

# uhttps

## Progetto di Sistemi Operativi 2

Andrea Mantoni

21 febbraio 2010

### Istruzioni d'uso (per l'utente finale)

#### **Compilazione e installazione**

Nella directory “src” sono forniti due Makefiles: quello predefinito è per i SO Unix-like, mentre “Makefile.win32” è per Windows.

I makefiles definiscono le macros richieste per la corretta compilazione dei sorgenti e linkano tutte le librerie necessarie.

Al termine della compilazione generano un eseguibile nella directory “bin”.

La directory “bin” contiene anche alcuni files per il testing del server.

#### **File di configurazione**

All'avvio il server legge le sue opzioni da un file di configurazione “uhttps.cfg” posto nella current working directory.

In alternativa, è possibile passare il percorso di un file di configurazione alternativo come primo argomento della linea di comando.

Il programma non accetta altri argomenti dalla linea di comando.

Nel caso in cui il file di configurazione non fosse leggibile o malformed le opzioni sono lasciate ai valori di default (elencati in seguito).

Il file di configurazione ha la seguente sintassi:

```
# comment  
option1=value1  
option2=value2
```

Le linee vuote e le linee che iniziano con l'asterisco “#” (commenti) sono ignorate.

Nella seguente tabella sono elencate tutte le opzioni supportate dal server:

Opzione	Valore di default	Descrizione
use_ipv6	yes	Abilita il supporto ad IPv6.
listening_port	80	La porta di ascolto del server.
root_pathname	.	La directory radice del server.

Opzione	Valore di default	Descrizione
log_pathname	access.log	Percorso dove scrivere il log. Sono accettati come file speciali anche stdout e stderr.
usersdb_pathname	uhttps_usersdb.txt	Percorso del database degli utenti (la sintassi è definita nel paragrafo seguente).
max_threads	10	Numero massimo di threads concorrenti generati dal server.
max_processes	1	Numero massimo di processi concorrenti generati dal server.
max_cached_file_size	1024	Dimensione massima di un file nella cache. (Se è "0" la cache è disattivata.)
max_cached_file_count	10	Numero massimo di files nella cache. (Se è "0" la cache è disattivata.)

La modalità del server (multi-processo o multi-thread) è stabilita mediante le opzioni `max_threads` e `max_processes`:

```

se ( max_threads >= max_processes )
    il server opera in modalità multithread
altrimenti
    il server opera in modalità multiprocesso

```

Il numero massimo di richieste HTTP che il server può accettare contemporaneamente è uguale a:  
`max( max_threads, max_processes )`

Se `log_pathname` non è un file speciale il processo principale è detachato dalla console di invocazione.

### ***Database degli utenti***

Il database degli utenti è memorizzato in un file di testo ASCII ad accesso sequenziale.

Ciò ha due vantaggi:

- portabilità del DB (non sarebbe stato lo stesso con un file binario!)
- possibilità di modificare direttamente il file con un editor di testo → non sono necessari tool specifici per la gestione degli utenti

La sua sintassi è la seguente:

```

# comment
username1:password1
username2:password2

```

## Chiusura del server

Il server può essere terminato in due modi:

1. **terminazione immediata**, mediante l'invio dei segnali `SIGTERM`, `SIGQUIT`, `SIGABRT` o gli eventi di console `CTRL_LOGOFF_EVENT` e `CTRL_SHUTDOWN_EVENT`. Questa modalità potrebbe causare la disconnessione forzata dei client attualmente connessi.
2. **terminazione dilazionata/“educata”**: attende che tutte le richieste in elaborazione siano completate prima di terminare.

E' richiesta mediante l'invio del segnale `SIGINT` o l'evento di console `CTRL_C_EVENT`.

Alla ricezione del **primo** segnale di terminazione il server chiude la porta di ascolto.

A questo punto se viene ricevuto un **secondo** segnale di terminazione la chiusura è immediata (come nel primo caso), altrimenti è dilazionata.

## Bugs e limitazioni note

- I metodi HTTP supportati sono GET, PUT e HEAD. Con gli altri metodi il server ritorna al client lo status code “405 Method Not Allowed”.
- Le URL possono avere lunghezza massima di 500 chars (costante definita a compile-time). Le URL più lunghe sono troncate.
- Anche la lunghezza massima degli headers HTTP è limitata a 500 chars.
- Non sono supportati gli headers HTTP multilinea (il messaggio viene interpretato come “malformed”).
- La versione per Windows non supporta la modalità multiprocesso (usa sempre il multithreading). L'uso del multiprocessing sarebbe stato particolarmente inefficiente a causa delle limitazioni della API di Windows (assenza di una funzione `fork`).
- Poiché i processi/threads secondari girano con gli stessi privilegi del processo/thread primario (root/Administrator), un utente remoto è abilitato a lanciare comandi che potrebbero compromettere il sistema.

## Descrizione dei sorgenti

### Convenzioni e name decorating

Il progetto è costituito da diversi moduli debolmente accoppiati tra loro.

Ogni modulo ha associato un header file “.h” che ne descrive l'interfaccia pubblica, ed un file sorgente omonimo “.c” che ne contiene l'implementazione.

Si è adottata la convenzione di nominare tutte le funzioni pubbliche di un modulo con il prefisso “`uhttps_`”, mentre le funzioni private hanno il prefisso “`_`” (underscore), e sono dichiarate statiche nel file di implementazione (.c).

Per migliorare la leggibilità e la mantenibilità del codice, sono state fatte le seguenti scelte:

- tutto il codice è thread-safe (indipendentemente da come è configurato il server), ciò comporta:
  - su Windows, la definizione della macro `_MT` che abilita le versione multithread della

standard library;

- su Unix, l'uso delle funzioni alternative reentrant con “\_r” postfisso (ad es. `time_r` al posto di `time`).
- minimizzazione del codice platform-specific:  
Si sono preferite le funzioni della standard library al posto di quelle non portabili (ad es. `fopen` al posto di `open` e `CreateFile`).

In generale si è cercato di usare la stessa logica di alto livello per entrambe le piattaforme, nascondendo le differenze tra le API di sistema ai livelli più bassi dell'implementazione.

### ***uhttps.h***

Contiene le includes e le definizioni comuni a tutti i moduli.

E' incluso da tutti i file sorgenti del progetto.

### ***main.c***

Rappresenta il processo/thread **principale** del server che apre la porta di ascolto e resta in attesa di nuove connessioni.

All'arrivo di una connessione crea un nuovo processo o thread a seconda della configurazione attuale.

**OSS.:** il server è di tipo **post-forked/threaded**: la creazione dei processi/threads secondari è effettuata dopo la accept della connessione con il client.

Per tenere traccia del numero di processi/threads generati utilizza un contatore `connected_clients_count`. L'incremento della variabile è effettuato dal processo/thread principale, il decremento dal thread secondario quando termina.

Poiché la variabile è acceduta in scrittura dai vari threads, l'accesso è protetto da un mutex su Unix, mentre su Windows è gestita mediante interlocked operations.

Se il server è configurato in modalità multiprocesso è il signal handler di `SIGCHLD` che decrementa il contatore quando un figlio termina.

### ***http\_parser.\****

Effettua il parsing dei messaggi HTTP.

La struct `uttps_http_request_parsed` rappresenta una request HTTP parsata.

La struct `uttps_http_response_parsed` rappresenta una response HTTP parsata.

I campi di queste structs contengono i valori dei singoli headers HTTP a cui il modulo `client_handler` può accedere direttamente.

Le funzioni pubbliche del modulo “trasformano” un messaggio HTTP bufferizzato in una struct e viceversa:

```
void uhttps_parse_http_request_headers( struct
    uttps_http_request_parsed* out_request,
    char* in_request );
void uhttps_parse_http_response_headers( char* out_response,
    struct uttps_http_response_parsed* in_response );
```

## ***client\_handler.\****

Gestisce una singola richiesta HTTP di un client:

legge la richiesta, la elabora ed invia la risposta attraverso un socket connesso.

Ha un'unica funzione pubblica:

```
void
uhttps_handle_http_request( socket_t client_connected_socket )
```

### **Logica:**

```
uhttps_handle_http_request( client_connected_socket )
1.  alloca il buffer per gli headers della response e le structs per il
    parsing
2.  imposta gli headers sempre presenti nella response (HTTP version,
    Server, Date, etc.)
3.  imposta un timeout di 10 sec per la recv con setsockopt → evita che
    un thread/processo resti bloccato a causa di un errore del client
4.  legge la lunghezza degli headers della request → funzione
    _read_request_headers_size (descritta in seguito)
5.  alloca il buffer per gli headers della request
6.  legge gli headers della request con recvn
7.  parse gli headers della request → modulo http_parser
8.  controlla che la request contenga l'Initial Request Line completo →
    se non lo è imposta lo status code "400 Bad Request"
9.  controlla se l'header Authorization è presente → se non lo è imposta
    lo status code "401 Forbidden" e richiede l'autenticazione del client
10. verifica se l'account è presente nel DB degli utenti → se non lo è
    imposta lo status code "403 Unauthorized"
11. se il metodo è PUT, legge anche il body della request con recvn
12. in base al metodo HTTP, invoca una funzione privata di gestione
    apposita. Se il metodo non è supportato imposta lo status code "405"
13. parse la response → modulo http_parser
14. invia la response con due sendn (una per gli headers ed una per il
    body)
15. registra nel log la request → modulo logger
16. chiude il socket
17. disalloca i buffers
```

Le funzioni `recvn` e `sendn` sono versioni modificate delle `readn` e `writen` tratte dal libro "Advanced Programming in the UNIX Environment" di Stevens, Rago.

Provvedono ad invocare più volte `recv` e `send` finché non sono stati ricevuti/inviati tutti i dati richiesti.

### **Logica della funzione privata `_read_request_headers_size`:**

Vengono letti i dati dal socket finché non si incontra il marcatore della fine degli headers (due newlines consecutivi).

La dimensione degli headers è uguale all'offset di questi newlines dall'inizio del messaggio (ottenuto con `strstr`).

Tuttavia, poiché lo standard HTTP non stabilisce una dimensione massima per gli headers si procede per tentativi, incrementando se necessario la dimensione del buffer della `recv` di un chunk per volta ed attendendo 1 secondo prima di riprovare.

**OSS.:** Qui la `recv` viene invocata con il flag `MSG_PEEK`, i dati vengono lasciati nel buffer

di sistema in modo che le `recv` seguenti possano leggerli.

### ***cfg\_parser.\****

Effettua il parsing del file di configurazione.

La `struct uhttp_configuration` contiene tutte le impostazioni del server.

All'avvio del server il modulo `main` ne crea un'istanza globale che viene importata dal modulo `client_handler`.

Poiché questa istanza è acceduta in **sola lettura** dai threads non è necessario un meccanismo di sincronizzazione.

Per rileggere il file di configurazione a seguito di modifiche si può inviare il segnale `SIGHUP` (o l'evento di console `CTRL_CLOSE_EVENT`) oppure riavviare il server.

Nel primo caso il thread principale attenderà la terminazione di tutti i threads secondari prima di scrivere nella istanza condivisa di `struct uhttp_configuration`, evitando così problemi di concorrenza con gli altri threads.

La reinizializzazione del server quindi è effettuata nel thread principale, non nel signal handler.

Il signal handler di `SIGHUP` si limita a settare un flag, la gestione vera e propria del segnale avviene nel thread principale.

### ***usersdb\_parser.\****

Un utente è rappresentato dalla

```
struct user
{
    char username[ _UHTTPS_MAX_USERNAME_LENGTH ];
    char password[ _UHTTPS_MAX_PASSWORD_LENGTH ];
};
```

Il database degli utenti è gestito in memoria come un array di `struct user` referenziato dalla

```
struct users_db
{
    struct user* data;
    size_t nelem;
};
```

All'avvio del server il modulo `main` ne crea un'istanza globale che viene importata dal modulo `client_handler`.

Per quanto riguarda l'accesso concorrente valgono le stesse considerazioni fatte per il modulo precedente.

Funzioni pubbliche:

```
void uhttps_usersdb_read( struct users_db* out_usersdb, const
    char* usersdb_pathname );
boolean uhttps_usersdb_login( const struct users_db* usersdb,
    const char* username, const char* password );
```

La prima effettua il parsing del database degli utenti.

La seconda convalida il login di un utente.

## ***logger.\****

Fornisce la funzionalità di logging al server.

E' utilizzato da tutti i moduli per accedere al file di log e stampare messaggi d'errore.

Funzioni pubbliche:

```
void uhttps_logfile_open( const String log_pathname );
void uhttps_logfile_close();
void uhttps_log_request( String host, String user, String
    request, int status, int bytes);
void uhttps_log_error( char* fmt, ... );
```

Le prime due funzioni aprono e chiudono il file di log.

La `uhttps_log_request` registra le operazioni effettuate secondo il Common Log Format.

L'ultima funzione stampa su `stderr` dei messaggi di debug (se il processo è stato detachato dalla console lo standard error viene reindirizzato sul file "stderr.txt")

## ***file\_cache.\****

Fornisce un meccanismo di caching per i file locali.

La struct `cached_file` rappresenta un singolo file nella cache:

```
struct cached_file
{
    char pathname[ _UHTTPS_MAX_PATHNAME_LENGTH ];
    size_t length;
    void* buffer;
};
```

La cache è implementata come un array di struct `cached_file` ad accesso sequenziale.

La cache segue una politica **FIFO**: mediante un **indice circolare** si sostituisce sempre il file che si trova da più tempo al suo interno.

L'accesso esclusivo alla cache è regolato da un mutex su Unix e da una `CRITICAL_SECTION` su Windows.

Tutti questi oggetti sono dichiarati statici all'interno del modulo, quindi non sono visibili all'esterno.

L'interfaccia pubblica si basa su due sole funzioni:

```
void uhttps_init_file_cache( struct uhttp_configuration* );
int uhttps_read_cached_file( struct cached_file* output,
    const char* pathname,
    const struct uhttp_configuration* current_config );
```

La prima inizializza la cache leggendo i files nella server root directory che non eccedono `max_cached_file_size`.

La seconda è utilizzata dal modulo `client_handler` per accedere ai files locali.

Tutte le richieste GET di file "passano" attraverso la cache, che in base ai propri parametri stabilisce se:

- il file richiesto si trova già nella cache (cache hit) → lo ritorna subito al chiamante
- il file non è presente (cache miss):
  1. lo legge dal disco;
  2. se la sua dimensione è minore di `max_cached_file_size` lo inserisce al posto di quello più vecchio;
  3. lo ritorna al chiamante.