti
- Inoltre, dovrebbe implementare anche Object.hashCode()
- Un' istanza di RMIClientSocketFactory può anche essere associata ad un oggetto remoto registry in modo che i clients possano usare un particolare socket di comunicazione con un oggetto registry

RMIServerSocketFactory interface

- Un' istanza di RMIServerSocketFactory associata ad un oggetto remoto è usata per ottenere il ServerSocket usato per accettare chiamate dai clients

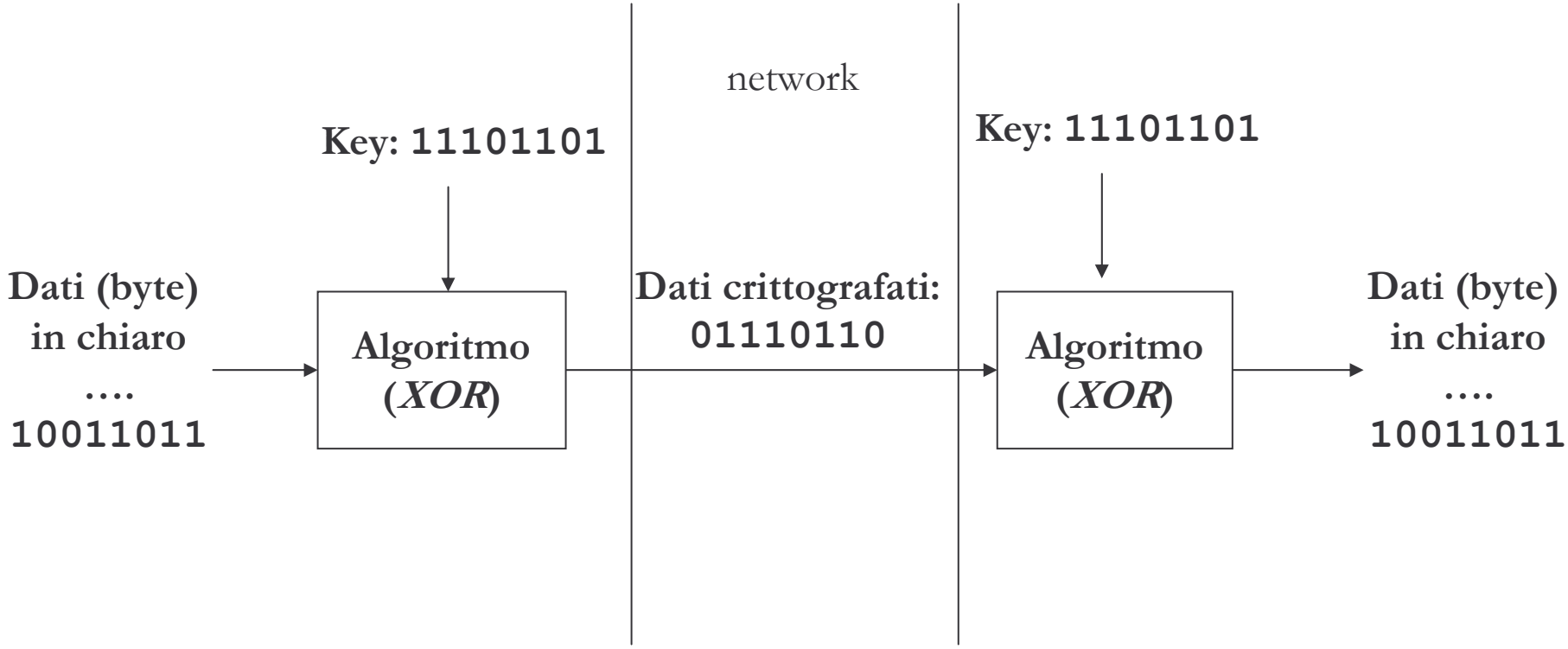
- Un' istanza di RMIServerSocketFactory può anche essere associata ad un oggetto remoto registry in modo che i clients possano usare un particolare socket di comunicazione con un oggetto registry

Custom RMI Socket Factory

- L'implementazione e l'uso di una custom RMI socket factory dovrebbe essere usata quando:
  - Client e Server RMI hanno bisogno di usare socket che criptano o comprimono i dati
  - Le applicazioni richiedono differenti tipi di socket per differenti oggetti remoti
- Prima della JavaTM 2 SDK, v1.2, era possibile creare e installare sottoclassi di java.rmi.server.RMISocketFactory usandole globalmente per tutte le connessioni create da RMI.
- Non era possibile comunque associare una differente socket factory su diversi oggetti che operavano sulla stessa JVM

Custom RMI Socket Factory: esempio

Per questo esempio, la nostra RMI socket factory creerà socket che useranno la *crittografia XOR*



Esempio: Custom RMI Socket Factory

- Questo tipo di crittografia proteggerà i dati da eventuali intrusioni durante la comunicazione tra due oggetti remoti. Ovviamente esistono altre tecniche per crittografare i dati ……!!
- I socket XOR usano una particolare implementazione di output e input stream per manipolare i dati scritti e letti dai socket

XorInputStream

```
import java.io.*;
public class XorInputStream extends FilterInputStream
{ private final byte pattern;

   public XorInputStream(InputStream in, byte pattern)
   {   super(in);
       this.pattern = pattern;
   }

    public int read() throws IOException
   {   int b = in.read();
       if (b != -1)
              b = (b ^ pattern) & 0xFF;
        return b;
   }

    [to be continued…]
```

XorInputStream

```
public int read(byte b[], int off, int len)
                              throws IOException
   {    int numBytes = in.read(b, off, len);
        if (numBytes <= 0)
           return numBytes;
        for(int i = 0; i < numBytes; i++)
           b[off + i] = (byte)((b[off + i] ^ pattern)& 0xFF);
        return numBytes;
     }
} // class XorInputStream
```

XorOutputStream

```
import java.io.*;
public class XorOutputStream extends FilterOutputStream
{  private final byte pattern;

    public XorOutputStream(OutputStream out, byte pattern)
   {    super(out);
        this.pattern = pattern;
    }

    public void write(int b) throws IOException
   {    out.write((b ^ pattern) & 0xFF);
    }
}
```

XorServerSocket

```
import java.io.*;
import java.net.*;
public class XorServerSocket extends ServerSocket
{  private final byte pattern;

   public XorServerSocket(int port, byte pattern)
       throws IOException
   {   super(port);
        this.pattern = pattern;
     }

   public Socket accept() throws IOException
   {   Socket s = new XorSocket(pattern);
        implAccept(s);
        return s;
     }
}
```

XorSocket

```
import java.io.*;
import java.net.*;
public class XorSocket extends Socket
{  private final byte pattern;
   private InputStream in = null;
   private OutputStream out = null;

    public XorSocket(byte pattern)
                throws IOException
    {   super();
                this.pattern = pattern;
    }

    public XorSocket(String host, int port, byte pattern)
                throws IOException
    {   super(host, port);
                this.pattern = pattern;
    }
   [to be continued…]
```

XorSocket

```
 public synchronized InputStream getInputStream()
        throws IOException
   {    if (in == null)
                in = new XorInputStream(super.getInputStream(),
                        pattern);
         return in;
    }

    public synchronized OutputStream getOutputStream()
        throws IOException
   {    if (out == null)
                out = new XorOutputStream(super.getOutputStream(),
                        pattern);
         return out;
    }
} // class XorSocket
```

XorClientSocketFactory

```
import java.io.*;
import java.net.*;
import java.rmi.server.*;
public class XorClientSocketFactory
    implements RMIClientSocketFactory, Serializable
{  private final byte pattern;
    public XorClientSocketFactory(byte pattern)
   {    this.pattern = pattern;
    }

    public Socket createSocket(String host, int port)
        throws IOException
    {   return new XorSocket(host, port, pattern);
    }
    [to be continued…]
```

XorClientSocketFactory

```
 public int hashCode()
  {   return (int) pattern;
    }
    public boolean equals(Object obj)
  {   return (getClass() == obj.getClass() &&
              pattern == ((XorClientSocketFactory)
                     obj).pattern);
    }
} //class XorClientSocketFactory
```

XorServerSocketFactory

```
import java.io.*;
import java.net.*;
import java.rmi.server.*;
public class XorServerSocketFactory
    implements RMIServerSocketFactory
{
   private byte pattern;
   public XorServerSocketFactory(byte pattern)
   {   this.pattern = pattern;
    }

    public ServerSocket createServerSocket(int port)
        throws IOException
    {   return new XorServerSocket(port, pattern);
    }
   public int hashCode() {[Same as client]}
   public boolean equals(Object obj) {[Same as client]}
}
```

interface Hello e HelloImpl

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

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
public class HelloImpl implements Hello
{  public HelloImpl() {}
   public String sayHello()
   {   return  "Hello World!";
    }
   public static void main(String args[])
   {   System.setSecurityManager(new SecurityManager());
       [to be continued…]
```

HelloImpl

```
    byte pattern = (byte) 0xAC;

    try
    {   HelloImpl obj = new HelloImpl();
        RMIClientSocketFactory csf =
                        new XorClientSocketFactory(pattern);
        RMIServerSocketFactory ssf =
                        new XorServerSocketFactory(pattern);
        Hello stub = (Hello) UnicastRemoteObject.
                        exportObject(obj, 0, csf, ssf);
        LocateRegistry.createRegistry(2002);
        Registry registry = LocateRegistry.getRegistry(2002);
        registry.rebind("Hello", stub);
        System.out.println("HelloImpl bound in registry");
    }
    catch (Exception e) {}
    }
} //class HelloImpl
```

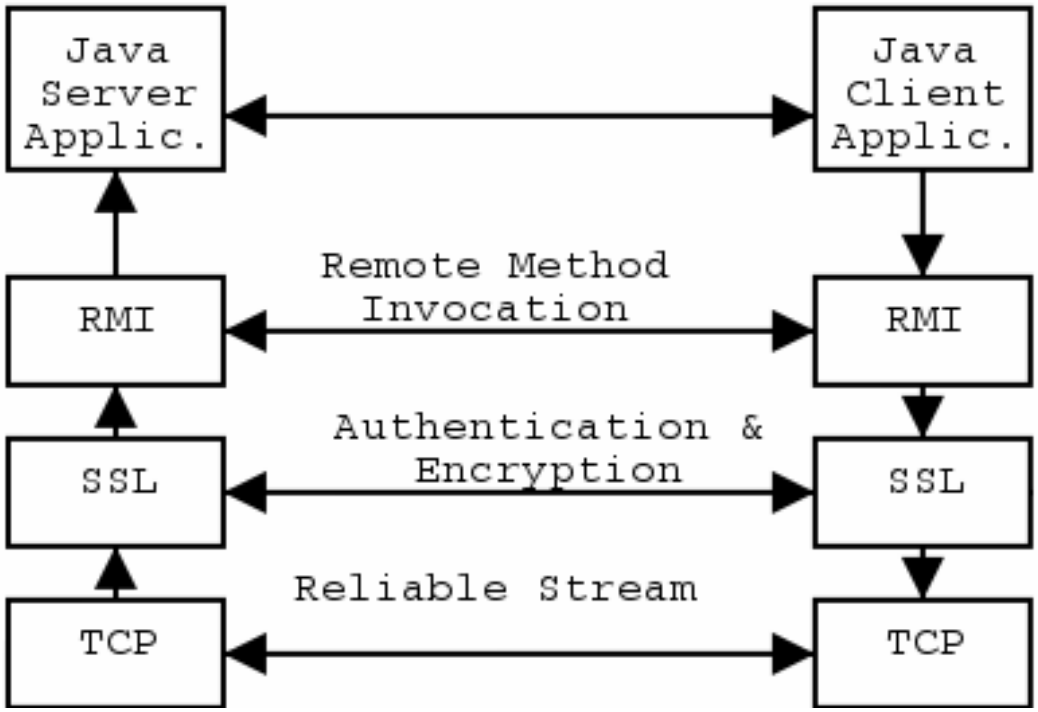
HelloClient

```
import java.rmi.*;
import java.rmi.registry.*;
public class HelloClient
{  public static void main(String args[])
   {    System.setSecurityManager(new SecurityManager());
               try
        {      Registry registry =
                       LocateRegistry.getRegistry(2002);
               Hello obj = (Hello) registry.lookup("Hello");
               String message = obj.sayHello();
               System.out.println(message);
       }
        catch (Exception e)
        {      System.out.println("HelloClient exception: "
   +
               e.getMessage());
               e.printStackTrace();
         }
     }
}
```

RMI e SSL

- È possibile usare i protocolli **SSL** (**Secure Sockets Layer**) e **TLS** (**Transport Layer Security**) in RMI
- Java mette a disposizione degli sviluppatori le librerie **JSSE** (**Java Secure Socket Extension**) cioè un implementazione java di SSL e TLS
- JSSE fornisce una serie di funzionalità per la crittografia dei dati e l'autenticazione nelle comunicazioni internet
- In RMI si possono utilizzare i socket di JSSE per le comunicazioni

RMI e SSL


Java
Server
Applic.
Java
Client
Applic.
Remote Method
Invocation
RMI
RMI
Authentication &
Encryption
SSL
SSL
Reliable Stream
TCP
TCP