ORGANIZZAZIONE DI SISTEMI OPERATIVI E RETI

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

SPECIFICHE DI PROGETTO A.A. 2007/2008 – VERSIONE 1.0

Il progetto consiste nello sviluppo di un'applicazione client/server. Client e server devono comunicare tramite socket TCP. Il server deve essere concorrente e la concorrenza deve essere implementata con i thread POSIX. Il thread main deve rimanere perennemente in attesa di nuove connessioni e le deve smistare ad un insieme di thread che hanno il compito di gestire una singola richiesta (i dettagli sono forniti più avanti).
L'applicazione da sviluppare consiste in un server web in grado di supportare il protocollo HTTP. Il server, che deve avere nome **`webserver`**, si occupa di gestire le richieste provenienti dai client. Il server deve essere implementato utilizzando il linguaggio C (*non sono ammesse eccezioni*). Non occorre implementare il relativo client HTTP, ma si può usare un qualsiasi browser esistente (come ad esempio Firefox).

Invocazione del server

La sintassi del comando **`webserver`** deve essere la seguente:

```
webserver <host> <porta> <cartella_base>
```

dove:

- `<host>` è l'indirizzo IP utilizzato[1];
- `<porta>` è la porta su cui il server è in ascolto;
- `<cartella_base>` è la cartella a partire dalla quale opera il server.

È importante notare che la cartella base è la cartella che contiene i file serviti dal server, e che corrisponde alla radice (`/`) del server stesso. In particolare il server interpreta i riferimenti assoluti dei file richiesti a partire dalla cartella base stessa. Ad esempio: se la cartella base è `/home/osor/public_html` la richiesta del file `/index.html` viene servita con l'invio del file `/home/osor/public_html/index.html`.

Gestione delle richieste

Il protocollo HTTP prevede lo scambio di opportuni messaggi tra client e server. I messaggi HTTP sono formati da una riga di richiesta o di risposta, seguita da una serie di intestazioni nella forma `Nome: valore`[2], ciascuna su una riga distinta. Oltre alle intestazioni, un messaggio HTTP può contenere un corpo, che è separato dal resto del messaggio per mezzo di una riga vuota[3].
Il server deve essere conforme ad un sottoinsieme delle specifiche HTTP/1.0, descritto nella RFC1945 (cfr. `http://www.w3.org/Protocols/HTTP/1.0/draft-ietf-http-spec.html`). In particolare il server deve soddisfare i seguenti requisiti.

- **Interpretazione delle richieste.**
  Il server deve essere in grado di servire richieste HTTP versione 1.0. In particolare il server supporta *esclusivamente* richieste di tipo *full request* aventi metodo `GET` come ad esempio:

  ```
  GET /index.html HTTP/1.0
  ```

---

[1] A scopo di test si può utilizzare l'indirizzo di loopback 127.0.0.1.
[2] Quindi il nome dell'intestazione termina con i due punti, ed è separato dal suo valore per mezzo di uno spazio.
[3] Il protocollo HTTP considera come interruzione di linea la sequenza di caratteri CR LF (`\r\n`).

Come argomento della richiesta il server accetta *soltanto* percorsi assoluti e non URI assoluti (cfr. RFC, § 5.1.2).
Il server deve anche accettare richieste HTTP versione 1.1, nel senso che non le deve scartare ma le deve servire come se fossero richieste HTTP versione 1.0. Il server deve ignorare gli header presenti nella richiesta, limitandosi ad interpretare esclusivamente la riga di richiesta. Per ulteriori informazioni fare riferimento alla RFC, in particolare ai §§ 4 e 5.
Le richieste di cartelle (inclusa la root `/`) devono essere trasformate in richieste del file `index.html` eventualmente contenuto al loro interno. Ad esempio la richiesta di `/dati/` deve essere considerata come la richiesta di `/dati/index.html`, sempre servita a partire dalla cartella base.
Il server deve essere flessibile nell'interpretare le linee di richiesta. In particolare il server deve accettare un numero qualsiasi di spazi o tabulazioni tra i campi e deve accettare indifferentemente CR, LF o CRLF come terminazione di linea (cfr. RFC, Appendice B).

- **Tipi MIME supportati.**
  Ogni file inviato dal server al client deve essere caratterizzato da una stringa, denominata tipo MIME, che descrive il tipo di informazione associata ad un certo file (documento HTML, foglio di stile, immagine ecc.). In genere il tipo di un file viene ricavato a partire dalla sua estensione. In particolare il server, nel momento in cui riceve una richiesta, deve analizzare l'estensione del file richiesto e individuare il tipo MIME corrispondente. Nel caso in cui l'estensione del file non corrisponda a nessun tipo MIME supportato il server deve restituire un codice di errore adeguato (vedere più avanti).
  Il server deve supportare i seguenti tipi MIME:
  - `/text/html`, per file HTML con estensioni `htm` e `html`;
  - `/text/css`, per file CSS con estensione `css`;
  - `/text/plain`, per file di testo con estensione `txt`;
  - `/image/jpeg`, per immagini JPEG con estensioni `jpg` e `jpeg`;
  - `/image/png`, per immagini PNG con estensioni `png`.
- **Risposta del server.**
  Il server deve inviare risposte di tipo HTTP versione 1.0. Le intestazioni che il server deve inviare sono limitate alle seguenti (cfr. RFC, § 7.1).
  - `Content-Type`, contenente il tipo MIME associato al file richiesto;
  - `Content-Length`, seguita dal numero di byte del corpo della risposta;
  - `Server`, in cui si può utilizzare un valore a piacere.

  Un esempio di risposta HTTP inviata in assenza di errori è il seguente:

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 79
Server: webserver/0.1 (FreeBSD)

<html><head><title>Index</title></head><body>Contenuto del sito</body></html>
```

  Per ulteriori informazioni fare riferimento alla RFC, in particolare ai §§ 4 e 6.
- **Gestione degli errori.**
  In caso di errore il server deve restituire il relativo codice di stato (cfr. RFC, § 6.1.1). I codici di stato supportati sono limitati ai seguenti.
  - `404`, nel caso in cui il file richiesto non esista;
  - `501`, nel caso in cui il server non riesca a soddisfare la richiesta.

  In particolare il server restituisce il codice di stato `501` nei seguenti casi:
  - la richiesta non è del tipo supportato (`GET`) o non è interpretabile;
  - si è verificato un errore di accesso al file (nel caso in cui il file esista);
  - il file richiesto ha un'estensione che non è associata a nessun tipo MIME supportato.

Il server deve inviare un breve messaggio nel corpo della risposta HTTP che descriva in modo sintetico l'errore che si è verificato. Per ulteriori informazioni fare riferimento ai §§ 9.4 e 9.5 della RFC.

Gestione della concorrenza

Il server è concorrente ed utilizza i thread per gestire le singole richieste. Il thread main, prima di mettersi in attesa di connessioni, prealloca un insieme (*pool*) di thread gestori. Il thread main assegna ad un thread gestore (libero) del pool ogni nuova connessione che riceve. Il thread in questione gestisce la richiesta e si mette in attesa di eseguirne un'altra. Il numero di thread del pool è specificato a tempo di compilazione (è una costante e non varia durante l'esecuzione del programma).
Possiamo schematizzare lo schema di funzionamento del server nel seguente modo:

1. il thread main crea `NUM_THREAD` thread gestori;
    a. il thread gestore si blocca finché non gli viene assegnata una richiesta;
2. il thread main si mette in attesa delle connessioni in ingresso;
3. quando riceve una connessione in ingresso, il thread main:
    a. controlla il numero di thread occupati (ovvero quelli che stanno attualmente eseguendo una richiesta);
    b. se tutti i thread del pool sono occupati si blocca in attesa che uno diventi libero;
    c. se c'è almeno un thread libero ne sceglie uno (senza nessuna politica particolare) e lo attiva;
4. il thread gestore:
    a. si sblocca nell'istante in cui gli viene assegnata una richiesta;
    b. riceve la richiesta e la interpreta;
    c. invia la relativa risposta al client;
    d. chiude la connessione e si blocca in attesa di nuove richieste.

La sincronizzazione deve essere realizzata *esclusivamente* per mezzo dei meccanismi forniti dalla libreria `pthread`: in particolare *non saranno accettate soluzioni che prevedano attese attive*.

Output prodotto dal programma

Una volta eseguito, **`webserver`** deve stampare a video delle informazioni descrittive sullo stato del server (creazione del socket di ascolto, creazione dei thread, connessioni accettate, richieste dei client ecc.).
L'output generato deve essere del tipo seguente:

```
$ ./webserver 127.0.0.1 4080 public_html
webserver: Porta: 4080
webserver: Indirizzo: 127.0.0.1
webserver: Descrittore del socket in ascolto: 3
webserver: Bind effettuata con successo
webserver: Server in ascolto con successo sul socket
webserver: [Thread #0] In attesa di richieste
webserver: [Thread #1] In attesa di richieste
webserver: [Thread #2] In attesa di richieste
webserver: [Thread #3] In attesa di richieste
webserver: Connessione con il client (socket 4)
webserver: Thread 0 disponibile (socket 4), occupati 1 su 4
webserver: [Thread #0] Ricevuta richiesta (socket 4): /index.html
webserver: [Thread #0] Risposta inviata (socket 4): /index.html
```

A scopo di test/debug conviene fare in modo che il server produca delle informazioni supplementari relative alla gestione delle richieste HTTP. Queste informazioni devono essere salvate (in modalità *append*) in un file di log nella stessa cartella dove si trova l'eseguibile **`webserver`**. In particolare il server deve salvare all'interno del file di log le richieste fatte dal client e le condizioni di errore descritte in precedenza.

Avvertenze e suggerimenti

- **Test con `telnet`.**
  Per testare la correttezza dell'header HTTP inviato dal server è possibile utilizzare il comando `telnet` invocato nella forma seguente

  ```
  telnet <indirizzo> <porta>
  ```

  dove `<indirizzo>` e `<porta>` sono gli stessi utilizzati in fase di invocazione del server. Il comando `telnet` consente di inserire manualmente la richiesta HTTP, e di ricevere la corrispondente risposta direttamente da terminale. In genere sotto UNIX `telnet` invia il carattere `\n` come terminazione di linea, quindi il server deve essere tollerante come descritto nella sezione "Interpretazione delle richieste". *L'uso di `telnet` non sostituisce – piuttosto completa – l'uso del browser in fase di test dell'applicazione*.
- **Uso della memoria.**
  È consentito utilizzare memoria allocata staticamente per il buffer di ricezione delle richieste e per le strutture dati necessarie all'elaborazione della richiesta stessa. È richiesta l'allocazione dinamica della memoria, invece, per caricare il file richiesto dal client in un buffer temporaneo prima dell'invio. In fase di allocazione della memoria è necessario conoscere la dimensione del file. A tale scopo si può utilizzare la funzione seguente:

  ```
  #include <sys/stat.h>
  #include <fcntl.h>
  int get_filesize( char* filename )
  {
   struct stat file_info;
   if( !stat( filename, &file_info ) )
       return file_info.st_size;
   return -1;
  }
  ```

  che restituisce la dimensione in byte del file di nome `filename`, mentre restituisce il valore -1 in caso di errore.
- **Analisi della richiesta.**
  La richiesta proveniente dal client deve essere interpretata analizzandone il contenuto. In fase di interpretazione della richiesta si consiglia di utilizzare la funzione `strtok` della libreria standard. Per ulteriori informazioni consultare la documentazione della funzione stessa (`man strtok`) o la pagina web `http://www.diee.unica.it/~armano/LT1/pdf/LT1-8b.pdf`.
- **Tipi MIME supportati.**
  Per la gestione dei tipi MIME supportati conviene utilizzare una struttura dati come la seguente:

  ```
  struct {
   char *extension;
   char *mimetype;
  } typeinfo[] = {
   {"pdf", "application/pdf" },
   {0,0}
  };
  ```

dove il campo `extension` è una stringa che rappresenta l'estensione e `mimetype` è una stringa che rappresenta il relativo tipo MIME. La struttura `typeinfo` viene definita con l'insieme di estensioni supportate, con relativi tipi MIME. Il valore speciale {0,0} può essere utilizzato come marcatore di fine array. Quando viene ricevuta una richiesta (valida) la struttura `typeinfo` viene scandita alla ricerca del tipo MIME corrispondente all'estensione del file richiesto.

- **Come realizzare il server**.
  Conviene realizzare il server in modo incrementale, cosicché si possa testare singolarmente il funzionamento di diverse porzioni del codice.
  - Per testare il corretto funzionamento della parte relativa ai socket si può partire da un server che è in grado di servire una sola richiesta alla volta (server mono-processo o iterativo).
  - Si può estendere il server così ottenuto introducendo la creazione dinamica dei thread (come descritto nei lucidi relativi alla programmazione distribuita).
  - Infine si può modificare il server multi-threaded utilizzando thread preallocati piuttosto che thread creati dinamicamente.

  Ovviamente lo scopo del progetto è creare un server concorrente che utilizzi un pool di thread preallocati. *Saranno accettate soltanto conformi alle specifiche descritte in precedenza*.

Valutazione del progetto

Il progetto viene valutato durante lo svolgimento dell'esame. La valutazione prevede le seguenti fasi.

1. **Compilazione dei sorgenti**. Il client e il server vengono compilati attivando l'opzione `-Wall` che abilita la segnalazione di tutti i warning. Si consiglia vivamente di usare tale opzione anche durante lo sviluppo del progetto, *interpretando i messaggi forniti dal compilatore*.
2. **Esecuzione dell'applicazione**. Il client e il server vengono eseguiti simulando una tipica sessione di utilizzo. In questa fase si verifica il corretto funzionamento dell'applicazione e il rispetto delle specifiche fornite.
3. **Esame del codice sorgente**. Il codice sorgente di client e server viene esaminato per controllarne l'implementazione.

Tutte le fasi sopra descritte saranno svolte in ambiente FreeBSD. *Se avete sviluppato l'applicazione con altre varianti UNIX (ad esempio Linux) accertatevi comunque che il programma compili ed esegua correttamente anche sotto FreeBSD.*