HTTP adaptation layer
per generico protocollo di scambio dati

Sandro Cavalieri Foschini 101786
Emanuele Richiardone 101790

Programmazione in Ambienti Distribuiti I - 01FQT
prof. Antonio Lioy
A.A. 2002-2003

Cos'è **HClib**?

- *HTTP Channel Library* – realizza un HTTP adaptation layer
- Libreria sviluppata in linguaggio C per realizzare un'interfaccia utile al programmatore per implementare con facilità una connessione HTTP.
- Il programmatore, anziché impiegare le funzioni di rete standard offerte dal SO *NIX, ha a disposizione 7 funzioni principali (più alcune altre di supporto), per inviare in maniera trasparente un qualsiasi dato attraverso un canale HTTP con la tradizionale architettura client/server.
- La comunicazione tra i due host si affida al protocollo affidabile TCP e segue le disposizioni dettate dallo standard HTTP/1.0 (RFC1945).
- I dati che transitano tra i due capi possono essere di tipo ASCII o binario. In entrambi i casi è possibile inviare il dato così come è (sfruttando per i binari la caratteristica *8-bit clean* del protocollo) oppure sfruttare le gli algoritmi di codifica e decodifica integrati nella libreria.
- Sono stati inseriti elementi del formato MIME 1.0 (RFC2045 e successivi). Tra le codifiche dati previste vi sono la base64 (qualsiasi dato) e la quoted-printable (solo per dati testuali).

Come creare un **client**?

ricevere dati da un server tramite GET

- `create_hclib()`
- `get_hclib()`

oppure

spedire dati a un server tramite POST

- `create_hclib()`
- `post_hclib()`

Come creare un client? **GET**

- `create_hclib()`
  - getaddrinfo()
  - socket()
  - connect()
  - setsockopt()

La funzione inizializza la connessione verso il server richiesto, imposta l'intervallo di tempo dopo il quale – in assenza di risposta del server– la connessione cade.

Richiede in ingresso il *nome* e *porta del server* a cui connettersi, il valore (in secondi) per il *timeout*.

Ritorna il valore del socket descriptor se l'operazione si è conclusa con successo.

- `get_hclib()`
  - send()
  - recv()
  - elabora

La funzione spedisce la richiesta, aspetta la risposta con i dati e, se necessario,decodifica.

Richiede in ingresso il *socket descriptor* creato dalla precedente funzione.

In uscita fornisce il *buffer* ricevuto e a sua *lunghezza*, la *codifica* a cui è stato sottoposto al buffer e il *tipo MIME* del dato ivi contenuto.

Come creare un client? **POST**

- `create_hclib()`
  - getaddrinfo()
  - socket()
  - connect()
  - setsockopt()

La funzione inizializza la connessione verso il server richiesto, imposta l'intervallo di tempo dopo il quale –in assenza di risposta del server– la connessione cade.

Richiede in ingresso il *nome* e *porta del server* a cui connettersi, il valore (in secondi) per il *timeout*.

Ritorna il valore del socket descriptor se l'operazione si è conclusa con successo.

- `post_hclib()`
  - elabora
  - send() (header)
  - send() (body)
  - recv()
  - check

La funzione codifica (se necessario) i dati, li spedisce con la richiesta e aspetta e verifica la risposta.

Richiede in ingresso il *socket descriptor* creato dalla precedente funzione, il *buffer* che contiene i dati da inviare, la sua *lunghezza*, l'*encoding* con il quale codificare i dati e il loro *tipo MIME*.

In uscita restituisce il valore 0 se l'operazione si e' conclusa con successo, oppure un codice d'errore.

Come creare un **server**?

- `listen_hclib()`
- `accept_hclib()`
- `send_hclib()`

→ *spedisce* dati a un client che ne fa richiesta con GET

oppure

- `listen_hclib()`
- `accept_hclib()`
- `recv_hclib()`

→ aspetta di *ricevere* dati da un client con POST

Come creare un server? **SEND**

- `listen_hclib()`
  - socket()
  - setsockopt()
  - bind()
  - listen()

La funzione inizializza il socket, lo fa attendere su una porta fissata e restituisce il socket “passivo”.
Richiede in ingresso il numero della porta su cui attendere e il numero backlog di connessioni client da tenere accodate in attesa.
Restituisce il socket descriptor in attesa, o un codice d’errore.

- `accept_hclib()`
  - accept()

La funzione (bloccante!) aspetta che un client si connetta sul sd inizializzato dalla funzione precedente. Ritorna un sd connesso.

- `send_hclib()`
  - elabora
  - recv()
  - check header
  - send()
  - close()

La funzione codifica (se necessario) i dati da inviare riceve l’header della richiesta e lo controlla, spedisce la risposta e chiude la connessione.
Richiede in ingresso il *socket descriptor connesso* dalla precedente funzione, il *buffer* da inviare, la sua *lunghezza*, l’*encoding* a cui verranno sottoposti i dati e il loro *tipo MIME*.

Come creare un server? **RECV**

- `listen_hclib()`
  - socket()
  - setsockopt()
  - bind()
  - listen()

La funzione inizializza il socket, lo fa attendere su una porta fissata e restituisce il socket "passivo".

Richiede in ingresso il numero della porta su cui attendere e il numero backlog di connessioni client da tenere accodate in attesa.

Restituisce il socket descriptor in attesa, o un codice d'errore.

- `accept_hclib()`
  - accept()

La funzione (bloccante!) aspetta che un client si connetta sul sd inizializzato dalla funzione precedente. Ritorna un sd connesso.

- `recv_hclib()`
  - recv()
  - check header
  - recv()
  - send()
  - close()
  - elabora

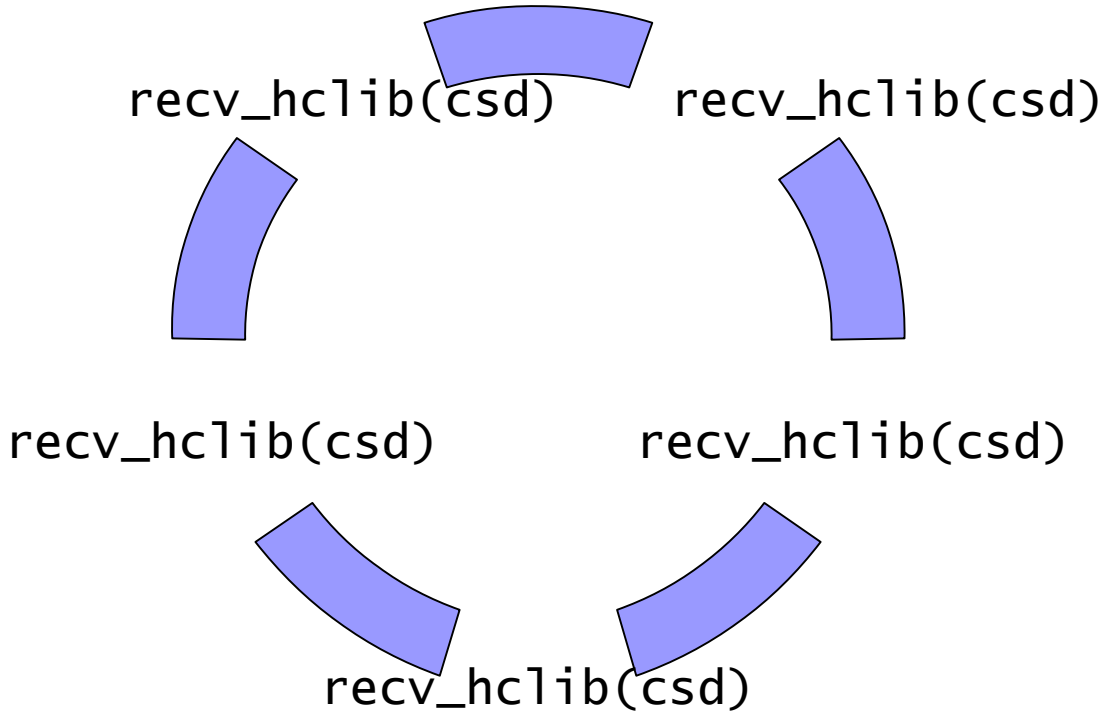
La funzione riceve l'header della richiesta, riceve il corpo, spedisce la risposta (di avvenuta ricezione) e -se necessario- decodifica i dati.

Richiede in ingresso il *socket descriptor* connesso dalla funzione precedente.

In uscita fornisce il *buffer* "riempito" con i dati spediti dalla POST, la sua *lunghezza*, quale *encoding* è avvenuto sul buffer e il *tipo MIME* dei suoi dati.

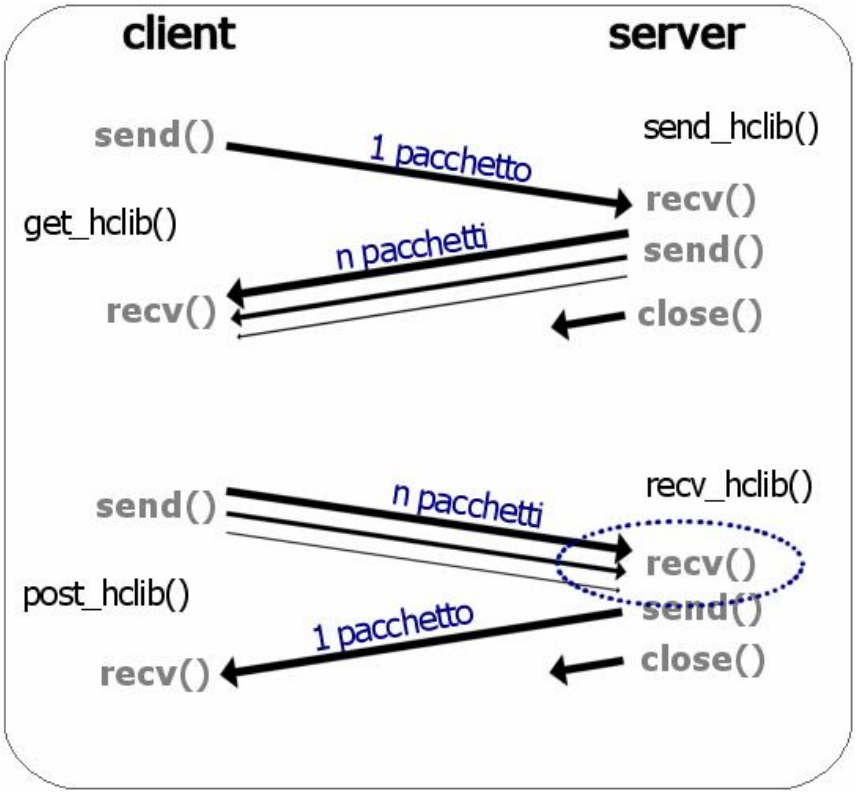
Come creare un server?

- `listen_hclib()`
- `csd=accept_hclib(lsd)`
- Ciclo di “ricezione”



- `close(lsd)`

Gestione **pacchetti TCP**



Il grafico illustra le due situazioni: mentre nella richiesta di GET non si verificano problemi (il lato server deve leggere una richiesta composta dal solo header che è contenuto in un solo pacchetto), nella richiesta POST la send() del client non sa quanti dati ricevere.

Problema: la recv() del server (evidenziata in blu nel grafico a fianco) non sa a priori quanti dati deve ricevere dalla send() del client, e in seguito a send() il client non può terminare la connessione in quanto dopo aver spedito deve ancora ricevere la conferma della ricezione corretta dal server.

La soluzione è utilizzare due send() dal lato client e due recv() dal alto server: si creano due connessioni una dopo l'altra.

Nella prima trasmissione è spedito il solo header, che contiene l'intestazione della richiesta con i campi HTTP e MIME.

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: così sa anche quando può terminare la ricezione.

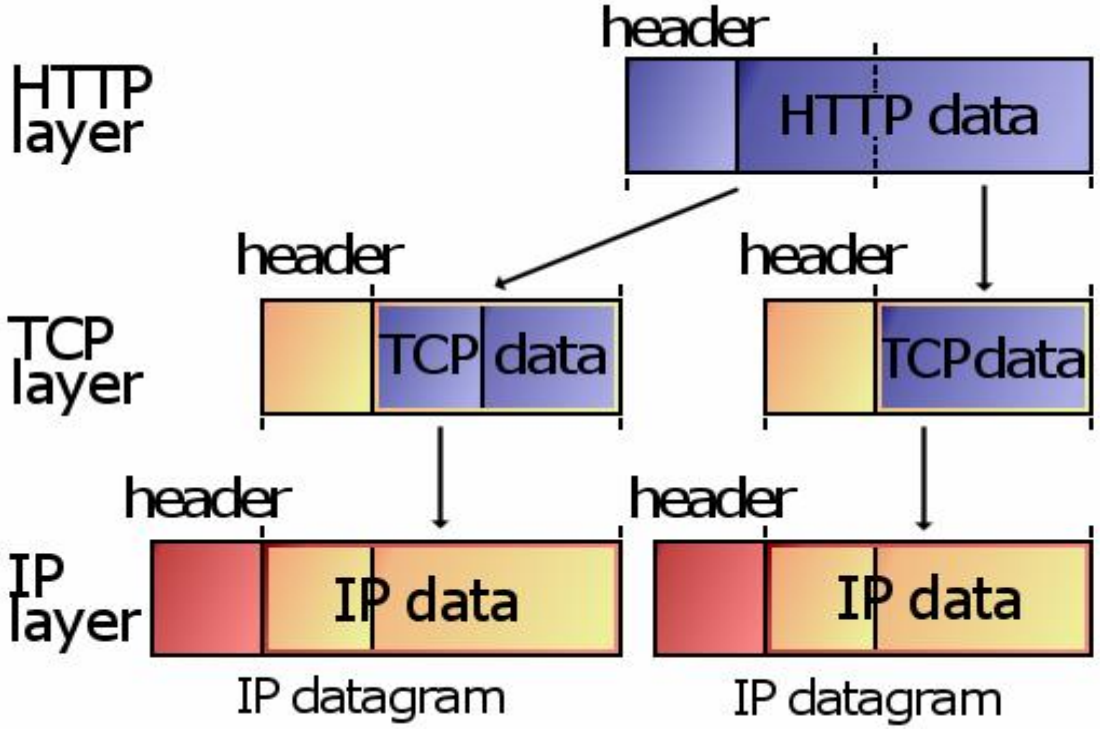
Frammentazione

Problema: ricezione di 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() cha abbia impostato come flag MSG_WAITALL: in questo modo la funzione legge tutto quello che arriva sullo stack fino ad una serie di eventi:

- Il numero di byte letti raggiunge il valore specificato
- La funzione riceve un segnale
- La connessione termina

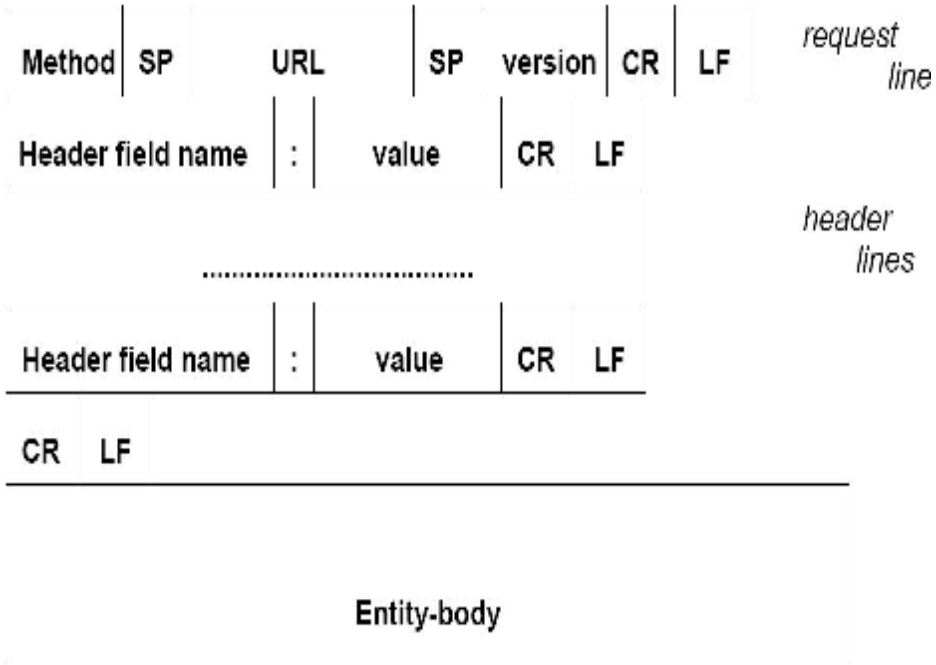


Ci interessa l'ultimo punto. Nel caso del GET di un dato dal server, dopo aver inviato il messaggio richiesto lungo magari molte MTU, chiude la connessione con close(); si è quindi stabilito il valore massimo di byte da leggere che rappresenta il massimo ricevibile dalla funzione.

La recv() grazie al suddetto flag MSG_WAITALL ritorna quando sono stati letti tutti i dati inviati.

Come è fatto un **pacchetto**?

Full-Request del client **`post_hclib()`**



POST / HTTP/1.0

Content-type: text/plain; charset="iso5589-1"

Content-lenght: 12

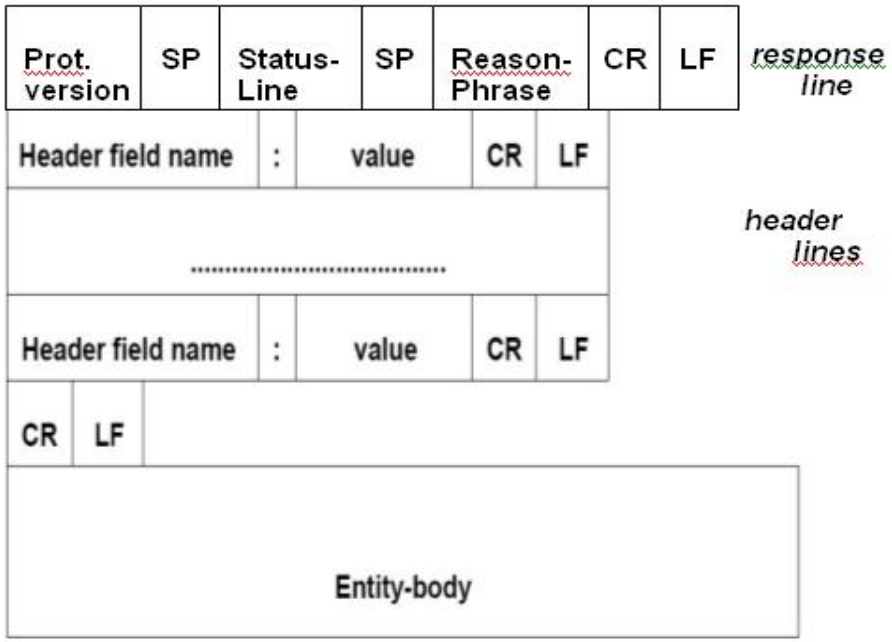
MIME-Version: 1.0

Content-transfer-encoding: 8bit

data, data, data...

Come è fatto un **pacchetto**?

Full-Response del server **`send_hclib()`**



HTTP/1.0 200 OK

Content-type: text/plain; charset="iso5589-1"

Content-lenght: 12

MIME-Version: 1.0

Content-transfer-encoding: 8bit

Server: HClib/1.0

data, data, data...

Come è fatto un **pacchetto**?

Full-Request del client `get_hclib()`

GET / HTTP/1.0

Full-Response del server `recv_hclib()`

HTTP/1.0 200 OK
Content-type: text/plain; charset="iso5589-1"
Content-lenght: 12
MIME-Version: 1.0
Content-transfer-encoding: 8bit
Server: HClib/1.0

get_hclib() – send_hclib()

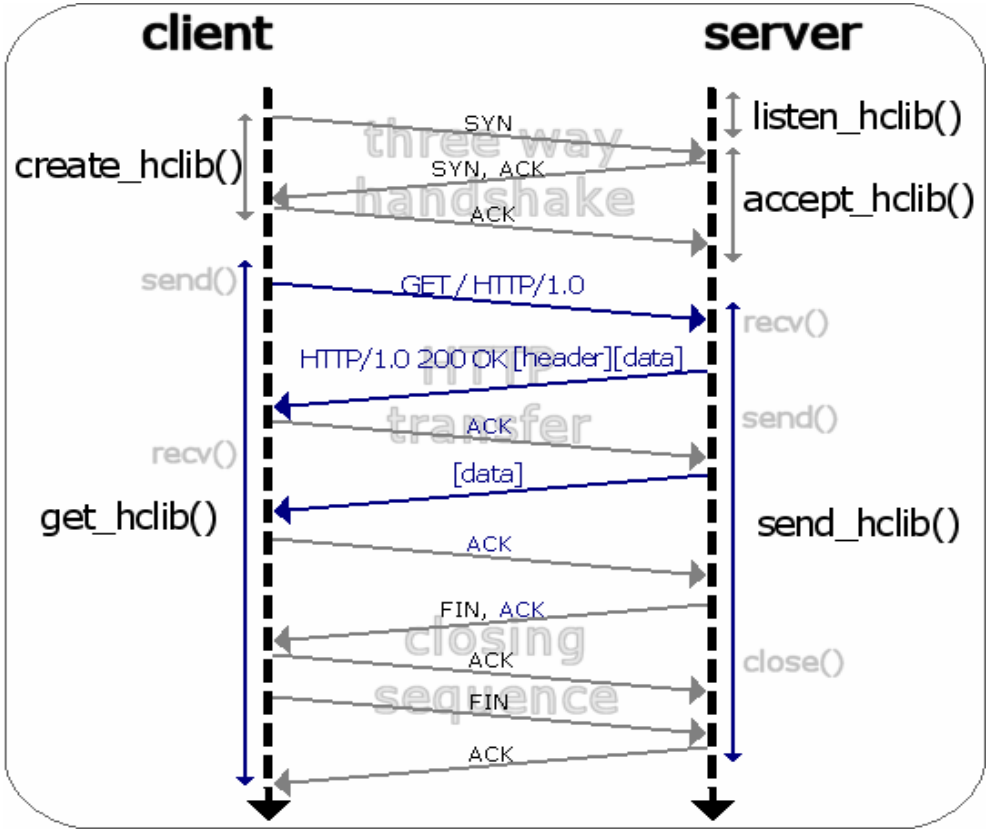


Diagramma a traliccio illustrativo sulle tempistiche e sul contenuto dei pacchetti in transito, nel caso di richiesta del client (GET) di un dato, che verrà spedito dal server con la SEND.

Sullo sfondo (in grigio chiaro) le chiamate alle funzioni di rete effettuate dalle funzioni della libreria (in colore nero). I pacchetti in blu contengono i dati che il programmatore vuole inviare sulla rete.

post_hclib() – recv_hclib()

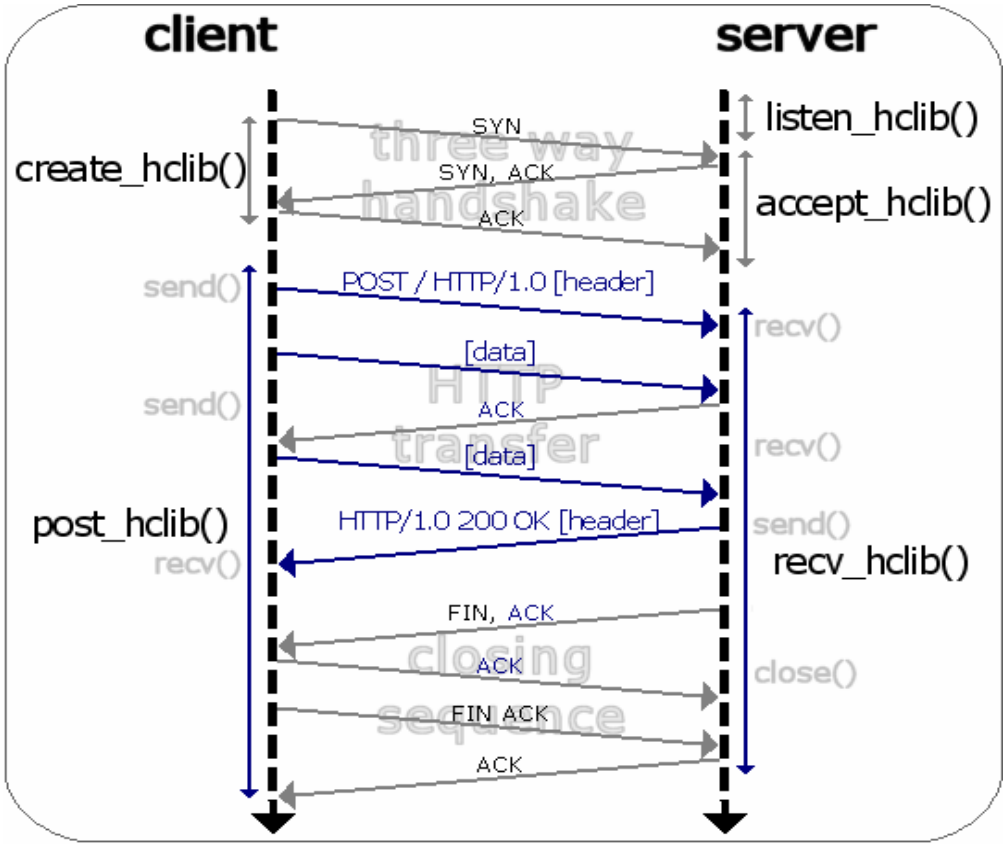


Diagramma a traliccio illustrativo sulle tempistiche e sul contenuto dei pacchetti in transito, nel caso di spedizione del client (POST) di un dato, che verrà ricevuto dal server con la RECV.

Sullo sfondo (in grigio chiaro) le chiamate alle funzioni di rete effettuate dalle funzioni della libreria (in colore nero). I pacchetti in blu contengono i dati che il programmatore vuole inviare sulla rete.

Codifiche

- quoted-printable
  - Utilizzata per testo.
  - 7 bit.
  - Adegua le ISO locali (8 bit).
  - Associa ai caratteri non stampabili, estesi e speciali una serie di tre caratteri (un ‘=’ e due caratteri ASCII standard che rappresentano il valore esadecimale del carattere).
  - Comporta un aumento della lunghezza del messaggio che può raggiungere il triplo della lunghezza originale.
- base64
  - Converte un dato qualsiasi perché agisce a livello binario.
  - 6 bit utilizzati della tabella ASCII (7 bit).
  - Comporta un aumento della lunghezza del messaggio pari a circa 1/3 della lunghezza originale.
  - Testo codificato formattato su righe di 76 caratteri (al massimo).