1) Server HTTP

Il server HTTP o server Web (ad esempio Apache o IIS), è un programma sempre in esecuzione che aspetta richieste dai client e ritorna le risorse specificate nelle richieste. Ci sono due tipi fondamentali di risorse disponibili nel Web server:

1. risorse statiche: le risorse esistono prima della richiesta e/o il loro contenuto è fissato;
2. risorse dinamiche: le risorse vengono create al momento della richiesta e in maniera automatica con programma.

In generale le risorse che un Web server gestisce sono principalmente pagine html, file di testo, formati per lo scambio di documento di testo (Adobe Portable Document Format, PostScript,), immagini (GIF, JPEG, PNG), animazioni e altri contenuti multimediali. Tutti queste risorse possono essere statiche o dinamiche, anche se generalmente vengono generate pagine html. Per la gestione delle risorse statiche è sufficiente un implementazione utilizzando le tecniche già viste la scorsa lezione:

1.1) Server Http

TinyHttpd ascolta su una porta specificata e serve semplici richieste HTTP:

```
GET /path/filename HTTP/1.0
```

Il Web browser manda una o più di queste linee per ogni documento da ottenere. Letta la richiesta, il server prova ad aprire il file specificato e ne spedisce il contenuto. Se il documento contiene referenze ad immagini o altro da mostrare online, il browser continua con richieste GET addizionali. Per ragioni di performance, TinyHttpd serve ogni richiesta in una sua thread. Perciò TinyHttpd può servire molte richieste concorrentemente.

```
import java.net.*;
import java.io.*;
import java.util.*;
public class TinyHttpd {
 public static void main( String argv[] ) throws IOException {
 ServerSocket ss =
 new ServerSocket(Integer.parseInt(argv[0]));
 while ( true )
 new TinyHttpdConnection( ss.accept() );
 }
}
class TinyHttpdConnection extends Thread {
 Socket sock;
 TinyHttpdConnection ( Socket s ) {
 sock = s;
 setPriority( NORM_PRIORITY - 1 );
 start();
 }
 public void run() {
 try {
 OutputStream out = sock.getOutputStream();
 BufferedReader d = new BufferedReader(new InputStreamReader(sock.getInputStream()));
 String req = d.readLine();
 System.out.println( "Request: "+req );
 StringTokenizer st = new StringTokenizer( req );
 if ( (st.countTokens() >= 2) && st.nextToken().equals("GET") )
 {
 if ( (req = st.nextToken()).startsWith("/") )
```

```
 req = req.substring( 1 );
 if ( req.endsWith("/") || req.equals("") )
 req = req + "index.html";
 try
 {
 FileInputStream fis = new FileInputStream ( req );
 byte [] data = new byte [ fis.available() ];
 fis.read( data );
 //Esercizio: scrivere header
 out.write( data );
 }
 catch ( FileNotFoundException e )
 {
 new PrintStream( out ).println("404 Not Found");
 }
 }
 else new PrintStream( out ).println( "400 Bad Request" );
 sock.close();
 }
 catch ( IOException e )
 {
 System.out.println( "I/O error " + e );
 }
}
}
```

1.2) Esempio d'uso del server

Dopo aver compilato il file `TinyHttpd.java` lo si inserisce nel class path. In una directory in cui esiste il file `prova.html` si avvia il daemon, specificando un numero di porta libero come argomento. Per esempio:

```
> java TinyHttpd 8080
```

Utilizzando un client standard HTTP che in questo caso è un browser Web tipo Internet Explorer o Netscape possiamo verificare il funzionamento del server appena creato. L'URL dovrà essere: `http://localhost:8080/prova.html`. Se il computer è collegato in rete e ha un IP address o un nome, allora le pagine potranno essere accessibili anche da altri computer specificando: `http://nome.dominio:8080/prova.html` oppure `http://ip.address:8080/prova.html`.

1.3) note sull'implemetazione

Abbassando la sua priorita' a `NORM_PRIORITY-1`, assicuriamo che il thread che serve le connessioni già stabilite non blocchi la main thread di `TinyHttpd` impedendo di accettare nuove richieste.
Anche se l'esempio serve allo scopo, in ogni casi ci sono tutta una serie di aspetti che l'implementazione di un Web server reale deve considerare:

1. consuma un sacco di memoria allocando un array enorme per leggere un intero file tutto d'un colpo. Un'implementazione piu' realistica userebbe un buffer e invierebbe i dati in più passi.
2. `TinyHttpd` inoltre non riconosce semplici directories. Non sarebbe difficile aggiungere alcune lineee per leggere directories e generare liste di link HTML come fanno molti web

server.

3. `TinyHttpd` soffre di limitazioni imposte dalla debolezza di accesso al filesystem. È importante ricordare che i pathnames dipendono dal sistema operativo, così come il concetto stesso di filesystem. TinyHttpd funziona, così com'è, su sistemi UNIX e DOS-like, ma richiede alcune variazioni su altre piattaforme. È possibile scrivere del codice più elaborato che usa informazioni sull'ambiente offerte da Java per adattarsi al sistema locale.
4. mancano tutti i meccanismi per la generazione di pagine dinamiche: CGI, SSI, php, servlet etc;
5. il problema principale con `TinyHttpd` è che non ci sono restrizioni sui files a cui puo' accedere. Con qualche trucco, il daemon spedirebbe un qualunque file del suo filesystem al client.

Facoltativo:
Per ovviare all'ultimo problema, potremmo fare in modo che prima di aprire il file, controllare che il path del file sia di un file (o directory) che sia nelle nostre intenzioni rendere pubblico. Un tale approccio è semplicemente realizzabile ma ha lo svantaggio di richiedere che in tutta la nostra applicazione ad ogni accesso di un file dobbiamo ricordarci di controllare se quel file è leggibile. Un approccio diverso è quello di appoggiarsi a delle caratteristiche standard di Java. In questo caso useremo un'estensione della classe `SecurityManager` per gestire in automatico l'accesso al file system. La nascita di Java è stata motivata dalla necessità di un linguaggio in cui la gestione degli accessi fosse sicura per poter far giare delle Applet in macchine ospiti. Quindi sono già presenti meccanismi che in automatico limitano l'accesso al File System. Questo meccanismo è appunto il `SecurityManager`. Di solito il `SecurityManager`
standard non permette di fare nessuna delle operazioni di seguito elencate:

1. `checkAccess(g)`: si può accedere ai dati del thred g?
2. `checkListen(p)`: si può accettare connessioni dalla porta p?
3. `checkLink(l)`: si può collegare con la libreria dinamica l?
4. `checkPropertyAccess(k)`: si puo accedere alla proprietà di sistema k?
5. `checkAccept(h,p)`: si può accettare connessioni dall'host h e porta p?
6. `checkWrite(f)`: si può scrivere il file f?
7. `checkRead(f)`: si può leggere il file f?

La classe `SecurityManager` impedisce tutte queste operazioni generando una eccezione `SecurityException` per ognuno dei metodi elencati. Quindi per fare in modo che il nostro server faccia tutto quello che gli serve eccetto per la lettura dei file non contenuti nella sottocartella, dobbiamo sovrascrivere i metodi in modo che non generino eccezzioni eccetto quando si tenta di leggere un file non consentito.

```
import java.io.*;

class TinyHttpdSecurityManager
extends SecurityManager {
  public void checkAccess(Thread
g) { };
  public void checkListen(int
port) { };
  public void checkLink(String
lib) { };
  public void
```

```
checkPropertyAccess(String key)
{ };
  public void checkAccept(String
host, int port) { };
  public void
checkWrite(FileDescriptor fd) { };

  public void
checkRead(FileDescriptor fd) { };
  public void checkRead( String
s )
  {
    if ( s.startsWith("/") ||
(s.indexOf("..") != -1) )
       super.checkRead(s);
  }
}
```

Il cuore di questo security manager e' il metodo `checkRead()`.
Controlla due cose: assicura che il pathname non sia assoluto, e che il pathname non contenga una coppia di punti (..) che risale l'albero delle directories.
Con questi vincoli siamo certi (almeno su un filesystem UNIX o DOS-like) di aver ristretto l'accesso alle solo subdirectories della directory corrente. Se il pathname è assoluto o contiene "..", `checkRead()` lancia una `SecurityException`.
Gli altri metodi che nell'esempio non fanno nulla a esempio checkAccess(), vengono invocati dal security manager e non impediscono al demone di fare il proprio lavoro.
Quando installiamo un security manager, ereditiamo implementazioni di molte "check" routines. L'implementazione default non permette di fare nulla; Lancia una security exception appena chiamata. Dobbiamo permettere cosicchè il daemon possa fare il suo lavoro; deve accettare connections, ascoltare sockets, creare threads, leggere property lists, ecc. Per questo sovrascriviamo i metodi default.
Per installare il security manager, aggiungere la seguente riga all'inizio del metodo `run` di `TinyHttpdConnection`:

```
System.setSecurityManager( new TinyHttpdSecurityManager()
);
```

Per catturare le security exception, aggiungere il seguente catch dopo la catch di `FileNotFoundException`:

```
    catch ( SecurityException e )
    {
        new PrintStream( out ).println( "403
    Forbidden" );
   }
```

Per la creazione delle pagine html, ma non solo, sono stati predisposti dei meccanismi da agganciare ai Web server in modo da aggiungere la generazione delle pagine dinamiche. Tra le varie tecniche adottate, la prima con un alto grado di flessibilità e potenza è stato lo standard CGI che vedremo nel prossimo paragrafo. Altre soluzioni alternative sono i SSI, linguaggi di scripting come PHP, Perl etc, le servlet e le JSP che vedremo più avanti.

2) CGI

La Common Gateway Interface è uno standard di comunicazione tra web-server e applicazioni. E' stato definito da Netscape. Lo standard CGI permette ad un web server di colloquiare con un'applicazione esterna in modo da integrare i servizi forniti dal server web alle particolali informazioni dinamiche fornite dall'applicazione. I vantaggi che si ottengono da questo approccio sono:

- sviluppo delle sole funzionalità peculiari del sistema (uso del server Web per tutte le funzionalità standard);
- gestione della concorrenza affidata al server Web;
- pre-elaborazioni delle richieste dei client fatta dal server Web (vedi ENVIRONMENT);
- sviluppatore delle applicazioni ignora quasi i problemi di comunicazione e di concorrenza;
- essendo uno standard, posso scrivere applicazioni portabili per quanto riguarda la comunicazione con i web server.