Java

accesso alla rete -- TCP

G. Prencipe
prencipe@di.unipi.it

Introduzione

- Storicamente, programmare su un sistema di macchine distribuite è sempre stato complesso
- Il programmatore doveva conoscere diversi dettagli sulla rete che collegava le macchine (e a volte anche dettagli legati all'hardware)
- Era necessario conoscere e comprendere i diversi livelli dei protocolli di rete, e c'erano molte differenti funzioni in ogni libreria di rete legate alle connessioni tra le macchine, alla "impacchettamento" e allo "spacchettamento" delle informazioni," all'invio dei pacchetti, ecc

Introduzione

- Comunque, l'idea di fondo del calcolo distribuito non è difficile, ed è astratta molto chiaramente nelle librerie Java
- In genere si vuole
  - Ottenere/inviare informazioni da/su qualche macchina remota
    - Programmazione di rete di base
  - Collegarsi a un database, che può essere distribuito sulla rete
    - *Java DataBase Connectivity* (*JDBC*), che è un'astrazione dai dettagli legati alla piattaforma *dell'SQL* (*Structured Query Language*, utilizzato per la maggiorparte delle transazioni su database)

Introduzione

- Fornire servizi tramite un server Web
  - *Servlet* e *Java Server Pages* (*JSP*)
- Eseguire metodi su oggetti che risiedono su macchine remote come se essi fossero locali
  - *Remote Method Invocation* (*RMI*)

Introduzione -- testi consigliati

- Java -- Network Programming and Distributed Computing
  - David Reilly, Michael Reilly -- Addison Wesley
- Java Networking Programming
  - Elliotte Rusty Harold -- O'Reilly
- An Introduction to Network Programming with Java
  - Jan Graba -- Pearson - Addison Wesley

Programmazione di rete di base

indice degli argomenti

Visione generale

- L'approccio di Java alla rete è piuttosto indolore
  - Infatti, gli accessi alla rete sono molto simili agli accessi ai file
  - La differenza è che il file è remoto, e la macchina remota può decidere cosa fare dell'informazione che stiamo richiedendo o inviando
- Quindi, il modello di programmazione utilizzato per programmare la rete è quello del *file*
  - La connessione di rete viene racchiusa in un oggetto stream, e l'invio e la ricezione di informazioni avvengono utilizzando le stesse chiamate di libreria utilizzate con gli stream

Visione generale

- Intoltre, il multithreading di Java è molto utile quando si devono gestire più connessioni alla volta
- Per quanto possibile, i dettagli "fisici" della rete sono stati astratti e vengono gestiti dalla JVM

Visione generale

- Java usa “naturalmente” la rete:
  - Per caricare le applet
    - Un’applet è una piccola applicazione Java che viene caricata tramite la rete ed eseguita all’interno di un browser web (Netscape, Internet Explorer, Mozilla, ecc.)
    - L’applet viene identificata dal tag <APPLET> (o anche <OBJECT> o <EMBED>) all’interno di una pagina HTML
    - Il codice eseguibile (.class o .jar) dell’applet viene prelevato tramite HTTP da un server web

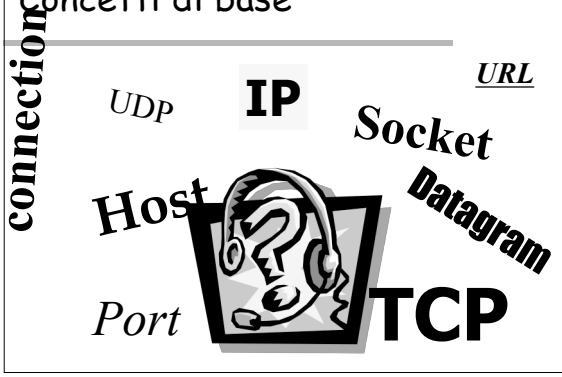
Visione generale

- Java usa “naturalmente” la rete:
  - Per caricare immagini o altre risorse
    - Queste risorse vengono identificate tramite una URL.
    - Se l’URL fa riferimento alla rete (http://…, ftp://…, ecc.), le classi di libreria di Java si occupano di contattare il server remoto e di recuperare i dati necessari.
- In questi casi, il programmatore non si deve preoccupare della rete: accesso **trasparente**!

Visione generale

- Ci sono però applicazioni in cui la rete gioca un ruolo **esplicito**
  - Esempio: un sistema che trasmette le quotazioni di borsa agli abbonati (via rete)
- In questi casi, il programmatore deve saper usare le *classi di libreria* di Java che forniscono il supporto per l’accesso alla rete
- Gran parte di quello che diremo oggi serve ad illustrare queste classi

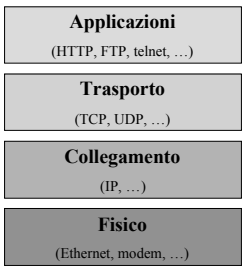
Concetti di base



Concetti di base

- Le applicazioni di rete comunicano fra loro usando (tipicamente) uno di due protocolli:
  - Il **Transmission Control Protocol** (TCP)
  - Lo **User Datagram Protocol** (UDP)

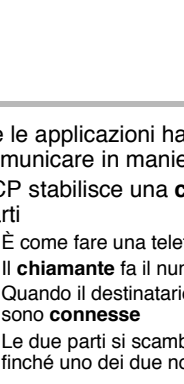
Concetti di base



- I programmi Java si situano fra le **applicazioni**
- Normalmente, non c'è bisogno di studiare I dettagli di TCP & UDP
- Basta usare le classi nel package (di sistema) *java.net* !
- In questo modo, l'applicazione funzionerà su qualunque sistema
  - Anche se poi la rete fisica è di altro tipo!

Concetti di base

- Però…
  - Per decidere *quali* classi usare, è importante capire la differenza fra i vari tipi di connessione disponibile
  - Ecco perché ci interessa conoscere quali sono le alternative fra cui possiamo scegliere



TCP

- Se le applicazioni hanno bisogno di comunicare in maniera **affidabile** e **sicura**
- TCP stabilisce una **connessione** fra le due parti
  - È come fare una telefonata:
  - Il **chiamante** fa il numero del **destinatario**
  - Quando il destinatario alza il telefono, le due parti sono **connesse**
  - Le due parti si scambiano **dati** (avanti e indietro) finché uno dei due non **riattacca**

TCP

- TCP si comporta come una compagnia telefonica (di quelle serie):
  - Garantisce che la chiamata verrà inoltrata al destinatario
  - Garantisce che la comunicazione non cadrà finché uno dei due non riaggancia
  - Garantisce che i dati (come le parole di una conversazione) arriveranno **tutti** dall'altra parte, e **nell'ordine giusto** (quello in cui sono stati inviati)

TCP

- Una connessione TCP è di tipo **punto-punto**: si svolge fra due sole parti, ma è **affidabile**
- HTTP, FTP, telnet, la posta elettronica… tutti usano TCP
  - È importante che i dati vengano ricevuti intatti
    - Tutti i pacchetti inviati arrivano
    - Nessun pacchetto spurio arriva
    - I pacchetti che arrivano sono in ordine
  - Cosa accadrebbe se la comunicazione non fosse affidabile? Caos totale!

TCP

- Per riassumere:

> **Definizione**: *TCP* è un protocollo orientato alla connessione che fornisce un flusso di dati affidabile fra due computer.

UDP

- Se le applicazioni hanno bisogno di comunicare in maniera **veloce**, e l'affidabilità della comunicazione non è importante
- UDP invia pacchetti di dati indipendenti, chiamati *datagrammi*
  - È come spedire una lettera
  - Il **mittente** scrive l'indirizzo e imbuca la lettera
  - **Forse**, **chissà quando**, la lettera arriva al destinatario

UDP

- UDP si comporta come un servizio postale (di quelli seri):
  - Non garantisce nulla (se non che incassa il francobollo)
  - Mediamente, i dati arrivano davvero, e più velocemente di quanto non farebbero con TCP
  - Però non c'è nessuna garanzia: i dati possono perdersi, o arrivare in ritardo, o fuori ordine
  - UDP fa del suo meglio, ma non promette niente

UDP

- UDP non ha connessioni; ogni datagramma viaggia per conto suo (e può avere un destino diverso dai suoi compari)
- Ci sono applicazioni in cui la maggiore velocità è più importante dell'affidabilità:
  - Trasmissioni audio/video via Internet (*iPhone,* teleconferenze)
  - Giochi di simulazione in rete (*Flight Simulator, Doom)*

UDP

- Per riassumere:

> **Definizione**: *UDP* è un protocollo orientato al pacchetto che fornisce mediamente tempi di consegna veloci, ma non offre altre garanzie.

Identificare una macchina

- Dal punto di vista fisico, ogni computer ha (tipicamente) una sola connessione alla rete
  - Scheda di rete
    - Ethernet, Token ring, AppleTalk
    - …
  - Connessione dial-up
    - Modem telefonico, Modem ISDN
    - …
  - Tutti i dati passano da quell'unica connessione

Identificare una macchina

- Per poter distinguere una macchina collegata in rete da un'altra ci deve essere un modo per identificare in modo univoco le macchine
- Questa distinzione avviene tramite l'indirizzo IP (Internet Protocol), che può esistere in due forme
  1. Domain Name Server (DNS): www.repubblica.it
  2. Come un indirizzo numerico: 4 numeri separati da .
- In entrambi i casi, l'indirizzo IP è rappresentato internamente da un numero a 32 bit (quindi ogni numero nell'indirizzo della forma 2. non può superare 255)

Le porte

- Un indirizzo IP non è sufficiente per identificare un unico server
  - Infatti, molti server possono esistere su una stessa macchina
- Un programma cliente sa come collegarsi a una certa macchina tramite il suo indirizzo IP, ma come riesce a connettersi a un determinato servizio offerto da quella macchina?
- Per distinguere i dati indirizzati a diverse applicazioni che girano sulla stessa macchina, si usano le **porte**

Le porte

- La porta *non è* una locazione fisica sulla macchina, ma solo un'astrazione software
  - I numeri di porta rappresentano un secondo livello di indirizzamento
  - L'idea è che se si chiede l'accesso a una porta particolare, si sta richiedendo il servizio associato a quel numero di porta
  - Tipicamente ogni servizio è associato a un unico numero di porta su una certa macchina server
  - È compito del cliente sapere su quale porta il servizio cercato è in esecuzione

Le porte

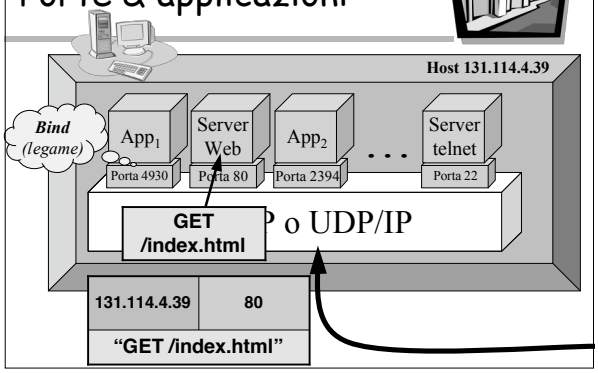
- Ogni porta su una macchina è identificata dal suo numero: 16 bit
- Tutti i dati che viaggiano sulla rete hanno come indirizzo la coppia (macchina,porta): 48 bit

Quindi: con *IPv4*, possiamo avere al massimo **4.294.967.296** macchine, su cui possono girare in contemporanea al massimo **281.474.976.710.656** applicazioni…. (ma in realtà di meno)

Le porte

- Le porte di una macchina sono divise in due gruppi:
  - Le porte 0-1023 ("*porte basse*") sono **protette** e **riservate** all'amministratore della macchina (*root* su UNIX, *Administrator* su Windows)
  - Le porte 1024-65535 ("*porte alte*") sono **liberamente** accessibili alle applicazioni utente
- In genere, le porte basse corrispondono a servizi standard (ftp, time, telnet, web, ecc.)

Porte & applicazioni



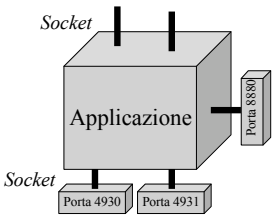
Socket

- Un **socket** (*presa*, come di corrente) è l'astrazione software utilizzata per rappresentare i "terminali" di una connessione tra due macchine
- Data una connessione tra due macchine, esiste un socket su ognuna delle macchine
- Si può immaginare un ipotetico "cavo" tra le due macchine con ognuna delle estremità del cavo inserita in un socket
- Chiaramente, l'effettiva connessione fisica fra le due macchine non è nota
  - Si tratta, appunto, di un'astrazione
  - Non si conosce più di quello che è necessario!!

Socket e porte

- In altre parole, un socket lega un **processo** del sistema operativo (ovvero, una particolare applicazione) a una **porta** (TCP/IP o UDP/IP)
- Un'applicazione può avere molti socket
- Un socket può essere *slegato* o *legato* a una (e una sola) porta
- Ogni porta può essere *libera* o *legata* a un (e uno solo) socket

Socket



- Un socket non legato a una porta non serve a molto…
- … ma può esistere come stato intermedio
- Quando un socket è legato a una porta, si dice che il socket / la porta è aperto / aperta
- Altrimenti, si dice che il socket / la porta è chiuso / chiusa

… e in Java?

- Il package *java.net* contiene le classi corrispondenti alle varie parti di una connessione di rete:

`Socket, ServerSocket`	Socket, connessioni TCP/IP
`DatagramPacket, DatagramSocket, MulticastSocket`	Connessioni UDP/IP, comunicazioni multicast
`URL, URLConnection`	Classi speciali per connessioni che usano il protocollo HTTP

TCP: socket client & server

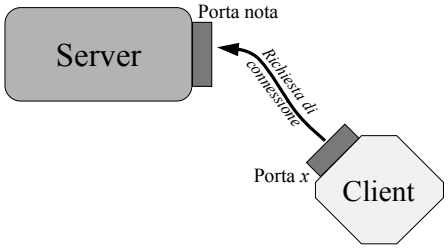
- Come abbiamo visto, una comunicazione TCP ha due parti:
  - Il chiamante, chiama perché **vuole qualcosa**: *client*
  - Il ricevente, viene chiamato perché **offre qualcosa**: *server*
- Nelle comunicazioni client/server, è il client che prende l'iniziativa
- Il server sta lì ad aspettare che qualcuno lo chiami

TCP: socket client & server

- Il server ha di solito un numero di porta *noto* a priori
  - Esiste un servizio, *inetd* (detto anche *super-server*), che fa da elenco telefonico: dato il *nome* di un servizio, vi dice a quale numero di porta trovate quel servizio
  - Su UNIX/Linux, trovate questi servizi in `/etc/services`
- Quando il client contatta il server presso questa porta:
  - Il server crea una seconda socket, dedicata alla conversazione con quel particolare client
  - La comunicazione avviene fra le due porte
  - La porta nota originale rimane libera per altri client

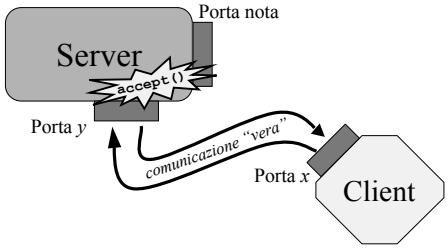
TCP: socket client & server

- Vediamo meglio cosa succede:



TCP: socket client & server

- Vediamo meglio cosa succede:



Socket in Java

- In Java, per attivare una comunicazione tra un cliente e un server
  1. Si crea un socket per connettersi all'altra macchina
  2. Si ottiene un **InputStream** e un **OutputStream** dal socket in modo da poter trattare la connessione con un oggetto stream di I/O
     - Tipicamente gli stream sono incapsulati in stream bufferizzati

Socket in Java

- Due classi del package *java.net* implementano i socket
- La classe **`Socket`** implementa i socket lato client, ed è la superclasse per altre implementazioni specializzate dei socket
- La classe **`ServerSocket`** implementa i socket lato server, e si usa per le "porte note" su cui si ricevono le richieste di connessione

Socket in Java

- Usando le classi di *java.net*, le applicazioni possono ignorare i dettagli dell'implementazione dei socket su ogni particolare sistema operativo
- Queste classi sono ottimizzate per le operazioni più comuni di comunicazione via rete
- Si possono usare tecniche avanzate (creare sottoclassi di **`SocketImpl`** e registrarle come *socket factory* presso **`Socket`**) per operazioni più sofisticate

Un client tipico

- Si dichiara un oggetto di classe **`Socket`**
- Lo si istanzia (**`new`**), specificando a quale server e su quale porta si desidera che venga stabilita la connessione
- Tutta la fase di collegamento viene svolta all'interno del costruttore: il programmatore non si deve preoccupare di nulla
- Si comunica con il server, inviando e ricevendo dati come necessario
  - Si utilizzano le librerie per gli stream
- Si chiude la socket

Un client tipico

```
import java.net.*;
/* … */
void client(String host, int port) {
     Socket socket;
     socket=new Socket(host,port);
     /* … usa il socket … */
     socket.close();
}
```

Costruttore socket

```
public Socket (String host, int port)
  throws UnknownHostException,IOException
```

- L'host viene specificato con una stringa
- La porta è sempre identificata da un intero tra 0 e 65535
- **`UnknownHostException`** è generata se il nome dell'host non può essere risolto, o se l'host non funziona
- Se la socket non può essere aperta per altri motivi, viene generata **`IOException`**

Esercizio

- Scrivere una classe **LowPortScanner** che individua quali delle prime 1024 porte di **localhost** ospitano servizi TCP

```
import java.net.*;…
public class LowPortScanner {
 public static void main (String[] args) {
…
 }
}
```

Esercizio -- soluzione

```
import java.net.*;
import java.io.*;
public class LowPortScanner {
    public static void main(String[] args){
        String host = "localhost";
        for (int i=1;i<1024;i++){
            try{
                Socket s = new Socket(host,i);
                System.out.println("Server sulla porta "+i+" di "+host);
            }
            catch (UnknownHostException e){
                System.err.println(e);
            }
            catch (IOException e){
                }
        }}}
```

Un server tipico

- Si dichiara un oggetto di classe **`ServerSocket`**
- Lo si istanzia (**`new`**), specificando quale porta si vuole usare come "porta nota"
- Si effettua una **`accept()`** sul server socket
  - Quando arriva una richiesta di connessione, la **`accept()`** ritorna e restituisce un nuovo socket, collegato con il client
- Si comunica con il client, inviando e ricevendo dati come necessario
- Si chiude il socket

Costruttore ServerSocket

```
public ServerSocket (int port)
 throws BindException,IOException
```

- Crea una server socket sulla porta specificata
- Se la porta non può essere creata, viene generata **`BindException`**, per due motivi
  - Esiste già una server socket alla porta specificata
  - Ci si vuole connettere a una porta tra 1 e 1024 senza essere amministratore

Esempio

- Scrivere una classe **LocalPortScanner** che controlla l'esistenza di porte su **localhost** cercando di creare **ServerSocket**

```
import java.net.*;…
public class LocalPortScanner {
 public static void main (String[] args) {
...
 }
}
```

Esempio -- soluzione

```
import java.net.*;
import java.io.*;
public class LocalPortScanner {
    public static void main(String[] args){
        for (int port=1;i<=65535;port++){
                try{
                        ServerSocket server = new SereverSocket(port);
                        System.out.println("Server sulla porta "+i+" di "+host);
                }
                catch (IOException e){
                        System.out.println("Esiste un server sulla porta "+port);
                }
}}}
```

accept()

- Un **ServerSocket** tipicamente opera con un ciclo che accetta connessioni
- Ad ogni ciclo, viene invocato il metodo **`accept()`**, che restituisce un oggetto **Socket** che rappresenta la connessione tra cliente remoto e server locale

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

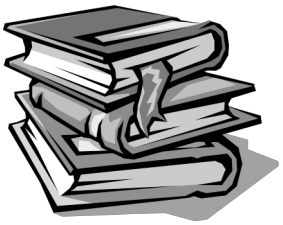
Server *single-threaded* e *multi-threaded*

- Una volta stabilita una connessione con il client, il server ha due scelte:
  - Può dedicarsi completamente al client, **ignorando** altre richieste di connessione che arrivino nel frattempo (server *single-threaded*)
  - Oppure può lanciare un nuovo *thread*, che si occuperà del client, e tornare **subito** a fare una nuova accept() sul server socket (server *multi-threaded*)
- Nel secondo caso, possono essere serviti più clienti in contemporanea: metodo più complesso, ma migliore

Un server tipico
(single-threaded)

```
import java.net.*;
/* … */
void server(int port) {
      ServerSocket ssocket;
      ssocket=new ServerSocket(port);
      while (!finito) {
            Socket csocket=ssocket.accept()
            /* … usa il csocket … */
            csocket.close();
      }
      ssocket.close();
}
```

Un server tipico



- "Bello, peccato che i server multi-threaded siano così complicati…"
- "Chissà quanti manuali tocca studiare per scrivere un server vero…"

Un server tipico
(multi-threaded, server principale)

```
import java.net.*;
/* … */
void server(int port) {
      ServerSocket ssocket;
      ssocket=new ServerSocket(port);
      while (!finito) {
            Socket csocket=ssocket.accept()
            MyServer s=new MyServer(csocket);
            s.start();
      }
      ssocket.close();
}
```

Un server tipico
(multi-threaded, server di un cliente)

```
import java.net.*;
class MyServer extends Thread
{
      Socket mioSocket;
      public MyServer(Socket s)
      {
            mioSocket=s;
      }
      public void run()
      {
            /* … usa mioSocket … */
            mioSocket.close();
      }
}
```

Riepilogando…

- Server single-threaded
  - Più facili da scrivere
  - Più veloci a reagire a una connessione
  - Possono servire **un solo cliente per volta**
  - Adatti a protocolli che richiedono connessioni brevi e risposte rapide
- Server multi-threaded
  - (poco) più complicati da scrivere
  - Tempi di reazione (di poco) più lunghi
  - Possono servire **un numero qualunque di clienti insieme**
  - Adatti a protocolli che hanno sessioni lunghe

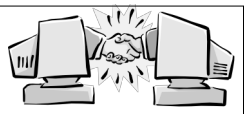
Test!

Protocollo	*Multi-thread*	*Single-thread*
time	No	✔
chargen	✔	No!
telnet	✔	No!
http	✔ *Server complessi, tante funzionalità (Apache, IE)*	✔ *Server semplici, alto traffico (khttpd)*
smtp	✔	✔
ssh	✔	No!

Come usare i socket

- Abbiamo visto finora come procurarsi un socket aperto verso il corrispondente:
  - Il cliente lo ottiene con `sock=new Socket(server,porta)`
  - Il server lo ottiene con `sock=serversocket.accept()`
- Ma una volta aperti, come si usano i socket per mandare e ricevere dati?

I/O con i socket



- La classe `Socket` ha due metodi che fanno al caso nostro:
  - `getInputStream()`
  - `getOutputStream()`
- Una volta ottenuti dal socket gli stream per fare input e output, tutto procede come per una normale lettura o scrittura verso un file

I/O con i socket

- Sia il client che il server possono fare:

```
import java.net.*;
import java.io.*;
/* … */
   InputStream in=sock.getInputStream();
   OutputStream out=sock.getOutputStream();
   /* usa in.read() per leggere dal socket
e
      out.write() per scrivere sul socket */
```

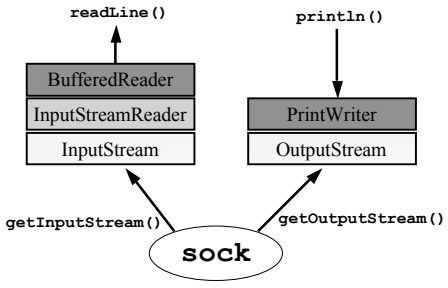
Esempio server

```
ServerSocket server = new ServerSocket(5776);
while (true) {
  Socket connection = server.accept();
  OutputStreamWriter out =
     new OutputStreamWriter(connection.getOutputStream());
  out.write("connesso!!");
  connection.close();
}
```

I/O con i socket

- Usare **`read()`** e **`write()`** è ***molto*** scomodo…
- Si può leggere/scrivere un byte alla volta (o al più un array di byte)
- Le classi di I/O di Java prevedono la possibilità di "arricchire" uno stream per avere maggiori funzionalità

I/O con i socket



I/O con i socket

```
import java.net.*;
import java.io.*;
/* … */
        BufferedReader in =
          new BufferedReader(
             new InputStreamReader(
               sock.getInputStream()));
        PrintWriter out =
          new PrintWriter(
             sock.getOutputStream());
/* continua */
```

I/O con i socket

```
String input, output;
/* … */
      input=in.readLine();
      /* … */
      out.println(output);
      /* … */
      out.close();
      in.close();
      sock.close();
```

Riceve i dati dal socket

Invia i dati al socket

Dettagli spinosi…

- *Naturalmente*, tante cose possono andare storte, in tutte le fasi:
  - Creazione dei socket
  - **`accept()`**
  - Lettura e scrittura sugli stream
  - Persino **`close()`**!
- Tutti gli intoppi vengono gestiti tramite **eccezioni**
- "Provare per credere"….

Dettagli spinosi…

- La classe **Socket** ha molti altri metodi utili:
  - **`getInetAddress()`, `getLocalAddress()`, `getPort()`, `getLocalPort()`**
    - Informazioni sulle connessioni
  - **`get/setSoTimeout()`, `get/setSoLinger()`, `get/setSoTcpNoDelay()`**
    - Legge e imposta alcuni parametri della connessione
  - **`get/setReceiveBufferSize()`, `get/setSendBufferSize()`**
    - Legge e imposta le dimensioni dei buffer del socket
- **ServerSocket** ne aggiunge altri di suo…

Ok, al lavoro!

Facciamo un po' di pratica…

Esercizio 0

- Scrivere un server **TCPEchoServer** che accetta come messaggi linee di testo dal cliente, conta i messaggi ricevuti, e manda indietro al cliente il messaggio (numerato)
  - Il server termina quando riceve il messaggio **STOP** dal cliente
- Scrivere il relativo cliente **TCPEchoClient**

Esercizio 1

- Scrivere un server **DaytimeServer** che, ad ogni connessione, risponda inviando al client la data e l'ora corrente (a questo provvede la classe `Date`)
  - Stampare a Console anche l'indirizzo e la porta del cliente che richiede il servizio
- Il server deve chiudere la connessione subito dopo aver inviato la data al client
- Scrivere il relativo cliente **DaytimeClient**
- Testare il server senza l'ausilio di **DaytimeClient**….
  - ….come si può fare??

Esercizio 1

- Scrivere un server **DaytimeServer** che, ad ogni connessione, risponda inviando al client la data e l'ora corrente (a questo provvede la classe `Date`)
  - Stampare a Console anche l'indirizzo e la porta del cliente che richiede il servizio
- Il server deve chiudere la connessione subito dopo aver inviato la data al client
- Scrivere il relativo cliente **DaytimeClient**
- Testare il server senza l'ausilio di **DaytimeClient**….
  - ….via telnet!!

Esercizio 1

- Scrivere un server **DaytimeServer** che, ad ogni connessione, risponda inviando al client la data e l'ora corrente (a questo provvede la classe `Date`)
  - Stampare a Console anche l'indirizzo e la porta del cliente che richiede il servizio
- Il server deve chiudere la connessione subito dopo aver inviato la data al client
- Scrivere il relativo cliente **DaytimeClient**
- Testare il server senza l'ausilio di **DaytimeClient**….
  - ….**telnet localhost 2000**

Esercizio 1

- Scrivere un server **DaytimeServer** che, ad ogni connessione, risponda inviando al client la data e l'ora corrente (a questo provvede la classe `Date`)
  - Stampare a Console anche l'indirizzo e la porta del cliente che richiede il servizio
- Il server deve chiudere la connessione subito dopo aver inviato la data al client
- Scrivere il relativo cliente **DaytimeClient**
- Testare il server senza l'ausilio di **DaytimeClient**….
  - ….oppure??

Esercizio 1

- Scrivere un server **DaytimeServer** che, ad ogni connessione, risponda inviando al client la data e l'ora corrente (a questo provvede la classe `Date`)
  - Stampare a Console anche l'indirizzo e la porta del cliente che richiede il servizio
- Il server deve chiudere la connessione subito dopo aver inviato la data al client
- Scrivere il relativo cliente **DaytimeClient**
- Testare il server senza l'ausilio di **DaytimeClient**….
  - ….via http!!

Esercizio 1

- Scrivere un server **DaytimeServer** che, ad ogni connessione, risponda inviando al client la data e l'ora corrente (a questo provvede la classe `Date`)
  - Stampare a Console anche l'indirizzo e la porta del cliente che richiede il servizio
- Il server deve chiudere la connessione subito dopo aver inviato la data al client
- Scrivere il relativo cliente **DaytimeClient**
- Testare il server senza l'ausilio di **DaytimeClient**….
  - ….**http://localhost:2000**

Esercizio 2

- Scrivere un server **DumpServer** che si limiti a stampare su **System.out** tutto ciò che arriva alla porta 4040
- Scrivere un client **DumpClient** che mandi sulla porta 4040 di un server dato una stringa passata come argomento
  - **DumpServer**: la porta deve essere fornita come argomento
  - **DumpClient**: il server e la porta a cui collegarsi devono essere passati come argomento
  - Gestire bene *tutte* le eccezioni!!

Esercizio 3 -- complesso

- Scriviamo un **server web** (semplificato)
- Il server riceve dal client un comando come `GET /un_certo_path/un_certo_file.html`
- Il server risponde inviando al client un codice di successo (si veda il protocollo HTTP), seguito dal contenuto del file indicato da GET
- Il ciclo si può ripetere; la connessione viene abbattuta dal client quando non vuole più chiedere altri file, oppure dopo 30 secondi senza comandi GET

Java

accesso alla rete -- TCP

fine