Programmazione in Ambienti Distribuiti I - 01FQT

Prof. Antonio Lioy

A.A. 2002-2003

HTTP adaptation layer
per generico protocollo di scambio dati

Sandro Cavalieri Foschini 101786
Emanuele Richiardone 101790

Indice

1
Introduzione

HClib è una libreria sviluppata in linguaggio C che realizza un HTTP adaptation layer, intendendo con ciò un'interfaccia utile al programmatore per implementare con facilità una connessione HTTP.

Il programmatore, anziché impiegare le funzioni di rete standard offerte dal sistema operativo *NIX, ha a disposizione 7 funzioni principali, più alcune altre di supporto, per inviare in maniera trasparente un qualsiasi dato (binario o testuale) attraverso un canale HTTP con la tradizionale architettura client/server. Il nome stesso delle funzioni della libreria ricorda *HClib*, ovvero *HTTP Channel library*.

Tutti i dati che transitano tra i due capi possono essere di tipo ASCII o binario. Nei due casi i dati possono essere spediti così come sono, oppure possono essere codificati e quindi decodificati senza essere modificati; vi sono per il testo altre codifiche che lo adeguano su 7 bit. I dati di tipo binario possono essere codificati oppure inviati direttamente sul canale sfruttando la caratteristica *8-bit clean* del protocollo HTTP.

La comunicazione tra i due host si affida al protocollo affidabile TCP e, in gran parte seguendo le disposizioni del *Request For Comment* 1945, segue lo standard HTTP 1.0 del maggio 1996. Si è preferita questa versione per la sua semplicità e diffusione.

Inoltre sono stati integrati anche elementi del formato MIME 1.0 (*Multipurpose Internet Mail Extensions*), conforme all'*RFC 2045* e successivi. La codifica cui può essere sottoposto il dato trasmesso si può eseguire impiegando varie conversioni, tra le quali il *quoted-printable* (per i dati di tipo testuale) ed il *base64* (per qualsiasi dato).

La seguente guida è articolata in due parti: nella prima (capitoli 1-6) si illustra schematicamente il modo d'uso delle funzioni per un loro corretto utilizzo da parte del programmatore che vuole utilizzare HClib.

Nella seconda (capitolo 7-15) si spiega il modo in cui le procedure operano a livello di rete: inizializzazione del canale HTTP e creazione dei pacchetti con intestazione conforme alle specifiche imposte dal protocollo.

2
Note sull'installazione

La libreria è composta dai seguenti file:

```
hclib.h
hclib_1.0.c
hclib_encode.h
hclib_subf.h
```

Il primo file è l'header file che definisce i prototipi delle funzioni, le strutture utilizzate, le altre librerie richieste alla compilazione e le definizioni; queste ultime, se ve ne è la necessità, possono qui essere modificate. Inoltre questo file sarà quello che verrà incluso in un programma che utilizzi le librerie libhc1.0.so o libhc1.0.a .

Il file successivo (*hclib_1.0.c*), che include i due seguenti *.h* , è il sorgente della libreria vera e propria; è possibile compilarla in modo statico, e quindi nel momento di linkare un programma con la nostra libreria essa viene inclusa nell'eseguibile, oppure in modo *shared*, che ha il grande vantaggio di non dover aggiornare ogni programma (che usi le *HClib*) ogni volta che si aggiornano le librerie. Illustreremo l'installazione con quest'ultimo modo più comodo, che permette al programma di usare in *run-time* le funzioni compilate in un file a parte (*libhc1.0.so*).

La libreria va quindi compilata (useremo sempre *gcc*) in */usr/lib*, la directory dove si situano di solito le librerie, e dove i compilatori vanno a cercarle:

```
gcc -shared -fPIC -o /usr/lib/libhc1.0.so hclib_1.0.c
```

Quindi si ottiene una libreria dinamica *libhc1.0.so* . L'opzione –fPIC indica che il codice deve essere *Position-Indipendent Code*. Il nome della libreria è importante in quanto deve essere composto da *lib*.so* , perché quando poi vogliamo utilizzarla, oltre al fatto che il codice del programma deve include l'header file *hclib.h*, dobbiamo compilare con l'opzione *–llibreria* , dove *libreria* nel nostro caso è *hc1.0* :

```
gcc -lhc1.0 –o eseguibile sorgente.c
```

Come nota inseriamo brevemente il metodo per compilare la libreria in modo statico, tramite il comando *ar* :

```
gcc -c hclib_1.0.c
ar cr /usr/lib/libhc1.0.a hclib_1.0.o
```

3
Funzioni client

Le funzioni per la gestione del canale HTTP dalla parte del client sono tre: una che esegue le normali operazioni per stabilire la connessione, e le altre due per spedire o ricevere dati.

3.1 Come creare un client

3.1.1 Ricezione di dati

I passi per poter ricevere dati sono semplici, si tratta di:

1) Ottenere il socket descriptor con la funzione **create_hclib()**
2) Si riceve il dato (binario o testo) con la **get_hclib()**

3.1.2 Spedizione di dati

Analogamente i passi per poter inviare dati:

1) Ottenere il socket descriptor con la funzione **create_hclib()**
2) Spedire il dato (binario o testo) con la **post_hclib()**

Per sapere il modo d'uso e i valori di ritorno delle sopraccitate funzioni si rimanda alle seguenti sezioni specifiche per ogni procedura.

3.2 create_hclib()

Funzione per la creazione del canale HTTP verso il server.

La funzione richiede in ingresso

1) il *nome del server* cui connettersi,
2) la sua *porta* e
3) il valore in secondi per il *timeout*.

Il valore di ritorno è il *socket descriptor* se l'operazione si è conclusa con successo.

```
int
create_hclib(const char *server, const char *port, const
unsigned int timeout);
```

```
IN:     const char *server  (nome/IP del server),
        const char *port    (porta/servizio del server),
        const unsigned int timeout  (valore in secondi del timeout)

RETURN: int (socket descriptor)
```

Il *nome del server* può essere immesso sia come IP numerico (ad esempio stringa "*192.168.0.1*") oppure nome di rete (stringa "*mioserver*"), così come la *porta* che può anche essere identificata del nome del servizio associato (ad esempio "*80*" oppure "*http*").

Si deve inoltre impostare *timeout*, l'intervallo di tempo dopo il quale -in assenza di risposta del server- la connessione cade. Questo valore è da esprimersi in secondi.

La procedura ritorna il valore intero positivo del *socket descriptor*, se è terminata con successo, un valore negativo altrimenti.

Esempio d'uso

```
#include "hclib.h"
#define TIME 5
[...]

int sd, listalen;
char lista[1024], mime[64], tenc[17];
[...]

if((sd = create_hclib("mioserver", "80", TIME)) < 0) return 1;

get_hclib(sd, lista, &listalen, tenc, mime);
printf("%s\n", lista);
return 0;
[...]
```

3.3 post_hclib()

Funzione per inviare il contenuto di un buffer al server.

La **post_hclib()** è la funzione responsabile della spedizione di dati al server.

Si è fissata una grandezza massima del buffer inviabile pari a 170kB, calcolata in modo che il massimo trasferimento dati sulla rete nel caso il contenuto venga codificato sia pari a 512kB.

Sul server deve essere attiva la corrispondente funzione di "ricezione" **recv_hclib()**, in grado di accettare i dati ed inviare la conferma di ricevimento.

```
int
post_hclib(int sockd, const void *buff, int bufflen, char
*tenc, char* mime);


 IN:     int sockd        (socket descriptor creato da create_hclib()),
         const void *buff (buffer da INVIARE),
         int bufflen      (lunghezza del buffer NON deve superare i 170kB),
         char *tenc       (transfer encoding con il quale codificare buff),
         char *mime       (mimetype del dato contenuto in buff)

 RETURN: int              (0 se OK, numero negativo se errore, vedi tabella)
```

In ingresso la funzione richiede:

1) il *socket descriptor* creato in precedenza con la **create_hclib()**
2) il *buffer* che contiene i dati da inviare,
3) *bufflen* che è il valore intero che esprime la sua lunghezza ,
4) *l'encoding* con il quale codificare i dati e
5) il programmatore deve specificare il *tipo MIME* dei dati.

Il valore di ritorno di questa funzione segue le specifiche dell'intera libreria: vale 0 se l'operazione si è conclusa con successo, negativo altrimenti (vedi tabella sezione 6.2).

Esempio d'uso

```
#include "hclib.h"
[...]

int  sd, i, bufflen;
char buff[] = "prova POST";
char tenc[] = "8bit";
char mime[] = "text/plain";
[...]

bufflen = strlen(buff);
if((sd = create_hclib("192.168.0.1","http", 5)) < 0) return 1;

i= post_hclib(sd, buff, bufflen, tenc, mime);

printf("return: %d\n", i);

return 0;
[...]
```

3.4 get_hclib()

Funzione per richiedere il contenuto di un buffer al server

La **get_hclib()** è la procedura responsabile della spedizione di dati al server; ha funzione simmetrica rispetto alla **post_hclib()**.

Sul server deve essere attiva la corrispondente funzione di "spedizione" **send_hclib()**, in grado di fornire i dati e restituirli alla **get_hclib()**.

```
int
get_hclib(int sockd, char *buff, int *bufflen,char *tencout,
char *mimeout);

 IN:     int sockd        (socket descriptor, creato con create_hclib())

 OUT:    char *buff       (buffer da RICEVERE),
         int *bufflen     (lunghezza del buffer ricevuto),
         char *tencout    (transfer encoding con il quale decodificare buff)
         char *mimeout    (mimetype del dato contenuto in buff)

 RETURN: int              (0 se OK, numero negativo se errore, vedi tabella)
```

In ingresso la funzione richiede:

1) il *socket descriptor* creato in precedenza con la **create_hclib()**

In uscita fornisce:

1) il *buffer* ricevuto
2) la lunghezza *bufflen* del buffer
3) *tencout*, ovvero la codifica che è stata applicata al buffer
4) *mimeout* che è il tipo MIME del dato contenuto nel buffer

I valori di errore restituiti sono sempre i soliti individuati dalla tabella riportata nella sezione 6.2.

Esempio d'uso

```
#include "hclib.h"
[...]

int   sd, i, bufflen;
char  buff[4096];
char  mime[64], tenc[17];
[...]

if((sd = create_hclib("elettra", "http", 5)) < 0) return 1;
```

```
i=get_hclib(sd, buff, &bufflen, tenc, mime);

printf("return: %d\n", i);

printf("\n%s\n%d, %s\n", buff, bufflen, mime);
[...]

return 0;
[...]
```

4
Funzioni server

Le funzioni per la gestione del canale dalla parte del server sono quattro: due che servono per l'inizializzazione della connessione e per attivare le impostazioni desiderate su di esse; le altre due sono le corrispondenti delle due funzioni client responsabili della trasmissione.

4.1 Come creare un server

3.1.1 Ricezione di dati

I passi per poter ricevere dati sono semplici, si tratta di:

1) Creare il socket descriptor in ascolto con la funzione `listen_hclib()`
2) Accettare il nuovo socket descriptor connesso con `accept_hclib()`
3) Si riceve il dato (binario o testo ASCII) con la `recv_hclib()`

3.1.2 Spedizione di dati

Analogamente i passi per poter inviare dati:

1) Creare il socket descriptor in ascolto con la funzione `listen_hclib()`
2) Accettare il nuovo socket descriptor connesso con `accept_hclib()`
3) Spedire il dato (binario o testo ASCII) con la `send_hclib()`

Per sapere il modo d'uso e i valori di ritorno delle sopraccitate funzioni si rimanda alle seguenti sezioni specifiche per ogni procedura.

4.2 listen_hclib()

Funzione per la creazione del canale HTTP verso il client.

La `listen_hclib()` è rivolta a costruire la struttura contenente le informazioni di rete tali da accettare connessioni client in ingresso.

Inoltre imposta le opzioni corrette sul socket descriptor in ascolto che ritorna, ad esempio pulisce lo stack da una precedente esecuzione di `bind()`).

```
int
listen_hclib(u_int16_t port, unsigned int backlog)

 IN:     u_int16_t port            (porta del server in attesa),
         unsigned int backlog      (numero di connessioni accodabili)
 RETURN: int sd                    (socket descriptor in ascolto)
```

In ingresso la funzione richiede:

1) Il numero *port* della porta su cui attendere
2) Il numero *backlog* di connessioni client da tenere accodate in attesa

Il valore di ritorno è il *socket descriptor* in attesa se positivo, un codice di errore se negativo.

Per l'esempio d'uso si veda quello della seguente funzione **accept_hclib()**.

4.3 accept_hclib()

Funzione per l'accettazione delle connessioni client.

La funzione è bloccante ed aspetta che un client si connetta sul socket descriptor inizializzato dalla **listen_hclib()**.

```
int
accept_hclib(int lsd)

 IN:     int lsd    (socket descr. inizializzato da listen_hclib())

 RETURN: int        (nuovo socket descriptor connesso)
```

In ingresso la funzione richiede:

1) Il socket descriptor in attesa inizializzato dalla **listen_hclib()**

Restituisce nuovi socket descriptor connessi; infatti è utilizzata per esempio all'interno di cicli che utilizzano le funzioni **recv_hclib()** e **send_hclib()** e che successivamente chiudono tale socket connesso.

Questa procedura è in pratica una ridefinizione della sys/socket.h `accept()`, ma che utilizza la funzione proprietaria della libreria **hcerror()** per riportare gli errori.

Esempio d'uso

```
#include "hclib.h"
[...]

#define BACKLOG 15
#define PORT    1235
[...]

char dati[1024];
int sd;
[...]


if((sd = listen_hclib(PORT, BACKLOG)) < 0) return 1;

for(;;){
     csd = accept_hclib(sd);

     riempidati(dati);
     i=send_hclib(csd, dati, sizeof(dati), “7bit”,
"text/plain");
     if(i < 0) continue;
}
return 0;
[...]
```

4.4 recv_hclib()

Funzione per ricevere i dati spediti con la post_hclib() dal client.

La **recv_hclib()** riceve le richieste che richiedono l'invio di dati generate dalla **post_hclib()** del lato client.

```
int
recv_hclib(int sockd, char *buff, int *bufflen, char *tencout,
char *mimeout)


IN: int sockd    (sd, creato con listen_hclib() e connesso da accept_hclib()),

OUT: char *buff       (buffer DA RICEVERE),
     int *bufflen     (lunghezza del buffer ricevuto),
     char *tencout    (transfer encoding con il quale decodificare buff)
     char *mimeout    (mimetype del dato contenuto in buff)

RETURN: int           (0 se OK, -1 se ha ricevuto GET anzichè POST, numero
negativo se errore)
```

In ingresso la funzione richiede:

1) Il *socket descriptor* creato in precedenza con **listen_hclib()** e connesso da **accept_hclib()**

In uscita fornisce:

1) il *buffer* "riempito" con i dati spediti dalla **post_hclib()**
2) la lunghezza *bufflen* del buffer
3) quale *encoding* è avvenuto sul buffer
4) *mimeout* che è il tipo MIME del dato contenuto nel buffer

I valori di ritorno sono quelli della tabella riportata nella sezione 6.2.

Nota - chiude il socket

Esempio d'uso

```
#include "libhc.h"
[...]

int   sd, csd;
int   i, bufflen;
char  buff[4096], mime[64], tenc[17];
[...]

if((sd = listen_hclib(80, 10)) < 0) return 1;

for(;;){
    csd = accept_hclib(sd);
    i=recv_hclib(csd, buff, &bufflen, tenc, mime);
    printf("return: %d\n", i);
    printf("\n%s\n%d, %s\n", buff, bufflen, mime);
}
return 0;
[...]
```

4.5 send_hclib()

Funzione per spedire i dati richiesti con la get_hclib() dal client.

La **send_hclib()** riceve le richieste che richiedono l'invio di dati generate dalla **get_hclib()** del lato client.

```
int
send_hclib(int sockd, void *buff, int bufflen, char *tenc, char
*mime)
```

```
 IN:  int sockd   (sd, creato con listen_hclib() e connesso da accept_hclib()),
      char *buff        (buffer DA INVIARE),
      int bufflen       (lunghezza del buffer ricevuto),
      char *tenc        (transfer encoding con il quale codificare buff)
      char *mime        (mimetype del dato contenuto in buff)

 RETURN: int             (0 se OK, -1 se ha ricevuto POST anzichè GET, numero
negativo se errore)
```

In ingresso la funzione richiede:

1) Il *socket descriptor* creato in precedenza con **listen_hclib()** e connesso da **accept_hclib()**
2) Il *buffer* da inviare
3) la lunghezza *bufflen* del buffer
4) quale *encoding* si desidera effettuare sul buffer
5) *mime* che è il tipo MIME del dato contenuto nel buffer

I valori di ritorno sono quelli della tabella riportata nella sezione 6.2.

Nota - chiude il socket

Esempio d'uso

```
#include "hclib.h"
[...]

int   sd, csd;
int   i, bufflen;
char  buff[] = "prova SEND";
char  tenc[]= “7bit”, mime[]="text/plain";
[...]

bufflen=strlen(buff);

if((sd = listen_hclib(80, 10)) < 0) return 1;

for(;;){
    csd = accept_hclib(sd);
    i=send_hclib(csd, buff, bufflen, tenc, mime);
    printf("return: %d\n", i);
  }
  return 0;
[...]
```

5
Funzioni aggiuntive

Per lo sviluppo della libreria sono state sviluppare una serie di funzioni di corredo, incluse nell'header file *hclib_subf.h* e *hclib_encode.h* : le elenchiamo brevemente.

5.1 break_line()

Funzione per la conversione di un testo in un testo formattato

```
int
break_line(char *dst, const char *src, unsigned int len);

 IN:    char *dst          (puntatore stringa di destinazione),
        const char *src    (puntatore al dato di origine),
        unsigned int len   (lunghezza dato di origine)

 RETURN: int               (lunghezza stringa di destinazione)
```

5.2 base64_encode()

Funzione per la codifica di un dato in base64

```
int
base64_encode(char *dst, const char *src, unsigned int len);


 IN:    char *dst          (puntatore stringa di destinazione),
        const char *src    (puntatore al dato di origine),
        unsigned int len   (lunghezza dato di origine)

 RETURN: int               (lunghezza stringa di destinazione)
```

5.3 base64_decode()

Funzione per la decodifica di una stringa in base64

```
int
base64_decode(char *dst, const char *src, unsigned int len)
```

```
 IN:     char *dst           (puntatore dato di destinazione),
         const char *src     (puntatore alla stringa origine),
         unsigned int len    (lunghezza stringa di origine)

 RETURN: int                 (lunghezza dato di destinazione)
```

5.4 qp_encode()

Funzione per la codifica di una stringa in quoted printable

```
int
qp_encode(char *dst, const char *src, unsigned int len)


 IN:     char *dst           (puntatore stringa di destinazione),
         const char *src     (puntatore alla stringa origine),
         unsigned int len    (lunghezza stringa di origine)

 RETURN: int                 (lunghezza stringa di destinazione)
```

5.5 qp_decode()

Funzione per la decodifica di una stringa in quoted printable

```
int
qp_decode(char *dst, const char *src, unsigned int len)

 IN:     char *dst           (puntatore stringa di destinazione),
         const char *src     (puntatore alla stringa origine),
         unsigned int len    (lunghezza stringa di origine)

 RETURN: int                 (lunghezza stringa di destinazione)
```

5.6 hc_error()

Funzione per la formattazione dei messaggi di errore

```
void
hcerror(const char *text)

IN: const char *text         (testo da formattare)
```

6
Valori di ritorno

6.1 Messaggi di errore

I messaggi di errore della libreria sono formattati dalla funzione **hc_error()** e visualizzati con la forma

(HClib error) LEVEL - TEXT: ERRNO

Sia le funzioni lato client che lato server possono ritornare i valori numerici negativi (a cui corrisponde un messaggio di errore visualizzato su *stderr*) riportate dalla seguente tabella:

6.2 Tabella riassuntiva

```
  0 : function returned successfully
-33 : GET/POST mismatch (used by server to identify client requests)
 -1 : ERROR - buffer to send/receive is too long (max is 170kB)
 -2 : ERROR - mimetype/encoding is too long
 -3 : ERROR - mimetype/encoding unknown
 -4 : ERROR - write on socket failure
 -5 : ERROR - read from socket failure
 -6 : WARNING - server has not closed the connection
 -7 : WARNING - received buffer is empty
 -8 : ERROR - server response is not a 200 OK
 -9 : ERROR - client(server) request(response) has an incorrect header
-10 : ERROR - server cannot send a 200 OK
-11 : ERROR - remote side has closed the connection
-12 : ERROR - server can't close the connection
```

7
Basi per la comprensione di HClib

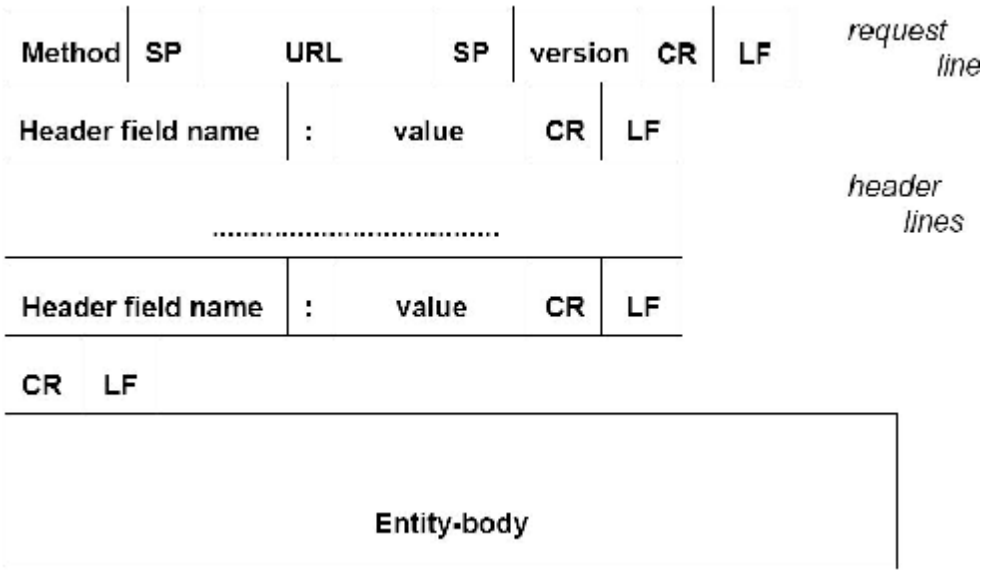
Una sommaria descrizione del protocollo HTTP è utile per il resto della trattazione.

Il protocollo HTTP si basa sul modello *request/response*. La transazione è sempre iniziata dal client che si collega ad un server per formulare una richiesta. Il server a sua volta invia la risposta e la transazione è conclusa.

L'HTTP, al contrario dei protocolli *FTP*, *RSH*, mail (*SMTP*) e molti altri che sono nati per determinate applicazioni (quindi per trasferire dati di un certo tipo), è stato pensato esplicitamente per trasferire dati di qualsiasi tipo.

È da notare che il messaggio HTTP segue la stessa struttura di un messaggio e-mail, ma non è necessario un trasferimento 7-bit compatibile. I messaggi *request* o *response* sono composti da un *header* e da un *body*; si inizia con un metodo e seguono nessuno o più header del messaggio, poi una riga vuota ed in seguito c'è il corpo del messaggio, nel quale si può trasferire sempre qualcosa di binario senza bisogno di successive codifiche.

La seguente illustrazione riporta l'esempio di un *HTTP request message*.



7.1 Scopo della libreria

HClib permette di attivare con facilità una connessione HTTP per inviare in maniera trasparente un qualsiasi dato (binario o testuale) avvalendosi della possibilità di codificare il dato in *quoted-printable* o *base64* oppure inviarlo direttamente sul canale utilizzando l'aspetto *8-bit clean* del protocollo.

Le funzioni della libreria si occupano della spedizione e ricezione del messaggio aggiungendo automaticamente un'intestazione al pacchetto conforme alle specifiche del protocollo.

8
Inizializzazione dei canali

Le tre funzioni per la creazione del canale HTTP svolgono le tipiche funzioni di rete per la creazione di un socket descriptor e l'impostazione delle corrette opzioni su di esso, quali gestione di timeout, numero di connessioni in entrata da accettare ed altre.

Quindi dal lato client chiamando la funzione **create_hclib()** non solo sarà inizializzato il socket descriptor, ma è chiamata anche la funzione `connect()` che in pratica avvierà il *TCP three-way handshake* creando il collegamento tra il socket locale e il suo remoto specificato tramite l'identificativo (indirizzo e porta) passato come parametro alla funzione.

Analogamente dal lato server con la funzione **listen_hclib()** non solo si ottiene il socket descriptor (con porta assegnata secondo il parametro che la funzione accetta in ingresso), ma è anche "attivato" dalla funzione `listen()` in modo da poter accettare connessioni in ingresso.

A questo punto è possibile richiamare la funzione della libreria **accept_hclib()** che preleva il primo collegamento disponibile nella coda delle richieste in attesa, restituendo un nuovo socket descriptor connesso con quello del client, su cui può iniziare la trasmissione e ricezione dei dati.

Si noti che la funzione è bloccante: nel caso non ci siano richieste in arrivo la procedura si blocca finché non ne riceve una da processare.
La trattazione segue elencando alcune particolarità di queste funzioni configurabili dal programmatore che utilizza la libreria.

8.1 Timeout

Le funzioni client **get_hclib()** e **post_hclib()** sono inizializzate dalla **create_hclib()** che con queste righe di codice

```
const unsigned int time_val;   //valore in secondi del timeout
int sd;                        //socket descriptor

struct timeval timeout;
[…]

timeout.tv_sec = time_val;
timeout.tv_usec = 0;
[…]

setsockopt(sd, SOL_SOCKET, SO_RCVTIMEO, &timeout,
sizeof(timeout));
[…]
```

imposta al socket descriptor - grazie alla funzione setsockopt() – l'opzione **SO_RCVTIMEO** che specifica il tempo di timeout in ricezione dopo il quale riportare un errore.

È possibile passare come parametro alla stessa **create_hclib()** il valore in secondi trascorsi i quali tutte le recv() di **get_hclib()** e **post_hclib()** andranno in timeout riportando questo errore all'utente.

NOTA: In effetti per semplicità è stato tralasciato il passaggio del parametro relativo ai microsecondi, anche se questo avrebbe permesso al programmatore di impostare un valore di timeout con l'approssimazione non di un microsecondo, ma di circa 100 millisecondi che è approssimativamente il limite di risoluzione dei sistemi *NIX).

Si è scelto di impostare il timeout solo alle funzioni client in quanto è sufficiente riscontrare questa situazione quando:

1) durante la **get_hclib()** non si riesce a ricevere tutti i dati spediti dalla **send_hclib()** del server;

2) durante la **post_hclib()** non si riceve la conferma che la **recv_hclib()** del server ha ricevuto tutti i dati.

Così risulta possibile che il server subito dopo aver spedito i dati si disconnetta e che comunque anche il client può interrompere la connessione in ogni momento. In questo caso il server non registrerà nessuna condizione d'errore.

8.2 bind() "occupato"

Talvolta, quando si fa prova a far ri-partire il server la funzione bind() fallisce riportando l'errore "Address already in use". Ciò è dovuto al fatto che il socket che era connesso al primo avvio del server sta ancora "occupando" la porta.

La soluzione al problema è o aspettare (circa un minuto…) che venga rimosso dalle procedure di sistema del kernel del SO, oppure adottare il sistema utilizzato dalla libreria dalla funzione server **listen_hclib()** che contiene queste righe di codice

```
int sd;        // socket descriptor
int yes=1;
[…]

if(setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes))
== -1)    {hcerror("setsockopt()"); return -3;}
[…]
```

che permettono al programma di riutilizzare la porta. Infatti con la funzione

setsockopt() si imposta al socket descriptor l'opzione **SO_REUSEADDR** che permette il riuso della porta su quell'indirizzo locale.

La funzione **create_hclib()** che ha incarico simile dal lato client non riporta questa soluzione, in quanto non è necessario impostare il socket client su una prefissata porta con la bind(). Dal lato client non ci preoccupiamo della porta locale (scelta "casualmente" dal kernel) ma solo della porta del server a cui dovremo connetterci (che chiaramente deve essere nota a priori).

8.3 Backlog

La funzione server **listen_hclib()** richiede il parametro *backlog* che ha una funzione analoga a quello della funzione di rete listen(), ossia specifica il numero di connessioni da tenere accodate in attesa. Ciò significa che le connessioni in ingresso dei client aspetteranno in questa coda finché non verranno processate dalla funzione **accept_hclib()**, e backlog è il limite di quante se ne possono accodare.

La maggior parte dei sistemi limita questo numero a circa 20, quindi per ottenere prestazioni ottimali si consiglia di inserire un numero compreso tra 15 e 20.

9
Costruzione dei pacchetti

I pacchetti sono inviati in rete come stringhe di testo contenenti l'intestazione (l'*header* HTTP) e, per le funzioni che lo prevedono, il campo dati (*l'Entity-Body*).

Nei prossimi capitoli si illustreranno i pacchetti spediti dalle funzioni client e server della libreria.

9.1 get_hclib() – HTTP Request con method GET

Questa funzione client deve spedire un pacchetto di richiesta dati al server, il quale con la funzione ***send_hclib()*** *controllerà la sintassi della richiesta e invierà il pacchetto dati desiderato.*

Il messaggio spedito dalla **get_hclib()** è conforme allo standard HTTP 1.0 contenuto nell'*RFC 1945*.

Esso è del tipo ***Full-Request***, contiene la ***Request-Line*** del tipo:

"GET<SP>/<SP>HTTP/1.0<CR><LF>"

che contiene il ***Method*** "GET", seguito dal ***Request-URI*** "/" e dalla versione del protocollo "HTTP/1.0".

Gli elementi sono separati dal carattere di spaziatura <SP>, e non sono ammessi altri caratteri <CR> o <LF> ad eccezione della sequenza finale che deve essere <CR><LF>.

NOTA: Questa non è una richiesta di tipo *Simple-Request* (definita in HTTP/0.9), poiché riporta come ultimo campo la versione HTTP, caratteristica non richiesta per una *Simple-Request*.

Per le esigenze della libreria HClib non è necessario che la richiesta contenga *Request-Header* o *Entity-Header* aggiuntivi, tutti previsti dallo standard come campi opzionali.

Trattandosi di una richiesta molto semplice, che non richiede una "configurazione" da parte del programmatore che usa la libreria HClib, questa è semplicemente spedita sul canale come stringa costante.

9.2 send_hclib() – HTTP Response

Questa funzione server deve spedire un pacchetto dati al client, che comprende un'intestazione che verrà controllata dalla funzione client ***get_hclib()*** *per l'accettazione del pacchetto.*

Il messaggio spedito dalla **send_hclib()** è del tipo HTTP ***Full-Response***, contiene come prima riga la ***Status-Line*** del tipo:

"HTTP/1.0<SP>200<SP>OK<CR><LF>"

che contiene la versione del protocollo "HTTP/1.0", seguita dal numero "200" corrispondente allo ***Status-Code*** e dalla ***Reason-Phrase*** "OK", che è il testo associato al precedente codice numerico. Anche qui l'unico carattere di spaziatura ammesso tra le parti è lo spazio <SP>, mentre la stringa deve finire con la sequenza <CR><LF>.

Nella libreria non sono stati implementati altri *status code* oltre il "200" che corrisponde alla condizione in cui la richiesta impartita dal client con la sua funzione **get_hclib()** sia stata ricevuta, interpretata e accettata con successo.

Unico ***Response-Header*** presente nel pacchetto HTTP inviato al client è il campo "Server: " che contiene nome e versione della libreria HClib utilizzata; il campo che può essere modificato dal programmatore cambiando il

```
#define HCVER "HClib/1.0"
```

presente nell'header file della libreria "hclib.h"

Seguono i due ***Entity-Header*** "Content-Lenght: " e "Content-Type: ", che definiscono informazioni sul contenuto dell'***Entity-Body*** (ovvero il contenuto del messaggio che si vuole spedire).

Questi stabiliscono:

Content-Length	Questo campo indica la lunghezza dell'*Entity-Body*. Essa è specificata con un numero decimale che indica il numero dei byte.
Content-Type	Indica il tipo e il sottotipo MIME dei dati inseriti nell'*Entity-Body*.

Come da specifiche dell'*RFC 1945* che definisce l'HTTP 1.0 il campo *Content-*

Length è obbligatorio, in quanto si conosce la lunghezza del campo dati spedito: essa è pari alla lunghezza del buffer da inviare nel caso il dato non sia codificato, o pari al valore dato in uscita dalle funzioni di codifica di cui la libreria è provvista. Si consulti la sezione "Codifiche" nel capitolo 13 per maggiori dettagli.

Il campo *Content-Type* può contenere i seguenti tipi MIME riconosciuti: "text", "message", "application", "image", "audio" e "video". L'RFC prevede anche l'uso del tipo "multipart", che permette più codifiche in un unico messaggio che non si è utilizzato in quanto la libreria spedisce messaggi di un solo tipo.

In caso il programmatore passi alla funzione un tipo MIME non riconosciuto, questa lo avvisa producendo un errore. Non sono attuati controlli sul sottotipo MIME, che risulta quindi libero.

Il pacchetto comprende anche due campi non standard HTTP, ma utili per la corretta comprensione del contenuto dei dati tra client e server.

Questi sono "*MIME-Version: "* e "*Content-Transfer-Encoding: "* che stabiliscono:

MIME-Version	La versione della codifica MIME utilizzata. Al momento l'unica versione esistente è l'1.0.
Content-Transfer-Encoding	Indica il tipo di trasformazione (codifica) che è stata usata per trattare il contenuto dei dati trasportati.

Il campo "MIME-Version: 1.0" è necessario poiché il contenuto del pacchetto è conforma all'*RFC 2045* "*Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*".

Il campo *Content-Transfer-Encoding* segue le specifiche dettate dagli *RFC 2045, 2046, 2047, 2048* e *2049.* I valori possibili del campo sono "7bit", "quoted-printable", "base64", "8bit", e "binary". Nel caso il programmatore passi come argomento alla funzione un *Content-Transfer-Encoding* scorretto la funzione si interrompe inviando un errore.

Nel caso che il *Content-Transfer-Encoding* sia di un tipo incompatibile con il tipo MIME, dichiarati dal programmatore, la funzione avverte con messaggio di *WARNING*. Infatti i dati potrebbero essere interpretati non correttamente dal client.

Segue l'*Entity-Body*: i dati che il programmatore vuole spedire all'utente che ne ha fatto richiesta col client. La sua presenza è segnalata dall'inclusione del campo "Content-Length" nell'header.

Il contenuto del pacchetto è rappresentato dalla seguente struttura:

```
#define HCVER "HClib/1.0"
#define MIMEver "1.0"

struct textstr{
    char *campo;
} text[] = {
    "HTTP/1.0 200 OK\r\n",
    "Content-type: ",
    "",                                  //2
    "\r\nContent-lenght: ",
    "",                                  //4
    "\r\nMIME-Version: ",
    MIMEver,                             //6
    "\r\nContent-transfer-encoding: ",
    "",                                  //8
    "\r\nServer: ",
    HCVER,
    "\r\n\r\n",
    "",                                  //12
};
```

9.3 post_hclib() – HTTP Request con method POST

Questa funzione client deve spedire un pacchetto di dati al server, che comprende un'intestazione. Il server con la funzione ***recv_hclib()*** *controllerà la sintassi dell'header e invierà conferma di avvenuta ricezione.*

Il primo campo del pacchetto inviato è una ***Full-Request***, contiene la ***Request-Line*** del tipo:

"POST<SP>/<SP>HTTP/1.0<CR><LF>"

che contiene il ***Method*** "POST", seguito dal ***Request-URI*** "/" e dalla versione del protocollo "HTTP/1.0".

Gli elementi sono separati dal carattere di spaziatura <SP>, e non sono ammessi altri caratteri <CR> o <LF> ad eccezione della sequenza finale che deve essere <CR><LF>.

Il metodo POST implica che il server di destinazione accetti l'entità inclusa nella richiesta, che è l'ultimo campo spedito.

L'header comprende tutti i campi già descritti per la funzione **send_hclib()** nel precedente capitolo 9.2.

In particolare sono di nuovo presenti i due ***Entity-Header***: il campo obbligatorio *Content-Lenght* che necessariamente deve specificare la

lunghezza in byte del messaggio spedito, e il campo *Content-Type* che ne descrive il tipo MIME.

Il programmatore è avvisato da un messaggio d'errore ritornato dalla funzione se tenta di spedire un dato con una codifica di tipo MIME non riconosciuta.

Seguono i due campi non standard HTTP, ma utili per la corretta comprensione del contenuto dei dati tra client e server: *MIME-Version* e *Content-Transfer-Encoding* che stabiliscono la versione del MIME usata (al momento l'unica definita è la 1.0) e la codifica con cui sono stati trattati i dati allegati.

Così come c'è un controllo per il tipo MIME, anche nel caso il programmatore passi come argomento alla funzione un *Content-Transfer-Encoding* scorretto la funzione si interrompe inviando un errore. Inoltre come nel caso della **send_hclib()** , viene fatto un controllo di ammissibilità tra questo campo e il tipo MIME.

Segue l'*Entity-Body*, i dati che il programmatore vuole spedire all'utente che ne ha fatto richiesta col client. La sua presenza è segnalata dall'inclusione del campo "Content-Length" nell'header.

Il contenuto del pacchetto è rappresentato dalla seguente struttura:

```
#define MIMEver "1.0"

struct textstr{
  char *campo;
} text[] = {
    "POST / HTTP/1.0",
    "\r\nContent-type: ",
    "",                                          //2
    "\r\nContent-lenght: ",
    "",                                          //4
    "\r\nMIME-Version: ",
    MIMEver,                                     //6
    "\r\nContent-transfer-encoding: ",
    "",                                          //8
    "\r\n\r\n",
    "",                                          //10
};
```

9.4 recv_hclib() – HTTP Response

*Questa funzione server deve ricevere il pacchetto di dati spedito dal client, che comprende un'intestazione. La funzione del client **post_hclib()** riceverà inoltre la conferma di avvenuta spedizione.*

Il messaggio spedito dalla **recv_hclib()** è del tipo HTTP ***Full-Response***,

contiene come prima riga la ***Status-Line*** del tipo:

"HTTP/1.0<SP>200<SP>OK<CR><LF>"

che contiene la versione del protocollo "HTTP/1.0", seguita dal numero "200" corrispondente allo ***Status-Code*** e dalla ***Reason-Phrase*** "OK", che è il testo associato al precedente codice numerico. Anche qui l'unico carattere di spaziatura ammesso tra le parti è lo spazio <SP>, mentre la stringa deve finire con la sequenza <CR><LF>.

Nella libreria non sono stati implementati altri status code oltre il "200" che corrisponde alla condizione in cui la richiesta impartita dal client con la sua funzione **post_hclib()** sia stata ricevuta, interpretata e accettata con successo.

Così come per il pacchetto inviato dalla **send_hclib()** l'unico ***Response-Header*** presente nel pacchetto HTTP inviato al client è il campo *Server* che contiene nome e versione della libreria HClib utilizzata e che il programmatore può modificare attraverso il

```
#define HCVER "HClib/1.0"
```

presente nell'header file della libreria "hclib.h"

Seguono i due ***Entity-Header*** *Content-Lenght*, numero decimale che indica il numero di byte della lunghezza dell'*Entity-Body* e *Content-Type*, il suo tipo MIME.

Anche questo pacchetto comprende due campi non standard HTTP, *MIME-Version* che riporta la versione della codifica MIME utilizzata e *Content-Transfer-Encoding* che indica la (eventuale) codifica a cui sono stati sottoposti i dati inviati, è utile per la corretta comprensione del contenuto dei dati tra client e server.

Tutti questi campi sono spediti da questa funzione alla corrispettiva parte client **post_hclib()** che ritornerà i dati in essi contenuti al programmatore come conferma di avvenuta ricezione (e interpretazione…) del pacchetto.

10
Ricezione degli header

La funzione **recv_hclib()** deve ricevere il messaggio spedito dalla **post_hclib()** del client correttamente intestato e restituirlo al programmatore.

Essa esegue quindi il *parsing* dell'intestazione ricavando i dati utili alla corretta interpretazione del contenuto del messaggio incluso nell'*Entity-Body*. La procedura di parsing è in grado di ricavare tutte le informazioni dell'header in qualunque ordine esse siano state spedite nel pacchetto. L'importante è che siano presenti tutti i campi indicati nel paragrafo 9.3, altrimenti verrà segnalato un errore.

```
[...]
  p1 = strCRLF(header);
  p2 = strCRLF(p1);
  p3 = strCRLF(p2);
  p4 = strCRLF(p3);
  ip1 = p2 - p1;
  ip2 = p3 - p2;
  ip3 = p4 - p3;
  ip4 = strCRLF(p4) - p4;
  p1[ip1-2] = '\0';
  p2[ip2-2] = '\0';
  p3[ip3-2] = '\0';
  p4[ip4-2] = '\0';
  if(sscanf(p1, "%s", cont) == 0)       [...]
  if(strcasecmp(cont, "Content-type:") == 0){
    strcpy(mimeout, (char *)&p1[strlen(cont)+1]);
    found = 1;
  } else {
    if(strcasecmp(cont, "Content-transfer-encoding:") == 0){
      strcpy(tencout, (char *)&p1[strlen(cont)+1]);
      found = 1;
    } else {
      if(strcasecmp(cont, "Content-lenght:") == 0){
        msg_len = atoi((char *)&p1[strlen(cont)+1]);
        found = 1;
      } else {
        if(strcasecmp(cont, "MIME-Version:") == 0){
          strcpy(mimever, (char *)&p1[strlen(cont)+1]);
          found = 1;
        }
      }
    }
  }
  if(sscanf(p2, "%s", cont) == 0)       [...]
  if(strcasecmp(cont, "Content-lenght:") == 0){
    msg_len = atoi((char *)&p2[strlen(cont)+1]);
    found++;
  } else {
    if(strcasecmp(cont, "Content-type:") == 0){
      strcpy(mimeout, (char *)&p2[strlen(cont)+1]);
      found++;
    } else {
      if(strcasecmp(cont, "Content-transfer-encoding:") == 0){
        strcpy(tencout, (char *)&p2[strlen(cont)+1]);
        found++;
      } else {
        if(strcasecmp(cont, "MIME-Version:") == 0){
```

```
          strcpy(mimever, (char *)&p2[strlen(cont)+1]);
          found++;
        }
      }
    }
  }
  if(sscanf(p3, "%s", cont) == 0)       [...]
  if(strcasecmp(cont, "Content-transfer-encoding:") == 0){
    strcpy(tencout, (char *)&p3[strlen(cont)+1]);
    found++;
  } else {
    if(strcasecmp(cont, "Content-lenght:") == 0){
      msg_len = atoi((char *)&p3[strlen(cont)+1]);
      found++;
    } else {
      if(strcasecmp(cont, "Content-type:") == 0){
        strcpy(mimeout, (char *)&p3[strlen(cont)+1]);
        found++;
      } else {
        if(strcasecmp(cont, "MIME-Version:") == 0){
          strcpy(mimever, (char *)&p3[strlen(cont)+1]);
          found++;
        }
      }
    }
  }
  if(sscanf(p4, "%s", cont) == 0)        [...]
  if(strcasecmp(cont, "Content-transfer-encoding:") == 0){
    strcpy(tencout, (char *)&p4[strlen(cont)+1]);
    found++;
  } else {
    if(strcasecmp(cont, "Content-lenght:") == 0){
      msg_len = atoi((char *)&p4[strlen(cont)+1]);
      found++;
    } else {
      if(strcasecmp(cont, "Content-type:") == 0){
        strcpy(mimeout, (char *)&p4[strlen(cont)+1]);
        found++;
      } else {
        if(strcasecmp(cont, "MIME-Version:") == 0){
          strcpy(mimever, (char *)&p4[strlen(cont)+1]);
          found++;
        }
      }
    }
  }
  if(found != 4){
    hcerror("ERROR recv_hclib() - server response has an incorrect header
: ");
    return -9;
  }

[...]
```

Si inizia con il definire, tramite la **strCRLF()** , i vari campi che compongono l'header definendoli tramite un puntatore di inizio e la lunghezza della stringa. Quindi si confrontano questi token con i campi aspettati, assegnando ad ogni variabile il suo giusto valore.

Anche la **get_hclib()** compie il parsing dell'header inviatogli dalla **send_hclib()** in modo da restituire al programmatore non solo il buffer ricevuto, ma anche la sua lunghezza, il transfer encoding utilizzato e il tipo MIME del dato richiesto.

Questi campi sono letti dagli header *Content-Leght*, *Content-Type* e *Content-Transfer-Encoding* indipendentemente dall'ordine in cui sono riportati nell'intestazione. Essendo campi *case insensitive* il confronto prescinde dal fatto che i nomi possano essere scritti in maiuscolo, minuscolo o in una combinazione dei due.

La funzione **post_hclib()** ha la particolarità si spedire inizialmente solo l'intestazione. Perché questa scelta?

In realtà si tratta di una scelta obbligata, perché è molto importante che l'intero header giunga alla corrispettiva funzione di ricezione **recv_hclib()** non frammentato, affinché questa riesca a processare un corretto parsing dello stesso.

Spedendo solamente l'intestazione si è sicuri che la funzione send() riesca a far recapitare interamente il pacchetto (contenente la sola intestazione) in un solo colpo sulla rete, siccome trattandosi di pochi byte la lunghezza dei dati è inferiore alla *MTU* (*Maximum Transfer Unit*) del supporto sul quale è trasmesso.

11
Ricezione del messaggio

Le funzioni della libreria utilizzano le procedure di rete `send()` e `recv()` per inviare e ricevere dati sui socket descriptor connessi.

L'utilizzo di queste specifiche funzioni di I/O anziché delle più comuni system call `write()` e `read()` , che operano su qualsiasi file descriptor, si è reso necessario poiché le prime supportano tutta una serie di parametri opzionali impostabili con l'utilizzo di opportuni flag, di cui abbiamo fatto uso per le specifiche necessità dei pacchetti inviati e ricevuti con la libreria. Caratteristica importante è il fatto che una `recv()` rimpiazza un ciclo di `read()` con la gestione di quanto si è letto e di quanto si deve leggere sullo stack TCP.

In particolare nelle funzioni **recv_hclib()** e **get_hclib()** che ad un certo punto devono ricevere una grossa quantità di dati (l'intero buffer spedito o richiesto dal programmatore), la funzione `recv()` preposta alla ricezione utilizza il flag **MSG_WAITALL**. L'impostazione di questo flag assicura la completa ricezione dell'intero pacchetto, finché il server non interrompe la connessione (con il comando `close()`). Il valore ritornato dalla funzione corrisponde quindi sicuramente alla dimensione (in byte) del pacchetto ricevuto che è la stessa di quello inviato.

Questo controllo può sembrare banale, invece è necessario poiché quando le funzioni **post_hclib()** e **send_hclib()** inviano con una `send()` i dati alle corrispettive funzioni in ascolto, è molto probabile che inviino un numero di byte inferiori alla lunghezza del buffer dati nel caso in cui il numero di dati sia superiore alla disponibilità del buffer interno del sistema operativo.

Nel caso in cui il numero di byte ricevuti sia zero è riportato l'errore

"`remote side has closed the connection`"

che comunica che il server ha chiuso la connessione.

12
Note sulla spedizione dei pacchetti

Nella scrittura di HClib ci si è imbattuti in un problema che riguarda la ricezione dei pacchetti tramite `recv()`. Infatti si deve gestire il problema di ricevere dati che non rimangono tutti in una singola *MTU*, quindi dati frammentati in diversi pacchetti a livello TCP. Sia la `read()` che la `recv()` senza flag ad ogni chiamata ritornano un solo pacchetto TCP, che al massimo può essere grande come l'*MTU* minima sul suo tragitto (*path MTU*).

Il problema può essere parzialmente risolto con un ciclo di `read()` o con una `recv()` che abbia impostato come flag MSG_WAITALL; in questo modo la funzione legge tutto quello che arriva sullo stack fino a una serie di eventi:

- il numero di byte letti raggiungono il valore specificato
- la funzione riceve un segnale
- la connessione termina

La sintassi della funzione, inclusa in *sys/socket* è:

```
ssize_t
recv(int socketd, void *buffer, size_t nbytes, int flag);
```

Ci interessa l'ultimo punto. Nel caso del GET di un dato il server, dopo aver inviato il messaggio richiesto lungo magari molte *MTU*, chiude la connessione con `close()` ; si è quindi stabilito il valore massimo di byte da leggere, *nbytes,* ad un massimo ricevibile dalla funzione. La `recv()` grazie al suddetto flag MSG_WAITALL ritorna quando sono stati letti tutti i dati inviati.

Un problema maggiore si è riscontrato nel caso della connessione POST, dove il client per iniziare la connessione spedisce un testo (composto dall'header POST e dal corpo del messaggio) che può essere lungo molte *MTU*. Quindi la `recv()` del server deve sapere quando interrompere l'ascolto e ritornare i dati letti sullo stack.
Nel caso precedente questo problema non esiste in quanto la `recv()` del server doveva catturare solo la stringa composta da "`GET / HTTP/1.0`", che sta in un pacchetto TCP che attraversi qualsiasi rete (dato che il minimo *MTU* è di 68 byte, considerando gli header).

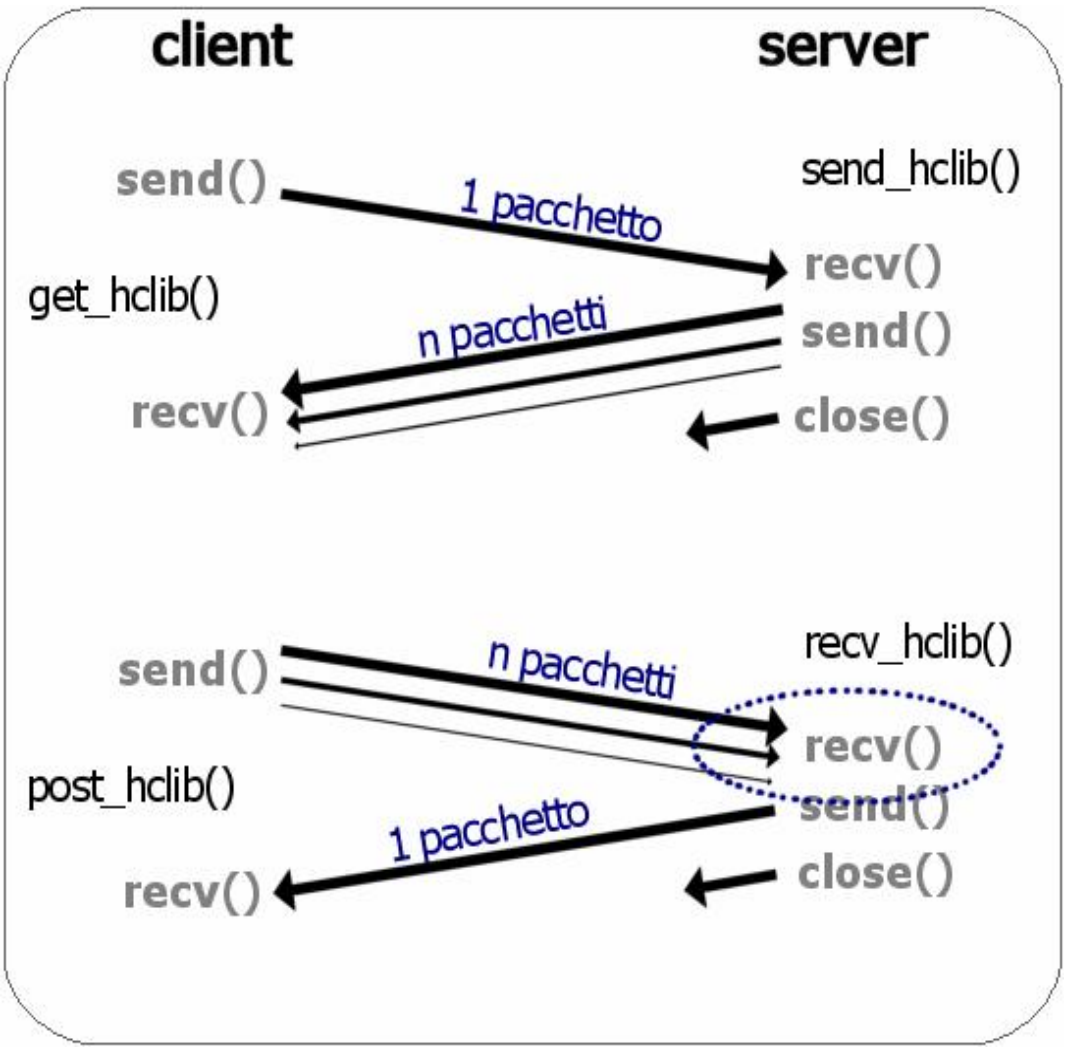
Per spiegare meglio la differenza di situazione riportiamo il seguente schema. La `recv()` del server non sa a priori quanti dati deve ricevere dalla `send()` del client, e in seguito a `send()` il client non può chiaramente terminare la connessione in quanto dopo aver spedito deve ancora ricevere la conferma della ricezione corretta dal server.

Allora si è pensato di risolvere la situazione utilizzando due send() dal lato client e due recv() dal lato server: si creano insomma due connessioni una dopo l'altra. Nella prima trasmissione, viene spedito l'header composto dal "POST / HTTP/1.0" e dai campi HTTP e MIME: si noti che la recv() non ha flag in questa ricezione, in quanto legge un solo pacchetto formato all'header. Inoltre prima di accettare altri dati con la seconda recv(), il server provvede ad elaborare l'header avendo quindi a disposizione il numero di byte del corpo.

```
[…]
recv(sockd, header, MAX_HEAD, 0);
[…elaborazione dell'header…]
recv(sockd, recv_buff, msg_len, MSG_WAITALL);
[…]
```

Infatti la send() successiva del client spedisce esattamente tutto il corpo, e il server può a questo punto utilizzare la seconda recv(), con il flag MSG_WAITALL, impostando il numero di byte da ricevere al numero di byte del corpo letto nell'header.

In questo modo si è sicuri che la recv() del server legga il buffer richiesto e quindi ritorni. Nello schema è evidenziata la recv() che non sa quando finire di ricevere, sostituita quindi da due recv() con i valori di passaggio opportuni. La connessione finisce con il server che utilizza la close(), e il client che riceve la conferma.



13
Codifiche

Le codifiche utilizzate nelle librerie sono basate sul MIME versione 1.0, definito nell' *RFC 1341 "MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies*" del 1992 e poi reso obsoleto dall' *RFC 1521* nel 1993. A sua volta è stato reso obsoleto dalla serie di *RFC 2045, 2046, 2047, 2048, 2049*. Per lo sviluppo della libreria si sono presi in considerazione questi ultimi, in particolare il primo, intitolato "*Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*". Quasi tutti questi *RFC* sono stati proposti da *Borenstein* e *Freed*.

Nel *2045* quindi, è spiegata la procedura per spedire dati codificati in modo che siano composti da stringhe da 6 o 7 bit, oltre chiaramente agli header che integrano l'HTTP specificando tali codifiche. In verità utilizzando un canale HTTP 1.0 pulito non ci sarebbe necessità di codificare, in quanto il formato del testo inviato è su 8 bit, ma sono stati introdotti per compatibilità su piattaforme di diversa codifica ISO estesa e retrocompatibilità con il vecchio *RFC 821*. Comunque possiamo scegliere fra i metodi di spedire dati:

- **binary** vale a dire i dati non sono codificati e vengono spediti così come sono (su 8 bit quindi).

- **7bit** ovvero i dati sono di tipo testo ASCII puro e sono spediti formattandoli su righe corte.

- **8bit** come sopra, ma la codifica può essere quella ISO locale.

- **base64** i dati, su 8 bit, sono codificati 3 per volta su stringhe lunghe 4 di caratteri a 6 bit. Ogni riga è lunga massimo 76 caratteri.

- quoted-printable
si presuppone che i dati siano testuali su 8 bit, e i caratteri estesi sono rappresentati con stringhe di 3 caratteri ASCII standard. Anche qui le righe sono lunghe massimo 76 caratteri.

Nella libreria identifichiamo questi tipi tramite una *struct*, associati ad un *int* che identifica la codifica da eseguire.

```
/*! \struct transferencoding
* Struttura per il riconoscimento del transfer encoding
*/
struct transferencoding{
  char *tp;  //!< tipo di encoding
  short num; //!< flag per encoding
} tenc_struct[] = {
  { "7bit",              1 },
  { "8bit",              1 },
  { "binary",            0 },
  { "quoted-printable", 2 },
  { "base64",            3 }
};
```

Nell'*RFC* si illustra anche il tipo *x-token*, per future aggiunte a questi tipi, ma nella libreria abbiamo preferito poter gestire tutte le casistiche con le nostre librerie, rendendo impossibile l'utilizzo di questo campo (fra l'altro poco utilizzato).

Del tipo binary non c'è molto da dire, i dati sono dei *char*, quindi 8 bit e sono spediti così come sono.

13.1 Caso 7bit, 8bit

Per questi tipi non c'è né una vera e propria codifica né decodifica, per entrambi i casi è chiamata la sotto funzione **break_line()** che provvede a ridurre le righe del testo da spedire a righe di lunghezza desiderata. E' stato imposto un massimo di 76 caratteri, come per il base64 e il quoted-printable (il limite potrebbe raggiungere i 1000 caratteri secondo l'*SMTP,* l'*RFC* invece non specifica una lunghezza precisa).

La funzione cerca l'ultimo spazio in una serie di 76 caratteri, e qui mette un <CR><LF>; se non ne viene trovato neppure uno, taglia la stringa.

13.2 Caso base64

Questa codifica lavora sostanzialmente sul valore binario del carattere, e prendendo 24 bit in ingresso - ovvero tre caratteri su 8 bit - ne restituisce 4 su 6 bit. *base64_table[]* è la stringa su cui lavora la funzione, che spostando con operatori di *shift* e azzerando con degli *and*, genera la quaterna di caratteri corrispondente alla terna.

```
char base64_table[] = {
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  "abcdefghijklmnopqrstuvwxyz"
  "0123456789+/"
};

quad[0] = base64_table[(triple[0] & 0xFC) >> 2];
// FC = 11111100
```

```
quad[1] = base64_table[((triple[0] & 0x03) << 4) | ((triple[1] & 0xF0) >>
4)];
// 03 = 00000011  ,  F0 = 11110000

quad[2] = base64_table[((triple[1] & 0x0F) << 2) | ((triple[2] & 0xC0) >>
6)];
// 0F = 00001111  ,  C0 = 11000000

quad[3] = base64_table[triple[2] & 0x3F];
// 3F = 00111111
```

Inoltre la funzione provvede anche a rendere il testo codificato formattato in righe di esattamente 76 caratteri, inserendo i caratteri di "a capo" <CR><LF>.

La decodifica base64 non considera tutti i caratteri che non siano quelli elencati nella *base64_table[]* con in più il carattere '=' , utilizzato come *padding*, indispensabile per codificare tutte le stringhe con un numero di caratteri non esattamente multiplo di 3. Oltre a ciò, la decodifica utilizza un metodo molto simile alla codifica lavorando sul valore binario dei caratteri e generando terne a partire da quaterne.

Si noti come la codifica comporti un aumento della lunghezza del messaggio pari a circa 1/3 della lunghezza originale.

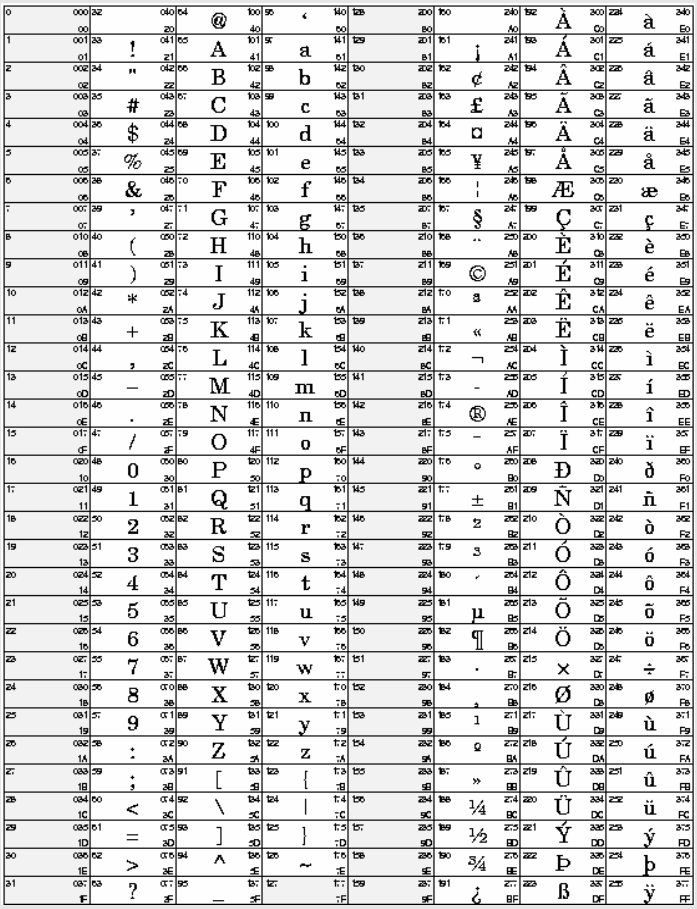
Esempi di tabelle ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	⌂

ASCII di base (stessa per tutte le codifiche ISO)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	|	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	⌂

ASCII estesa standard



ISO-5589-1 ovvero LATIN-1

13.3 Caso quoted-printable

La quoted printable è una codifica che associa ai caratteri *non stampabili*, ai caratteri *estesi* (cioè diversi per ogni codifica locale) e ai caratteri *speciali* una serie di tre caratteri composta da un '=' e due caratteri ASCII standard che rappresentano il valore esadecimale del carattere.

Come esempio il carattere '&' (che ha valore decimale 38) in esadecimale ha valore 26, e quindi in quoted-printable verrà codificato come *"=26"*.

Per poter fare ciò agevolmente si è creata una grande struttura che contiene - ordinati per codice ASCII - la stringa corrispondente al carattere (che può anche essere un solo carattere nel caso delle lettere, numeri e caratteri di base); inoltre la struttura contiene un campo che definisce la lunghezza di tale stringa (quindi 1 oppure 3):

```
struct quoted {
  char          *ch;
  unsigned short lench;
} qp[] = {
    {"=00", 3}, {"=01", 3}, {"=02", 3}, {"=03", 3}, {"=04", 3}, {"=05", 3},
    {"=06", 3}, {"=07", 3}, {"=08", 3}, {"=09", 3}, {"=0A", 3}, {"=0B", 3},
    {"=0C", 3}, {"=0D", 3}, {"=0E", 3}, {"=0F", 3}, {"=10", 3}, {"=11", 3},
    {"=12", 3}, {"=13", 3}, {"=14", 3}, {"=15", 3}, {"=16", 3}, {"=17", 3},
    {"=18", 3}, {"=19", 3}, {"=1A", 3}, {"=1B", 3}, {"=1C", 3}, {"=1D", 3},
    {"=1E", 3}, {"=1F", 3}, {" ",   1}, {"=21", 3}, {"=22", 3}, {"=23", 3},
    {"=24", 3}, {"=25", 3}, {"=26", 3}, {"'",   1}, {"(",   1}, {")",   1},
    {"=2A", 3}, {"+",   1}, {",",   1}, {"-",   1}, {".",   1}, {"/",   1},
    {"0",   1}, {"1",   1}, {"2",   1}, {"3",   1}, {"4",   1}, {"5",   1},
    {"6",   1}, {"7",   1}, {"8",   1}, {"9",   1}, {":",   1}, {"=3B", 3},
    {"=3C", 3}, {"=3D", 1}, {"=3E", 3}, {"?",   1}, {"=40", 3}, {"A",   1},
    {"B",   1}, {"C",   1}, {"D",   1}, {"E",   1}, {"F",   1}, {"G",   1},
    {"H",   1}, {"I",   1}, {"J",   1}, {"K",   1}, {"L",   1}, {"M",   1},
    {"N",   1}, {"O",   1}, {"P",   1}, {"Q",   1}, {"R",   1}, {"S",   1},
    {"T",   1}, {"U",   1}, {"V",   1}, {"W",   1}, {"X",   1}, {"Y",   1},
    {"Z",   1}, {"=5B", 3}, {"=5C", 3}, {"=5D", 3}, {"=5E", 3}, {"=5F", 3},
    {"=60", 3}, {"a",   1}, {"b",   1}, {"c",   1}, {"d",   1}, {"e",   1},
    {"f",   1}, {"g",   1}, {"h",   1}, {"i",   1}, {"j",   1}, {"k",   1},
    {"l",   1}, {"m",   1}, {"n",   1}, {"o",   1}, {"p",   1}, {"q",   1},
    {"r",   1}, {"s",   1}, {"t",   1}, {"u",   1}, {"v",   1}, {"w",   1},
    {"x",   1}, {"y",   1}, {"z",   1}, {"=7B", 3}, {"=7C", 3}, {"=7D", 3},
    {"=7E", 3}, {"=7F", 3}, {"=80", 3}, {"=81", 3}, {"=82", 3}, {"=83", 3},
    {"=84", 3}, {"=85", 3}, {"=86", 3}, {"=87", 3}, {"=88", 3}, {"=89", 3},
    {"=8A", 3}, {"=8B", 3}, {"=8C", 3}, {"=8D", 3}, {"=8E", 3}, {"=8F", 3},
    {"=90", 3}, {"=91", 3}, {"=92", 3}, {"=93", 3}, {"=94", 3}, {"=95", 3},
    {"=96", 3}, {"=97", 3}, {"=98", 3}, {"=99", 3}, {"=9A", 3}, {"=9B", 3},
    {"=9C", 3}, {"=9D", 3}, {"=9E", 3}, {"=9F", 3}, {"=A0", 3}, {"=A1", 3},
    {"=A2", 3}, {"=A3", 3}, {"=A4", 3}, {"=A5", 3}, {"=A6", 3}, {"=A7", 3},
    {"=A8", 3}, {"=A9", 3}, {"=AA", 3}, {"=AB", 3}, {"=AC", 3}, {"=AD", 3},
    {"=AE", 3}, {"=AF", 3}, {"=B0", 3}, {"=B1", 3}, {"=B2", 3}, {"=B3", 3},
    {"=B4", 3}, {"=B5", 3}, {"=B6", 3}, {"=B7", 3}, {"=B8", 3}, {"=B9", 3},
    {"=BA", 3}, {"=BB", 3}, {"=BC", 3}, {"=BD", 3}, {"=BE", 3}, {"=BF", 3},
    {"=C0", 3}, {"=C1", 3}, {"=C2", 3}, {"=C3", 3}, {"=C4", 3}, {"=C5", 3},
    {"=C6", 3}, {"=C7", 3}, {"=C8", 3}, {"=C9", 3}, {"=CA", 3}, {"=CB", 3},
    {"=CC", 3}, {"=CD", 3}, {"=CE", 3}, {"=CF", 3}, {"=D0", 3}, {"=D1", 3},
    {"=D2", 3}, {"=D3", 3}, {"=D4", 3}, {"=D5", 3}, {"=D6", 3}, {"=D7", 3},
    {"=D8", 3}, {"=D9", 3}, {"=DA", 3}, {"=DB", 3}, {"=DC", 3}, {"=DD", 3},
    {"=DE", 3}, {"=DF", 3}, {"=E0", 3}, {"=E1", 3}, {"=E2", 3}, {"=E3", 3},
    {"=E4", 3}, {"=E5", 3}, {"=E6", 3}, {"=E7", 3}, {"=E8", 3}, {"=E9", 3},
    {"=EA", 3}, {"=EB", 3}, {"=EC", 3}, {"=ED", 3}, {"=EE", 3}, {"=EF", 3},
    {"=F0", 3}, {"=F1", 3}, {"=F2", 3}, {"=F3", 3}, {"=F4", 3}, {"=F5", 3},
    {"=F6", 3}, {"=F7", 3}, {"=F8", 3}, {"=F9", 3}, {"=FA", 3}, {"=FB", 3},
    {"=FC", 3}, {"=FD", 3}, {"=FE", 3}, {"=FF", 3},
};
```

Si può accostare questa *struct* alle tavole ASCII precedenti per confrontare i valori dei caratteri ai loro sostituti in esadecimale.

Scrivendo la funzione **qp_encode()** e **qp_decode** si è dovuto tener conto del fatto che i *char* sono *signed*, ovvero la prima parte ASCII di base, fino al carattere 127, sono i numeri positivi +2^4 , invece la parte estesa, da 128 a 255, sono espressi dai numeri negativi, i -2^4 .

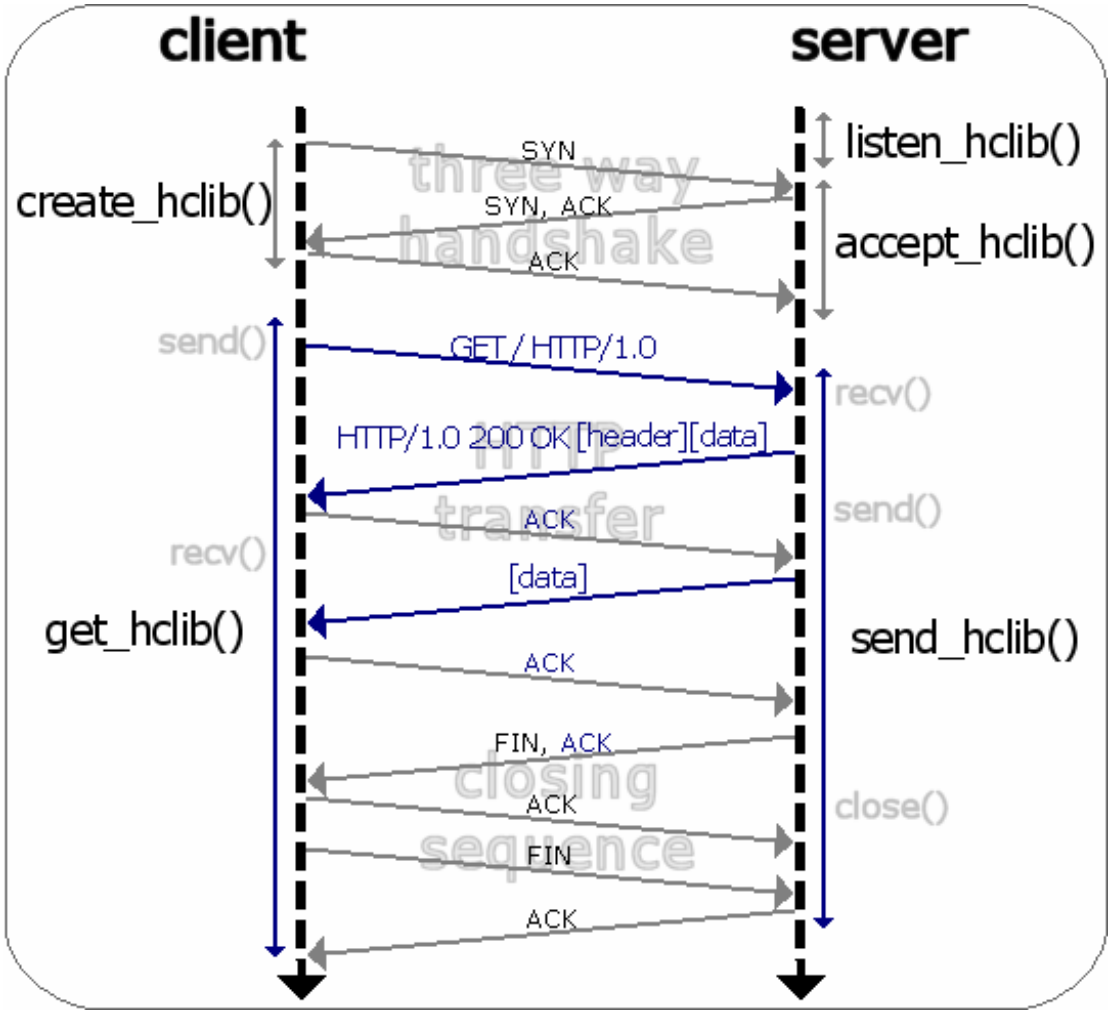
Infine la funzione deve inserire, quando chiude una riga lunga 76 caratteri, un '=', per permettere alla funzione di decodifica di ripristinare la riga originale, rimuovendo quindi gli eventuali <CR><LF> introdotti.

Si noti come la codifica comporti un aumento della lunghezza del messaggio variabile, che può raggiungere nel caso peggiore il triplo (caso in cui tutti i caratteri originali sono codificati con i tre previsti dalla struttura).

14
Illustrazione della connessione

Sono riportate le analisi di una connessione di tipo "POST" e di tipo "GET", con un diagramma a traliccio illustrativo. Qui è quindi possibile riscontrare tutto quanto è stato detto sulle tempistiche e sul contenuto dei pacchetti trasmessi il client e il server.

14.1 Analisi di get_clib() e send_hclib()



E' riportata in forma ridotta la cattura della connessione eseguita con il comando:

```
tcpdump -X -s0 -i xl0
```

I programmi client e server utilizzati sono quelli allegati e servono per la memorizzazione di file su un server (a cui viene assegnato un nome numerico progressivo e estensione basata sull'encoding).

Il server (dal nome "*maia*") genera una lista dei file presenti, da spedire al client ("*elettra*").

```
01:41:39.945123 elettra.32883 > maia.1235: S 2564468443:2564468443(0) win 5840 <mss
1460,sackOK,timestamp 228698 0,nop,wscale 0> (DF)
[…]
```

```
01:41:39.945476 maia.1235 > elettra.32883: S 2149186059:2149186059(0) ack 2564468444
win 65535 <mss 1460,nop,wscale 1,nop,nop,timestamp 22845636 228698> (DF)
[…]
```

```
01:41:39.945520 elettra.32883 > maia.1235: . ack 1 win 5840 <nop,nop,timestamp 228698
22845636> (DF)
[…]
```

```
01:41:39.945567 elettra.32883 > maia.1235: P 1:17(16) ack 1 win 5840
<nop,nop,timestamp 228698 22845636> (DF)
0x0000       4500 0044 d383 4000 4006 e4ac c0a8 006f E..D..@.@......o
0x0010       c0a8 00c4 8073 04d3 98da aedc 8019 fa0c .....s..........
0x0020       8018 16d0 ad19 0000 0101 080a 0003 7d5a ..............}Z
0x0030       015c 98c4 4745 5420 2f20 4854 5450 2f31 .\..GET./.HTTP/1
0x0040       2e30 0d0a                               .0..
```

```
01:41:39.949108 maia.1235 > elettra.32883: . 1:1449(1448) ack 17 win 33304
<nop,nop,timestamp 22845636 228698> (DF)
0x0000       4500 05dc 082f 4000 4006 aa69 c0a8 00c4 E..../@.@..i....
0x0010       c0a8 006f 04d3 8073 8019 fa0c 98da aeec ...o...s........
0x0020       8010 8218 fd5d 0000 0101 080a 015c 98c4 .....].......\..
0x0030       0003 7d5a 4854 5450 2f31 2e30 2032 3030 ..}ZHTTP/1.0.200
0x0040       204f 4b0d 0a43 6f6e 7465 6e74 2d74 7970 .OK..Content-typ
0x0050       653a 2074 6578 742f 7573 2d61 7363 6969 e:.text/us-ascii
0x0060       0d0a 436f 6e74 656e 742d 6c65 6e67 6874 ..Content-lenght
0x0070       3a20 3136 3138 310d 0a43 6f6e 7465 6e74 :.16181..Content
0x0080       2d74 7261 6e73 6665 722d 656e 636f 6469 -transfer-encodi
0x0090       6e67 3a20 3762 6974 0d0a 5365 7276 6572 ng:.7bit..Server
0x00a0       3a20 4843 6c69 622f 312e 300d 0a0d 0a31 :.HClib/1.0....1
0x00b0       3220 2020 2020 2020 2020 2020 2020 2020 2...............
0x00c0       2020 2e2f 2e0a 3132 2020 2020 2020 2020 .../..12........
 […]
0x05c0       2020 2020 2020 2020 2020 2020 2020 2020 ................
0x05d0       2e2f 3237 2e71 756f 0a31 3620           ./27.quo.16.
```

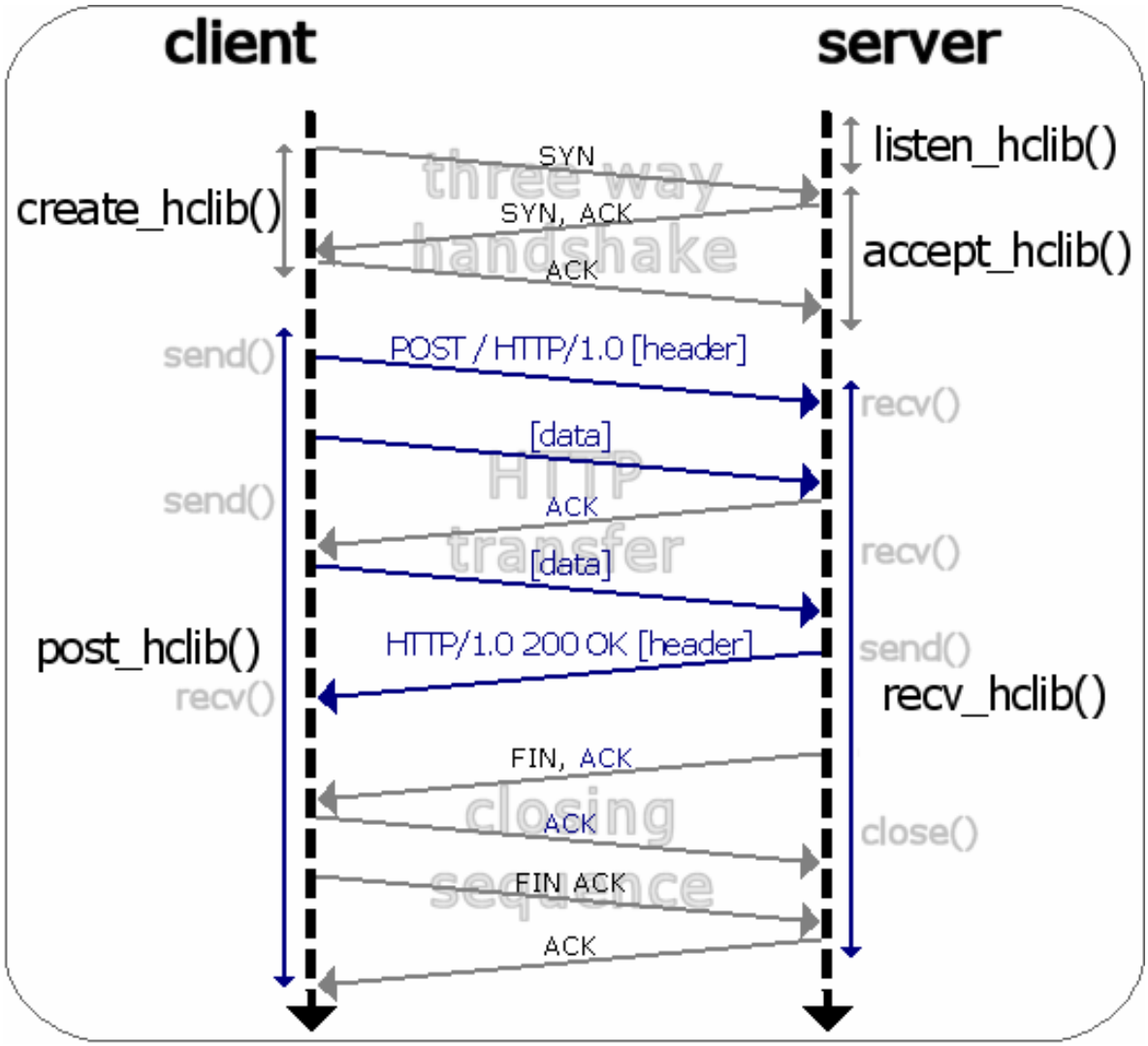
```
01:41:39.949148 elettra.32883 > maia.1235: . ack 1449 win 8688 <nop,nop,timestamp
228698 22845636> (DF)
[…]
```

```
01:41:39.950054 maia.1235 > elettra.32883: P 1449:2539(1090) ack 17 win 33304
<nop,nop,timestamp 22845636 228698> (DF)
0x0000       4500 0476 0830 4000 4006 abce c0a8 00c4 E..v.0@.@.......
0x0010       c0a8 006f 04d3 8073 8019 ffb4 98da aeec ...o...s........
0x0020       8018 8218 0f09 0000 0101 080a 015c 98c4 .............\..
0x0030       0003 7d5a 2020 2020 2020 2020 2020 2020 ..}Z............
0x0040       2020 2020 2e2f 3238 2e71 756f 0a31 3620 ...../28.quo.16.
0x0050       2020 2020 2020 2020 2020 2020 2020 2020 ................
0x0060       2e2f 3239 2e71 756f 0a31 3620 2020 2020 ./29.quo.16.....
[…]
0x0460       2020 2020 2020 2020 2020 200d 0a2e 2f36 ............../6
0x0470       362e 7175 6f0a                          6.quo.
```

```
01:41:39.950077 elettra.32883 > maia.1235: . ack 2539 win 11584 <nop,nop,timestamp
228698 22845636> (DF)
[…]
```

```
01:41:39.950188 maia.1235 > elettra.32883: F 2539:2539(0) ack 17 win 33304
<nop,nop,timestamp 22845636 228698> (DF)
[…]
01:41:39.984083 elettra.32883 > maia.1235: . ack 2540 win 11584 <nop,nop,timestamp
228702 22845636> (DF)
[…]
01:41:40.083668 elettra.32883 > maia.1235: F 17:17(0) ack 2540 win 11584
<nop,nop,timestamp 228711 22845636> (DF)
[…]
01:41:40.083967 maia.1235 > elettra.32883: . ack 18 win 33303 <nop,nop,timestamp
22845650 228711>
[…]
```

14.2 Analisi di post_clib() e recv_hclib()



Anche qui si riporta in forma ridotta la cattura della connessione. Si noti che la porta server da cui spedire dopo un GET è la 1235, invece la porta dalla quale si ricevono file tramite POST è la 1234.

La lista dei file presenti (generata nel sottocapitolo precedente) non è codificata ed è spedita come *7bit*, invece i file ricevuti possono essere codificati in *quoted-printable*, in *base64* oppure non codificati (ovvero *binary*) a seconda della volontà dell'utente del programma.

```
01:40:38.191520 elettra.32880 > maia.1234: S 2508471433:2508471433(0) win 5840 <mss
1460,sackOK,timestamp 222522 0,nop,wscale 0> (DF)
[…]
```

```
01:40:38.191949 maia.1234 > elettra.32880: S 1752986620:1752986620(0) ack 2508471434
win 65535 <mss 1460,nop,wscale 1,nop,nop,timestamp 22839460 222522> (DF)
[…]
```

```
01:40:38.191995 elettra.32880 > maia.1234: . ack 1 win 5840 <nop,nop,timestamp 222522
22839460> (DF)
[…]
```

```
01:40:38.192541 elettra.32880 > maia.1234: P 1:117(116) ack 1 win 5840
<nop,nop,timestamp 222522 22839460> (DF)
0x0000       4500 00a8 59b5 4000 4006 5e17 c0a8 006f E...Y.@.@.^....o
0x0010       c0a8 00c4 8070 04d2 9584 3c8a 687c 73fd .....p....<.h|s.
0x0020       8018 16d0 b25f 0000 0101 080a 0003 653a ....._........e:
0x0030       015c 80a4 504f 5354 202f 2048 5454 502f .\..POST./.HTTP/
0x0040       312e 300d 0a43 6f6e 7465 6e74 2d74 7970 1.0..Content-typ
0x0050       653a 2074 6578 742f 7072 696e 7461 626c e:.text/printabl
0x0060       650d 0a43 6f6e 7465 6e74 2d6c 656e 6768 e..Content-lengh
0x0070       743a 2032 3632 360d 0a43 6f6e 7465 6e74 t:.2626..Content
0x0080       2d74 7261 6e73 6665 722d 656e 636f 6469 -transfer-encodi
0x0090       6e67 3a20 7175 6f74 6564 2d70 7269 6e74 ng:.quoted-print
0x00a0       6162 6c65 0d0a 0d0a                     able....
```

```
01:40:38.192661 elettra.32880 > maia.1234: . 117:1565(1448) ack 1 win 5840
<nop,nop,timestamp 222522 22839460> (DF)
0x0000       4500 05dc 59b6 4000 4006 58e2 c0a8 006f E...Y.@.@.X....o
0x0010       c0a8 00c4 8070 04d2 9584 3cfe 687c 73fd .....p....<.h|s.
0x0020       8010 16d0 f010 0000 0101 080a 0003 653a ..............e:
0x0030       015c 80a4 3d33 4348 3220 616c 6967 6e3d .\..=3CH2.align=
0x0040       6365 6e74 6572 3d33 4574 6f6c 6c61 7269 center=3Etollari
0x0050       3d35 4264 6f74 3d35 4463 756f 7265 3d35 =5Bdot=5Dcuore=5
0x0060       4264 6f74 3d35 446f 7267 3d30 443d 3041 Bdot=5Dorg=0D=0A
0x0070       3d30 443d 3041 693d 3044 3d30 413d 0d0a =0D=0Ai=0D=0A=..
[…]
0x05c0       3030 3020 6d6f 6465 6c20 3630 203d 3236 000.model.60.=26
0x05d0       616d 703d 3342 3d0d 0a20 5641           amp=3B=...VA
```

```
01:40:38.194179 maia.1234 > elettra.32880: . ack 1565 win 32580 <nop,nop,timestamp
22839461 222522> (DF)
[…]
```

```
01:40:38.194249 elettra.32880 > maia.1234: P 1565:2743(1178) ack 1 win 5840
<nop,nop,timestamp 222523 22839461> (DF)
0x0000       4500 04ce 59b7 4000 4006 59ef c0a8 006f E...Y.@.@.Y....o
0x0010       c0a8 00c4 8070 04d2 9584 42a6 687c 73fd .....p....B.h|s.
0x0020       8018 16d0 47d5 0000 0101 080a 0003 653b ....G.........e;
0x0030       015c 80a5 5873 7461 7469 6f6e 2033 3130 .\..Xstation.310
0x0040       3020 3d30 443d 3041 2020 2020 2020 2020 0.=0D=0A........
0x0050       2020 2020 2020 2020 2020 6d6f 6465 6c20 ..........model.
0x0060       3432 3d33 432f 413d 3345 3d33 432f 5444 42=3C/A=3E=3C/TD
0x0070       3d33 453d 3044 3d30 413d 0d0a 2020 2020 =3E=0D=0A=......
[…]
0x04b0       453d 3044 3d30 4120 2020 2020 2020 2020 E=0D=0A.........
0x04c0       2020 2020 203d 3044 3d30 413d 3030      .....=0D=0A=00
```

```
01:40:38.195694 maia.1234 > elettra.32880: P 1:136(135) ack 2743 win 33304
<nop,nop,timestamp 22839461 222523> (DF)
0x0000       4500 00bb 081f 4000 4006 af9a c0a8 00c4 E.....@.@.......
0x0010       c0a8 006f 04d2 8070 687c 73fd 9584 4740 ...o...ph|s...G@
0x0020       8018 8218 8bd8 0000 0101 080a 015c 80a5 .............\..
0x0030       0003 653b 4854 5450 2f31 2e30 2032 3030 ..e;HTTP/1.0.200
0x0040       204f 4b0d 0a43 6f6e 7465 6e74 2d74 7970 .OK..Content-typ
0x0050       653a 2074 6578 742f 7072 696e 7461 626c e:.text/printabl
0x0060       650d 0a43 6f6e 7465 6e74 2d6c 656e 6768 e..Content-lengh
0x0070       743a 2032 3632 360d 0a43 6f6e 7465 6e74 t:.2626..Content
0x0080       2d74 7261 6e73 6665 722d 656e 636f 6469 -transfer-encodi
0x0090       6e67 3a20 7175 6f74 6564 2d70 7269 6e74 ng:.quoted-print
0x00a0       6162 6c65 0d0a 5365 7276 6572 3a20 4843 able..Server:.HC
0x00b0       6c69 622f 312e 300d 0a0d 0a             lib/1.0....
```

```
01:40:38.195765 maia.1234 > elettra.32880: F 136:136(0) ack 2743 win 33304
<nop,nop,timestamp 22839461 222523> (DF)
[…]
```

```
01:40:38.195893 elettra.32880 > maia.1234: . ack 136 win 5840 <nop,nop,timestamp
222523 22839461> (DF)
[…]
```

```
01:40:38.201590 elettra.32880 > maia.1234: F 2743:2743(0) ack 137 win 5840
<nop,nop,timestamp 222523 22839461> (DF)
[…]
```

```
01:40:38.201886 maia.1234 > elettra.32880: . ack 2744 win 33303 <nop,nop,timestamp
22839461 222523>
[…]
```

15
Bibliografia

Richard W. Stevens, 1998
"UNIX Network Programming: Networking APIs: Sockets and XTI", Vol. 1 2nd Edition. Prentice-Hall PTR.
http://www.kohala.com/~rstevens

Mark Mitchell, Jeffrey Oldham, Alex Samuel , 2001
"Advanced Linux Programming", New Riders Publishing.

Brian "Beej" Hall, 2001
"Beej's Guide to Network Programming Using Internet Sockets".
http://www.ecst.csuchico.edu/~beej/guide/net/

Brian W. Kernighan Dennis M. Ritchie, 1989
"The C programming language" 2nd edition, Prentice-Hall INC.

Gli *RFC* sono tratti da:
http://www.ietf.org/rfc

Le tavole ASCII sono tratte da:
http://www.asciitable.com
http://www.alltheweb.com/?cat=img