Sockets

Nelle pagine seguenti richiameremo il concetto di socket in Java.

- I *Socket* sono un'astrazione attraverso cui possono comunicare due programmi in esecuzione su macchine differenti trasmettendosi dati attraverso flussi (*streams*) in input ed in output.

- I socket, ovviamente, non sono parte di RMI ma sono gestiti in Java attraverso un altro pacchetto: `java.net`.

- RMI usa i socket Java per gestire la comunicazione tra processi; ecco perchè è importante avere un'idea di come funzionino.

- I Socket fecero la loro prima comparsa a metà degli anni '70 sotto Unix. Nei successivi 30 anni, i socket sono divenuti uno strumento fondamentale della programmazione distribuita.

- Il pacchetto `java.net` contiene due classi principali per la comunicazione su TCP/IP: `Socket` e `ServerSocket`.

- Queste due classi vengono usate nel seguente modo:
  - `Socket` abilita una singola comunicazione tra due processi che si conoscono. Per poter scambiare informazioni, entrambi i programmi devono creare un'istanza di `Socket`.
  - `ServerSocket`, gestisce presso il server, la connessione iniziale tra client e server. Cioè, quando un client si connette ad un server usando un'istanza di `Socket`, in realtà comunicherà inizialmente con un `ServerSocket` del server. `ServerSocket` crea immediatamente un'istanza di `Socket` e lo assegna al client.

  In un certo qual senso `Socket` è simile ad una linea telefonica mentre `ServerSocket` è simile ad un centralino che crea manualmente una linea per consentire la comunicazione tra server e client.

- Per *creare* una connessione di rete via socket con una macchina remota sono necessarie due informazioni: l'*indirizzo IP* e il numero della *porta* usata dalla macchina remota.

- L'indirizzo IP può essere in formato simbolico, come `www.oreilly.com`, oppure in numerazione ottale, come ad esempio `204.148.40.9`.

- La porta è una sorta di indirizzo relativo (cioè locale alla macchina) ed è rappresentato da un intero minore o uguale di 65535.

Sockets dal punto di vista del Client

- La procedura attraverso cui un client Java usa un socket consiste in tre passi:
  - Si crea un socket a partire dall'IP e dalla porta associata al server.
  - Si ricavano i flussi in input (`InputStream`) ed output (`Outputstream`) associati al socket. L'`InputStream` è usato per ricevere dati mentre l'`Outputstream` serve per spedire dati.

– Si chiude il socket quando terminato. Quest'ultimo passo consente al server di disallocare le risorse allocate per la comunicazione col client.

Vediamo un esempio (molto semplificato) di un'applicazione *WebBroswer* che tenta di recuperare la pagina web principale presso un *WebServer* su una macchina remota.

```
public class WebBroswer {
// ... dichiarazioni varie
  private void askForPage(Socket webServer) throws IOException {
    BufferedWriter request;
    request = new BufferedWriter(new OutputStreamWriter(webServer.
                                   getOutputStream()));
    request.write(``GET / HTTP/1.0'');
    request.flush();
  }
```

```
  private void receivePage (Socket webServer) throws IOException {
    BufferedReader webPage=null;
    webPage = new BufferedReader(new InputStreamReader(webServer.getInputStream()));
    String nextLine;
    while (null!=(nextLine=webPage.readLine())) {
       System.out.println(nextLine);
    };
    return;
}
...............
  public void main (URL url) {
    Socket webServer;
    webServer = new Socket(url, 80);
    try {
        askForPage(webServer);
        receivePage(webServer);
        webServer.close();
        }
    catch (IOException ex) { .......}
```

```
}
```

- L'applicazione assume che il WebServer sia in attesa di connessioni sulla porta 80. Usando questa informazione, il client apre un socket verso il webserver, chiede una pagina, e riceve la pagina in risposta. Dopo aver visualizzato la pagina il client chiude il socket.

- Come si può notare il client richiede una pagina al webserver e poi riceve una risposta. Questi due passi sono possibili solo perchè tra client e server vi è un protocollo di comunicazione. Il protocollo usato qui è HTTP. Un protocollo molto semplice in cui il client spedisce, attraverso uno stream, un testo ASCII contenente uno tra 8 possibili messaggi (GET, HEAD, PUT, POST, DELETE, TRACE, CONNECT, OPTIONS).

- HTTP è un esempio di protocollo privo di stato in cui, dopo la ricezione di una risposta, la comunicazione tra client e server è potenzialmente terminata. Il server non conserva alcuna informazione sullo stato del client; successive richieste non possono basarsi su precedenti richieste o risposte.

Sockets dal punto di vista del server

Fino adesso abbiamo visto l'uso dei socket dal punto di vista del client, supponendo la presenza di un server disposto a dialogare su una certa porta. Adesso vedremo le operazioni che deve fare il server per comunicare via socket con uno o più clients. Il codice del server effettua le seguenti operazioni:

- Crea un'istanza di `ServerSocket` indicando la porta su cui quell'istanza di `ServerSocket` ascolterà in attesa di connessioni.

- Invoca il method `accept()` di `ServerSocket`, il quale ha la seguente segnatura:

  ```
  public Socket accept() throws IOException.
  ```

  - Il metodo `accept()` è bloccante; se nessun client si presenta il server si mette in attesa fintantochè non riceve una richiesta da parte di un client.
  - Il metodo `accept()` crea e ritorna un'istanza di `Socket`. L'istanza è creata all'interno del metodo `accept()` per un singolo

client ed incapsula la connessione tra il server e un determinato client.

Il cuore di un WebServer (molto semplificato) potrebbe essere il seguente:

```
public void main() {
  ServerSocket serverSocket;
  try {
       serverSocket = new ServerSocket(80);
      }
  catch (IOException e) {......}
  while (true) {
        try {
             Socket client = serverSocket.accept();
             processClientRequest(client);
             client.close();
}
```

dove

```
private void processClientRequest(Socket client) throws IOException {
  System.out.println(‘‘Cliente connesso alla porta ’’
     + client.getPort() +
     ‘‘ dalla macchina ‘‘ + client.getInetAddress());
  readRequest(client);
  sendResponse(client);
```

Dove `readRequest` e `sendResponse` sono due procedure che si occupano di leggere la richieste fatta dal client e di tornare una risposta utilizzando il socket `client` di tipo `Socket`.

Security Socket Layer

I Socket servono per trasferire dati tra due processi in esecuzione su host differenti. Ogni volta che avviene tale trasfermimento esistono per lo meno tre rischi riguardanti la sicurezza dei dati:

- Durante il trasferimento da un'applicazione all'altra, l'informazione può essere letta da un terzo agente che non è autorizzato a farlo. Il modo tipico di risolvere questo problema è quello di *criptare* l'informazione.

- Sempre durante il trasferimento, l'informazione potrebbe essere inquinata/alterata da un attacker. La soluzione tipica a tale problema è quello di attaccare all'informazione un "secure checksum".

- Infine, quando si passano delle informazioni ad un agente è necessario essere sicuri della sua identità. La soluzione a ciò consiste nell'utilizzare una terza applicazione che si occupi di validare ed autorizzare i partecipanti ad uno scambio di informazioni.

Ovviamente questi sono dei criteri generali: non tutti gli utenti necessitano di essere autenticati e non tutti i dati devono essere criptati. Comunque, esistono delle tecnologie che consentono di utilizzare socket in modo sufficientemente sicuro.
SSL (Secure Socket Layer) è un protocollo standard per autenticare gli utenti ed allo stesso tempo criptare i dati che poi vengono trasferiti attraverso socket standard. SSL ha tre interessanti ed utili caratteristiche:

- È un protocollo pubblico. SSL è stato definito ed implementato da Netscape ma la specifica è pubblica ed è stata oggetto di numerosi e severi controlli.

- È comunemente molto usato. Quasi tutti i linguaggi che usano sockets hanno una libreria SSL. Inoltre è facile definire una versione sicura di un protocollo di comunicazione usando SSL invece di standard Sockets. Ad esempio HTTPS (la versione sicura di HTTP) è definita usando SSL invece di sockets ordinari.

- È sufficientemente sicuro, sebbene non al 100%.

SSL è definito come un protocollo di comunicazione sviluppato sopra socket standard. Più precisamente, in una connessione SSL si crea una connessione su socket oridinari. Questa connessione è utilizzata per stabilire il criterio da seguire per lo scambio di informazione in maniera sicura tra le due (versione SSL, algoritmi di criptaggio, etc). Dopo questa fase di negoziazione, chiamato SSL handshake, si usa il socket per trasmettere i dati in forma criptata.

Custom Socket Factories

- In Java RMI le “socket factory” vengono utilizzate per ridefinire le classi `Socket` e `ServerSocket` al fine di controllare le invocazioni remote a livello di programmazione di rete. Ad esempio, le socket factories possono essere utilizzate per imporre
  - l’autenticazione dei server
  - la codifica dei dati attraverso strumenti crittografici e di compressione,
  - un timeout sulla comunicazione.
- Ovviamente le socket factories non vanno necessariamente utilizzate su tutti i server di una applicazione. Ad esempio, si potrebbe pensare ad un accesso sicuro con protocollo di autenticazione per ottenere la referenza remota (magari attraverso un registro RMI) ad un server e poi non fare ulteriori controlli nel momento in cui la referenza viene usata.

- Le socket factory vengono associate ad un server remoto al momento della sua esportazione. Quando un server remoto viene esportato è possibile specificare sia la “server socket factory” che la “client socket factory”.

- Si noti che uno stub di un server remoto fa riferimento (all’oggetto) client socket factory con cui il server remoto è stato esportato. Quindi il client oltre a ricevere l’oggetto stub riceve anche la client socket factory. Di conseguenza. la la classe client socket factory deve implementare `Serializable` e il suo codice verrà caricato dinamicamente dal client al momento del passaggio dello stub.

- Le classi server socket factory devono implementare l’interfaccia `java.rmi.server.RMIServerSocketFactory` e devono fornire un’implementazione del metodo `createServerSocket` di tale interfaccia.

- Le classi client socket factory devono implementare l’interfaccia `java.rmi.server.RMIClientSocketFactory` e devono fornire un’implementazione del metodo `createSocket` di tale interfaccia.

- I metodi `equals` e `hashCode` ereditati dalla classe `Object` vanno ridefiniti. Infatti la definizione originale del metodo `equals` di `Object` sarebbe troppo restrittiva distinguendo socket factories solo perchè associati a servers differenti. Ciò, ad esempio, non consentirebbe una prassi comune in Java RMI secondo cui un socket su un dato host ed una data porta viene spesso utilizzato da più servers (però nella stessa JVM) o più clients.

- Java RMI supporta varie socket factory. In particolare, a partire dalla versione 5.0, le classi `javax.rmi.ssl.SslRMIClientSocketFactory` e `javax.rmi.ssl.SslRMIServerSocketFactory` per comunicare attraverso i protocolli SSL e TSL usando la Java Secure Socket Extension (JSSE).