Lezione 2 Applicazioni Web

0) Server Iterativi e concorrenti

Il server appena visto gestisce una sola connessione alla volta infatti non viene nuovamente invocata `accept()` per ascoltare nuove connessioni fino a quando non ha finito con quella corrente. Questo tipo di server viene detto **iterativo**.
Un server più realistico dovrebbe accettare più richieste (connessioni in questo caso) contemporaneamente e viene detto **concorrente**. In generale la concorrenza del server può essere ottenuta in vari modi. In C ad esempio possiamo ottenere la concorrenza attraverso concorrenza dei processi (fork), thread e I/O asincorno. Normalmente in Java si usano i thread per implementare un server concorrente. In ogni caso bisogna ricordare che la definizione di server concorrente riguarda **non** l'implementazione, ma il fatto che sia in grado di accettare e servire più richieste contemporaneamente.

0.1) Server concorrente

Vediamo ora un esempio di server concorrente che implementa il servizio di echo. Tale servizio prevede che una richiesta del client sia una qualsiasi sequenza di carattere, e che il server esaudisca la richiesta inviandola tale e quale indietro al client. Il server multi-thread che vediamo, accetta una nuova connessione, e appena ottenuto l'oggetto che rappresenta il socket, crea un nuovo thread a cui delega il compito di rispondere al client relativo alla connessione. Dopo aver creato il thread slave, il thread principale ritorna ad aspettare ulteriori connessioni attraverso il SocketServer.

```
import java.net.*;
import java.io.*;
import java.util.*;


public class TinyServer {
   public static void main( String argv[] ) throws IOException
   {
    ServerSocket ss = new ServerSocket(7);
    while ( true )
    {
        Socket temp = ss.accept();
        new ManageConnection( temp );
    }
   }
}
//classe privata
class ManageConnection extends Thread {
  Socket sock;
  ManageConnection ( Socket s )
  {
    sock = s;
    start();
  }
  public void run() {
    try {
      OutputStream out = sock.getOutputStream();
      PrintStream pout = new PrintStream(sock.getOutputStream());
      BufferedReader d = new BufferedReader(new
InputStreamReader(sock.getInputStream()));
      String req = null;
      while ( (req=d.readLine())!=null )
```

```
      {
        pout.println( req );
      }
      sock.close();
    }
    catch ( IOException e )
   {
     System.out.println( "I/O error " + e.getMessage() );
    }
 }
}
```

Per controllare il funzionamento del server possiamo utilzzare come client un qualsiasi client telnet aggiungendo dopo il nome dell'host anche la porta del server al quale connettersi, che in questo caso è la porta numero 7. Ad esempio:
`>telnet localhost 7`
Permette di testare il funzionamento del server che gira nella medesima macchina del client.

Nell'esempio il server aspetta la richiesta del client leggendo i caratteri dal client finché non vede i caratteri di fine linea. Diverso è l'approccio quando utilizziamo un protocollo binario.

Esercizi

Scrivere un metodo per leggere una richieste (o analogamente una risposta) di un protocollo Client/Server binario. Il protocollo prevede che le richieste e le risposte siano di lunghezza variabile e la lunghezza stessa sia indicata nel primo byte binario della richiesta (max 255 byte per messaggio). Il metodo deve ritornare la lunghezza della richiesta se questa viene letta correttamente, altrimenti -1 se non si è letta una richiesta corretta (ovvero di lunghezza corretta). Nota che se non si ricevono byte, è un errore. Almeno deve essere ricevuto il byte della lunghezza, se questo byte vale 0, allora abbiamo ricevuto una richiesta corretta.

Problema, se richieste e risposte possono essere più lunghi di 255 bytes, come posso rappresentare la lunghezza? Scrivete un metodo che legge una richiesta di un protocollo Client/Server binario. Il protocollo prevede che le richieste e le risposte siano di lunghezza variabile e la lunghezza sia indicata nei primi due byte binari del messaggio stesso. Si assuma che il byte più significatgivo sia il primo (Big endian o Network byte order)

Per implementare entrambi gli esercizi studiare prima la documentazione dei metodi read(), read(byte[] b) e read(byte[] b,int off,int len) dell'interfaccia InputStream.

Per la consegna utilizzate l'apposita funzionalità di moodle.

1) Struttura del Web

Vedremo in questa lezione come sono strutturate le applicazioni web.
Il Web è composto da 3 elementi:

- URL: i nomi delle risorse nel Web;
- HTTP: il modo in cui vengono trasferite le risorse nel Web;
- HTML: il linguaggio delle risorse più importanti del Web.

1.1) URL

Gli Uniform Resource Locator ci permettono di individuare in maniera univoca una risorsa nel Web.

Un Url è composto dalle seguenti parti:

- protocollo: normalmente http, altri valori sono ftp, telnet, mailto;
- user e password (opzionali): sono le credenziali per accedere al Web Server;
- l'indirizzo del Web Server: può essere specificato tramite un I.P. address (157.138.20.15) o un nome (www.dsi.unive.it);
- una porta (opzionale): indica il numero di porta TCP alla quale connettersi;
- il path della risorsa (file normalmente) cercata;
- i parametri della richiesta (opzionali): importanti per le applicazioni Web permettono di scambiare utili parametri tra il browser e l'applicazione tramite ad esempio i FORM HTML.

Concludendo gli Url servono a individuare univocamente una risorsa nella rete.

1.2) HTTP

L'Hyper Text Transfer Protocol stabilisce delle regole per il trasferimento delle informazioni.
Il protocollo HTTP prevede come vedremo meglio più avanti, le seguenti fasi:

- il browser apre una connessione col Web server specificato nel URL;
- il browser invia la richiesta al Web server attraverso la connessione;
- il Web server invia la risposta al browser attraverso la connessione;
- il server chiude la connessione.

Le richieste inviate al server normalmente sono richieste di lettura di file (file HTML, testo, immagini etc.). Le richieste di lettura di file avvengono tramite il richieste HTTP di tipo GET. Talvolta le richieste fatte al Web server riguardano l'invio di dati dal Browser al Server. Queste richieste avvengono tramite richieste HTTP di tipo POST.

1.3) HTML

L'HyperText Markup Language è il linguaggio con il quale scrivere i documenti scambiati nel Web. Anche se non tutti i file che vengono scambiati nel Web sono scritti in HTML, i file che permettono un'interazione nelle applicazioni Web sono scritti in HTML. Allo stesso modo in cui nella busta con cui inviamo una lettera ci possiamo accludere una foto o una banconota e lettera illustra al destinatario il motivo degli altri oggetti inviati, così il file HTML specifica al browser l'aspetto grafico e alcuni aspetti semantici della pagina da visualizzare. Il linguaggio HTML ha una sintassi molto semplice che prevede i seguenti TAG principali:

- `<html>` è il tag di apertura che indica l'inizio del contenuto HTML del file;
- `<body>` è il tag di inizio della parte "visibile" del file HTML;
- `<a href="URL">` è il tag principale per specificare gli aspetti semantici di un file html, cioè i collegamenti allo stesso o altri file. Questi collegamenti vengono specificati tramite il loro "nome" ovvero l'Url;
- `<form action="URL">` è il tag che indica l'inizio di un modulo dove l'utente può inserire o selezionare dei dati da inviara dal browser al Web server (vedremo più avanti come).

I tag HTML possono avere un corrispondente tag di chiusura composta nel seguente modo: `</tag>`. Ad esempio i tag di chiusura per i tag elencati prima sono: `</html>,</body></a></form>`.

1.4) Form

I form sono usati principalmente per trasferire dati dal client al server. All'interno della coppia di tag di apertura e chiusura del form possono comparire una varietà di altri tag quali:

- `<INPUT NAME=nome TYPE=tipo VALUE=valore>` che serve per introdurre un testo. Può

avere uno dei seguenti tipi:
    - `TEXT`: per introdurre un testo libero di una riga;
    - `SUBMIT`: visualizza un pulsante per la sottomissione;
    - `PASSWORD`: per introdurre un testo nascosto;
    - `FILE`: per introdurre un file.
- `<TEXTAREA NAME=`*`nome`* `ROWS=`*`n`*`>` per introdurre più linee di testo.

2) Peculiarità delle applicazioni Web

Analizzando il funzionamento del protocollo HTTP possiamo vedere alcune sue particolarità che condizionano il funzionamento delle applicazioni Web.
Prima di tutto ogni richiesta di risorsa richiede una nuova connessione dal browser al Web server, questo comporta un rallentamento dovuto al fatto che la gestione della connessione TCP ne incrementa le prestazioni con l'uso della connessione stessa.
Sempre perché dopo l'invio della risorsa il server chiude la connessione, non c'è nessuna interazione tra browser e server fino alla prossima richiesta da parte del client (PULL). Inoltre il protocollo HTTP è senza stato e questo ci complica le cose quando dobbiamo mantenere delle informazioni riguardo le richieste precedenti.
Se vogliamo mantenere delle informazioni tra una richiesta e la successiva possiamo farlo tramite l'url rewriting e parametri nascosti oppure i cooky.
Altro problema è rappresentato dal fatto che il Web server ritorna file HTML (gli altri tipi o sono più stupidi o non sono supportati da tutti i browser) e i tipi di funzionalità è quella permessa dal linguaggio HTML all'inzio ideato solo per facilitare la navigazione, ora invece si tende ad usare il browser per implementare le funzionalità di un generico client.
Una delle funzionalità aggiunte al linguaggio HTML è **Javascritp** un linguaggio di scripting in grado di esegure script anche complessi nel browser. Javascript permette di aumentare l'interattività del browser senza bisogno di interagire col server, quindi una risposta più veloce e inoltre. Nel caso ciò non basti si può ricorrere all'uso di **Applet** Java che prevedono la realizzazione di un client ad hoc da includere nelle pagine HTML.