Lezione 1 Socket java

0) Introduzione

Modalità esame:

- L'esame consiste in:
    1. un progetto che prevede la realizzazione di un’applicazione Web e una Android che comunicano tra loro;
    2. la stesura della documentazione;
    3. la discussione orale del progetto;
    4. un orale con domande sul programma del corso
- il progetto può essere svolto in gruppi di 2/3 persone. Eventualmente anche da soli;
- la consegna del progetto deve avvenire una settimana prima della data fissata per l'orale. Eventuali deroghe vanno concordate prima con il docente.

Sono disponibili anche progetti più complessi che possono valere come tesi triennale. E' anche possibile proporre al docente un proprio progetto o implementato in un linguaggio diverso. In ogni caso, è necessario accordarsi preventivamente.

Il protocollo TCP/IP prevede che la comunicazione tra due entità sia molto complessa dal punto di vista della sequenza di pacchetti da inviare, della temporizzazione degli degli invii e della gestione degli errori. I **socket** sono stati introdotti nel sistema operativo BSD 3 decenni fa come un'API che permetteva di usare la comunicazione TCP/IP in maniera semplice: come se si trattasse di una semplice accesso ad un file locale. Java supporta un accesso ancora più semplificato e object-oriented aisocket. Questo rende la comunicazione di rete sensibilmente più semplice anche rispetto al C.

1) Client/Server

Oltre ai protocolli a livello trasporto e sottostanti, molto importanti dal punto di vista del programmatore sono i protocolli a livello di applicazione. La prima grossa distinzine tra i protocolli e quella tra:
1) protocolli **Client/Server**;
2) protocolli **Peer To Peer**.
I protocolli Client/Server sono molto più semplici da descrivere e per certi versi da implementare rispetto ai protocolli Peer To Peer. Nei protocolli Client/Server la temporizzazione degli eventi è molto semplice: il **server** rimane perennemente in attesa di richieste da parte dei **client**. Ricordiamo che stiamo parlando di applicazioni quindi sia il client che il server sono due applicazioni (processi) che possono girare sulla stessa macchina o su macchina diverse.
Un esempio di client è il browser che usiamo per navigare nel Web, mentre il relativo server è il processo perennemente in attesa delle nostre richieste di pagine che indichiamo attraverso parte dell'URL. Altri esempi noti di applicazioni che sfruttano il paradigma Client/Server sono: e-mail, ftp, telnet.
I protocolli Peer To Peer hanno una temporizzazione molto più libera, esempi dei quali sono Napster, Gnutella etc. In questa lezione ci occuperemo dei protocolli Client/Server anche se i socket possono essere indipendentemente usati anche per programmare applicazioni non Client/Server. Per fissare le idee definiremo: client il processo che inizia la conversazione; server il processo che aspetta le richieste.
Per i nostri scopi, la più importante differenza tra client e server è che il client può in qualunque istante creare una socket per iniziare una conversazione con un server, mentre un server si deve preparare ad ascoltare in anticipo le possibili conversazioni in arrivo.

2) Porte standard (well-known)

Un client ha bisogno di due informazioni per connettersi a un processo server su Internet:

1) un **indirizzo** IP (o un nome di host per recuperare l'indirizzo IP tramite un server DNS);
2) un numero di **porta** (dato che il protocollo prevede la possibilità di comunicazioni contemporanee anche di tipo diverso, la porta serve al sistema operativo per individuare il processo destinatario della comunicazione stessa).
Una processo server ascolta su una porta predefinita (cioé nota a priori al client) mentre attende una connessione. I client selezionano il numero di porta corrispondente al servizio desiderato.
I numeri di porta sono codificati nelle RFC (Es. Telnet 23, FTP 21, ecc.) ma possono anche essere scelti (quasi) arbitrariamente per nuove applicazioni. In Unix le porte sotto la 1024 non possono essere usate dalle applicazioni degli utenti normali, bisogna essere "root".
Viceversa il numero di porta del client è irrilevante e viene tipicamente assegnato automaticamente dal sistema operativo durante la richiesta di cumunicazione. (Per conoscere il numero di porta creato automaticamente dal sistema possiamo usare una funzione apposita dell'API dei Socket).
Quando una client inizia la comunicazione con un server socket, la richiesta include la specifica della porta e dell'indirizzo del client: così la richiesta raggiunge correttamente non solo la macchina dove risiede il processo server, ma anche l'opportuno processo server.

3) Protocolli con e senza connessione (TCP e UDP)

Oltre a specificare indirizzo e porta il protocollo TCP/IP prevede che si deve specificare anche il tipo di trasporto per la comunicazione sulla rete. I trasporti previsti da TCP/IP sono due:

1. **UDP**: senza connessione (*connectionless*).
2. **TCP**: con connessione (*connection-oriented*);

I due tipi di trasporto offrono comunicazioni molto diversi in termini di prestazioni e affidabilità.

3.1) UDP

Un protocollo connectionless somiglia al servizio postale. Le applicazioni possono inviarsi brevi messaggi, ma non viene tenuta aperta una connessione tra un messaggio e l'altro.
Il protocollo **non** garantisce che la consegna dei dati, e **non** garantisce che questi arrivino nell'ordine corretto.
Per contro, la spedizione di un singolo dato è più efficiente nelle reti locali usando UDP rispetto a TCP ed è possibile inoltre effettuare comunicazioni in broadcast e multicast. Un messaggio trasmesso viene detto *pacchetto*.
In Java la classe `DatagramSocket` usa il protocollo connectionless UDP per inviare le informazioni.

3.2) TCP

Un protocollo connection-oriented offre l'equivalente di una conversazione telefonica. Dopo aver stabilito la connessione, due applicazioni possono scambiarsi dati.
La connessione rimane in essere anche se nessuno parla. Il protocollo garantisce che non vengano persi dati e che questi arrivino sempre nell'ordine corretto. Dato che il protocollo TCP si appoggia al protocollo IP che è inaffidabile, il costo da pagare per l'affidabilità è quello di prestazioni più basse specie nella fase iniziale della connessione (*handshake* e *slow-start*).
Con un connection-oriented protocol, il socket del client stabilisce, alla sua creazione, una connessione con il socket del server. Stabilita la connessione, un protocollo connection-oriented assicura la consegna affidabile dei dati, ovvero:

1. Ogni byte inviato viene consegnato. Ad ogni spedizione, il socket si aspetta di ricevere un acknowledgement che i byte siano stati ricevuti con sucesso. Se la socket non riceve l'acknowledgement entro un tempo prestabilito, il socket invia novamente i byte. Il socket continua a provare finché la trasmissione ha successo, o finché decide che la consegna è impossibile.
2. I byte sono letti dal socket ricevente nello stesso ordine in cui sono stati spediti. Per il modo in cui

la rete funziona, i byte possono arrivare alla macchina destinataria in un ordine diverso da quello in cui sono stati spediti, ma è compito dell'implementazione TCP riordinarli.

Un protocollo affidabile permette alla socket ricevente di riordinare i paccheti ricevuti, così che possano essere letti dal programma ricevente nell'orine in cui erano stati spediti. Al contrario dei messaggi trasmessi dal protocollo UDP, nel protocollo TCP non c'è il concetto di messaggio e anche se la nostra applicazione invia le informazioni in blocchi distinti, il protocollo TCP sono considerati un flusso continuo di informazione e il protocollo TCP può legittimamento unire o spezzare tali blocchi e l'unica garanzia è che viene preservato l'ordine dei singoli byte trasmessi.
La classe Java `Socket` usa il protocollo connection-oriented TCP.

4) Socket

La classe `java.net.Socket` rappresenta un singolo lato di una connessione socket indifferentemente su un client o su un server.
Inoltre, il server usa la classe `java.net.ServerSocket` per attendere connessioni dai client. Un applicazione server crea un oggetto `ServerSocket` e attende, bloccato in una chiamata al suo metodo `accept()`, finché non giunge una connessione. A quel punto, il metodo accept() crea un oggetto `Socket` che il server usa per comunicare con il client.
Un server può mantenere molte conversazioni simultaneamente usando un *solo* oggetto della classe `ServerSocket` e un oggetto della classe `Socket` per ogni client attivo.

4.1) `java.net.Socket`

I constructori della classe creano il socket e lo connettono allo host e porta specificati nel costruttore.
Una volta creato l'oggetto socket, i metodi `getInputStream()` e `getOutputStream()` ritornano oggetti della classe `InputStream` e `OutputStream` che possono essere usati come se fossero canali di I/O su file.
I metodi `getInetAddress()` e `getPort()` restituicono l'indirizzo remoto e la porta remota a cui il socket è connesso.
Il metodo `getLocalPort()` ritorna la porta locale usata dal socket .

4.2) `java.net.ServerSocket`

Questa classe è usata dai server per ascoltare le richieste di connessione. Quando si crea una `ServerSocket`, si specifica su quale porta si ascolta.
Il metodo `accept()` iniza ad ascoltare su quella porta, e si blocca finche' un client non richiede una connessione su quella porta.
A quel punto, `accept()` accetta la connessione, creando e ritornando un oggetto `Socket` che il server può usare per comunicare col client.

5) Client

Un applicazione client apre una connessione con un server costruendo un oggetto `Socket` che specifica hostname e port number del server desiderato:

```
try
{
    Socket sock = new
Socket("www.dsi.unive.it", 80);
    //Socket sock = new
Socket("157.138.20.6", 80);
}
catch ( UnknownHostException e )
```

```
{
// Il nome dell'host non e' valido
    System.out.println("Can't find
host.");
}
catch ( java.io.IOException e )
{
// Si e' verificato un errore di
connessione
    System.out.println("Error connecting
to host.");
}
```

5.1) Read raw byte

Una volta stabilita la connessione, input e output streams possono essere ottenuti con i metodi `getInputStream()`e `getOutputStream()` della classe `Socket` .

```
try
{
    Socket socket = new
Socket("www.dsi.unive.it", 80);
    InputStream in =
socket.getInputStream();
    OutputStream out =
socket.getOutputStream();
    // Write a byte
    out.write(42);
    // Read a byte
    int back = in.read();
    if (back<0) System.out.println("no
byte read");
    else System.out.println("1 byte read:
"+back);
    socket.close();
}
catch (IOException e )
{
  ...
}
```

5.2) Incaspulamento con `DataInputStream` e `PrintStream`

Incapsulando `InputStream` e `OutputStream` è possibile accedere agli streams in modo più semplice. Infatti con questi nuovi oggetti abbiamo a disposizione i metodi per inviare e leggere i più comuni tipi di dati Java.

```
try {
    Socket socket = new Socket("www.dsi.unive.it", 80);
    InputStream in = socket.getInputStream();
    DataInputStream din = new DataInputStream( in );

    OutputStream out = socket.getOutputStream();
    PrintStream pout = new PrintStream( out );

    // Say "Hello" (send newline delimited string)
```

```
    pout.println("Hello!");
    // Read a newline delimited string
    String response = din.readLine();
    socket.close();
} catch (IOException e ) { }
```

5.3) altri metodi dei socket

Tra i vari metodi messi a disposizione dalla classe Socket, ne vediamo ora alcuni dei più utili:

1. `getInetAddress()`: ritorna l'indirizzo IP dell'altro lato della connessione (remoto); Utile ad un server per conoscere l'indirizzo IP del client. Il cliente invece conosce già l'indirizzo del server perché prima di effetturare la connessione deve conoscere IP address e porta del processo server;
2. `getLocalAddress()`: ritorna l'indirizzo IP locale. Potrebbe essere utile ad una macchina con più indirizzi IP (proxy, router, gateway etc.);
4. `getPort()`: ritorna la porta dell'altro lato della connessione; Utile ad un server per conoscere la porta del client;
5. `getLocalPort()`: ritorna la porta locale della connessione; Utile al client per conoscere il numero di porta creato automaticamente dal sistema.

6) Server

Il lato server di un protocollo connection-oriented tipicamente esegue questa sequenza:

1. crea un oggetto della classe `ServerSocket`;
2. si mette in attesa di connessioni tramite il metodo `accept` dell'oggetto appena creato;
3. per ogni nuova connessionerichiesta da un client, il metodo `accept` ritorna un nuovo oggetto di tipo `Socket`;
4. tramite l'oggetto Socket possiamo accedere agli oggetti `InputStream` e `OutputStream` per leggere e scrivere bytes da e verso la connessione;
5. l'oggetto `ServerSocket` può ritornare ad aspettare altre connessioni attraverso il metodo `accept`.

6.1) Accettare connessioni

```
try {
    ServerSocket master = new ServerSocket( 8080 );
    while ( true )
    {
        // wait for connection
        Socket slave = master.accept();

        InputStream in = slave.getInputStream();
        OutputStream out = slave.getOutputStream();
        // Read a byte
        byte res = (byte) in.read();
        // Write a byte
        out.write(43);
        slave.close();
    }
    //master.close();
} catch (IOException e ) { }
```

6.2) Incaspulamento con `DataInputStream` e `PrintStream`

```
try {
    ServerSocket master = new ServerSocket( 8080 );
    while ( true ) {
        // wait for connection
        Socket slave = master.accept();

        InputStream in = slave.getInputStream();
        DataInputStream din = new DataInputStream( in );
        OutputStream out = slave.getOutputStream();
        PrintStream pout = new PrintStream( out );
        // Read a string
        String request = din.readLine();
        // Say "Goodbye"
        pout.println("Goodbye!");
        slave.close();
    }
    //master.close();
} catch (IOException e ) { }
```

7) Esercizi

1. iscriversi al corso su moodle.unive.it con chiave TAW2013;
2. scrivere un client per il servizio di echo;